Corruption de la mémoire lors de l'exploitation

Samuel Dralet[1] and François Gaspard[2]

[1] MISC Magazine
zg@miscmag.com
[2] MISC Magazine kad@miscmag.com

Résumé Les intrusions informatiques sont de plus en plus répandues aujourd'hui. Et plus le temps passe, plus elles évoluent. Si il y a une dizaine d'années, les attaquants ne s'inquiétaient pas beaucoup de laisser des traces sur les systèmes, ce n'est plus le cas aujourd'hui. De plus en plus de méthodes utilisées lors d'une intrusion tentent de ne rien écrire sur le système compromis. Cet article est un petit état de l'art des différentes techniques permettant de n'utiliser que la mémoire du système lors d'une intrusion. Les concepts de *remote userland execve*, *syscall proxy* et *remote library injection* sont expliqués ici. Le logiciel Plato développé dans le cadre de cet article sera également présenté à la fin de l'article. Plato apporte une amélioration dans les techniques *anti-forensics* sous Linux permettant de ne rien écrire sur le disque lors d'une intrusion informatique.

1 Introduction

De tous les grands philosophes grecs, il faut en retenir au moins deux : Platon et Aristote. Depuis plus de 2000 ans, notre monde est sujet à une bataille sans fin entre les idées de ces deux grands philosophes. Pour le premier, ce qui est perçu tous les jours n'est que le reflet de la réalité. Selon lui la réalité est à chercher autre part, là haut, où notre esprit joue un rôle important. C'est le monde des idées. Pour le second, Aristote, c'est tout le contraire. La réalité est bien le monde matériel qui est vu tous les jours, le monde d'en bas, accessible à nos 5 sens.

Que viennent faire deux philosophes dans un article de sécurité ? C'est simple. Dans le monde actuel tel qu'il est perçu, personne ne sait encore très bien où se trouve la réalité (et les récentes découvertes en mécanique quantique sèment encore plus le doute). Dans un système d'exploitation moderne, la réalité d'un système d'exploitation est évidente : c'est ce qui est présent en mémoire vive. Une fois le système en marche, il est possible de créer, supprimer ou même effacer des fichiers. Rien n'y changera, le système est déjà chargé en mémoire.

Et c'est là le plus important. Le système, le noyau ou encore les applications utilisateurs de type serveur qui sont en exécution, sont tous en mémoire. Il est souvent courant de l'oublier, mais il est inutile de garder l'intégrité des fichiers sur le disque si lorsqu'ils sont chargés en mémoire, plus aucun contrôle

n'est effectué. Alors que c'est à cet instant précis que le système devient le plus vulnérable. La tendance aujourd'hui du côté des « méchants » est d'agir au niveau de la mémoire et de ne jamais rien écrire sur le disque, de manière à rester le plus discret possible tout en ayant un contrôle total sur le système exploité. La réplique des « gentils » ne s'est pas fait attendre, de plus en plus de techniques *forensics* se focalisent aujourd'hui sur l'analyse de la mémoire vive.

Ainsi, les techniques des attaquants, pour n'utiliser que la mémoire vive lors d'une attaque, ont considérablement évolué ces dernières années. Au point qu'il est maintenant possible de ne plus jamais toucher au disque dur lors d'une intrusion informatique. C'est précisément ce sujet qui sera traité, une grande partie de ces techniques vont être présentées. Après avoir expliqué leurs qualités et leurs défauts, un schéma d'attaque fonctionnel sous Linux n'utilisant que la mémoire vive sera proposé. Cependant aucun joli code source dans cet article ou sur une page web quelconque ne sera fourni. Le but n'est pas de propager encore plus de logiciels malveillants qu'il n'y en a déjà, mais plutôt de bien faire prendre conscience aux lecteurs de l'importance de la mémoire vive dans le monde de la sécurité informatique d'aujourd'hui.

2 Une intrusion informatique

2.1 Les différentes étapes lors d'une attaque

Lors d'une attaque d'un système par un attaquant (un vrai « pirate », pas un *script kiddie*), plusieurs étapes (au nombre de 5) sont réalisées afin qu'il puisse obtenir le contrôle total du système qu'il vise :

1. Récolter des informations
2. Attaque et pénétration
3. Augmentation des privilèges si nécessaire
4. Installation d'une backdoor
5. Effacer les traces

Peu importe le système visé, ces étapes seront toujours les mêmes, certaines étant plus critiques que d'autres dans le sens où elles vont générer automatiquement des écritures sur le disque visé. C'est précisément ce qui va être évité par la suite.

Récolter des informations Lors de cette première étape, la tâche de l'attaquant est de récolter un maximum d'informations sur la cible visée. Les informations recueillies peuvent provenir principalement de trois sources différentes.

La première source, et la plus évidente, est la source humaine. Pour obtenir des informations sur sa cible, l'attaquant peut utiliser la technique bien connue du *Social Engineering*. Elle consiste à soutirer des informations auprès des personnes proches de l'entreprise ou détenant des informations sur celle-ci (par exemple les employés travaillant pour l'entreprise visée). Si la cible est un particulier, un envoi de mail à celui-ci peut permettre d'obtenir des informations pertinentes (l'utilisation d'une boîte mail avec *webmail* dans un pays étranger permettra de garder l'anonymat).

La deuxième source est le *world wide web*. Cette source est parfois négligée, mais permet en général d'obtenir une foule de renseignements sur la cible. Cependant pour ne pas se disperser dans sa recherche, il est fondamental de bien la cibler. Une bonne connaissance des mots clés de Google (qui reste malheureusement pour nous européens le moteur de recherche le plus efficace ... donc incontournable) est fondamental. Des moteurs spécialisés dans certains types d'informations peuvent également être utilisés pour obtenir des informations utiles : le status d'une entreprise ou encore la composition de son conseil d'administration par exemple. Une fois les membres du conseil connu, la technique du *Social Engineering* peut être utilisée à nouveau pour obtenir des informations supplémentaires. Une bonne utilisation des mots clés de Google peut également permettre d'obtenir des renseignements très intéressants sur le système visé, comme par exemple des fichiers de mots de passe. Un exemple d'ouvrage sur le sujet est celui de Johnny Long [13].

Il est important de noter que ces deux types de recherche n'impliquent pas les systèmes informatiques adverses. La cible ne peut pas se douter qu'elle est visée (du moins si le *Social Engineering* est pratiqué correctement).

La dernière source d'informations, et la plus utile pour l'attaquant, concerne le réseau de l'entreprise visée. Pour obtenir des renseignements, les techniques de *scan* de ports, de *fingerprinting*, etc sont utilisées. Des paquets sont donc envoyés et peuvent être sauvegardés par le système adverse (c'était aussi le cas avec l'envoi de mail pour le *Social Engineering*). La partie devient alors plus dangereuse pour l'attaquant.

Les informations recueillies sont par exemple :

- Les différentes routes pour arriver aux machines visées.
- Les ports ouverts, fermés ou filtrés.
- Des informations provenant d'un serveur DNS (requêtes AXFR, requête inverse, ...).
- Les plages réseaux attribuées.
- Les versions des différents serveurs présents.

Attaque et pénétration Une fois les informations recueillies, l'attaquant sait précisément où il doit agir pour obtenir un accès à la machine adverse. En général, il cible un serveur tournant sur la machine adverse connu pour avoir une faille de sécurité et l'utilise à profit pour pénétrer sur le système. Il s'agit d'exploitation.

Le terme exploitation en sécurité informatique provient du fait qu'un attaquant profite d'une faille de sécurité dans un programme vulnérable afin d'affecter sa sécurité. L'exploitation est réalisée à l'aide d'un programme appelé exploit. Typiquement, un exploit est utilisé soit en local sur le système vulnérable, soit à distance. En local, il servira principalement à augmenter ses privilèges. A distance, il permettra tout simplement d'accéder à un système. Dans ce cas, l'exploit s'utilisera le plus souvent contre un programme de type serveur (serveur web, mail, ftp, ...) tournant sur ce système.

Parmi les failles les plus connues, il existe (termes francisés) :

- un buffer ou heap overflow,
- un format string,
- une race condition.

D'après Ivan Arce [2], un exploit peut être divisé en trois parties : le vecteur d'attaque, la technique d'exploitation et le *payload*. Le vecteur d'attaque est le moyen pour déclencher le bug, par exemple envoyer un certain type de paquet à un serveur. La technique d'exploitation est l'algorithme utilisé pour modifier le flux de contrôle du programme visé. La dernière partie, le *payload* n'est qu'une suite d'instructions exécutées par le programme vulnérable une fois contrôlé.

Les plus anciens *payloads* permettaient simplement d'ajouter un utilisateur sur la machine mais il était nécessaire d'avoir un service de type telnet sur celle-ci pour pouvoir utiliser ce nouvel utilisateur. Ensuite sont apparus des *payloads* ajoutant des services réseaux au démon inetd :

```
echo 'ingreslock stream tcp nowait root /bin/sh sh -i' \ >>/tmp/bob ;
/usr/sbin/inetd -s /tmp/bob
```

Des *shellcodes* sont ensuite développés où l'idée est simplement d'obtenir directement un *shell* pour que l'attaquant puisse interragir avec le système qu'il compromet. Il peut ainsi lancer autant de commandes qu'il souhaite à partir de son *shell*. C'est ici un premier élément intéressant dans le cadre de la corruption de la mémoire, car aucun fichier sur le disque n'est changé ou créé. Par rapport à la technique du service ajouté à inetd par exemple, aucun fichier n'est créé dans **/tmp**, réduisant considérablement les chances d'être détecté.

Au niveau implémentation, les *shellcodes* étaient écrits auparavant directement en assembleur (et certains le sont encore aujourd'hui), mais par la suite des techniques, permettant de les écrire dans des langages plus évolués comme

le C ou le python, sont apparues.

Il n'est pas possible cependant d'exécuter simplement une suite d'instructions qui va lancer un *shell*, il faut aussi que le *shell* arrive sur la machine de l'attaquant. Ce dernier devra donc connaître la topologie du réseau qu'il attaque (*firewall* ou pas ?) pour que le *shell* puisse lui revenir. Il s'agit des *shellcodes* de type *bind shellcode*, *connect back shellcode* et *findsocket shellcode*.

Un *bind shellcode* est un *shellcode* qui *bind* un *shell* sur un port. C'est à dire qu'un nouveau port sur la machine est ouvert et en se connectant à celui ci, l'attaquant obtient un accès à un *shell* sur la machine exploitée.

Un *connect back shellcode* est un *shellcode* qui établit une connexion de la machine attaquée vers la machine de l'attaquant. De cette manière, aucun port n'est ouvert sur la machine en écoute. Ce type de *shellcode* permet la plupart du temps d'outre-passer un *firewall* qui bloque les connections entrantes sur des ports non autorisés.

Un *findsocket shellcode* quant à lui permet de réutiliser la socket ouverte utilisée lors de la connexion entre l'exploit et le serveur vulnérable. Quand un exploit tire parti d'une vulnérabilité dans un serveur distant, il établit une connexion à ce serveur pour pouvoir prendre le contrôle du flux d'exécution. Une connexion est donc créée entre sa machine et la machine visée. L'idée est simplement de réutiliser cette connexion pour exécuter un *shell*. Le grand avantage de cette technique est que l'attaquant utilise un canal de communication connu et fonctionnel entre lui et la machine visée. Cette technique ne crée donc pas de nouvelle connexion suspecte.

Il sera question par la suite des nouveaux types de *payloads* avancés comme ceux créés à partir de MOSDEF, de Inline-egg ou encore de Shellforge.

Augmentation de privilèges Une fois que l'attaquant a pris possesion du système visé, il peut vouloir augmenter ses privilèges. Si le service visé par exemple fonctionne avec les droits d'un simple utilisateur, il est intéressant (sinon primordial) d'obtenir les droits du super utilisateur pour avoir un contrôle total de la machine. Généralement, un exploit local est utilisé contre un programme vulnérable. Dans le monde Unix, le programme vulnérable pourra être un programme ayant les droits root avec le bit *s* (*suid bit*) mis ou bien le noyau lui-même.

Il faut noter que cette étape n'est pas tout le temps nécessaire. Il est possible en effet que le serveur vulnérable compromis par l'attaquant ait déjà les droits du super utilisateur. Heureusement (et malheureusement pour l'attaquant), ce type de serveur se fait de plus en plus rare aujourd'hui.

Installation d'une *backdoor* Une *backdoor* est simplement un dispositif permettant de revenir facilement sur la machine compromise sans passer par toute la phase d'exploitation. Les *backdoors* peuvent aller du simple ajout d'un utilisateur, en passant par l'ajout d'un service sur la machine, jusqu'à la modification d'un service existant. Pour ce dernier point, l'attaquant peut modifier le noyau, le processus qui tourne en mémoire ou encore le binaire sur le disque. Il peut même modifier le BIOS (voir à ce sujet les dernières recherches récentes sur l'ACPI [4]).

Ici, l'éventail des possibilités est énorme, mais il est intéressant de constater que toutes les *backdoors* n'ont pas besoin d'être écrites sur le disque. Tout dépend si l'attaquant modifie le comportement d'un serveur en *backdoorant* le binaire lui-même et en le relançant, ou s'il modifie directement le processus en mémoire.

Effacer les traces La dernière étape pour un attaquant, juste avant de quitter la machine, est d'effacer ses traces. Les fichiers de logs seront analysés pour y enlever toutes traces qui permettraient de remonter jusqu'à l'attaquant. Si l'exploit agit par exemple contre un serveur Apache, il y aura des traces dans le fichier **access.log**. Si un fichier a dû être copié sur le disque, l'attaquant devra également effacer ce fichier de manière sécurisée pour qu'il ne puisse être recupéré par la suite (par exemple avec la commande shred sous Linux).

2.2 Intrusion tout en mémoire

Les différentes étapes qui viennent d'être présentées sont les étapes effectuées lors d'une intrusion classique. Quasiment à chaque étape, des informations sont écrites sur le disque de la machine cible. Au moment de récolter des informations, il peut y avoir par exemple un système de logs qui loggue tous les paquets envoyés au système. Ainsi, si l'attaquant se connecte à un port ou s'il envoie un paquet SYN sur un port, une ligne de log est écrite. Il est bien évident que si un tel système est mis en place, il est difficile voire impossible d'empêcher cela.

L'étape une n'étant pas le sujet de cet article, la suite se focalisera sur les étapes deux à quatre.
L'étape « attaque et pénétration » est la plus importante. C'est à cet instant que l'attaquant utilisera un exploit. Comme mentionné plus tôt, un exploit comporte trois parties : le vecteur d'attaque, la technique d'exploitation et le *payload*. Le vecteur d'attaque et la technique d'exploitation ne sont pas expliqués ici. Beaucoup d'articles sur le sujet ont déjà vu le jour, le lecteur désirant en savoir plus sur le fonctionnement d'un *buffer overflow* pourra trouver toute la documentation nécessaire sur Internet. La partie la plus intéressante est la partie concernant l'écriture et les fonctionnalités du *payload*, tout simplement parce que c'est au cours de cette étape que l'attaquant pourra effectuer l'action qu'il souhaite sur la machine exploitée.

Il faut noter aussi qu'aujourd'hui, les mots *payload* et *shellcode* sont souvent utilisés pour signifier la même chose. Historiquement, un *shellcode* exécutait un

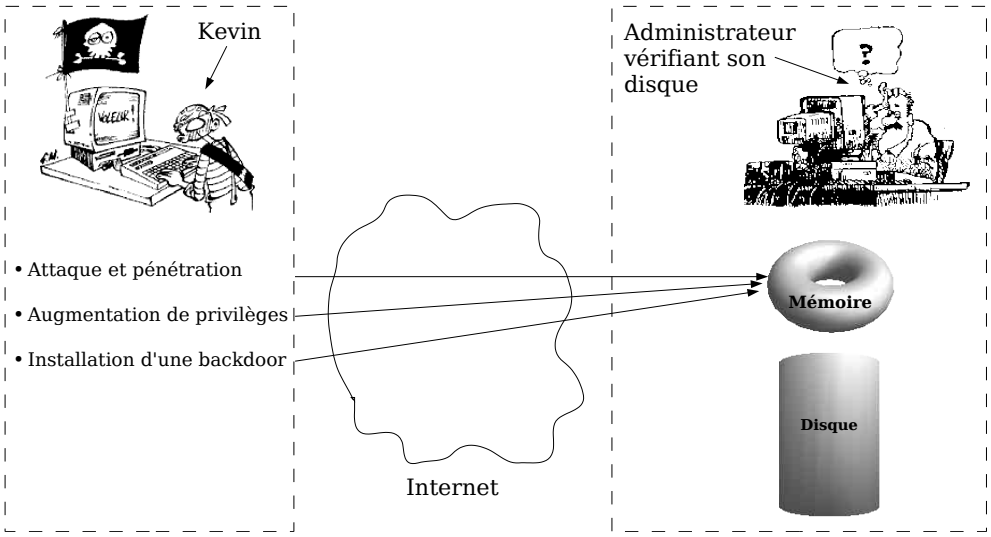

Fig. 1. Intrusion en mémoire, étape 2 à 4

shell, mais aujourd'hui les *shellcodes* sont devenus plus complexes par nécessité : évasion hors d'un *chroot*, installation de *backdoor*, exécution de binaires à distance, élévation de privilèges et la liste n'est pas exhaustive.

A ces fonctionnalités s'ajoutent la notion de forensics dans le sens où l'attaquant développe des *shellcodes* évolués en prenant soin de ne rien écrire sur le disque. Le plus souvent, ces *shellcodes* fonctionnent en 2 parties (un serveur et un client). Après avoir exploité un processus sur une machine distante, un *shellcode* est lancé et attend des données (des portions de code représentées le plus souvent par des *shellcodes*) du client distant. Le code reçu, il le charge et l'exécute soit dans son propre espace d'adressage soit dans l'espace d'adressage d'un autre processus. Tout se passe en mémoire.

Développer de tels *shellcodes* à la « main » serait presque de la folie furieuse. Des outils sont donc apparus. Ils permettent globalement à partir d'un simple programme C ou Python d'obtenir un *shellcode*, aussi complexe soit il.

Concernant l'étape 4 « augmentation de privilièges », elle ne sera pas tout le temps nécessaire. Si le serveur exploité a des droits suffisants pour réaliser les opérations désirées, il ne sera pas nécessaire d'augmenter ceux-ci (typiquement à l'aide d'un exploit local). Dans le cas contraire, il faudra augmenter les privilèges. Cette partie étant un peu spéciale, elle sera uniquement abordée lors de la partie sur Plato qui est l'implémentation proposée dans cet article pour réaliser une intrusion uniquement en mémoire.

L'étape 5 « effacer les traces » ne sera pas abordée dans cet article. Il est évident que si une intrusion informatique se déroule uniquement en mémoire, il n'y aura presque pas de traces sur le système. De plus, les éventuelles traces laissées sur le disque dépendent fortement des types d'exploits utilisés, des services visés et surtout du niveau de compétence de l'attaquant. A noter tout de même qu'un administrateur vigilant (ce qui n'est pas le cas à la figure 1) pourrait très bien vérifier ce qui se trouve en mémoire, c'est tout à fait possible mais plus difficile. Ce point sera abordé à la fin de l'article dans la partie sur les protections.

3 Exécution à distance et techniques existantes

Il a été question de *shellcodes*. Certaines techniques ont pour objectif de pouvoir les exécuter sur une machine distante. Un *shellcode* n'est en fait qu'une suite d'instructions déjà préparées à être exécutées. Mais il peut être question de binaire complet (nmap par exemple) à récupérer et exécuter. C'est le rôle des *syscall proxy* et des *remote userland execve*. Il existe un point commun entre ces techniques, celui de ne jamais rien écrire sur le disque.

3.1 Ecriture de payloads

Inline-egg Inline-egg [5] est un module python qui fournit à l'utilisateur un ensemble de classes permettant d'écrire des *shellcodes* plus facilement, le vecteur d'attaque et la technique d'exploitation devant dans ce cas être écrit aussi en python. Il est développé par Gera de la société Core-Security. Une version GPL est disponible sur le site de la société[3] mais Inline-egg a aussi été intégré dans le logiciel Core-Impact[4].

D'habitude, les *shellcodes* sont la plupart du temps écrit directement en assembleur. Une fois compilés, ils sont insérés dans l'exploit sous forme de chaines de caractères :

```
char shellcode[]=
    "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3"
    "\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80";
```

Ce *shellcode* exécute simplement **/bin/sh**. Pour modifier ce *shellcode*, par exemple pour lui ajouter un **setuid(0)** et **setgid(0)**, il est nécessaire de toucher au code assembleur, de le recompiler et de changer la chaîne de caractères dans l'exploit. Si pour de petits *shellcodes* comme celui là ce n'est pas encore trop contraignant, inutile de dire que ce n'est pas le cas pour des *shellcodes* plus complexes. Une certaine souplesse lors de la création de *shellcodes* est donc la

[3] `http://www.coresecurity.com`
[4] `http://www.coresecurity.com/products/coreimpact/index.php`

bienvenue. Inline-egg part de cette constatation.

L'idée d'Inline-Egg est de créer des *shellcodes* en concaténant simplement des appels système (ou *syscalls*), facilitant ainsi le changement des arguments passés aux appels système. Si on veut un *shellcode* (ou *egg*) exécutant **/bin/sh** juste après un appel à **setuid()**, il suffit de concaténer ces deux appels système pour créer le *shellcode* :

```
setuid(0)
execve('/bin/sh',('sh','-i'))
```

Il suffit de reproduire cet enchainement en python en utilisant Inline-Egg qui compilera le code en assembleur :

```
#!/usr/bin/python

from inlineegg.inlineegg import *
import socket
import struct
import sys

def stdinShellEgg():
   egg = InlineEgg(Linuxx86Syscall)

   egg.setuid(0)
   egg.setgid(0)
   egg.execve('/bin/ls',('ls','-la'))

   return egg
...
```

Le *shellcode* ainsi créé comporte 3 appels système. Il est possible de cette manière de concaténer autant d'appels système qu'il en existe. Cependant, le version GPL d'Inline-egg n'est pas terminée, tous les appels système ne sont pas implémentés. Malgré tout, il permet déjà d'utiliser un certain nombre d'entre eux comme ceux en rapport à la couche réseau, un *bind shellcode* pouvant être implémenté en quelques lignes simplement.

Inline-egg ne se contente pas seulement d'offrir la possibilité de concaténer des appels système, il est également possible d'utiliser des boucles (**while**, **do-while**), des conditions **if** et même de créer ses propres fonctions. Par exemple une condition peut être créée de cette manière :

```
uid = egg.getuid()
no_root = egg.If(uid, '!=', 0)
no_root.write(1,'You are not root!\n', len('You are not root!\n'))
no_root.exit(1)
```

```
no_root.end()
egg.write(1,'You are root!\n', len('You are root!\n'))
```

Une dernière chose intéressante avec Inline-egg est la possibilité de créer des *shellcodes* pour plusieurs systèmes d'exploitations différents (tous sous une architecture i386) : Linux, Freebsd, Openbsd, Solaris et Windows. Concernant Windows, il n'y a pas d'appels système à proprement parler. Les appels système doivent plutôt être vus comme n'importe quelles fonctions dans n'importe quelles librairies (ou **dll**). Pour pouvoir utiliser Inline-egg contre un système Windows, il sera nécessaire de fournir deux adresses de fonctions, celles de **LoadLibrary** et de **GetProcessAddress** qui se trouvent dans la librairie **kernel32.dll**. Il existe plusieurs méthodes pour connaître ces adresses (traverser la liste des librairies en mémoire et regarder la table d'export, utiliser des valeurs *hardcodé* par service pack, ...), néanmoins l'utilisation d'Inline-egg pour la création de *shellcode* Windows n'est pas encore assez fonctionnelle, du moins pour ce qui est de la version *open source* d'Inline-egg

MOSDEF MOSDEF est l'abréviation de « *Most Definitely* ». Cet outil a été développé par Dave Aitel, fondateur de la société Immunitysec célèbre pour son outil CANVAS et sa mailing list Daily Dave.

MOSDEF permet lui aussi d'envoyer des *shellcodes* à une machine cible lors d'une exploitation. Cependant contrairement à Inline-egg ou à Shellforge (qui sera vu après), il permet d'envoyer plusieurs *shellcodes* vers la machine cible. Il fonctionne en mode client-serveur, c'est à dire que lors d'une exploitation, un serveur est installé dans la mémoire du processus exploité. Il s'agit d'un *multi-stage shellcode*, un *shellcode* en plusieurs étapes. Lors d'une exploitation, par exemple avec un *buffer overflow*, il arrive souvent qu'il n'y ait pas assez d'espace dans le buffer contrôlé. L'idée est alors d'envoyer un petit *shellcode* qui va attendre sur une socket un *shellcode* plus avancé. Ce deuxième *shellcode* pourra alors être placé sur la pile où il y a moins de contraintes d'espace et sera exécuté par le premier *shellcode*. Il est ainsi possible d'exécuter plusieurs *shellcodes* dans le processus exploité.

MOSDEF utilise ce principe, mais il est également un compilateur C et un assembleur écrit en python. C'est à dire que ce ne sont pas de simples *shellcodes* qui sont envoyés mais du code ressemblant à du C compilé avec MOSDEF. Le langage C créé n'est pas aussi puissant que le langage C connu, mais offre néanmoins certaines fonctionnalités basiques comme l'utilisation de tableaux, fonctions, boucles **while** et **if**. Le code C créé est compilé du côté client, envoyé au serveur et exécuté dans l'espace d'adressage du processus.

Le principal problème de MOSDEF vient du fait qu'implémenter un compilateur n'est pas si aisé. Si l'attaquant souhaite exécuter du code complexe sur la machine cible, il ne pourra le faire avec MOSDEF. Par exemple utiliser l'appel système **ptrace()** est impossible. Il faudra également connaître la syntaxe

du langage C créé pour pouvoir l'utiliser. Pour finir, la version *open source* est non-documentée et il semble difficile de l'utiliser tel quelle.

Shellforge Shellforge est un générateur de *shellcodes* ou *payloads* développé en python par Philippe Biondi. La version actuelle est la version G2 [4] encore en *pré-release* mais avec l'immense avantage d'être multi-plateformes.

Comment fonctionne Shellforge ? D'un point de vue utilisateur, c'est assez simple. Il suffit de lui fournir un programme en C, et Shellforge le transforme directement en *payload*. Seulement le fichier source C doit avoir quelques spécificités : toutes les variables globales doivent être déclarées en statique de manière à ne pas utiliser la **GOT**[5], il ne doit pas y avoir de déclaration **include** et il est nécessaire certaines fois d'utiliser les fonctions internes à Shellforge (par exemple celles qui permettent le traitement des adresses IPs). Une fois le *payload* obtenu, Shellforge offre plusieurs options d'affichages qui vous permettent de le récupérer sous forme de chaines de caractères (comme il est courant de le voir dans un exploit), en mode *raw* ou dans un fichier source C.

D'un point de vue architecture c'est un peu plus compliqué. Avant d'expliquer comment fonctionne Shellforge, il est nécessaire de faire un bref rappel sur la déclaration des appels système. Sur une plate-forme Linux, chaque appel système est représenté par un nombre déclaré dans le fichier **/usr/include/asm/unistd.h** :

```
#define __NR_exit         1
#define __NR_fork         2'
#define __NR_read         3
#define __NR_write        4
```

Et chaque appel système peut être appelé avec les macros définies dans ce même fichier, à condition de connaître le nombre d'arguments dont il a besoin. Pour l'appel système **setuid()** par exemple :

```
#define __NR_setuid  23

#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(arg1))); \
__syscall_return(type,__res); \
}

static inline _syscall1(int, setuid, int, uid)
```

[5] Global Offset Table

Shellforge procède de la même façon, il appelle les appels système directement en utilisant par contre ses propres macros (les numéros des appels système restant les mêmes) :

```
#define _sfsyscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("pushl %%ebx\n\t"     \
                  "mov %2,%%ebx\n\t"    \
          "int $0x80\n\t"       \
                  "popl %%ebx"          \
    : "=a" (__res) \
    : "0" (__NR_##name),"g" ((long)(arg1))); \
__sfsyscall_return(type,__res); \
```

Ces macros ont l'avantage par rapport à celle de la **Libc** de pouvoir extraire **errno** de la valeur de retour, et de préserver le registre **%ebx**.

Pour toutes ces déclarations, Shellforge utilise la librairie **sflib** et pour chaque architecture il existe un répertoire :

- hpux_hppa,
- linux_arm,
- linux_i386,
- linux_mips,
- linux_powerpc,
- macos_powerpc,
- solaris_sparc,
- freebsd_i386,
- linux_alpha,
- linux_hppa,
- linux_m68k,
- linux_mipsel,
- linux_sparc,
- openbsd_i386.

Dans chaque répertoire sont présents 3 fichiers, le fichier **sfsysnr.h** pour les numéros des appels système, le fichier **sfsyscall.h** pour les macros et le fichier **sflib.h** pour les prototypes (pour chaque **__NR_name**, **name()** est déclaré dans ce fichier). Il est évident que pour ajouter une plate-forme supportée par Shellforge, il suffit de créer un nouveau répertoire dans **sflib** avec les trois fichiers .h qui vont bien.

Ensuite, pour obtenir à partir d'un fichier source C un *shellcode* pour l'architecture souhaitée, il suffit d'utiliser l'option –**arch** en ligne de commande :

```
shellforge.py -C --arch=freebsd_i386 hello.c
```

C'est aussi simple que cela.

Après avoir généré le *shellcode*, Shellforge offre des fonctionnalités qui permettent de le transformer. Il suffit de spécifier sur la ligne de commande le *loader* souhaité :

– XOR *loader*,
– alphanumeric *loader*,
– stack relocation *loader*.

Ces *loaders* qui dépendent du processeur peuvent être chainés ensemble.

Nous avons utilisé assez souvent Shellforge pour développer l'outil que nous allons présenter à SSTIC, pour être d'avis que c'est certainement le générateur de *payloads* le plus puissant dans sa discipline (et ce n'est pas pour être chauvin[6]). Il est portable et il offre une grande souplesse de fonctionnement : tests des *shellcodes* faciles à mettre en oeuvre, modifications rapides et sans connaitre l'assembleur. Shellforge a été testé avec un *shellcode multi-stage* et une fonction **read()** qui attendait en entrée un *payload* pour l'exécuter. Il a été possible aussi de coder un *shellcode* qui injecte une libraire dans un processus, ce dont les autres générateurs sont incapables. Le seul « souci » rencontré est l'utilisation des fonctions de la **Libc** qui ne sont pas des appels système tel que **memset()**, **memcpy()** etc (ce que Philippe Biondi appelle des idioms [5] et qui sont dépendants du processeur). La solution, en tout cas sur un système Linux, est d'utiliser les implémentations fournies dans la **glibc**. Pour **memset()** par exemple, il existe le fichier **glibc-2.3.2/string/test-memset.c** dans lequel il y a une implémentation simpliste :

```
char *
simple_memset (char *s, int c, size_t n)
{
  char *r = s, *end = s + n;
  while (r < end)
    *r++ = c;
  return s;
}
```

3.2 *Syscall proxy*

Rappel sur les appels système
Fonctionnement d'un point de vue utilisateur

Le système d'exploitation offre aux processus s'exécutant en mode utilisateur un ensemble d'interfaces lui permettant d'interagir avec les dispositifs matériels tels que le processeur, les disques, les imprimantes, etc. Les systèmes UNIX

[6] En tous les cas pour Samuel, François étant belge, ça ne le tracasse pas beaucoup :-)

implémentent l'essentiel des interfaces entre le processus en mode utilisateur et le matériel par le biais d'appels système émis vers le noyau. Il faut faire la différence ici entre une API[7] et un appel système. La première est une définition de fonctions spécifiant comment obtenir un service donné, alors que le second est une requête explicite faite au noyau via une interruption logicielle.

Sous Linux, la principale API est la **Glibc** (*GNU Library C*). Cette librairie offre un ensemble de fonctions agissant comme des interfaces vers les appels système du noyau. Elle agit comme un *wrapper*.

Pour mieux illustrer ce principe, voici un exemple :

```
$ cat example.c
int main(void)
{
printf("Fonction printf ! \n");
return 0;
}
$ gcc -o example example.c
$ ./example
Fonction printf !
$
```

example.c est un simple programme qui affiche une chaîne de caractères à l'écran. Pour voir ce qu'il se passe au niveau des appels système lors de son exécution, il suffit d'utiliser la commande **strace** qui trace les signaux et les appels système d'un programme en exécution.

```
$ strace ./example
1: execve("./example", ["./example"], [/* 34 vars */]) = 0
2: uname({sys="Linux", node="Joshua", ...}) = 0
...
9: open("/lib/libc.so.6", O_RDONLY)        = 3
10: read(3,"\177ELF\1\1\1\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\20U\1\000"...,512)
         = 512
...
21: mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
          -1, 0) = 0x40017000
22: write(1, "Fonction printf ! \n", 19)    = 19
23: munmap(0x40017000, 4096)                = 0
24: exit_group(0)                           = ?
$
```

La sortie de la commande **strace** montre les différents appels système utilisés par le binaire **example** : **execve()** permet de lancer le programme, **open()**

[7] Application Programmer Interface

ouvre le fichier **/lib/libc.so.6** qui représente la **Libc** nécessaire au bon fonctionnement du programme (tout programme compilé avec la **Libc** possède une ligne similaire). La ligne 22 est la plus intéressante :

```
22: write(1, "Fonction printf ! \n", 19)    = 19
```

L'appel système *write()* est utilisé avec 3 arguments. Sa définition d'après la page **man** est la suivante :

```
ssize_t write(int fd, const void *buf, size_t count);
```

Le premier argument est le descripteur de fichiers. Il représente dans l'exemple **stdout** qui est la console. Le deuxième argument est la chaîne de caractères à imprimer. Et le troisième est la taille de cette dernière. La fonction **printf()** a donc bien été transformée en un appel système **write()** avec les bons paramètres.

Fonctionnement d'un point de vue noyau

La gestion d'un appel système dans le noyau est réalisée par le gestionnaire des appels système, qui réalise principalement les opérations suivantes :

1. Il sauvegarde le contenu de la plupart des registres dans la pile noyau.
2. Il invoque la routine de service de l'appel système.

La table des appels système est enregistrée dans le tableau **sys_call_table** et contient 255 entrées. Chaque entrée de cette table contient l'adresse de la routine de service correspondante (**exit()** pour la première entrée, **fork()** pour la seconde, ...). A chaque appel système correspond donc un numéro. Au moment d'en appeler un, l'interruption **0x80** est levée et le contrôle est donné au gestionnaire.

La fonction **system_call()** implémente ce fameux gestionnaire. Il commence par sauvegarder sur la pile le numéro de l'appel système ainsi que tous les registres du processeur susceptibles d'être utilisés par le gestionnaire[8], les paramètres de l'appel système étant passés par les registres dans le cas où 5 arguments maximum sont nécessaires. Dans le cas contraire (avec l'appel système **mmap()** par exemple), les paramètres sont passés par la pile (ou *stack*) puisqu'il n'y a pas assez de registres.

La routine de service de l'appel système correspondant au numéro stocké dans le registre **eax** est ensuite appelée.

[8] Certains registres sont en fait sauvegardés automatiquement par l'unité de contrôle comme *eflags, cs, eip, ss et esp.*

Un exemple valant mieux qu'un long discours, le programme suivant démontre en assembleur comment faire appel à un appel système, en l'occurence dans cet exemple **sys_open()** qui ouvre le fichier **/boot/System.map** :

```
$ /bin/cat sys_open.c
#include <linux/unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define FILENAME "/boot/System.map"

int
main( void )
{
        long __res;

        __asm__ volatile ("int $0x80"
              : "=a" (__res)
              : "0" (__NR_open), "b" (FILENAME), "c" (O_RDONLY) );
}
$ make sys_open
cc     sys_open.c   -o sys_open
$ strace ./sys_open
[...]
munmap(0x40014000, 14874)               = 0
getpid()                                = 354
open("/boot/System.map", O_RDONLY)      = 3
_exit(3)                                = ?
```

Les appels système (ou *syscalls*) sont la base du fonctionnement de la technique appelée *syscall proxy*. Celle-ci est une des deux grandes techniques permettant d'exécuter des binaires sur une machine distante sans jamais rien écrire sur le disque.

Théorie du *syscall proxy* La première implémentation du concept du *syscall proxy* a été développée par Maximiliano Caceres (est-ce l'inventeur du concept ?) de la société Core Security Technologies et a été utilisée dans leur produit phare Core-Impact, un outil de tests de pénétration.

Le *syscall proxy* a certainement été la première technique évoluée pour exécuter des binaires sur la machine distante tout en restant en mémoire. Son principe est simple : appeler un appel système sur une machine distante depuis une machine locale. Plusieurs appels système concaténés ensemble représentent l'exécution d'un programme à l'image des *shellcodes* créés avec Inline-egg. Pour ouvrir, lire et fermer un fichier par exemple, trois appels système sont utilisés : **open()**,

read() et **close()** avec les paramètres qui vont bien (figure 2). Le programme (un exploit par exemple) n'est donc pas téléchargé sur le système distant mais exécuté localement grâce à la couche d'abstraction fournie par le *syscall proxy*. Celui-ci joue le rôle d'interface entre le processus (pour la machine locale) et le système d'exploitation (pour la machine distante). Le processus ne communique donc plus directement avec l'OS sous-jacent mais via la nouvelle interface *syscall* formalisée (figure 3).

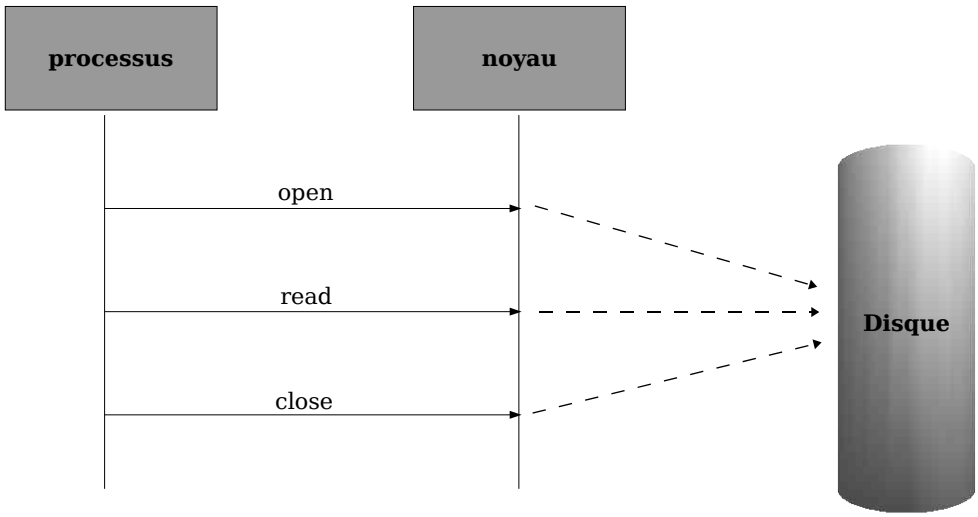

Fig. 2. Lecture d'un fichier par un processus

A noter que seuls les appels système sont exécutés sur la machine distante, tout le traitement arithmétique (une addition par exemple), les boucles, les conditions, l'allocation mémoire, etc sont faits sur la machine locale. **memset()** ou **strcpy()** par exemple n'ont aucun intérêt d'être exécutés sur la machine distante. Il est évident par contre que l'algorithme du programme ne changera pas quelle que soit la machine (locale ou distante). La seule différence est avec quels privilèges seront appelés les appels système.

Implémentation d'un *syscall proxy* D'après le rappel sur les appels système, leurs exécutions se déroulent en trois étapes : le numéro de l'appel système et les arguments à fournir, l'exécution de l'appel système et la valeur de retour à récupérer.

La première étape est le rôle du client *syscall proxy*. Pour chaque appel système, il construit une requête que comprendra le serveur dans laquelle sont renseignés le numéro de l'appel système, et ses arguments. Il adapte le code ou

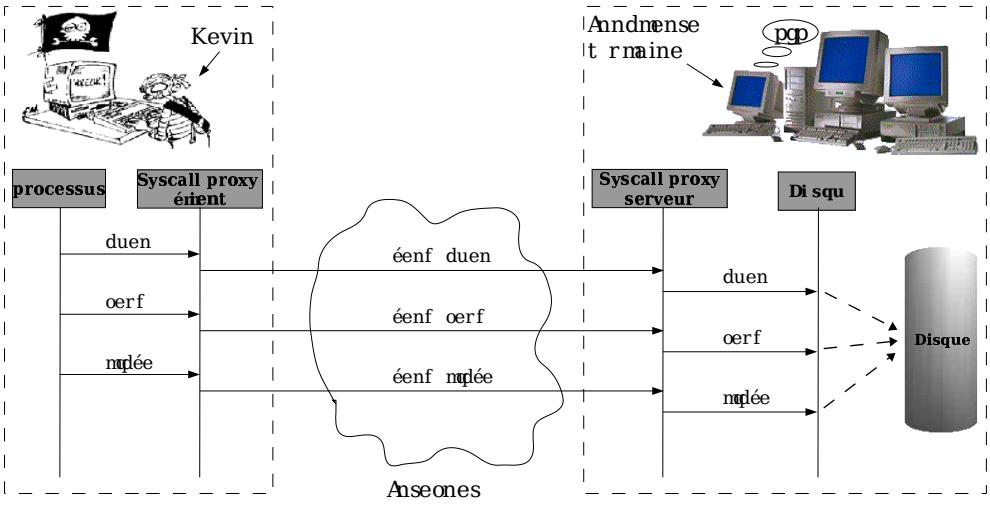

Fig. 3. Lecture d'un fichier par un processus à travers un *syscall proxy*

la commande à invoquer directement aux contraintes et conventions du serveur.

Le serveur *syscall proxy* reçoit la requête, l'interprète comme il se doit, c'est à dire qu'il copie sur la pile les arguments et le numéro de l'appel système (registre **eax**) et lève l'interruption **0x80**. Les privilèges évoqués plus tôt, avec lesquels s'exécute l'appel système sont ceux du processus ayant lancé le serveur *syscall proxy*.

La troisième et dernière étape est de renvoyer au client *syscall proxy* les résultats retournés par l'exécution de l'appel système. Cela peut être par exemple la valeur du *file descriptor* dans le cas d'un **open()**, ou bien encore le buffer contenant les données dans le cas d'un **read()**.

Le serveur *syscall proxy* est la plupart du temps un *shellcode* lancé après l'exploitation d'un processus et donc résident en mémoire. Des méthodes d'optimisation [6] sont utilisées pour le rendre le plus petit possible.

Le client *syscall proxy* peut lui être représenté sous forme de librairie qui *hook* les appels système appelés par le programme via la librairie **Libc**. Ces *hooks* construisent les requêtes à envoyer au serveur *syscall proxy* en fournissant les informations nécessaires et récupèrent les résultats renvoyés par ce serveur. Cette librairie peut être chargée via la variable d'environnement **LD_PRELOAD** de manière à ne jamais avoir à modifier le fichier source.

CORE-IMPACT CORE-IMPACT est un produit commercial. N'ayant pu le tester, les informations qui suivent sont tirées de l'article de Nicolas Fischbach

intitulé « Les tests de pénétration automatisés : CORE-IMPACT » paru dans le magazine MISC n°11.

Les serveurs *syscall proxy* sont livrés dans le produit sous forme d'agents. Ils peuvent être installés de manière éphémère et discrète sur le système distant après exploitation de la vulnérabilité. Ces agents obtiennent les droits du processus qui a été exploité et vont exécuter ce qui est appelé dans CORE-IMPACT des modules (recherche d'information, augmentation de privilèges, etc). Les agents se doivent d'être sûrs pour éviter de rendre l'infrastructure testée plus vulnérable et donc d'augmenter les risques de compromission par des tiers.

Ces agents ont plusieurs niveaux d'application, allant du niveau 0 au niveau 3. Les agents de niveau 0 s'exécutent en mémoire seulement, comprennent un mini-serveur *syscall proxy* et dès que l'on se détache de celui-ci, il se détruit. Une fois cet agent installé, il est possible d'exécuter un mini-*shell* qui fournit quelques commandes de base. L'agent de niveau 1 est plus complexe (il offre beaucoup plus de fonctionnalités) et donc plus difficile à déployer. La manière la plus automatisée consiste à déployer un agent de niveau 0, lui faire obtenir les droits maximum (administrateur ou root) pour pouvoir ensuite l'*upgrader* en agent 1. C'est en quelque sorte un *multi-stage shellcode* avec élévation de privilèges entre le premier et le second *shellcode*. Un détail important cependant à noter. A la différence de l'agent de niveau 0, celui de niveau 1 installe temporairement des fichiers sur le système. Le troisième agent, le *localagent* de niveau 3 est le point de départ (*agent source*) de tout test de pénétration. Il est donc fixe et intégré dans la console IMPACT. C'est à partir de cet agent que s'exécuteront les modules sélectionnés.

UW_sp_xxx_x86 Casek du groupe Uberwall a présenté lors d'une conférence au CCC des implémentations de *syscall proxy*. Le serveur est sous forme d'un *shellcode* (qui est supposé être exécuté après exploitation d'un processus sur une machine distante) et implémenté selon l'algorithme de Maximilio :

```
1: channel ← set_up_communication()
2: channel .send(ESP)
3: while channel.has_data() do
4:     request ← channel.read()
5:     copy request in stack
6:     pop registers
7:     int 0x80
8:     push EAX
9:     channel.send(stack)
10: end while
```

Algorithme 1 : Simple serveur pour un *syscall proxy*

Le client prépare la pile localement, *package* le tout sous forme de requête qu'il envoie au serveur distant. Celui-ci exécute la requête du client et lui renvoie les résultats. Plusieurs *shellcodes* sont implémentés, l'algorithme du *syscall proxy* est toujours la même, seule la manière dont il est exécuté diffère (en mettant un port sur écoute, en réutilisant la même socket, etc).

Côté client, ici aussi l'algorithme est toujours identique (préparation de la pile, etc). Seules les fonctionnalités changent : *scanner* de port, infection d'un processus à distance, etc. Pour développer ses clients, il a utilisé la librairie UWsplib (qu'il identifie comme étant une *ultra light libc*) dans laquelle chaque appel système est réimplémenté : **sp_open()**, **sp_read()**, etc. Il est cependant important de noter que :

1. Il est regrettable que cette librairie ne soit pas fournie en tant que ... librairie. D'après les sources disponibles sur le site d'Uberwall, toutes ces fonctions **sp_xxx()** sont implémentées dans chacun de ses clients *syscall proxy*.
2. Dans le cas de clients légers, cette démarche peut convenir. Mais dans le cas où il est nécessaire d'exécuter nmap par exemple, réécrire un tel outil avec les fonctions **sp_xxx()** peut s'avérer très lourd.

Il est évident qu'utiliser une librairie via la variable d'environnement **LD_PRELOAD** aurait été beaucoup plus judicieux. Casek parle justement dans sa présentation (slide 25) de la librairie UWskyzoexec mais c'est apparemment au stade de développement.

3.3 *Userland execve*

La deuxième grande technique pour exécuter des binaires sur un système distant est appelée *remote userland execve*. Ici l'exécution du binaire n'est pas décomposée entre la machine cliente et le serveur, le binaire est complètement envoyé dans la mémoire de la machine distante, puis executé.

Description d'un *execve* Les fonctions de la famille **exec** (**execl()**, **execlp()**, **execle()**, **execv()**, **execvp()**) sont utilisées pour exécuter un binaire. Elles utilisent toutes l'appel système **execve()**. Lors d'un appel à celui-ci dans un programme, l'image du processus est remplacé par l'image du binaire à exécuter. En clair, le programme qui appelle **execve()** est retiré de la mémoire et le binaire à exécuter est chargé. Le segment de code, le segment de données, la pile ainsi que la *heap* sont remplacés. Par conséquent, **execve()** ne retourne pas à l'ancien processus puisque toutes les structures de ce dernier ont été écrasées par le nouveau binaire. Pour pouvoir retourner à l'ancien processus, il faut utiliser un **fork()**, de la même manière que la fonction **system()** de la **Libc**. Lors d'un appel à cette fonction, le processus est dédoublé à l'aide d'un **fork()** et c'est le

fils qui invoque **execve()**. Le processus père peut donc continuer à fonctionner.

Le remplacement de l'image du processus se déroule en plusieurs étapes. Tout d'abord, **execve()** *unmap* toutes les pages qui forment l'espace d'adressage du processus courant. Le nouveau programme est ensuite chargé en mémoire. Si ce dernier est un programme dynamique, il utilise donc des libraries (contrairement aux binaires statiques), l'éditeur de liens dynamique doit alors être chargé en mémoire aussi. La pile est initialisée avec les variables d'environnement, les autres paramètres nécessaires à la fonction **main()** et les paramètres nécessaires à l'éditeur de liens dynamiques.

La figure 4 représente l'état de la pile lors d'un appel à **execve()**.

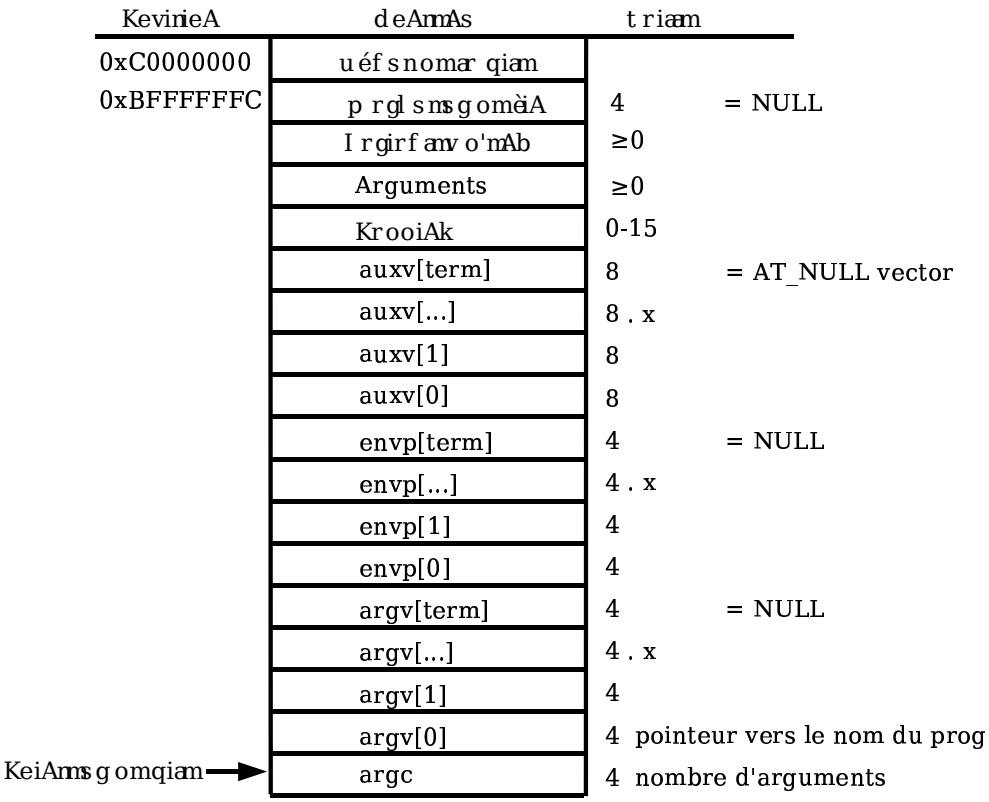

Fig. 4. Etat de la pile lors d'un appel à **execve()**

En haut de la pile, il y a les variables d'environnements et les arguments. Ensuite, si le binaire est un binaire dynamique, un vecteur **auxv** (de type

Elf_aux) est présent. Ce vecteur contient les informations nécessaires à l'éditeur de liens dynamiques. Si le binaire est un binaire statique, ce vecteur est absent. En dessous, se trouve les tableaux de pointeurs vers les variables d'environnement (**envp**) et les arguments (**argv**). Enfin, il y a **argc** tout en bas de la pile représentant le nombre d'argument.

Si l'éditeur de liens dynamiques est présent et chargé en mémoire, le point d'entrée (*entry point*) pointe vers lui. Dans le cas contraire, ce sera simplement le point d'entrée du binaire lui-même.

Description d'un execve() en *userland* Le problème principal avec l'appel système **execve()** est qu'il nécessite la présence du binaire sur le disque. L'autre souci est la possibilité qu'il soit interdit par un mécanisme de sécurité tel qu'un patch noyau, empêchant l'exécution du binaire. L'idée intéressante est donc de simuler le comportement d'un appel système **execve()** pour exécuter un binaire déjà présent en mémoire. De cette manière, le binaire n'est pas écrit sur le disque et aucune trace n'est laissée.

L'implémentation d'un **execve()** en espace utilisateur doit effectuer toutes les étapes normalement réalisées par l'appel système **execve()** :

1. *Unmapper* les pages contenant l'ancien processus
2. Si le binaire est un binaire dynamique, charger en mémoire l'éditeur de liens dynamiques
3. Charger le binaire en mémoire
4. Initialiser la pile
5. Déterminer le point d'entrée
6. Transférer l'exécution au point d'entrée

Remote userland execve L'idée d'un *remote userland execve* est simplement d'exécuter un binaire sur la machine cible sans rien écrire sur le disque. A la place d'exécuter un binaire présent sur le disque, le binaire est envoyé via une socket par le réseau, réceptionné dans l'espace d'adressage du processus, et exécuté.

Une autre utilisation intéressante est de pouvoir exécuter un exploit local. Une fois que l'attaquant a pris possession d'une machine à l'aide d'une faille dans Apache par exemple, il se retrouve à ce moment dans l'espace d'adressage du processus Apache qui tourne avec les droits *nobody*. L'attaquant souhaite alors obtenir les permissions root pour installer sa backdoor. Il dispose d'un exploit local qui lui permettra d'augmenter ses privilèges. Il l'envoie sur la machine distante et l'exécute via le *remote userland execve* sans jamais l'écrire sur le disque.

Bien sûr, la plupart des exploits locaux permettent d'obtenir un *shell* avec les droits root. Il existe des scénarios où l'attaquant ne souhaite pas de *shell* root

puisque qu'il préfère rester en mémoire pour laisser le moins de traces possibles. L'exploit local, à la place de lancer un *shell*, pourra donc par exemple donner les droits root au processus Apache exploité. De cette manière, l'attaquant disposera d'un processus Apache avec les pleins droits lui permettant d'exécuter tout ce qu'il désire (par exemple insérer un module dans le noyau à partir de ce processus sans jamais l'écrire sur le disque).

Ul_exec et rexec La première implémentation publique d'un *userland execve* est celle développée par The Grugq [18]. Sa librairie, Ul_exec, permet d'exécuter des binaires dynamiques sans l'aide de **execve**. Les 6 étapes présentées au point 3.3.2 sont respectées pour arriver à ce résultat. A ce jour, la librairie Ul_exec est la seule permettant d'exécuter des binaires dynamiques. Les autres implémentations qui seront vues par la suite permettent d'exécuter seulement des binaires statiques. Le fait de pouvoir exécuter des binaires dynamiques est intéressant mais aggrandit considérablement la taille du code. La gestion des variables d'environnement est également présente dans Ul_exec.

Concrètement, son outil est simple à utiliser. Il est fourni sous forme de librairie **libulexec.so** qu'il suffit d'ajouter à la compilation. Pour faire appel à la librairie, il faut utiliser la fonction Ul_exec() de cette manière :

```
[...]
char * buffer = NULL;
[...]
buffer=mmap( NULL,flen,(PROT_READ|PROT_WRITE|PROT_EXEC),
                       (MAP_PRIVATE|MAP_ANON),-1,0);
if ( buffer == (void *)-1 ) {
        perror("mmap()");
        return;
}
[...]
ul_exec( buffer, argc, argv );
[...]
```

Sur des noyaux Linux ¿ 2.6.11, il est nécessaire de patcher la librairie car elle fonctionnait seulement avec une pile pré-définie et les dernières versions du noyau *randomizent* l'adresse du début de la pile. Le patch suivant permet de connaître le sommet de la pile en *runtime* :

```
/*get STACK top on boxes with random stack adresses (PaX, kernel > 2.6.11)*/
fd = fopen( "/proc/self/maps", "r" );
if ( !fd ) {
        return;
}

memset( stack_top, '\0', sizeof(stack_top) );
```

```
while( (ptr = fgets(buffer, sizeof(buffer)-1, fd)) != NULL ) {
        if ( strstr(ptr, "[stack]") ) {
                char * tmp;

                tmp = strstr( ptr, "-" );
                if ( tmp != NULL ) {
                        snprintf(stack_top,sizeof(stack_top),"0x%s",&tmp[1]);
                        stack_top[sizeof(stack_top)-1] = '\0';
                }
                break;
        }
}
fclose( fd );
```

A noter encore que pour compiler la librairie Ul_exec, il est nécessaire d'utiliser Diet Libc[9], une librairie C optimisée pour la création d'objet ELF[10] de petite taille.

Peu après Ul_exec, The Grugq a développé une autre librarie, appelé **libgdbrpc** permettant d'exécuter des binaires à distance. Cette librairie fonctionne en concert avec la librairie Ul_exec, le tout étant appelé rexec. Pour pouvoir interagir avec la machine visée, il est nécessaire qu'un serveur soit installé sur celle-ci. Lors d'une exploitation, le serveur peut être le processus exploité. Mais il est également possible que le serveur soit un programme spécialement conçu à cette occasion. The Grugq appelle ces serveurs des IUD :

- *Inter Userland Device* (IUD), qui permet d'accéder à son propre espace d'adressage.
- *Intra Userland Device* (IUD), qui permet d'accéder à n'importe quel espace d'adressage, donc aussi ceux d'autres processus.

Un bon IUD pour The Grugq, est IUD qui permet à l'attaquant de manipuler des registres et des régions mémoires, qui est un outil standard sous Linux et qui accepte des commandes en format texte. La solution la plus évidente est donc bien évidemment GDB. L'avantage de GDB est qu'il permet de débugger des programmes à distance (peu de personnes le savent). C'est cette fonctionnalité que The Grugq utilise. Pour pouvoir débugger un programme à distance, un *stub* doit être lancé sur la machine cible pour interagir entre GDB (qui est sur la machine de l'attaquant) et les processus visés (figure 5).

Dans le cas de rexec, l'utilisation d'un *stub* lui permet d'exécuter des binaires à distance sans rien écrire sur le disque. Les différentes étapes pour y arriver sont les suivantes :

[9] `http://www.fefe.de/dietlibc/`
[10] Executable and Linkable Format

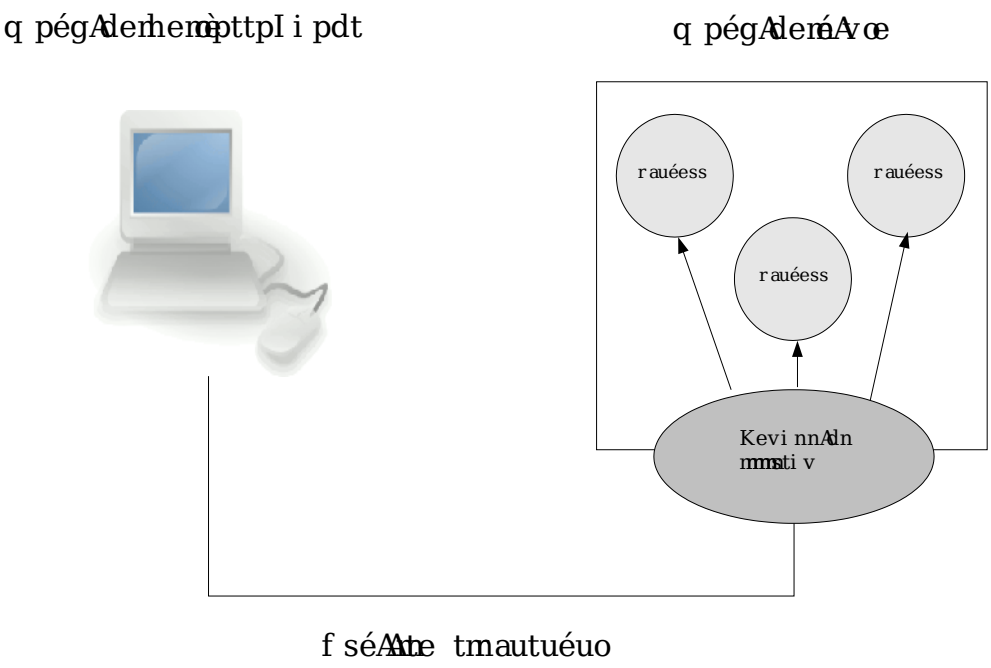

Fig. 5. Utilisation de gdb pour débugger à distance

1. Utiliser **libgdbrpc** comme IUD
2. *Uploader* le binaire à exécuter dans la mémoire de la machine cible
3. Charger le binaire dans un espace d'adressage
4. Transférer l'exécution au binaire chargé à l'aide de Ul_exec

L'idée d'utiliser les fonctionnalités de GDB pour s'en servir comme serveur est une bonne idée. Malheureusement, il n'est pas possible d'utiliser rexec à partir d'un exploit. Il faut installer la *libgdbrpc* à partir d'un *shell*, celui-ci devant être obtenu lors de l'exploitation, par exemple avec un *connect back shellcode* ou un *bind shellcode*. Il y a donc quelque chose d'écrit sur le disque à un moment précis. Par contre, utiliser un IUD séparé du processus exploité permet à l'attaquant de revenir sur le système à n'importe quel moment et de toujours rester en mémoire. Il sera cependant vu, avec l'implémentation de Plato, qu'il est possible à partir de l'exploit, d'installer un IUD permettant de revenir par la suite (une *backdoor*) sans jamais rien écrire sur le disque.

SELF SELF (pour *Shellcode ELF loader*) est une implémentation d'un *userland execve* développée par Pluf et Ripe [18]. Cette implémentation, contrairement à Ul_exec, a été developpée dans un but d'attaque, c'est à dire pour fonctionner dans un exploit. La principale différence entre Ul_exec et SELF est que ce

dernier ne peut exécuter des binaires dynamiques mais seulement des binaires statiques. Il est évident qu'implémenter un *userland execve* pour des binaires statiques est plus facile que pour des binaires dynamiques, seulement la raison de l'utilisation de binaires statiques n'est pas seulement là. Lors d'une intrusion sur un système, il n'est pas souvent pratique d'avoir des binaires dynamiques puisque des librairies peuvent être nécessaires. C'est la raison pour laquelle les programmes (souvent les exploits) sont le plus souvent compilés en statique. Ils sont également strippés (c'est à dire que les symboles sont enlevés) pour prendre moins de place (inutile d'avoir un binaire statique de plusieurs mégas bytes.) Pour finir, ils sont de plus en plus protégés par chiffrement avec des programmes comme Burneye[11] ou Shiva[12] pour compliquer le *reverse engineering* du binaire.

Le fonctionnement de SELF est peu plus subtile que Ul_exec, il se compose de trois parties le *lxobject*, le *builder* et le *jumper*, indépendantes les unes des autres mais fonctionnant toutes en harmonie.

Le *lxobject* est un type spécial d'objet qui contient tous les élements nécessaires pour remplacer l'image d'un processus par un autre. Le *lxobjet* est construit sur la machine de l'attaquant par le *builder* et se compose également de trois parties :

- Le binaire ELF compilé en statique à exécuter sur la machine cible.
- Une pile pré-construite contenant le contexte nécessaire lors du lancement du binaire. Comme le binaire est un binaire statique, le vecteur **auxv** nécessaire à l'éditeur de liens dynamiques est absent.
- Le *shellcode loader*, celui-ci est simplement une implémentation d'un *userland execve*. Sa principale utilité est de remplacer l'image du processus cible par celui du binaire statique. Il *unmappe* donc les pages de l'ancien processus pour charger les segments appartenant au binaire statique.

Le *jumper* est simplement le *shellcode* envoyé lors de l'exploitation. Il attend un *lxobject* et donne le contrôle de l'exécution au *shellcode loader*. Ce dernier se charge de remplacer l'image du processus exploité par l'image du binaire statique à exécuter.

Il y a cependant un problème majeur dans cette implémentation : le binaire exécuté sur la machine cible est exécuté à la place du processus exploité. Il est donc impossible d'exécuter plusieurs fois le même binaire ou d'exécuter des binaires différents ! C'est totalement contraignant, l'idéal étant d'avoir un système similaire, mais permettant des exécutions multiples.

Pitbull Peu après la sortie de SELF, un de ses auteurs, Pluf, a sorti une technique dérivée de SELF qui permet d'exécuter un binaire plusieurs fois sur la machine cible [15]. La principale différence par rapport à SELF est que le *lxobject* est construit sur la machine cible et plus sur la machine de l'attaquant.

[11] `http://packetstormsecurity.org/groups/teso/burneye-1.0.1-src.tar.bz2`
[12] `http://www.securereality.com.au/archives/shiva-0.95.tar.gz`

Pitbull est le nom donné à ce logiciel.

Pas besoin d'aller bien loin pour comprendre comment Pitbull fonctionne : il utilise tout simplement un *fork()*. Un processus fils est créé et c'est l'image du processus fils qui est remplacée par le binaire à exécuter. Une amélioration intéressante cependant par rapport à SELF est le fait que Pitbull permet de stocker les binaires envoyés dans l'espace d'adressage du processus père (le processus exploité). De cette manière, s'il est nécessaire d'exécuter plusieurs fois le même binaire, il n'y a aucun besoin de le renvoyer au *jumper*. Pitbull utilise toujours des binaires statiques pour fonctionner.

Malgré que cette technique soit plus intéressante que SELF, elle possède néanmoins un inconvénient majeur. Si un processus Apache par exemple est exploité et le *shellcode* (par exemple le *jumper* de Pitbull qui attend les binaires à exécuter) est lancé, le démon continue de tourner malgré tout puisqu'il y a de fortes chances que ce soit un de ses fils qui soit exploité. Maintenant dans le cas où seulement un seul processus tourne, il est impossible que le processus revienne à son état normal. Après avoir envoyé le *shellcode*, la mémoire de ce processus est en effet corrompue, ses registres, ses régions mémoire, les données dans les buffers, sa pile ... sont modifiés. Et il est impossible de sauter au point d'entrée du programme (il faudrait reconstruire tout ce qui a été écrasé). Le processus est donc condamné à mourir et comme il est unique, le serveur ne sera pas relancé. Il pourrait être envisageable de relancer le serveur avant que le processus ne se termine (il y a souvent un *exit()* à la fin des *shellcodes*). Mais encore une fois, c'est impossible puisque si le démon est relancé, il devra mettre un port défini sur écoute (*binder* un port). Or ce port y est déjà puisque le processus condamné est toujours actif. Le chat se mord la queue. Une solution est étudiée lorsqu'il sera question de Plato.

Impurity La dernière implémentation d'un *userland execve* présentée ici s'appelle Impurity. Il s'agit d'une suite de scripts créée par Alexander E. Cuttergo [7] permettant d'exécuter un binaire sur une machine distante sans rien écrire sur le disque. Il est question également d'un *shellcode* en deux parties, un *multi-stage shellcode*. Comme SELF et Pitbull, il est orienté pour l'exploitation. La différence cependant est qu'il ne remplace pas l'ancienne image du processus. Pour ce faire, le binaire doit être compilé d'une certaine manière :

1. Le binaire est compilé en statique comme dans SELF (aucun problème de librairies à charger).
2. D'habitude, le segment **PT_LOAD** qui contient la section **.text** (le segment exécutable) est mappé à l'adresse **0x08048000**. Or à cette adresse, il y a déjà l'autre binaire qui est *mappé*. Comme Impurity ne veut pas *unmapper* les pages de l'ancien binaire, il faut placer le binaire à exécuter autre part. Ce sera dans la pile. Le segment contenant la section **.text** est ainsi *linké* pour démarrer à une autre adresse, ici **0xbfff1000** qui est une adresse dans

la pile. Si maintenant la pile est non exécutable, il est toujours possible de placer le code dans une région mémoire allouée avec *mmap* et avec les protections **PROT_EXEC**. Les arguments et les variables d'environnement sont également enlevés pour plus de facilité, il n'y a donc pas ici de pile à préconstruire.

3. Le binaire doit également être *linké* pour que le segment de code et le segment de données ne soient pas disjoints (contrairement à un binaire compilé normalement). De cette manière, le chargement du binaire en mémoire sera beaucoup plus simple (plus besoin de charger chaque segment indépendamment).
4. Les binaires compilés en statique avec la **Libc** sont souvent de tailles importantes, même s'ils sont *strippés*. Le binaire sera donc compilé avec la Diet Libc.

Le fonctionnement d'Impurity se base aussi sur un *multi-stage shellcode*. Le premier *shellcode* reçoit le binaire à exécuter, le place dans la pile et saute à son point d'entrée.

Pour chaque binaire à exécuter sur la machine distante, il faudra passer par ces 4 étapes. Si le fonctionnement d'Impurity est intéressant puisque le processus exploité n'est pas remplacé, il est toute fois relativement contraignant de devoir recompiler chaque binaire de cette manière. Néanmoins, l'approche choisie par Impurity pour exécuter un binaire en *userland* est beaucoup plus facile à implémenter.

4 *Backdooring* de processus et injection de librairies

Différentes techniques permettant de ne rien écrire sur le disque lors d'une intrusion viennent d'être abordées. Ces techniques peuvent être utilisées lors de l'étape 2 « Attaque et penétration » d'une intrusion. La troisième étape concernant l'élévation de privilèges ne vaut pas la peine de s'y attarder. Pour rappel, l'objectif est d'obtenir des privilèges plus importants pour avoir un contrôle total de la machine et notamment pouvoir relancer le serveur exploité. Pour ce faire, les techniques précédentes (telles que le *remote userland execve* ou le *syscall proxy*) sont aussi utilisées à ce stade de l'intrusion pour envoyer sur la machine exploitée un exploit local et l'exécuter.

Reste alors l'étape 4 « installation d'une *backdoor* ». Cette étape est nécessaire à l'attaquant s'il souhaite revenir facilement sur la machine cible sans passer par toute la phase d'exploitation. L'objectif principal reste toujours le même : ne rien écrire sur le disque. La *backdoor* doit donc elle aussi être installée en mémoire vive. Inutile de corrompre un binaire et de relancer un serveur, il y aura des informations inscrites sur le disque. La solution la plus évidente reste donc de placer la *backdoor* dans la mémoire d'un serveur déjà lancé sur la machine.

4.1 L'ancienne méthode : l'injection de code assembleur

L'ancienne méthode pour infecter un processus était d'injecter directement du code assembleur dans son espace d'adressage à l'aide de l'appel système **ptrace()**. Les rappels d'usage concernant **ptrace()** ne seront pas expliqués ici, un article sur cet appel système dans ces actes lui étant entièrement consacré. Le lecteur peut également consulter l'article sur l'utilisation de la mémoire vive dans MISC 25 [11].

Pour pouvoir injecter du code, il est nécessaire d'avoir un minimum d'espace. Deux solutions donc : soit l'injection se fait dans la mémoire déjà allouée qui contient des instructions ou des données qui ne sont plus nécessaires (voir à ce propos l'article dans MISC 25 [11]), soit un nouvel espace dans la mémoire du processus est alloué, à l'aide par exemple de l'appel système **mmap()**. Mais quelle que soit la solution, il est impossible de réaliser des *backdoors* évoluées avec ce type d'injection. Son seul avantage est qu'elle peut fonctionner autant avec des binaires statiques que dynamiques.

4.2 Injection de librairies en local

Une méthode plus pratique pour corrompre la mémoire d'un processus est d'injecter directement une librairie. La librairie peut ainsi contenir toutes les opérations à faire réaliser par le serveur à corrompre. Pour injecter une librairie, il est nécessaire de passer par la fonction **dlopen()**, dépendante de la librairie **libdl.so** :

```
void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
```

dlopen() charge dans la mémoire du processus une librairie présente sur le disque, l'emplacement de la librairie sur le disque étant spécifié par **filename**. Le champs **flag** permet de contrôler la résolution des symboles importés. La fonction **dlopen()** marche de concert avec les fonctions **dlsym()** et **dlclose()**. **dlsym()** permet d'obtenir l'adresse de l'emplacement où est chargé le symbole en mémoire (typiquement une fonction). Une fois cette adresse obtenue, la fonction peut être appelée. **dlclose()** permet simplement de décharger une librairie précédemment chargée à l'aide de **dlopen()**. Ces trois fonctions appartiennent à la **libdl**, il est donc nécessaire que cette librairie soit déjà chargée. Malheureusement, ce n'est pas souvent le cas. Il faut alors passer par les fonctions **_dl_open()**, **_dl_sym()** et **_dl_close()**.

```
void *_dl_open(const char *file, int mode, const void *caller);
void *_dl_sym(void *handle, const char *name, void *who);
void *_dl_close(void *_map);
```

Ces trois fonctions sont similaires à celles de la **libdl** sauf qu'elles sont contenues dans la **Libc** toujours présente dans les binaires compilés en dynamique. Il est donc préférable d'utiliser les fonctions de la **Libc** que celle de la **libdl**, les fonctions de la **libdl** n'étant de toute façon que des fonctions de passage vers les fonctions de la **Libc**. A noter le paramètre **caller** dans **_dl_open()** qui n'est pas nécessaire ici et qui peut être mis à **NULL**.

Pour que le processus charge la librairie, un petit code assembleur qui effectue le **_dl_open()** est injecté dans la mémoire du processus à l'aide de **ptrace()** :

```
_start:
          jmp string
begin:
          pop eax                                   ; char *file
          xor ecx, ecx                              ; *caller
          mov edx,0x1                               ; int mode

          mov ebx, 0x12345678                       ; addr of _dl_open()
          call ebx                                  ; call _dl_open!
          add esp, 0x4
          int3                                      ; breakpoint
string:
          call begin
          db "/tmp/evillibrary.so", 0x00
```

L'adresse **0x12345678** doit être changée par la véritable adresse de **_dl_open()**. Une fois la librairie chargée, il faut maintenant *hijacker* (ou détourner) une fonction pour placer la *backdoor*. Pour cela, il est nécessaire tout d'abord de connaître son emplacement en mémoire, en utilisant soit la *link map* si le binaire a été *linké* avec un ancien éditeur de liens [1], soit la PLT[13] pour les nouveaux éditeurs de liens [11]. C'est cette dernière technique qui est retenue pour la simple et bonne raison qu'elle fonctionne dans tous les cas. Elle permet notamment de connaître l'adresse de **_dl_open()**.

Une fois l'adresse de l'ancienne fonction obtenue, il faut la remplacer par la nouvelle fonction qui est définie dans la librairie injectée (ici **/tmp/evillibrary.so**). Pour ce faire, la méthode la plus simple est soit d'utiliser la **GOT** (**GOT redirection**), soit la **PLT** (**PLT redirection**), sachant qu'avec la **GOT** il n'y aura aucun problème même si le noyau a été compilé avec PaX[14]. Dans les deux cas, l'entrée de la fonction à détourner est remplacée par l'adresse de la nouvelle fonction se trouvant dans la librairie qui est chargée. Dans cette nouvelle fonction, il sera souvent nécessaire d'appeller la fonction d'origine, il faudra alors utiliser **dlsym()**. Des techniques plus avancées comme **ALTPLT** [14] et **CFLOW** [10] sont également possibles (et même mieux) mais elles ne seront pas abordées ici

[13] Procedure Linkage Table
[14] `http://pax.grsecurity.net/`

pour ne pas égarer le lecteur[15].

Pour citer un exemple, une librairie peut être chargée dans un processus Apache à l'aide de **dlopen()** et la fonction **read()** peut être *hijacké*. La nouvelle fonction **read()**, dès qu'elle reçoit une chaîne de caractères spécifique, lance un *bind shell* :

```
new_read (fd, buf, count)
1: o_read ← dlsym("libc.so","read")
2: size ← o_read(fd, buf, count)
3: if buf = MAGIC_WORD then
4:    launch bind shell
5: end if
6: return size
```

Algorithme 2 : Redirection de la fonction read

Une implémentation presque similaire a été développée par Ares [3] dans l'article faisant suite à celui de phrack [1].

Le grand problème de cette technique, malgré le fait qu'elle soit très efficace, est que la librairie est écrite sur le disque avant d'être chargée (ici dans le répertoire **/tmp**), **dlopen()** et **_dl_open()** nécessitant tous les deux comme premier argument l'emplacement de la librairie sur le disque. Une solution pour contrer ce problème est de placer la librairie dans une partition **tmpfs** ou **ramdisk**, qui sont des partitions créées en mémoire vive [11]. La librairie n'est alors jamais écrite sur le disque.

4.3 Injection de librairies à distance

Comme dans le cas de l'exécution d'un binaire à l'aide d'un *remote userland execve* ou d'un *syscall proxy*, il est intéressant d'installer la librairie dans la mémoire du processus à distance et sans passer par le disque (pas de **tmpfs** ou de **ramdisk**). L'idée, comme dans le cas d'un *userland execve*, est de recevoir la librairie par le réseau en écoutant sur une socket. Une fois réceptionnée, elle est placée dans la mémoire du processus et est ensuite chargée à l'aide de **_dl_open()**. Malheureusement, **_dl_open()** a besoin d'une librairie présente sur le disque et non déjà en mémoire. Il est alors nécessaire de procéder autrement.

Dans l'article *remote library injection* de Jarko Turkulainen [10], deux solutions sont proposées. L'idée de la première (*On-Disk Remote Library Injection*) est simplement, une fois la librairie réceptionnée dans la mémoire du processus distant, d'écrire cette librairie sur le disque. Elle peut ensuite facilement être chargée en mémoire. Comme ça a été dit, si cette technique est utilisée, il est

[15] Est-il toujours là ? :-)

nécessaire de passer par un **tmpfs** ou un **ramdisk** pour que rien ne soit écrit sur le disque. Les différentes étapes du coté serveur permettant d'y arriver sont les suivantes :

1. Ouvrir un fichier sur le disque (**tmpfs** ou **ramdisk**)
2. Réceptionner sur la socket la librairie envoyée et la sauvegarder dans la pile, dans le tas ou dans un plage mémoire *mappée*
3. Ecrire dans le fichier la librairie stockée dans la pile
4. Fermer le fichier créé
5. Résoudre l'adresse de **_dl_open()**
6. Appeler **_dl_open()** avec la librairie écrite sur le disque

La deuxième technique proposée par Jarko Turkulainen (*In-Memory Remote Library Injection*) est légèrement différente. Dans celle-ci, rien n'est écrit sur le disque, pas même sur une partition de type **tmpfs** ou **ramdisk**. Son idée est de *hooker* les fonctions d'opérations sur fichier utilisées par **dlopen()**, c'est à dire **open()**, **read()**, **lseek()**, **mmap()** et **fxstat64()**.

Soit l'exemple de **open()**. Cette fonction qui est une fonction de la **Libc** et qui agit comme un *wrapper* vers l'appel système **open()**, est utilisée pour ouvrir la librairie présente sur le disque. Comme elle est présente seulement en mémoire, il faut procéder autrement. Si le nom de fichier passé en paramètre à la fonction **open()** est la librairie à charger, alors un descripteur de fichier spécial est retourné. Il est en fait utilisé tout au long du processus de chargement de la librairie lors des appels subséquents aux autres opérations sur fichier et contiendra aussi bien l'adresse de base où est chargée la librairie, la taille de la librairie ou encore la position courante dans le fichier chargé en mémoire. Ces informations sont utilisées ensuite par les autres opérations sur fichier.

L'idée est intéressante, cependant il semble que ce soit assez compliqué à mettre en oeuvre. Le concept n'est que théorique et aucune implémentation publique n'existe à ce jour. Une autre idée plus intéressante et plus simple pour charger une librairie déjà présente en mémoire est de recoder complètement la fonction **dlopen()**. La nouvelle fonction ainsi codée permettra de charger une librairie déjà présente en mémoire.

5 Constatations

5.1 Constatations générales

Le serveur exploité sur la machine cible est condamné à mourir

Une fois un serveur exploité sur une machine cible, Openssh, Apache, ProFTP ou un autre, il n'est plus possible de relancer le processus. Sa mémoire a été corrompue, les registres ont maintenant des valeurs différentes, des pointeurs ont probablement été écrasés, ... et il est impossible de deviner les valeurs initiales. Envisager de relancer le binaire en sautant simplement à son point d'entrée est quasiment impossible puisqu'il faut pour cela remettre la pile dans son état initial. Le processus est donc condamné à mourir. Ce problème est néanmoins très intéressant et ouvert. Il n'y a pas encore de solution générique à l'heure actuelle.

Comme il a été stipulé précédemment, dans le cas d'un processus Apache, le problème ne se pose pas puisque c'est souvent le processus fils qui est exploité et contrôlé, laissant le processus père fonctionner et lancer des fils normalement. Dans le cas des services gérés par inetd, ce dernier peut relancer les processus se terminant anormalement (ce qui peut permettre par exemple de *bruteforcer* les adresses lors d'une exploitation ...). Il peut y avoir cependant des problèmes avec des démons qui ne *fork* pas et qui ne se relancent pas. Dans ce cas, une fois l'exploitation terminée, le processus meurt et le service n'existe plus, empêchant de se reconnecter au port. Il est donc impératif dans ce cas de le relancer.

Le serveur ne peut être relancé juste avant de mourir

La solution de relancer le serveur juste avant que le processus exploité ne meurt ne peut fonctionner. Simplement parce que le processus a déjà *bindé* le port du serveur et qu'il est impossible de relancer un serveur qui va écouter lui aussi sur le même port. Il faut donc relancer le serveur après que le processus se termine.

Le serveur ne peut être relancé par manque de privilèges

Par souci de sécurité, très peu de démons aujourd'hui sont lancés avec les permissions root sur les serveurs. Donc, lors de l'exploitation d'un de ces démons, l'attaquant obtiendra les privilèges du processus exploité. Même s'il a une solution pour relancer le service, il n'aura pas les permissions suffisantes pour le faire. Une élévation de privilèges est donc souvent nécessaire.

5.2 Constatations à propos du *userland execve*

Le binaire ne doit pas être recompilé d'une manière (trop) spécifique

Une implémentation comme celle d'Impurity est intéressante car plus facile à implémenter, mais beaucoup trop contraignante dans le sens où le binaire doit être recompilé (segments non disjoints, *entry point* différent, ...). S'il est nécessaire de le faire de cette manière pour tous les exécutables afin de les lancer

sur la machine distance, cela peut vite devenir contraignant. Il est donc préférable de simuler le fonctionnement d'un **execve()** le plus juste possible, c'est à dire en remplacant l'image d'un processus par l'image du binaire à lancer, de la même manière que Ul_exec, de SELF et de Pitbull.

Utiliser des exécutables compilés en dynamique n'est pas une bonne idée

Ul_exec permet d'exécuter des binaires dynamiques par rapport à SELF et à Pitbull. Il est évident qu'implémenter un *userland execve* pour des binaires dynamiques est un peu plus compliqué, mais ce n'est pas la raison pour laquelle il ne faut pas utiliser une implémentation pour des binaires dynamiques.

Si l'exécutable est dynamique, il est nécessaire que toutes les librairies dont il a besoin soient présentes sur le système cible. Ce n'est pas forcément toujours le cas. Maintenant, si les librairies sont présentes sur le système cible, elles ne portent pas forcément le même nom ou n'ont pas la même version (les noms dépendent des versions des librairies). Et comme le nom des libraires nécessaires à l'exécution d'un binaire dynamique sont hardcodées dans ce dernier (la commande ldd permet de connaitre les noms des librairies utilisées par le binaire), il peut vite devenir difficile d'implémenter un *userland execve* pour des binaires dynamiques.

Un *userland execve* doit *forker* pour ne pas tuer le processus utilisé

Dans la solution proposée par SELF, le processus utilisé sur la machine cible est remplacé en mémoire par l'image du binaire à exécuter. C'est une très mauvaise idée car dans ce cas le processus initial ne peut plus être réutilisé. Une implémentation d'un *userland execve* doit donc *forker* avant d'exécuter réellement le binaire, exactement comme la fonction **system()** de la Libc. Après le **fork()**, c'est au fils d'exécuter le binaire. De cette manière, il est possible de réutiliser le processus père pour lancer un autre binaire.

Les pages ne doivent pas être swappées sur le disque

Dans le cas d'un *userland execve*, tout le binaire va se retrouver dans la mémoire de la machine cible. Il se peut donc qu'à un moment donné, le noyau manque de place et *swap* sur le disque une partie des pages contenant l'exécutable. Une partie du binaire est alors écrite sur le disque. Pour l'éviter, l'utilisation de l'appel système **mlock()** peut empêcher certaines pages d'être *swappées* sur le disque.

Le binaire à exécuter sur la machine cible doit être le plus petit possible

Dans le cas d'un *userland execve*, il est avantageux d'avoir un binaire de petite taille, de manière à écraser le moins possible la mémoire de la machine cible. Et surtout, les pages risquent d'êtres moins *swappées* sur le disque. Il est dès lors intéressant de compiler le binaire avec la *Diet Libc* pour le rendre le plus petit possible.

5.3 Constatations à propos du *syscall proxy*

Beaucoup trop de communications réseaux

L'algorithme d'un *syscall proxy* nécessite beaucoup d'échanges de données entre la machine locale et la machine exploitée. Il y a toujours minimum 2 échanges par appel système : l'envoi de la requête par le client au serveur et l'envoi du résultat de l'exécution de l'appel système par le serveur au client. Ce résultat peut être simplement une valeur de retour ou un entier (dans le cas de l'exécution de l'appel système **open()**) mais aussi un *buffer* (dans le cas de l'exécution de l'appel système **read()** où le client souhaite récupérer le contenu de ce qui a été lu). C'est donc évident que le *syscall proxy* peut vite souffrir de lenteur si plusieurs appels système sont exécutés.

Il est nécessaire de réimplémenter tous les appels système

En prenant la solution d'une librairie qui sera chargée avec la variable d'environnement **LD_PRELOAD** du côté du client, il est nécessaire de réimplémenter la totalité des appels système. Ça concerne donc environ 200 appels système si l'on veut avoir la chance de pouvoir envoyer tous les outils souhaités (nmap par exemple). Un travail colossal.

L'appel système fork() ne peut pas être utilisé

La fonction **fork()** fait partie des appels système standards d'UNIX. Cette fonction permet à un processus de se dupliquer, par exemple en vue de réaliser un second traitement, parallèlement au premier. Il existe une filiation dans les processus : le créateur d'un nouveau processus est appelé le père et le nouveau processus, le fils. Le fait qu'il y ait une duplication de processus rend quasi impossible la reproduction de l'appel système **fork()** dans un *syscall proxy*. Le client ne saura communiquer qu'avec le processus père. Aucun moyen n'existe pour rattacher le processus fils à cette communication déjà existante.

5.4 *Remote Userland Execve* ou *Syscall Proxy* ?

Pour pouvoir trancher entre les deux solutions, nous avons effectué quelques *benchmarks* dans le but de mieux se rendre compte de l'efficacité de chaque technique (même si nous avions déjà notre avis sur la méthode qui s'avérerait la plus rapide). Ces *benchmarks* ne sont cependant pas optimaux dans le sens où les essais réalisés n'ont pas été assez nombreux et les architectures pas assez diversifiées. Néanmoins, les résultats obtenus sont déjà assez significatifs et permettent déjà de tirer quelques conclusions.

D'un point de vue architecture, les *benchmarks* ont été effectués entre deux machines jouant le rôle de client et serveur. Sur chacune d'entre elles a été installé un serveur ntpd. Le premier s'est synchronisé avec un serveur ntpd en ligne sur Internet. Le second s'est synchronisé avec le premier. Il a fallu attendre plusieurs jours pour une totale synchronisation des deux machines de manière à réduire au maximum la dérive. Les noyaux installés sur chaque machine sont un 2.4.13-pre9 pour le client et un 2.4.31 pour le serveur.

Au niveau fonctionnel, l'objectif a été de reproduire l'exécution du programme suivant en utilisant les techniques de *syscall proxy* et de *remote userland execve* :

```
int
main( void )
{
    int fd;
    char buffer[ 4192 ];

    fd = open( "/tmp/same_file", O_RDONLY );
    read( fd, buffer, sizeof(buffer) );
    printf( "%s", buffer );
    close( fd );
}
```

Le fichier **/tmp/same_file** a été créé sur chaque machine avec le même contenu, permettant ainsi de ne pas fausser les résultats.

Côté *remote userland execve*, nous avons repris l'outil SELF qui s'avérait être adapté à ce que nous voulions faire. Pour le *syscall proxy*, nous avons du implémenter notre propre solution afin de pouvoir fournir en entrée le programme source précédent. Nous utilisons côté client une librairie que nous chargeons avec la variable d'environnement **LD_PRELOAD** et qui *hook* les appels système (en l'occurence ceux utilisés par le programme de test).

Les résultats obtenus correspondent quelle que soit la technique, à la différence entre l'heure à laquelle le client s'est connecté au serveur et l'heure à laquelle le

serveur a coupé la connexion. Nous avons effectué 3 tests pour chaque technique. Les temps sont calculés en seconde à partir du 1er janvier 1970.

Tab. 1. Résultats pour le *syscall proxy*

Essai	**Temps de départ**	**Temps d'arrivé**	**Différence (seconde)**
1	1144575001.544476	1144575001,844928	0.300452
2	1144575222.945634	1144575223.264958	0.319324
3	1144575389.394808	1144575389.702290	0.307482
Moyenne			**0.309086**

Tab. 2. Résultats pour le *remote userland execve*

Essai	**Temps de départ**	**Temps d'arrivé**	**Différence (seconde)**
1	1144576301.802521	1144576301.921963	0.119442
2	1144576566.364178	1144576566.489811	0.125633
3	1144576756.478594	1144576756.639001	0.160407
Moyenne			**0.135160**

Sans équivoque, on peut voir que le *remote userland execve* est au moins deux fois plus rapide que le *syscall proxy*. Ceci n'est pas une surprise et confirme bien notre impression.

5.5 Constatations à propos de l'injection de librairies à distance

La librairie ne pourra pas être injectée dans le processus exploité

Comme on l'a dit, le processus exploité est condamné à mourir. Il n'est dès lors d'aucune utilité de l'infecter en injectant une librairie dans sa mémoire avant même que le processus ne soit relancé.

L'injection de librairies pour des binaires compilés en statique ne fonctionnera pas

Les binaires statiques n'étant liés à aucune librairie, il est évident qu'il n'est pas possible de charger une librairie dans le processus. Une solution est d'injecter directement du code assembleur, mais cette technique n'est pas très adéquate pour des *backdoors* évoluées. Une autre solution beaucoup mieux adaptée mais

aussi beaucoup plus compliquée à mettre en oeuvre est d'insérer du code dans le processus à l'aide de la méthode **ET_REL** [14] et de détourner les fonctions à l'aide des méthodes **CFLOW** ou **EXTSTATIC** [10]. Ces méthodes sont adaptées en effet même pour des binaires statiques.

6 Plato

Il est maintenant temps de proposer une implémentation reprennant toutes les idées abordées jusqu'à maintenant. Nous avons développé dans ce but un logiciel permettant de ne toucher qu'à la mémoire vive d'un système lors d'une intrusion informatique. Ce logiciel pourra entre autre être utilisé lors d'un test de pénétration où il est nécessaire de ne jamais touché au disque. Il pourra également fonctionner avec n'importe quel exploit.

Plato part d'une hypothèse : le processus exploité ne *fork* pas et ne se relance pas automatiquement, la contrainte la plus forte est ainsi prise en compte. Sur le schéma (figure 6), le serveur exploité est Apache configuré pour fonctionner avec un seul processus. Si le serveur vient à *forker* (comme Apache en temps normal), le schéma d'attaque sera alors beaucoup plus simple.

Nous commençons l'explication du logiciel à partir du premier *payload* envoyé vers le serveur exploité. Le but final avoué est d'installer une librairie dans la mémoire du serveur exploité permettant ainsi d'exécuter à distance autant de binaires que souhaités et surtout à n'importe quel moment. Pour arriver à ce résultat, 5 étapes sont nécessaires (figure 6).

6.1 Attaque et pénétration

C'est lors de cette étape (attaque et pénétration) que le premier *shellcode* est envoyé au système cible. Comme le processus est condamné à mourir, il est nécessaire de prendre le contrôle d'un autre processus sur le système, mais cette fois-ci de manière saine (étape 2 et étape 3), c'est à dire sans corrompre sa mémoire, ses registres, ses données, etc. Les étapes 1, 2 et 3 sur la figure 6, utilisent un *shellcode* et sont donc intégrées dans l'exploit de l'attaquant. Et c'est seulement après l'étape 3 que Plato prend le contrôle.

6.2 Augmentation de privilèges si nécessaire

Comme le processus est condamné à mourir, il faut le relancer (étape 4). Dans ce cas, des droits supérieurs à ceux obtenus sont peut-être nécessaires. Plato doit donc être capable de lancer un exploit local pour augmenter les droits du processus dont il a le contrôle. Une fois les droits adéquats obtenus, il peut relancer sans problème le serveur.

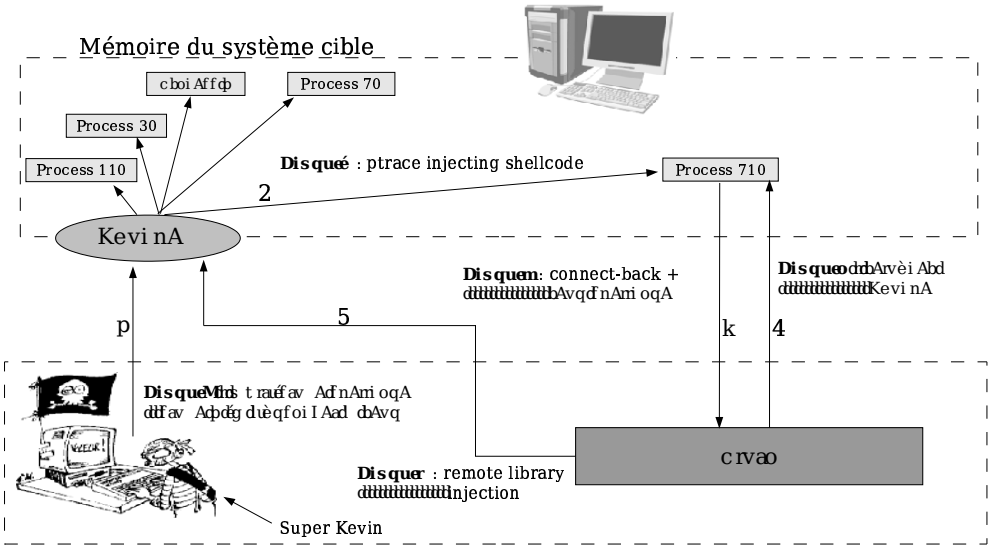

Fig. 6. Etapes nécessaires pour la mise en marche de Plato

6.3 Installation d'une *backdoor*

L'étape concernant l'installation d'une *backdoor* est la moins évidente. L'idée ici est de corrompre le serveur en lui injectant une librairie, librairie qui contiendra un *userland execve*, un peu à la manière de Ul_exec. Cependant, l'implémentation d'un *userland execve* dans Plato concerne uniquement pour des binaires statiques, de manière à ne pas avoir à se soucier des éventuelles librairies nécessaires au fonctionnement d'un binaire.

Comme la solution pour corrompre le serveur est d'injecter une librairie, il est évident que le serveur doit être compilé en dynamique (c'est à 99% une certitude).

La partie injection de librairies dans le serveur est la plus intéressante (étape 5). Il a déjà été dit qu'une solution pour injecter une librairie à distance sans jamais écrire quelque chose sur le disque est soit de passer par une partition de type **tmpfs** (*On-Disk remote Library Injection*) ou alors de recoder **dlopen()** (*In-Memory Remote Library Injection*).

Nous proposons ici une troisième méthode, un peu farfelue mais pas si dénuée de sens que ça. L'idée est simplement de rapatrier sur le système de l'attaquant le binaire correspondant au serveur à corrompre. Une fois sur la machine de l'attaquant, il peut être *backdooré* de n'importe quelle manière (par exemple avec un outil comme ELFsh en utilisant l'injection **ET_REL** et le détournement par **ALTPLT** [14]). Une fois *backdooré*, il peut ensuite être renvoyé vers le système

cible et exécuté en mémoire à l'aide d'un *userland execve*. De cette manière, le serveur est corrompu en mémoire mais pas sur le disque. Ajoutez à cela que si le binaire est compilé en statique, il peut être *backdooré* facilement sur le système de l'attaquant à l'aide de la technique **CFLOW** [14]. Cette dernière méthode est donc plus puissante, même si elle est moins évidente.

7 Protections

Il n'est pas question de « rabacher » (et c'est bien le terme) les méthodes de protection classiques qui peuvent être appliquées sur un système. Mettre en place un *firewall*, choisir et éventuellement auditer les applications en production, mettre en place des protections contre les techniques d'exploitation tels que les *buffers overflow*, ... toutes ces mesures de sécurité sont expliquées en long et en large sur tous bons sites de sécurité qui se respectent.

Il est préférable de voir des méthodes de protection plus spécifiques, plus axées sur le sujet traité dans ce document.

7.1 ptrace() et les appels système

L'appel système **ptrace()** a souvent été évoqué au cours de cet article. D'origine, **ptrace()** est un appel système servant à tracer et débugger un processus en mode utilisateur (GDB l'utilise). Il est présent par défaut sur la plupart des systèmes Unix standard. Pour un attaquant, il sert plutôt à injecter des données (que ce soit de simples données ou bien du code à exécuter) dans des processus en cours d'exécution.

La mesure de sécurité à prendre en compte est évidente : il suffit de contrôler ou d'interdire l'utilisation de **ptrace()** tout simplement. Comme il est impossible d'altérer la mémoire d'un processus à l'aide de **/proc/pid/mem**, si l'appel à **ptrace()** est interdit, les techniques d'injection de données ne fonctionneront plus. Un simple module noyau qui détourne la table des appels système fait l'affaire (sans aucun risque d'effet de bord puisque **ptrace()** est principalement utilisé par les debuggeurs). L'autre solution est d'utiliser grsecurity [17] qui propose cette fonctionnalité.

Bien évidemment, à l'image de **ptrace()**, il est tout à fait possible d'offrir ces mêmes mesures de sécurité pour d'autres appels système sensibles tel que **mmap()** par exemple.

D'une manière général, tous les appels systèmes peuvent être contrôlés ou interdits. Systrace [16] par exemple est un outil développé par Niels Provos (développeur d'OpenSSH entre autres) présent sur BSD et Linux permettant de renforcer la sécurité de son système[16]. Il permet de contrôler de manière

[16] Sauf quand il contient des failles de sécurité :)

très détaillée le comportement et les droits des applications au niveau appel système. Pour chaque programme exécuté sur une machine, Systrace permet de définir un ensemble d'appels système que le logiciel a le droit ou non d'exécuter : ouverture/écriture de fichiers, lecture/écriture d'une socket TCP/IP, allocation de mémoire, etc. L'ensemble de ces règles est appelé politique de sécurité. Cette politique de sécurité est générée de manière interactive et les accès non définis génèrent une alarme.

7.2 Audit des processus en cours d'exécution

Une autre solution de protection de sécurité (il est plus question de prévention) est de vérifier l'intégrité des processus. A l'image de ProcShow [20] développé par Nicolas Waisman, il suffit de récupérer un maximum d'informations quant à la structure ELF, les mappages mémoire des processus, de désassembler le segment de code du processus, etc pour en vérifier les anomalies. Par exemple, il peut être envisagé que l'attaquant ait injecté du code dans le processus mais n'ait pas modifié le binaire correspondant par souci. Donc, une comparaison des structures ELF du processus et du binaire (un point d'entrée différent, du code injecté dans le *padding*,...) peut permettre de détecter une éventuelle *backdoor* injectée dans le processus. Du moins si l'attaquant n'a pas pris soin à l'aide d'un code noyau de cacher les différences entre le binaire et le processus, ce qui est tout à fait possible.

Dans cette optique, Samuel Dralet a développé une librairie [8] qui permet d'extraire les structures ELF d'un processus. L'utilisation de cette librairie et de la librairie **libelfsh** [9] peut apporter des résultats convaincants en matière de détection. Nous empiétons cependant sur le domaine du forensics ou analyse post-mortem.

8 Conclusions

Nous arrivons à la fin de notre sujet. Beaucoup de choses ont été presentées, mais beaucoup d'autres n'ont pas pu l'être pour ne pas encore allonger plus l'article[17]. Nous pensons notamment à plus de détails concernant Plato, aux techniques permettant de cacher des fichiers à distance dans la mémoire vive (*remote DHIS* [12]) ou encore à l'injection dans le noyau d'une *backdoor* à l'aide de **kmem**, le tout à partir d'un exploit et sans jamais rien écrire sur le disque. Néanmoins, cet article permettra au lecteur de se faire une bonne idée des techniques principales permettant de corrompre un système sans jamais rien écrire sur le disque.

Les deux techniques principales, le *syscall proxy* et le *remote userland execve*, sont vraiment des techniques très pratiques pour exécuter du code à distance sans

[17] Si si on vous assure...

rien écrire sur le disque (et qui ne sont pas si compliquée à mettre en oeuvre), avec une petite préférence tout de même pour le *remote userland execve*. La technique d'injection de librairies à distance est quant à elle très utile pour laisser une *backdoor* sur un système. Nous espérons avoir ainsi un peu démystifié ces techniques tout au long de cet article.

Mais il y a une chose principale à retenir : la mémoire vive est tout aussi importante, si non plus, qu'une mémoire de masse. Il est primordial pour tout administrateur et expert en sécurité qui se respectent d'en avoir conscience.

Un dernière remarque pour terminer, mais là c'est plus personnel, c'est que nous croyons plus en la théorie de Platon que celle d'Aristote, d'où peut-être le nom donné à notre logiciel. Mais là à chacun sa propre opinion ...

Références

1. Anonymous, *Runtime Process Infection*, `http://www.phrack.org/show.php?p=59&a=8`
2. Arce I., *The Shellcode Generation*, `http://www.coresecurity.com/files/files/51/TheShellcodeGeneration.pdf`
3. Ares, *Runtime Process Infection 2*, `http://ares.x25zine.org/ES/txt/0x4553-Runtime_Process_Infecting.htm`
4. Biondi P. *Shellforge*, `http://www.secdev.org/projects/shellforge/files/`
5. Biondi P. *About Unix Shellcodes*, `http://www.secdev.org/conf/shellcodessyscan04.pdf`
6. Cáceres M., *Syscall Proxying - Simulating remote execution*, `http://www1.corest.com/common/showdoc.php?idx=259&idxseccion=11`
7. Cuttergo A. E., *The joys of impurity*, `http://archives.neohapsis.com/archives/vuln-dev/2003-q4/0006.html`
8. Dralet S., *LibMemProc*, `http://forensics-dev.blogspot.com/2006/03/libmemproc.html`
9. ELFsh crew *ELFsh*, `http://elfsh.devhell.org/`
10. ELFsh crew, *Embedded ELF Debugging : the middle head of Cerberus*, `http://www.phrack.org/show.php?p=63&a=9`
11. Gaspard F. et Dralet S., *Techniques anti-forensics sous Linux : utilisation de la mémoire vive*, MISC 25.
12. Gaspard F. et Dralet S., *Distributed Hidden Storage*, `http://dhis.devhell.org`
13. Long J., Lointier P. Google Hacking : Mettez vos données sensibles à l'abri des moteurs de recherche.
14. Mayhem, *The Cerberus ELF Interface*, `http://www.phrack.org/show.php?p=61&a=8`
15. Pluff, *Perverting Unix Processes*, `http://www.securityfocus.com/archive/1/428244/30/0/threaded`
16. Provos N., *Systrace*, `http://www.citi.umich.edu/u/provos/systrace/`

17. Spengler B., *Grsecurity*, `http://www.grsecurity.net`
18. The Grugq, *Announcing Userland Exec*, `http://cert.uni-stuttgart.de/archive/bugtraq/2004/01/msg00002.html`
19. Turkulainen J., *Remote Library Injection*, `http://www.nologin.net/Downloads/Papers/remote-library-injection.pdf`
20. Waisman N., *Procshow*, `http://www.remoteassessment.com/darchive/191006388.html`