solutions linux opensource www.solutionslinux.fr

09 déc 2008

Protection de l’espace d’adressage : état de l’art sous Linux et OpenBSD

Catégorie : Sécurité Tags : misc

~~Retrouvez cet article dans : **Misc 23**~~

La protection de l’espace d’adressage d’un processus permet de rendre plus difficile l‘exploitation de failles de sécurité. Plusieurs techniques existent, notamment mises en oeuvre au niveau du noyau Linux, qui se montrent très efficaces, pour un coût presque négligeable en performances et en temps d’administration. Sous Windows, par manque de contrôle sur les éléments de bas niveau du système, les techniques les plus avancées sont difficiles à mettre en oeuvre et les solutions actuelles, regroupées sous le nom de HIPS (Host instrusion prevention systems) n’ont pas la même efficacité.

1. Prévention générique d’exploitation de failles de sécurité

Introduction

Beaucoup savent qu’une simple erreur de programmation peut être exploitée par un attaquant pour exécuter du code arbitraire avec les privilèges du

processus vulnérable. La possibilité d'exploiter certaines failles est souvent due à des détails d'implémentation de bas niveau, rarement connus du programmeur : par exemple le fait que des données soient mélangées à des structures de contrôle, que ce soit sur la pile (variables locales et adresse de retour sur architecture Intel), sur le tas (données et métadonnées de gestion du tas) ou dans une chaîne de format. De nombreux domaines, a priori orthogonaux, se trouvent mélangés. Qui aurait prévu que les choix a priori indépendants des encodages ASCII et des opcodes d'Intel ou l'adresse virtuelle choisie par le linker pour une fonction jouent un rôle dans la facilité d'exploitation d'une faille ? De ce fait, déterminer l'exploitabilité ou non d'une faille de sécurité est délicat. Il est cependant possible de rendre plus difficile, voire impossible, l'exploitation de certaines erreurs de programmation de manière générique, sans connaître la faille en question a priori.

Classification

Dans cet article, nous nous limiterons aux failles de sécurité dont l'exploitation se fait en contrôlant le flot d'exécution du processus. Cela exclut par exemple les erreurs logiques et un cas particulier de dépassement de tampon où l'on pourrait écraser une variable vitale gérant nos privilèges.
Les étapes de l'exploitation d'une faille (prenons ici pour simplifier un buffer overflow) sont généralement les suivantes :

- 1 Recherche d'une faille dans un programme ;
- 2 Communication avec le processus vulnérable afin de déclencher le dépassement de tampon ;
- 3 Injection de code arbitraire dans l'espace d'adressage du processus (facultatif) ; Protection de l'espace d'adressage : état de l'art sous Linux et OpenBSD
- 4 Contrôle du flot d'exécution du programme (éventuellement pour le rediriger vers le code arbitraire préalablement injecté) ;
- 5 Détournement du processus pour réaliser diverses actions (installation de rootkit, récupération de certaines données, lancement d'un nouvel exploit) ;

Le séquencement de ces étapes n'est pas strict, on peut par exemple imaginer utiliser le contrôle du flot d'exécution du programme pour injecter du code arbitraire ou, ce qui arrive plus fréquemment, pour rendre des données injectées exécutables (nous reviendrons là-dessus).Des mécanismes de prévention générique existent pour prévenir chacune de ces étapes :

- 1 Détection d'erreurs de programmation par analyse statique (sparse, Coverity, Microsoft PREfix/PREfast, etc.) ;

- 2 Empêcher que les conditions de la faille (par exemple l'overflow) se produisent à l'exécution (libsafe, BCC (bound checking GCC) et CASH) ;
- 3 Empêcher l'injection de code arbitraire dans l'espace d'adressage d'un processus [0] (PaX, OpenBSD's W^X) ;
- 4 Empêcher le contrôle du flot d'exécution par l'attaquant (SSP/ Propolice), obscurcissement de l'espace d'adressage (PaX, Ozone, Whentrust) ;
- 5 Empêcher un processus contrôlé par l'attaquant de faire certaines actions en limitant ses droits (Système de contrôle d'accès, SELinux, TrustedBSD, RSBAC, GrSecurity's RBAC, LIDS, McAfee Entercept, CISCO CSA).

Dans cet article, nous traiterons des étapes 3 et 4 sous Linux et OpenBSD. L'étape 5 sera toutefois abordée car les cinq étapes ci-dessus ne sont pas totalement disjointes, et la correspondance avec les étapes d'exploitation n'est pas stricte : par exemple, un système de contrôle d'accès peut participer à empêcher l'injection de code arbitraire dans l'espace d'adressage d'un processus (nous y reviendrons). Le cas de Windows sera à peine abordé, nous y reviendrons sans doute dans un prochain article.

2. Éléments de système

Nous allons décrire quelques éléments du fonctionnement d'un système d'exploitation sur architecture Intel.
Une grande partie de la sécurité d'un système d'exploitation moderne n'est possible que grâce à l'existence d'une MMU
(Memory Management Unit) dans le processeur utilisé. Grâce à celle-ci :

0 Dans cet article, « injecter du code » signifiera injecter ce code dans une zone exécutable ou injecter ce code dans une zone non exécutable, puis rendre cette zone exécutable.

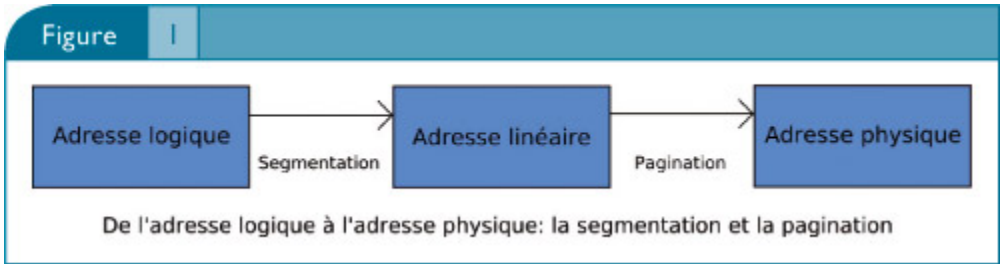
- Il existe un niveau de privilège matériel qui sépare le noyau du code exécuté en mode utilisateur.
- Le noyau contrôle quels périphériques sont accessibles pour Figure 1 un processus en mode utilisateur (à l'aide du champ IOPL du registre EFLAGS et du bitmap de permissions enregistré dans le TSS). En règle générale, seul le noyau peut accéder directement aux périphériques et, en mode utilisateur, il faut utiliser des services du noyau (appels système) pour que le processus puisse exécuter des instructions (contrôlées) en mode noyau et accéder au matériel.
- La mémoire d'un processus est virtualisée. Il se voit seul dans son espace d'adressage.

Ces trois éléments sont essentiels pour la sécurité car ils rendent possible une séparation des privilèges entre processus : le noyau étant un passage obligé pour la communication entre les processus et le matériel, il est le superviseur de la sécurité. En effet, grâce à l'espace d'adressage virtuel, un processus doit passer par le noyau pour pouvoir par exemple modifier l'espace d'adressage d'un autre processus (ptrace(), création de zones de mémoire partagée..). C'est ce qui permet des modèles de séparation de privilège simples (par uid/gid) ou complexes (mandatory access control par exemple). Cette vision des choses est très importante lorsque l'on étudie les systèmes de contrôle d'accès (RSBAC, SELinux, TrustedBSD…), c'est-à-dire lorsque l'on suppose qu'un attaquant possède déjà le contrôle d'un processus : le fait qu'il soit ensuite obligatoire de passer par le noyau pour « sortir de ce processus » (en ouvrant des fichiers, en infectant l'espace d'adressage d'un autre processus, en ouvrant des connexions réseau…) est alors capital.

Segmentation et pagination

Dans cet article cependant, nous allons voir comment l'on peut réagir un niveau auparavant, afin d'empêcher le contrôle du processus par l'attaquant. Ce qui nous intéresse surtout est le procédé de virtualisation de l'espace d'adressage et nous allons l'étudier rapidement dans le cadre de l'architecture Intel IA32 en mode protégé.
Un programme travaille avec des adresses dites « logiques » : dans l'instruction ~~mov eax, dword ptr ss:[esp+4]~~, si ~~esp~~ vaut 0xBFFFFF80, le mot contenu à l'adresse logique pointée par ss :0xBFFFFF84 est mis dans le registre EAX.
Afin de savoir quelle adresse (physique) il doit utiliser sur le bus mémoire, le processeur effectue deux transformations :

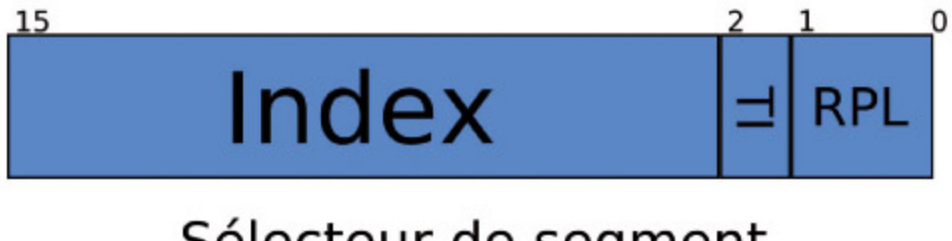
- 1 La première, appelée « la segmentation », transforme l'adresse logique en adresse linéaire (aussi appelée adresse virtuelle) ;
- 2 La deuxième, appelée « pagination », transforme l'adresse linéaire en adresse physique.



Unité de segmentation

Nous l'avons vu, les adresses mémoire utilisées par un programmeur sont des adresses logiques. Celles-ci sont de la forme « selector:offset » où « selector » est un sélecteur de segment de 16 bits et offset un décalage de 32 bits. Les sélecteurs de segment sont en pratique rarement spécifiés, car par défaut, (c'est-à-dire sans préfixe segment override), le segment de code (cs) est utilisé pour tous les instruction fetch, le segment de pile (ss) pour tous les push, pop et références utilisant ESP ou EBP, le segment de données (ds) pour toutes les références sauf celles liées à la pile ou aux chaînes, le sélecteur ES pour les instructions liées aux chaînes de caractères. Nous allons décrire, de manière peu détaillée (les lecteurs intéressés peuvent se reporter à la documentation [Intel]) le mécanisme de segmentation.
C'est dans la segmentation (et non pas dans la pagination) que l'on retrouve les fameux rings, c'est-à-dire les niveaux de privilèges Intel. Ils vont de 0 à 3, 0 étant le privilège le plus fort. En pratique les systèmes d'exploitation courants n'utilisent que les privilèges 0 (mode noyau) et 3 (mode utilisateur).
Considérons par exemple l'adresse logique précédente avec 7B pour valeur de SS (7B:0xBFFFF884). Le sélecteur de segment se décompose en trois champs (voir figure ci-dessous).

Figure 2



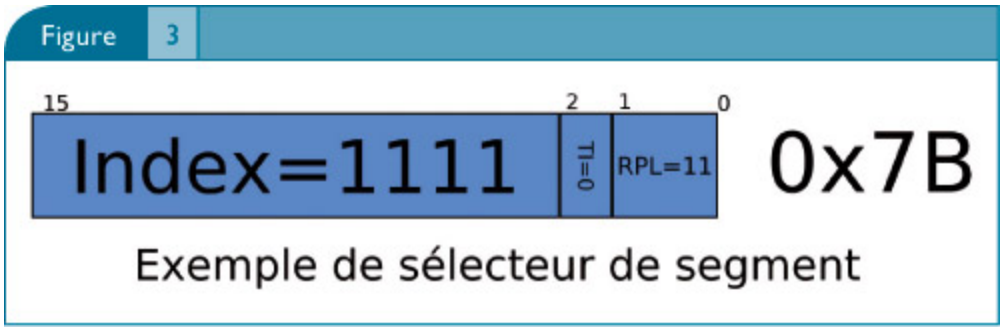
Sélecteur de segment

Le champ TI (table indicator) indique dans quelle table de descripteurs, GDT (global descriptor table) ou LDT (local descriptor table), le descripteur de segment correspondant doit être cherché.

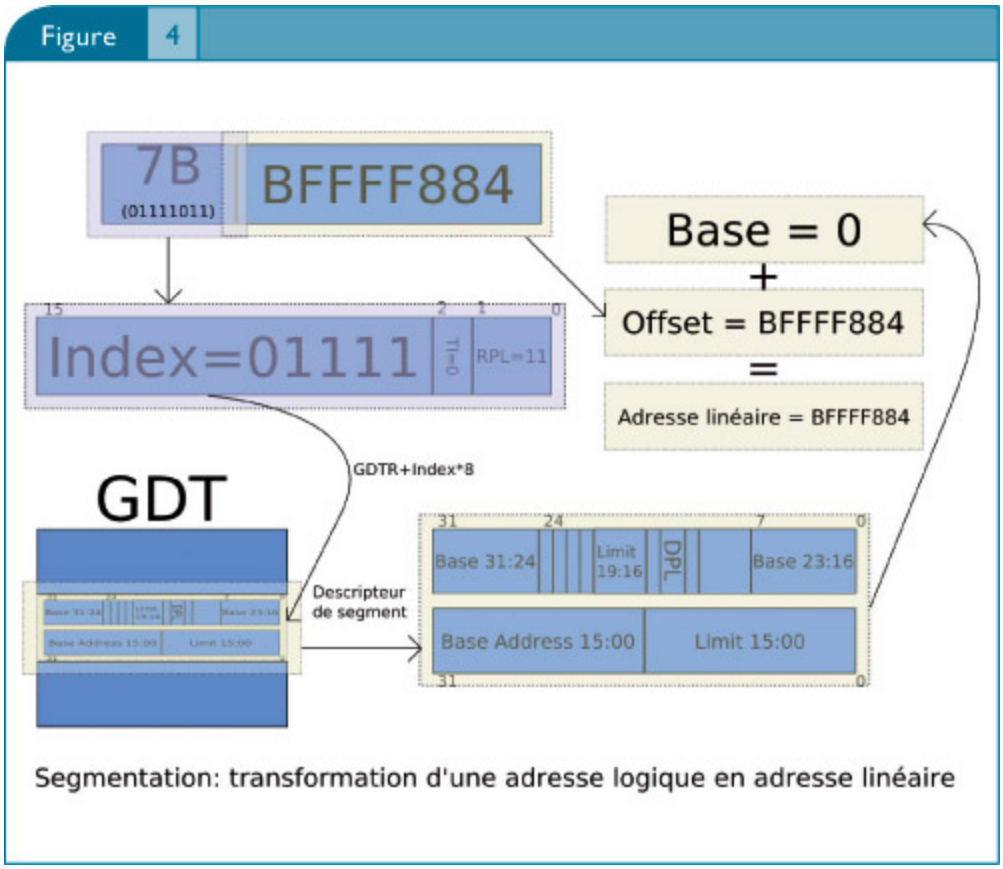
- 2 Le champ Index indique l'index du descripteur de segment dans la GDT ou la LDT
- 3 Le champ RPL indique le requested privilege level, i. e. le plus haut privilège nécessaire pour avoir accès au segment. Dans le cas de CS, ce champ indique le CPL (current privilege level, c'est-à-dire le niveau de privilège courant (typiquement 0 en mode kernel et 3 en mode utilisateur).

Ainsi, notre sélecteur SS dont la valeur est 0x7B indique que le descripteur de

segment à utiliser est le numéro 15 (0b1111) dans la GDT et que le RPL est 3.



Grâce au sélecteur de segment, le processeur repère un descripteur de segment. Selon l'indicateur de table (TI), le descripteur de segment est recherché dans la LDT ou dans la GDT (les adresses linéaires de ces tables sont situées dans les registres LDTR et GDTR). À l'aide du champ index, le processeur repère le descripteur de segment désiré dans la table. Ce descripteur de 8 octets contient de nombreux champs, entre autres un champ base et un champ limit : l'addition de ce champ base et de l'offset donne l'adresse linéaire, le champ limit permet de limiter la taille du segment. Parmi les autres champs importants, citons le champ DPL (descriptor privilege level) qui, combiné au CPL (derniers bits de CS) et au RPL du descripteur utilisé permet de vérifier les privilèges.
Par défaut, la majorité des systèmes d'exploitation actuels utilisent le Flat memory model, c'est-à-dire qu'ils utilisent zéro comme base pour les principaux segments (ceux dont les sélecteurs sont chargés dans cs, ds, ss, es), au moins pour les segments utilisés en mode user, et 0xFFFFF comme limite (ce qui donne une limite de 4 Go au segment). L'offset d'une adresse logique peut ainsi dans la majorité des cas être identifiée à l'adresse linéaire. Ceci explique les nombreuses confusions entre adresse linéaire et adresse logique.
La segmentation joue en général un rôle peu important dans les systèmes d'exploitation modernes, son utilisation est cependant obligatoire sur processeur Intel, pour des raisons historiques. Nous verrons cependant plusieurs méthodes pour tirer parti de la segmentation. Vous pouvez explorer vos descripteurs de segments à l'aide du programme [DTDUMPER].



Unité de pagination

La pagination constitue la deuxième étape de la translation d’adresse : elle transforme les adresses linéaires en adresses physiques que le processeur envoie sur le bus mémoire. L’espace d’adressage linéaire est découpé en pages (en général de 4 Ko), la pagination fait la correspondance entre les pages et les cadres de pages (pages en mémoire physique). En pratique, sur la plupart des systèmes d’exploitation, c’est elle qui joue le rôle le plus important, elle permet de vérifier les droits d’accès aux pages (droit en écriture, privilège nécessaire pour accéder à la page)... et grâce à elle des mécanismes avancés sont possibles :

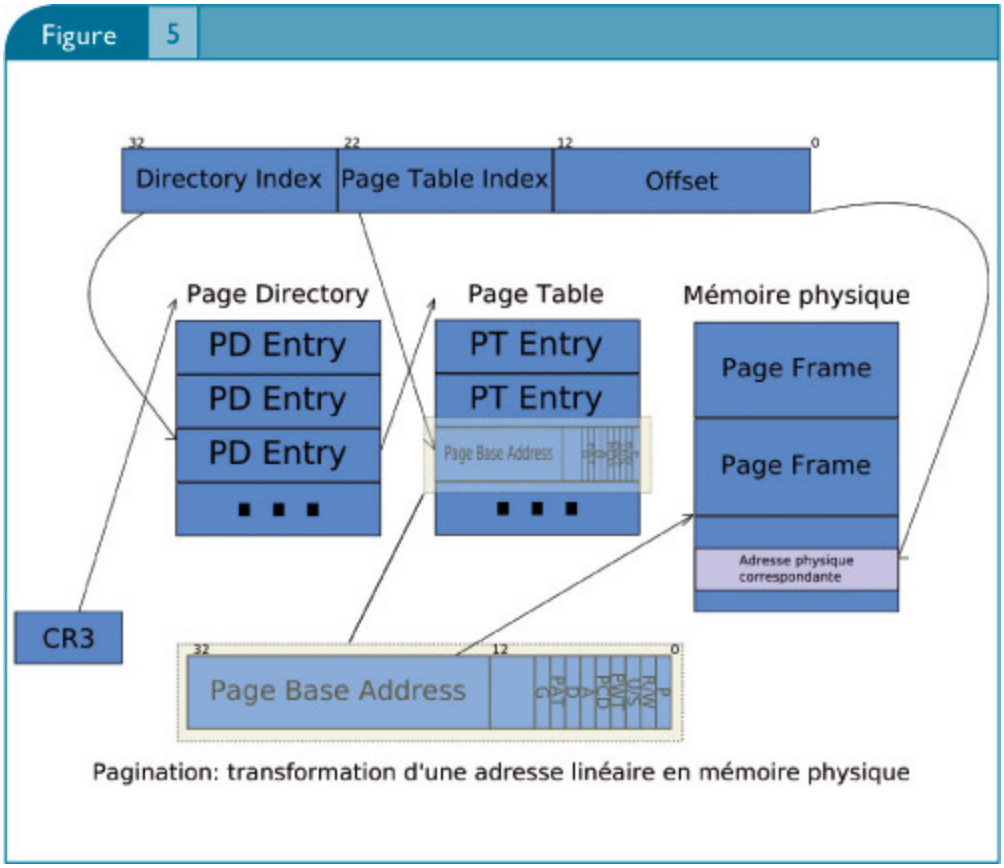
- Un cadre de page n’est pas forcément alloué en mémoire physique pour une page effectivement utilisable par le processus : l’utilisation de cette page provoquera une page fault et le handler correspondant du noyau allouera le cadre de page correspondant en mémoire physique, en y plaçant éventuellement les données nécessaires :
  - Swaping : des cadres de page peuvent être déchargés de la mémoire physique et placés sur le disque.

    - On demand paging : des fonctions telles que ~~mmap()~~ ou ~~MapViewOfFile~~ sont utilisées pour mapper un fichier en mémoire, les pages ne seront effectivement mappées qu’à l’utilisation.
- Le copy on write : deux processus sans mémoire partagée peuvent partager le même cadre de page ; lorsque l’un d’eux écrira sur sa page, le cadre de page sera dupliqué. Ce procédé est particulièrement avantageux pour mapper des bibliothèques.

Il existe plusieurs types de pagination sur les processeurs Intel.

Nous allons pour simplifier décrire ici rapidement le mode de pagination simple avec pages de 4 Ko (sans PSE, sans PAE, sans IA32e). Dans ce mode, la pagination s’effectue à deux niveaux : une adresse linéaire contient un index de Directory, un index de Table (de 10 bits chacun, qui permettent de repérer une entrée dans une table de 1024 entrées) et un offset de 12 bits qui permet de se repérer à un octet près dans une page de 4 Ko.

Figure 5



Pagination: transformation d'une adresse linéaire en mémoire physique

Le registre cr3 contient l’adresse physique du Page Directory, cette adresse, combinée au Directory Index nous fournit l’adresse d’un PDE (Page Directory Entry), qui lui-même permet de repérer une Page table qui, combinée au Table Index, fournit l’adresse du PTE (Page table entry) de la page. Celui-ci contient un champ Page base address de 20 bits qui permet de repérer le cadre de

page (Page Frame) en mémoire physique ainsi que divers flags qui indiquent notamment les droits de la page (Read/Write), ses privilèges (User/Supervisor; notons qu’il n’y a ici que deux niveaux et non pas 4 Rings), si elle est présente en mémoire physique, si elle a été accédée...
Nous ne détaillerons pas les PDE qui sont similaires aux PTE (voir Figure). Remarquons qu’un PTE permet de marquer une page comme étant disponible en écriture ou non (W), mais qu’il n’existe pas de flag pour marquer une page disponible en exécution.
L’absence de ce flag pose de gros problèmes de sécurité (il fut ensuite réintroduit en fanfare sous la dénomination de flag NX, nous y reviendrons).
Notons que comme nous ne perdons pas d’information, les 32 bits d’une adresse linéaire permettent bien d’adresser 4 Go de mémoire.

Espace d’adressage d’un processus

Voyons à titre d’exemple comment se crée l’espace d’adressage d’un processus sous Linux 2.4. On peut voir la partie utilisateur de l’espace d’adressage linéaire d’un processus en lisant le fichier ~~/proc/pid/maps~~. Sous Linux 2.4, tous les descripteurs de segment qui nous intéressent ont une base nulle; on peut donc ici directement confondre l’espace d’adressage logique (tel qu’il est vu par le processus) et l’espace d’adressage linéaire (après segmentation).

```
$ cat /proc/self/maps
08048000-0804c000 r-xp 00000000 03:03 32672 /bin/cat [1]
0804c000-0804d000 rw-p 00003000 03:03 32672 /bin/cat [2]
0804d000-0806e000 rwxp 00000000 00:00 0 [3]
40000000-40016000 r-xp 00000000 03:03 179102 /lib/ld-2.3.2.so
40016000-40017000 rw-p 00015000 03:03 179102 /lib/ld-2.3.2.so
40017000-40018000 rw-p 00000000 00:00 0
4001d000-40145000 r-xp 00000000 03:03 179607 /lib/libc-2.3.2.so
40145000-4014d000 rw-p 00127000 03:03 179607 /lib/libc-2.3.2.so
4014d000-40150000 rw-p 00000000 00:00 0
40150000-4019f000 r--p 00000000 03:03 942989 /usr/lib/locale/locale-archive
bfffe000-c0000000 rwxp fffff000 00:00 0 [pile]
```

L’espace d’adressage d’un processus est changé lors de l’appel système ~~sys_execve()~~ qui remplace le contexte d’exécution d’un processus (la création d’un nouveau processus est due à l’appel système ~~sys_fork()~~).
L’espace d’adressage linéaire au-dessus de 3 Go (0xC0000000) est réservé au noyau et est mappé dans chaque processus (c’està- dire que les pages situées au-dessus de 0xC0000000 dans chaque processus pointent vers les mêmes cadres de page). Décrivons rapidement quelques étapes du remplacement du contexte d’exécution d’un processus réalisées par la fonction ~~sys_execve()~~.
Les exécutables sous Linux utilisent généralement le format ELF. Un fichier

ELF peut être vu de deux manières, selon le header considéré : le Program Header offre une vision adaptée à un chargeur de programme, alors que la section header table offre une vision adaptée à un linker statique.

Le Program Header décrit le fichier exécutable comme une suite de segments. Les segments d'un programme sont des informations pour le chargeur de programme, ils n'ont absolument rien à voir avec la segmentation Intel. Voici le program header de « ~~cat~~ ».

```
$ readelf -l cat
Elf file type is EXEC (Executable file)
Entry point 0x8048b70
There are 7 program headers, starting at offset 52
Program Headers:
Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
PHDR 0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
INTERP 0x000114 0x08048114 0x08048114 0x00013 0x00013 R 0x1
[Requesting program interpreter: /lib/ld-linux.so.2]
LOAD 0x000000 0x08048000 0x08048000 0x03a2d 0x03a2d R E 0x1000
LOAD 0x003a3 0x0804ca30 0x0804ca30 0x001d0 0x0033c RW 0x1000
DYNAMIC 0x003a7c 0x0804ca7c 0x0804ca7c 0x000c8 0x000c8 RW 0x4
NOTE 0x000128 0x08048128 0x08048128 0x00020 0x00020 R 0x4
GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4
Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .init .plt .text .fini .rodata
03 .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
04 .dynamic
05 .note.ABI-tag
06
```

Voici quelques étapes du chargement d'un programme :

- 1 La fonction ~~flush_old_exec()~~ du noyau libère les ressources utilisées par le précédent programme, en particulier toutes les régions mémoire user.
- 2 La pile est allouée, juste en dessous de la limite de l'espace utilisateur. Nous pouvons la voir ici de 0xbfffe000 à 0xc0000000.
- 3 Les segments de type ~~PT_LOAD~~ sont chargés en mémoire. On peut voir ici qu'il y a une zone mémoire avec les permissions ~~read~~ et ~~execute~~ [1] et une zone mémoire avec des permissions ~~read~~ et ~~write~~ [2]. On retrouve bien ces zones dans notre fichier maps. Notons que le deuxième segment a une ~~MemSiz~~ supérieure à sa ~~FileSiz~~. Cela est à l'origine de la création de la zone [3] qui contient les données non initialisées (section ~~.bss~~) et qui sert de point de départ au tas (heap).
- 4 Grâce au Program Header (repéré par son type ~~PT_INTERP~~), le noyau localise le chargeur dynamique (~~ld-linux.so~~), le charge en mémoire et lui

transfère l’exécution.
- 5 Le chargeur dynamique utilise l’appel système ~~sys_mmap()~~ pour charger les bibliothèques dynamiques (entrées ~~DT_NEEDED~~ dans la table dynamique) en mémoire. Ces bibliothèques dynamiques sont elles-mêmes au format ~~ELF~~, de type ~~ET_DYN~~.
- 6 Le chargeur dynamique réalise des relocations pour le programme principal et les bibliothèques chargées. C’est lui qui met éventuellement en place le mécanisme GOT/PLT.
- 7 Le chargeur dynamique transfère l’exécution au point d’entrée du programme

Le format ELF est très complexe (il est par exemple beaucoup plus complexe que le format PE de Microsoft). Les lecteurs intéressés peuvent se reporter à la spécification TIS [ELF].
Le fichier maps de la page précédente correspond à un noyau 2.4. Avec un noyau 2.6 nous verrions quelques différences :

- Presque tout en haut de l’espace d’adressage linéaire(0xffffe000), une page est réservée par le noyau. Elle contient une bibliothèque ELF appelée ~~vDSO~~ qui est utilisée pour réaliser des appels systèmes. Celle-ci est chargée automatiquement en mémoire par le kernel dans tous les processus. Selon le type de processeur, cette bibliothèque réalise des appels système en utilisant int ~~0x80/irot~~ ou ~~sysenter/sysexit~~.
- Les adresses des bibliothèques dynamiques sont changées car l’allocation se fait maintenant du haut vers le bas.
- À partir du noyau 2.6.12 on peut constater une faible randomisation des adresses des bibliothèques dynamiques et de la pile (nous reviendrons dessus dans la partie consacrée à Exec-Shield).

3. Prévention d’exécution de code arbitraire

Nous décrivons ici plusieurs techniques visant à empêcher la prise de contrôle du processus par un attaquant. Comme nous l’avons expliqué, nous nous limiterons aux prises de contrôles fortes qui permettent de détourner le flot d’exécution du programme.
Cela est réalisé en :

- 1 Empêchant l’attaquant de détourner le flot d’exécution du programme ;
- 2 Empêchant l’attaquant d’injecter du code arbitraire.

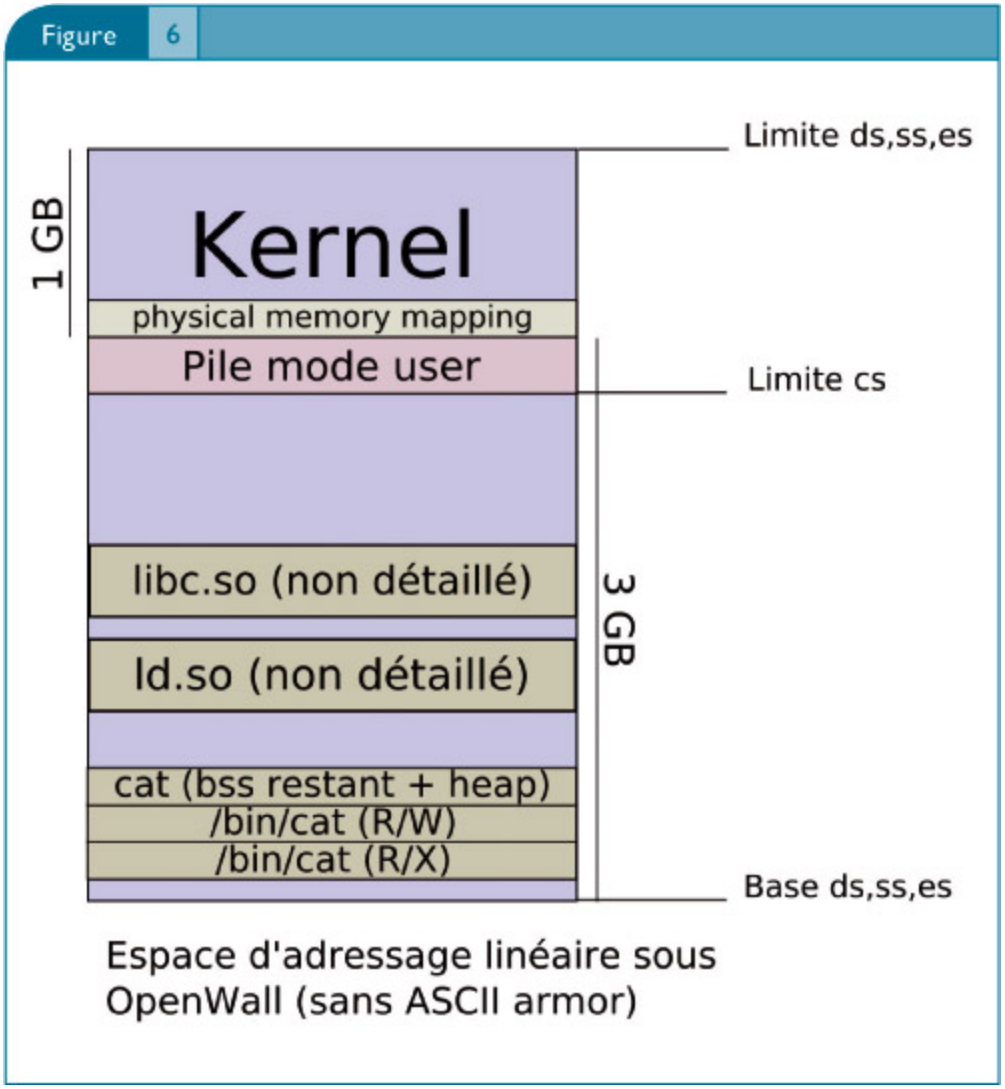
Un attaquant peut simplement détourner le flot d’exécution du processus sans injecter de code arbitraire. En revanche le contrôle est souvent beaucoup plus simple et plus fort en injectant du code arbitraire puis en détournant le flot d’exécution du processus vers celui-ci. Les choses se compliquent si l’on considère que l’on peut détourner le flot d’exécution du programme vers du code existant dans l’espace d’adressage, de sorte à injecter du code arbitraire et à l’exécuter.

OpenWall

Le premier patch pour le noyau Linux destiné à rendre plus difficile l’exploitation d’erreurs de programmation fut OpenWall (sorti en 1998) [Openwall]. Bien avant l’ère des exploits de format strings (~2000) et avant la démocratisation des heap overflows, la plupart de ces erreurs de programmation avaient pour conséquence un dépassement de tampon sur la pile. La technique classique d’exploitation étant d’écraser une adresse retour afin d’exécuter du code injecté sur la pile, il semblait souhaitable de rendre la pile non exécutable.

On peut certes marquer la pile comme étant non exécutable (c’est-à-dire changer les données internes au noyau, les virtual memory areas), mais à cause de l’absence de flag marquant le droit d’exécution dans les PTE, cela n’impliquera pas de changement dans les structures utilisées par le processeur et n’aura donc aucun effet utile. Pour pallier ce défaut de l’architecture Intel, Solar Designer utilisa une technique relativement simple afin de rendre la pile non exécutable : en mode utilisateur, le descripteur de segment référencé par le registre cs possède une limite inférieure à l’adresse linéaire de la pile. Les autres descripteurs de segment sont inchangés.

Ainsi, lorsqu’un accès à la pile est réalisé relativement aux registres ds, ss et es, aucune différence n’apparaît. La base du descripteur de segment (0) est ajoutée à l’offset demandé, la somme ne dépassant pas la limite du segment (4 Go), aucune protection matérielle due à la segmentation ne vient perturber l’accès à la mémoire. En revanche, lorsque le processeur tente d’exécuter du code sur la pile, le descripteur de segment référencé par le sélecteur contenu dans le registre cs est utilisé. La somme de la base de ce descripteur (0) et de l’offset demandé dépasse la limite de ce segment (modifiée dans ce but), le processeur lève une exception de protection générale et le noyau Linux tue le processus.



Les limites de cette protection sont nombreuses. Il reste en effet possible d'injecter du code exécutable ailleurs que sur la pile et d'utiliser du code existant. Solar Designer fut également le premier à proposer d'utiliser la technique du return-to-libc afin de passer outre sa protection. Mais on peut également imaginer utiliser de nombreuses techniques plus avancées, par exemple des return-to-libc chaînés (voir l'article de Rafal Wojtczuk dans Phrack [Wojtczuk]), afin de recopier son code exécutable dans une autre zone mémoire puis de l'exécuter.
Afin de déjouer ces techniques visant à exploiter du code existant dans l'espace d'adressage, Solar Designer a proposé l'une des premières techniques d'obfuscation de l'espace d'adressage, l'armure ASCII. Le principe est que les bibliothèques sont chargées à une adresse inférieure à 0x01010000 afin que leurs adresses contiennent toujours un zéro. Cela peut en effet empêcher l'injection d'une adresse de retour cible (par exemple celle de ~~system()~~) dans l'espace d'adressage d'un processus via des chaînes de caractères. Cette méthode n'est bien entendue utile que s'il est difficile pour un attaquant

d’injecter des zéros et il est toujours possible d’utiliser du code existant dans les zones mémoires correspondants au fichier exécutable principal, qui lui ne se trouvera pas dans l’ASCII armor.

Malgré ses limites très importantes, ce patch extrêmement simple déjoue un grand nombre des exploits publics visant des stack overflows (ce qui n’a certes pas grand chose à voir avec la sécurité réelle, n’en déplaise aux éditeurs d’antivirus).

PaX

PaX [PaX] est né en 2000, il était alors plus une preuve de concept exploitant une des idées du projet PleX86 (utiliser les TLB séparés pour pouvoir instrumenter séparément la lecture/écriture et l’exécution d’une page sur architecture Intel) qu’un véritable patch de sécurité. Il a évolué en introduisant successivement l’ASLR (Address Space Layout Randomization) en juillet 2001 puis le VMA mirroring, (base de SEGMEXEC et RANDEXEC) en juillet 2002, ainsi qu’en démocratisant des méthodes de compilation permettant d’obtenir des exécutables ET_DYN. Il supporte plus de dix architectures (Alpha, i386, ia64, Mips, Mips64, Parisc, PPC, PPC64, Sparc, Sparc64, x86_64) et sert de base aux distributions sécurisées Hardened Gentoo et Adamantix ainsi qu’au patch GrSecurity.
Nous l’avons vu, la pagination Intel fournit nativement un flag permettant de rendre ou non une page disponible en écriture dans les PTE. Idéalement, les protections d’une page devraient pouvoir être une combinaison arbitraire de PROT_READ, PROT_WRITE, PROT_EXEC (cette terminologie est celle utilisée notamment dans l’appel système ~~mprotect()~~ qui est utilisé pour changer les protections d’une zone mémoire), mais nous avons vu que seul PROT_WRITE avait un sens au niveau matériel. Le premier rôle de PaX est d’émuler une sémantique de pages non exécutables sur les processeurs Intel qui ne le supportent pas, c’est-à-dire de donner une signification à PROT_EXEC (cette fonctionnalité s’appelle NOEXEC). Il existe deux techniques : la première, PAGEEXEC, induit une surcharge, la deuxième, SEGMEXEC, n’introduit pas de surcharge réelle mais ne permet d’utiliser que 1,5 Go de l’espace d’adressage virtuel utilisateur. Depuis l’introduction des Pentium 4 5xxj (puis 6xx supportant l’EM64T ou x86_64) et des Athlons 64, les processeurs supportent nativement cette fonctionnalité (appelée souvent flag NX). Dans ce cas, PaX (PAGEEXEC) utilise la fonctionnalité native du processeur (ce qui malheureusement nécessite l’utilisation du mode PAE du processeur, fastidieux et plus lent).

PAGEEXEC

Nous décrivons rapidement le fonctionnement de PAGEEXEC, pour plus d’informations, le lecteur pourra se reporter à la documentation de PaX [PaX]. Lorsque le processeur ne supporte pas physiquement la possibilité de marquer une page comme étant non exécutable, PaX utilise la séparation des TLB introduite avec les processeurs Pentium pour émuler cette fonctionnalité. Les TLB (Translation Lookaside Buffers) sont des caches situés dans le processeur qui contiennent les entrées récemment utilisées des Page Directories et Page Tables afin d’accélérer la traduction d’adresses linéaires en adresses physiques. Dans les processeurs récents, les TLB sont séparés selon que l’adresse translatée est utilisée pour récupérer du code (instruction ~~fetch~~) (ITLB) ou pour écrire/lire en mémoire (DTLB). De plus, c’est au noyau d’invalider explicitement les TLB lorsqu’il modifie une Page Table ou un Page Directory. Cela signifie qu’il est possible pour le système d’exploitation de contrôler de manière fine les PTE contenus dans les TLB et en mémoire. Le fonctionnement de PAGEEXEC est le suivant :

- Les pages à rendre non exécutables sont marquées « superviseur » (flag U/S dans le PTE).
- Toute utilisation d’une telle page par le processus en mode utilisateur génère une faute. Le Page fault handler du noyau vérifie si cette faute est due à une tentative d’exécution de code (en comparant EIP du mode utilisateur et l’adresse de la page). Dans ce cas le processus est terminé (en réalité, les fonctionnalités EMUTRAMP et EMUSIGRT qui servent à émuler deux cas communs de génération de code à la volée, trampolines GCC et signal return du noyau sur la pile (vieilles libcs), compliquent ce processus).
- Dans le cas où l’accès demandé n’est pas en exécution, le PTE est modifié afin de permettre un accès à la page depuis le mode utilisateur, puis un accès est réalisé afin de charger le PTE correspondant dans le DTLB.
- Le drapeau ~~supervisor~~ est alors repositionné dans le PTE afin que tout accès en exécution soit à nouveau détecté.
- Lorsque l’exécution du processus en mode utilisateur reprend, tout accès en lecture/écriture utilisera le DTLB et ne provoquera pas de nouvelle faute alors qu’un accès en exécution passera par le PTE avec le drapeau superviseur.

Ce procédé est relativement coûteux en temps processeur. C’est pourquoi une autre méthode, SEGMEXEC, a été introduite en 2002. Cependant, fin 2004, PAGEEXEC a été amélioré dans le noyau 2.6. Il combine maintenant pagination et segmentation : une limite au segment de code est établie, au-dessus de la page exécutable la plus basse. De ce fait, toutes les pages situées au-dessus de cette limite ne sont pas exécutables grâce à la segmentation, ce qui a pour

conséquence de ne pas imposer une surcharge lors de l’utilisation de ces pages. Les pages non exécutables situées en dessous de cette limite sont traitées avec les TLB séparés, comme expliqué ci-dessus. D’un point de vue pratique, cette nouvelle méthode est relativement efficace car les « grosses » zones de données (le tas par exemple), sont situées au-dessus de la limite de cs. La surcharge n’existe que pour les zones de données qui n’ont pas pu être logées au-dessus de cette limite (par exemple les sections ~~.data~~ des bibliothèques). Des mesures de performances sont disponibles sur [paxperf].

SEGMEXEC

L’introduction de SEGMEXEC dans la deuxième moitié de 2002 a été un pas majeur pour PaX. Grâce à cette nouvelle méthode, les surcharges induites par PaX sont devenues négligeables, au prix cependant d’une complexité d’implémentation accrue. SEGMEXEC est une méthode ingénieuse permettant d’obtenir des pages non exécutables en utilisant la segmentation, grâce à une technique appelée le VMA mirroring. Le prix à payer est que la taille de l’espace des adresses logiques réellement utilisables par un processus en mode utilisateur passe de 3 Go à 1,5 Go. L’idée est de séparer complètement le segment de code utilisateur du segment de données utilisateur. Nous l’avons vu, en temps normal les segments utilisateurs ont une base de 0 et une limite de 4 Go et se chevauchent donc parfaitement dans l’espace d’adressage linéaire : une adresse 0xAABBCCDD utilisée pour lire/écrire des données ou pour exécuter du code se traduira invariablement par une adresse linéaire 0xAABBCCDD. Imaginons maintenant que le segment de code et le segment de données soient disjoints dans l’espace d’adressage linéaire de la manière suivante :

- Un descripteur de segment de données « User » (~~USER_DS~~) a pour adresse de base 0 et pour limite 1,5 Go, celui-ci est référencé par les sélecteurs chargés dans ds, es et ss
- Un descripteur de segment de code « User » (~~USER_CS~~) a pour adresse de base 0x60000000 (1,5 Go) et pour limite 1,5 Go, celui-ci est référencé par le sélecteur chargé dans le registre cs

Grâce à cette technique, une page située dans le segment de code (c’est-à-dire entre 1,5 et 3 Go dans l’espace d’adressage linéaire) sera exécutable (accessible depuis cs) mais pas lisible/inscriptible (rappelez-vous que les accès aux données se font par rapport aux sélecteurs ds, ss ou es), alors qu’une page située dans le segment de données (c’est-à-dire entre 0 et 1,5 Go dans l’espace d’adressage linéaire) ne sera pas exécutable.
Afin d’être certain que des pages situées dans le segment de données ne

puissent jamais être exécutées (le but étant par la suite de pouvoir prouver que l’injection de nouveau code exécutable dans l’espace d’adressage est impossible, cf. la partie dédiée à MPROTECT), il faut s’assurer qu’un autre sélecteur de segment moins restrictif ne puisse être chargé dans cs. C’est pourquoi PaX s’assure que la GDT utilisée par les processus utilisant SEGMEXEC possède un seul descripteur de segment de code en DPL3. En effet, il ne faut pas qu’un processus contrôlé par l’attaquant (par exemple à l’aide de return-to-libc chaînés) puisse être amené à charger dans cs un descripteur de segment lui permettant d’exécuter des pages contenues dans le segment de données (on pourrait imaginer par exemple que l’attaquant exécute un far return dont l’opcode était déjà présent dans la partie exécutable de l’espace d’adressage du processus).

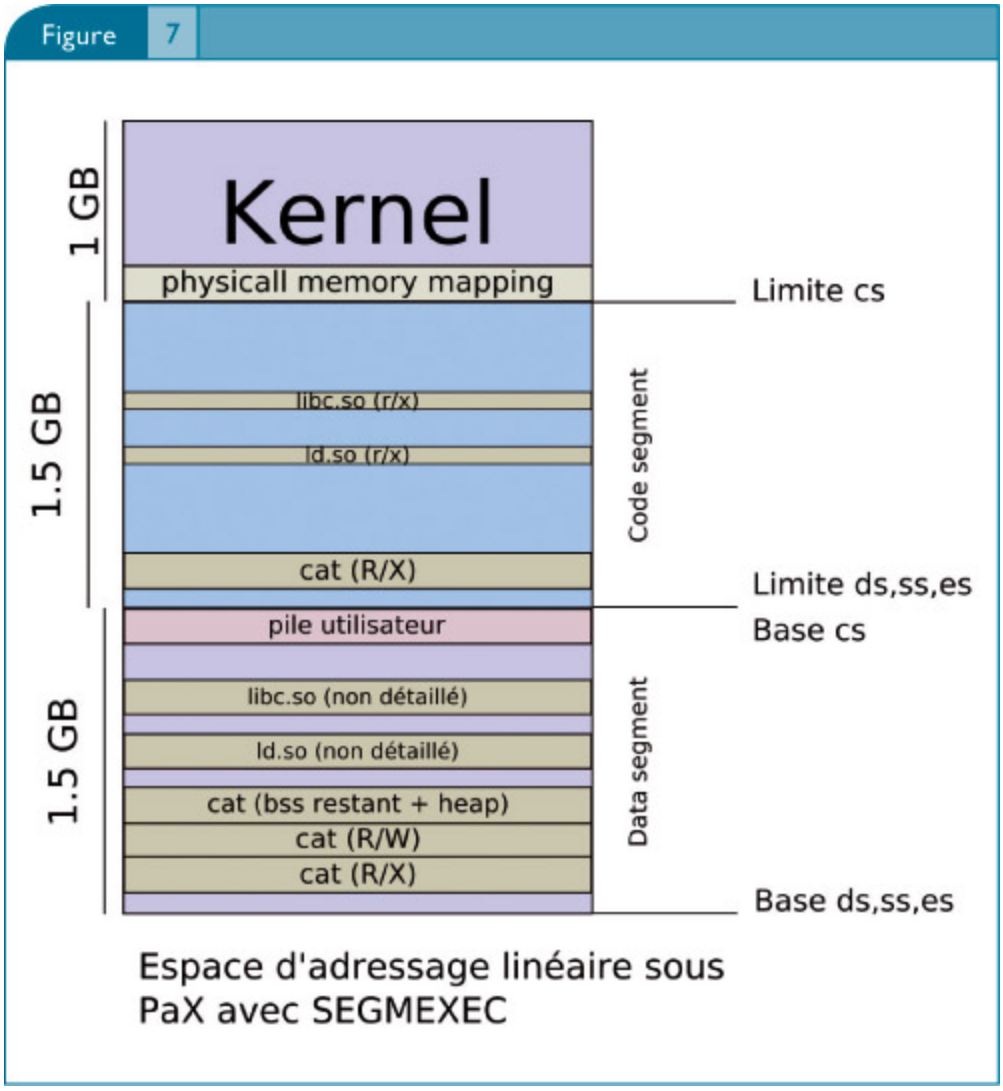
Avons-nous à ce stade réellement une sémantique de pages non exécutables ? Pas vraiment, car notre nouvelle protection PROT_ EXEC n’est pas ici combinable avec les protections PROT_WRITE et PROT_READ : il n’est pas possible de lire des données qui seraient situées dans une page exécutable. Cela est bloquant; si vous faites attention à la sortie de ~~readelf -l /bin/cat~~ ci dessus, vous verrez que la section ~~.rodata~~ (qui contient les données en lecture seule) par exemple se retrouve dans le segment (au sens ELF) de code. En pratique, il est, à de nombreuses occasions, nécessaire de pouvoir lire une page exécutable.

La solution à ce problème, le VMA mirroring, est ce qui rend SEGMEXEC complexe. Toute page disponible dans le segment de code doit être disponible également dans le segment de données. Pour cela, pour toute page présente dans le segment de code (adresses linéaires de 1,5 à 3 Go), un PTE « miroir » est ajouté dans le segment de données, référençant le même cadre de page que le PTE présent dans le segment de code (notez bien qu’il n’y a toujours qu’un cadre de page correspondant en mémoire physique !).
Notre espace d’adressage linéaire présente l’aspect suivant où l’on peut remarquer trois zones exécutables mirorées (voir également figure 7) :

```
08048000-0804c000 r-xp 00000000 03:03 32672 /bin/cat [1]
0804c000-0804d000 rw-p 00003000 03:03 32672 /bin/cat
0804d000-0806e000 rw-p 00000000 00:00 0
20000000-20016000 r-xp 00000000 03:03 179102 /lib/ld-2.3.2.so [2]
20016000-20017000 rw-p 00015000 03:03 179102 /lib/ld-2.3.2.so
20017000-20018000 rw-p 00000000 00:00 0
2001d000-20145000 r-xp 00000000 03:03 179607 /lib/libc-2.3.2.so [3]
20145000-2014d000 rw-p 00127000 03:03 179607 /lib/libc-2.3.2.so
2014d000-20150000 rw-p 00000000 00:00 0
20150000-2019f000 r--p 00000000 03:03 942989 /usr/lib/locale/locale-archive
5fffe000-60000000 rw-p 00000000 00:00 0 [pile]
68048000-6804c000 r-xp 00000000 03:03 32672 /bin/cat [1]
```

```
80000000-80016000 r-xp 00000000 03:03 179102 /lib/ld-2.3.2.so [2]
8001d000-80145000 r-xp 00000000 03:03 179607 /lib/libc-2.3.2.so [3]
```

De nombreuses modifications sont nécessaires afin que les deux pages « miroir » restent synchronisées en cas de changement d’état (lorsque le kernel gère un page fault, un ~~mremap()~~, un ~~mprotect()~~..), ce qui rend le VMA mirroring complexe.



Tous ces changements influent sur la segmentation et changent l’espace d’adressage linéaire.
Pour un processus, qui lui ne « voit » que l’espace d’adressage logique, rien n’a changé, si ce n’est que son espace d’adressage logique utilisable en mode User a maintenant une taille de 1,5 Go (en effet, les segments de code et de données ont tous les deux une taille de 1,5 Go et l’utilisation d’un offset supérieur à 1,5 Go produirait une exception de protection générale).

Le programme [DTDUMPER] af f iche précisément les descripteurs de

segments

```
DT Dumper
julien@cr0.org
GDT size: 0x7F (16 entries), GDT LA: 0xC03028D0
LDT’s selector is 0x68 (entry #13 in GDT)
TSS’s selector is 0x60 (entry #12 in GDT)
IDT size: 0x7FF (256 entries), IDT LA: 0xC0302000
CS: 0x23 DS: 0x2B SS: 0x2B ES: 0x2B
Dumping GDT (0xC03028D0)
[...]
(#004) 0x23 60C5FB000000FFFF Code D/B=32bts AVL=0 Present DPL=3 TYPE= _RA
BASE=0x60000000 LIMIT=0x5FFFFFFF
(#005) 0x2B 00C5F3000000FFFF Data D/B=32bts AVL=0 Present DPL=3 TYPE= _WA
BASE=0x00000000 LIMIT=0x5FFFFFFF
[...]
```

MPROTECT

Nous avons vu deux méthodes, PAGEEXEC et SEGMEXEC, permettant d’obtenir des pages non exécutables, même sur une architecture Intel ne les supportant pas (sur les architectures le supportant nativement, PAGEEXEC utilise bien sûr le support natif du processeur). Les restrictions MPROTECT de PaX permettent d’utiliser ces pages non exécutables afin de contribuer à rendre impossible l’injection de code arbitraire dans l’espace d’adressage d’un processus. Le noyau Linux maintient pour chaque zone de mémoire linéaire (VMA pour Virtual Memory Area) un ensemble de drapeaux qui déterminent si la zone est exécutable, inscriptible, potentiellement exécutable ou potentiellement inscriptible (c’est-à-dire après un appel à ~~mprotoct()~~): VM_EXEC, VM_WRITE, VM_MAYEXEC, VM_MAYWRITE.
Les restrictions MPROTECT font en sorte qu’un VMA ne possède jamais une combinaison d’un drapeau WRITE et d’un drapeau EXEC. Pour cela :

- Les mappings anonymes (pile et tas en particulier) ainsi que les mappings de mémoire partagée sont automatiquement marqués VM_WRITE|VM_MAYWRITE.
- Les mappings de fichiers sont marqués avec VM_WRITE|VM_MAYWRITE si la protection PROT_WRITE a été demandée lors de l’appel de ~~mmap()~~ et avec VM_EXEC|VM_MAYEXEC dans le cas contraire.

Une exception existe afin de gérer le cas des fichiers ELF (le plus souvent des bibliothèques) ayant des relocations dans le segment de code (entrée DT_TEXTREL dans la table dynamique) : le linker dynamique doit pouvoir changer les droits d’une page, afin d’effectuer une relocation, puis rechanger les droits. Cette exception peut être supprimée si l’on est certain d’avoir un système ne contenant aucun fichier ELF avec des relocations dans le segment (au sens ELF) exécutable. Nous reviendrons sur ce sujet dans la partie

suivante, consacrée à la mise aléatoire de l’espace d’adressage.
Les restrictions MPROTECT ne laissent qu’une possibilité à un attaquant pour introduire du nouveau code exécutable dans un processus dont il aurait pris le contrôle (en utilisant du code existant, à l’aide de return-to-libc enchaînés par exemple) : charger en mémoire à l’aide de mmap() un fichier dans lequel il a injecté du code, en demandant la protection PROT_EXEC.

C’est avec cette même méthode que les bibliothèques utilisées par un programme sont chargées en mémoire. C’est le rôle d’un système de contrôle d’accès (RSBAC, SELinux, partie RBAC de GrSecurity) de prévenir cette attaque.
Si un tel contrôle d’accès est correctement mis en place, et si l’exception liée aux relocations TEXTREL est supprimée, on peut avec PaX prouver qu’il est impossible d’injecter du code arbitraire dans l’espace d’adressage du processus et cela sans faire aucune hypothèse sur le degré de contrôle d’un attaquant sur le processus !
Même sans la restriction du mappage de fichier avec la protection PROT_EXEC, notons que l’attaquant doit pouvoir bénéficier d’un grand contrôle du processus vulnérable (en ne s’appuyant que sur du code déjà présent dans l’espace d’adressage !) pour pouvoir lui faire créer un fichier, lui faire mapper ce fichier avec la protection PROT_EXEC, puis lui faire exécuter son code (évidemment le fichier peut être créé indépendamment, si l’attaquant a déjà un compte local par exemple).
Les restrictions MPROTECT sont évidemment incompatibles avec les quelques programmes générant du code à la volée (par exemple la machine virtuelle JAVA). Pour pallier ce problème, PaX permet de marquer tout fichier exécutable afin de désactiver certaines fonctions, dont MPROTECT.

Address space layout randomization (ASLR)

Comme nous l’avons mentionné, nous nous limitons dans cet article aux techniques permettant à l’attaquant d’avoir un contrôle fort du flot d’exécution du processus. Le contrôle le plus fort est l’exécution de code arbitraire, nous avons vu dans la partie précédente que PaX, s’il est bien utilisé, le prévient totalement.
Cependant il est également possible d’obtenir un contrôle relativement puissant du processus en réutilisant du code existant et en détournant le flot d’exécution. Le degré de contrôle est toutefois sans commune mesure avec celui obtenu avec l’exécution de code arbitraire.
C’est pourquoi des exploits utilisent souvent ces méthodes pour ensuite réussir à exécuter du code arbitraire, par exemple en utilisant ~~mprotect()~~ pour rendre un buffer exécutable puis en retournant sur celui-ci.
Dans le cas de PaX, ceci n’est pas possible, mais il reste quand même

souhaitable d'empêcher le contrôle du processus à l'aide de code existant (sans compter que PaX n'est pas toujours utilisé de manière optimale et que les restrictions MPROTECT sont parfois retirées).
L'idée derrière la randomisation de l'espace d'adressage est d'empêcher toute adresse écrite en dur d'avoir un sens. Un exploit fiable doit au maximum éviter les adresses écrites en dur et utiliser des techniques plus avancées ou avoir recours à l'information leak, mais c'est loin d'être toujours possible. Pour une version compilée donnée d'un programme les adresses dans le fichier ELF (ET_EXEC) sont par exemple considérées comme stables. La randomisation de l'espace d'adressage utilisateur se décompose en trois parties :

- 1 Randomisation de la pile utilisateur (RANDUSTACK)
- 2 Randomisation des zones mmap()ées (sans adresse fixée) (RANDMMAP)
- 3 Randomisation de l'exécutable principal (RANDEXEC)

Les deux premières randomisations sont simples (voir [ASLR26] pour une implémentation d'ASLR pur). Pour RANDUSTACK, les bits 4 à 27 sont randomisés (alignement sur 16 octets). Cela est réalisé au niveau de deux fonctions de ~~fs/exec.c~~ responsables de la création de la pile lors de l'appel système ~~execve()~~. Pour RANDMMAP, la fonction ~~arch_get_unmapped_area()~~ (~~mm/mmap.c~~) est modifiée.
Celle-ci est responsable de la recherche d'une zone de mémoire libre lorsque le drapeau MAP_FIXED n'a pas été utilisé (c'est-àdire lorsque l'appelant ne souhaite pas spécifier d'adresse précise où sera créé son mapping, qu'il soit anonyme où d'un fichier). Les bits 12 à 27 sont ici randomisés (alignement sur une page). Grâce à RANDMMAP, les adresses des bibliothèques, du tas lorsque la libc utilise ~~mmap()~~ pour le gérer (rare en pratique), et des requêtes manuelles de mappage de fichier ou anonyme qui n'utilisent pas le drapeau MAP_FIXED (c'est presque toujours le cas) sont automatiquement aléatoires.
Cependant, les adresses de l'exécutable principal (ET_EXEC) ne sont pas randomisées par RANDMMAP. Comme le tas, lorsqu'il est géré avec ~~brk()~~ (ce qui est courant) débute dans la section ~~.bss~~, mappée avec l'exécutable principal, celui-ci n'est pas non plus randomisé (en fait une faible randomisation est tout de même ajoutée par PaX en « gâchant » volontairement de l'espace). Cela est évidemment très gênant, d'autant plus qu'un attaquant peut ainsi retrouver les adresses de fonctions dans les bibliothèques (comme ~~system()~~), en utilisant ~~dl-resolve()~~ et le mécanisme GOT/PLT [Wojtczuk] et passer ainsi outre la randomisation des bibliothèques.
Le problème avec l'exécutable principal est que celui-ci est placé en mémoire à partir d'un fichier ELF, au format ET_EXEC qui contient énormément d'adresses « en dur ». De plus ce fichier ne contient pas les informations de relocations nécessaires pour le reloger !
La solution consiste à générer des fichiers exécutables de type ET_DYN (type normalement réservé pour les bibliothèques). La PaX Team a introduit en 2003

une méthode pour générer de tels exécutables [ET_DYN]: en utilisant un crt1.o relogeable (utilisant la GOT), l'option -shared de GCC, et en spécifiant un lieur dynamique à utiliser, il est possible de générer un fichier exécutable ET_DYN. Notons que ceci n'était pas révolutionnaire, la libc est déjà un tel exécutable (vous pouvez essayer de l'exécuter !).
La génération de fichiers exécutables ET_DYN est également l'occasion d'éviter les relocations dans le code (cf. TEXTREL mentionnée dans la partie MPROTECT) en utilisant le switch -fPIC permettant d'obtenir du code indépendant de la position. Avec ces modifications, il est maintenant possible de randomiser l'exécutable principal : cela est fait dans la fonction load_elf_binary() du noyau et cela fait partie de l'option RANDMMAP de PaX.
Redhat s'est appuyé sur cette idée pour introduire en 2003 un switch -pie (position independant executable) à ld (et -fPIE, similaire à -fPIC, dans GCC), réalisant exactement les opérations décrites plus haut. Redhat a ensuite introduit le switch -z relro [RELRO] permettant de rassembler toutes les sections qui ne sont disponibles en écriture que pour des raisons de relocation et de créer un program header spécifique (PT_GNU_RELRO) afin que le lieur dynamique puisse s'il le souhaite réaliser un mprotect() sur cette zone après que les relocations aient été faites pour la mettre en lecture seule. Grâce à ce changement, on peut rendre la GOT disponible en lecture seule après le chargement du programme.

Lorsque notre système n'utilise que des exécutables de type ET_DYN, nous pouvons être dans une situation de full ASLR, où tout l'espace d'adressage du processus est aléatoire. C'est le cas dans les distributions [Adamantix], Hardened [Gentoo], et en partie dans les dernières Redhat. Il faut cependant garder à l'esprit plusieurs choses :

- Certaines erreurs de programmation permettent de faire du leak d'information, et de donner à l'attaquant des informations précieuses sur l'espace d'adressage, parfois des adresses complètes.
- Il est possible d'affaiblir la randomisation à l'aide d'un bourrage avec des instructions de type «nop». Cela est particulièrement réaliste sur la pile où des bits de poids faible (à partir du 5ième) sont aléatoires.
- Tout n'est ici qu'une question de probabilités : si l'attaquant a de la chance ou peut utiliser la force brute aussi longtemps qu'il le souhaite, il finira par réussir. Il faut absolument utiliser des techniques anti-bruteforce, telles que SegvGuard ou celle utilisée dans [GRSecurity].
- L'espérance mathématique lors d'une tentative de brute force est très différente selon que l'on crashe tout le démon (one-shot) ou seulement un fils. En effet, si les processus que l'on attaque sont tous issus d'un fork() du même démon, l'espace d'adressage est toujours semblable et nous en apprenons un peu plus après chaque tentative
- Lors d'une attaque locale il est facile d'obtenir des informations sur la

mémoire d'un processus, même sans privilèges (par exemple ~~/proc/pid/maps~~). Il faut utiliser un patch tel que [PAX+OBS] ou un système de contrôle d'accès pour l'éviter.

Notons pour être complet qu'il existe dans PaX une méthode, appelée RANDEXEC, ingénieuse, mais très coûteuse en performances, permettant de randomiser les exécutables de type ET_EXEC. L'idée repose sur le fait que beaucoup de branchements sont réalisés de manière relative et que, pour ceuxci, il est possible de reloger l'exécutable principal « tel quel ». L'exécutable est relogé, mais un miroir (utilisant le VMA-mirroring décrit dans la section SEGMEXEC) existe à l'adresse originale.

Ce miroir à l'adresse originale n'est pas exécutable (cette partie repose sur PAGEEXEC ou SEGMEXEC [notons que dans le cas de SEGMEXEC il n'y aura pas 3 miroirs comme on pourrait s'y attendre car on n'a pas besoin d'accéder aux données de manière relative, et on ne veut pas qu'il soit possible d'accéder au code de manière absolue]).
Les adresses absolues peuvent donc être utilisées pour écrire ou lire des données. Si une adresse absolue est utilisée pour exécuter du code, une exception se produira et PaX va utiliser une heuristique pour déterminer s'il s'agit d'une exécution légitime ou d'une attaque. Cette méthode est simplement une preuve de concept et ne doit pas être réellement utilisée à cause de la faiblesse de l'heuristique et des coûts en performances. Elle a été retirée dans les dernières version de PaX pour noyaux 2.6.
Le programme [PAXTEST] est capable de vérifier la randomisation par des tests statistiques. On retrouve des valeurs proches des valeurs théoriques (16 bits pour RANDMMAP et 24 pour RANDUSTACK), sauf pour le heap qui bénéficie d'un traitement spécial (randomisation par « perte de place ») :

```
Anonymous mapping randomisation test : 16 bits (guessed)
Heap randomisation test (ET_EXEC) : 13 bits (guessed)
Heap randomisation test (ET_DYN) : 25 bits (guessed)
Main executable randomisation (ET_EXEC) : No randomisation
Main executable randomisation (ET_DYN) : 17 bits (guessed)
Shared library randomisation test : 16 bits (guessed)
Stack randomisation test (SEGMEXEC) : 23 bits (guessed)
Stack randomisation test (PAGEEXEC) : 24 bits (guessed)
```

Exec-Shield

Exec-Shield a été annoncé en 2003 par Ingo Molnar, un « kernel hacker » réputé de chez Redhat. À son annonce, exec-shield était une évolution d'OpenWall, dont l'idée était d'avoir une limite dynamique (au lieu de statique) pour le segment de code, correspondant pour tout processus à l'adresse mémoire exécutable la plus haute de son espace d'adressage, ce qui permet,

contrairement à OpenWall, de rendre non exécutables d'autres zones que la pile.
De plus, Exec-Shield utilise également l'armure ASCII et y place les mappings PROT_EXEC (donc les bibliothèques), afin de rendre potentiellement plus difficiles certains return-to-libc et de concentrer le maximum de code exécutable le plus bas possible afin d'avoir le segment de code le plus petit possible. Voici les descripteurs de segment d'un processus [DTDUMPER] et son fichier maps (noyau 2.4), on peut voir la limite d'exécutabilité à 0x804A000:

```
DT Dumper
julien@cr0.org
(#004) 0x23 00C0FB0000008049 Code D/B=32bts AVL=0 Present DPL=3 TYPE= cRA
BASE=0x00000000 LIMIT=0x08049FFF
(#005) 0x2B 00CFF3000000FFFF Data D/B=32bts AVL=0 Present DPL=3 TYPE= eWA
BASE=0x00000000 LIMIT=0xFFFFFFFF
00c15000-00d3d000 r-xp 00000000 03:02 765565 /lib/libc-2.3.2.so
00d3d000-00d45000 rw-p 00127000 03:02 765565 /lib/libc-2.3.2.so
00d45000-00d48000 rw-p 00000000 00:00 0
00d5a000-00d70000 r-xp 00000000 03:02 765562 /lib/ld-2.3.2.so
00d70000-00d71000 rw-p 00015000 03:02 765562 /lib/ld-2.3.2.so
08048000-0804a000 r-xp 00000000 03:02 162889 /root/es/dtdumper
0804a000-0804b000 rw-p 00001000 03:02 162889 /root/es/dtdumper
09da3000-09dc4000 rw-p 00000000 00:00 0
200b3000-200b5000 rw-p 00000000 00:00 0
bffad000-c0000000 rw-p fffb8000 00:00 0
```

Depuis 2003, Exec-Shield a légèrement évolué. Outre quelques corrections de bugs exploitables, il a repris l'idée de randomisation utilisée dans PaX et tire en particulier profit des exécutables ET_ DYN générés à l'aide des switches ~~-fPIE~~ et ~~-pie~~ que Redhat utilise de plus en plus pour générer les exécutables « sensibles ». Il est intéressant de noter qu'Exec-Shield a été voulu comme un patch simple et léger, la sécurité n'étant pas la préoccupation fondamentale de son auteur. L'approche d'Exec-Shield a de nombreuses limites :

- On n'a pas de réelle sémantique des pages non exécutables, cela se traduit par un problème important : certaines zones de données seront situées sous la limite du segment de code et seront donc exécutables. En particulier, toutes les zones correspondant aux sections ~~.data~~ ou ~~.bss~~ des bibliothèques seront donc exécutables. Pour certains programmes, cela peut être bien pire si une zone exécutable est chargée à des adresses hautes pour une raison ou pour une autre.
- Exec-Shield a longtemps été incompatible avec le vDSO (~~linux-gate.so.1~~, le système introduit dans Linux 2.6 permettant de tirer parti des appels système rapides de Intel à l'aide de sysenter/sysexit) : en effet, il s'agit d'une zone exécutable située tout en haut de l'espace d'adressage, ce qui est incompatible avec une limite basse pour le segment de code.

Cependant, en juin 2005, Roland McGrath a proposé un patch grâce auquel il est possible de reloger le vDSO, qui est donc maintenant relogé et même randomisé dans l’ASCII armor comme une bibliothèque normale.
- Il n’y a pas d’équivalent aux restrictions MPROTECT de PaX : un attaquant ayant acquis un certain degré de contrôle du processus (par exemple à l’aide de returnto- libc enchaînés) peut faire un ~~mprotect()~~ afin de rendre la pile (et donc tout l’espace d’adressage car la pile est située en haut !) exécutable ! Contrairement à PaX qui distingue complètement l’ASLR de l’anti-injection de code exécutable, Exec-Shield repose indirectement sur l’ASLR pour rendre potentiellement plus difficile (et non pas impossible) l’injection de code exécutable.
- La randomisation dans Exec-Shield est plus faible que celle de PaX, notamment à cause du fait que les bibliothèques et l’exécutable principal doivent êtres « groupés » sous la limite du segment de code :

```
Anonymous mapping randomisation test : 8 bits (guessed)
Heap randomisation test (ET_EXEC) : 13 bits (guessed)
Heap randomisation test (ET_DYN) : 13 bits (guessed)
Main executable randomisation (ET_EXEC) : No randomisation
Main executable randomisation (ET_DYN) : 12 bits (guessed)
Shared library randomisation test : 12 bits (guessed)
Stack randomisation test : 19 bits (guessed)
```

La « fonctionnalité » la plus controversée introduite par Exec-Shield est sans nul doute PT_GNU_STACK. Les autres fonctionnalités s’inscrivent bien dans le modèle « faire le plus possible dans un patch simple et léger sans trop se soucier de la sécurité », mais PT_GNU_STACK introduit de réels problèmes et est un non-sens du point de vue de la sécurité. PT_GNU_STACK est un drapeau du program header permettant de marquer un fichier ELF (programme ou bibliothèque) comme « ayant besoin d’une pile exécutable ».
Ce marquage est réalisé de manière automatique par la toolchain (GCC, ld,...) en détectant des constructions ayant besoin d’une pile exécutable (en pratique, PT_GNU_STACK se limite aux trampolines GCC). Le lieur dynamique tire parti de cette information lors du chargement du programme principal ou d’une bibliothèque pour, si l’un au moins de ces composants réclame une pile exécutable, appeler la fonction ~~make_stack_executable()~~.
Le problème de cette approche est qu’il y a inévitablement des faux positifs et des faux négatifs. De plus, un lieur dynamique supportant cette fonctionnalité est incompatible avec PaX, puisque PaX verrouille à l’~~execve()~~ les « privilèges » (droit de mprotect PROT_EXEC par exemple) du processus, alors que PT_GNU_STACK est par essence dynamique.
Avec un lieur dynamique supportant PT_GNU_STACK, si une bibliothèque marquée comme utilisant une pile exécutable est chargée, la pile sera alors marquée exécutable (le processus et donc l’attaquant, a le droit de rendre la pile exécutable dans le modèle PT_GNU_STACK), PaX l’interdira, ce qui

conduira le lieur dynamique à abandonner le chargement.
Notons aussi que l'existence même d'une fonction «~~make_stack_executable()~~» est hallucinante [1] puisque cette fonction devient évidemment la cible privilégiée pour un attaquant capable d'appeler du code de son choix existant dans le processus : en effet, comme nous l'avons mentionné, rendre la pile exécutable sous exec-shield le désactive complètement. Nous voyons là encore qu'exec-shield repose complètement sur sa randomisation (dont la limitation à 12 bits pour les bibliothèques est ici particulièrement ennuyeuse).
Malgré ses nombreuses limitations, grâce à la notoriété d'Ingo Molnar, Exec-Shield fait peu à peu son entrée dans le noyau Linux. La version 2.6.12 du noyau a été la première à inclure une version allégée de la randomisation de l'espace d'adressage d'Exec-Shield (désactivable avec ~~/proc/sys/kernel/randomize_va_space~~). Voici le résultat de [paxtest] :

```
Anonymous mapping randomisation test : 8 bits (guessed)
Heap randomisation test (ET_EXEC) : No randomisation
Heap randomisation test (ET_DYN) : No randomisation
Main executable randomisation (ET_EXEC) : No randomisation
Main executable randomisation (ET_DYN) : No randomisation
Shared library randomisation test : 10 bits (guessed)
Stack randomisation test : 19 bits (guessed)
```

OpenBSD's W^X

W^X (W xor X) est apparu dans OpenBSD 3.3 en mai 2003. Il a pour but d'empêcher l'existence de pages à la fois disponibles en écriture et en exécution. Dans la version 3.3 d'OpenBSD, W^X ne fonctionnait que sur les processeurs supportant le marquage non exécutable d'une page. W^X pour i386 est apparu avec la version 3.4 d'OpenBSD en novembre 2003.
L'approche d'OpenBSD est de ne réaliser que des modifications simples dans le noyau et de réaliser le gros du travail en userland. Là encore, sur i386, la segmentation est utilisée : le segment de code a une limite à 512 Mo, ce qui signifie que toute adresse (dans l'espace logique) supérieure à 0x20000000 ne sera pas exécutable.
La première étape pour la réalisation de W^X (utile même sur les processeurs ayant un marquage NX) a été de s'arranger pour ne plus avoir besoin de zones à la fois exécutables et disponibles en écriture. Pour cela le trampoline ~~'sigreturn'~~ a été sorti de la pile (contrairement à ce qui a été fait sous Linux, il n'est cependant pas intégré dans la libc sous OpenBSD) et la GOT et la PLT ne sont plus disponibles en écriture (le linker met/retire les protection à l'exécution selon les besoins, cette approche est à mettre enparallèle avec celle de [RELRO]).

D'autres assainissements ont également été réalisés, par exemple les sections ~~.ctors~~ et ~~.dtors~~ qui contiennent des pointeurs ne sont plus disponibles en

écriture et la section ~~.rodata~~ n'est plus exécutable : vous pouvez consulter à ce sujet les slides de Theo [W^X].
La deuxième étape dans la réalisation de W^X sous i386 fut de s'arranger pour mapper toutes les zones de données au-dessus de la limite des 512 Mo et toutes les zones exécutables en dessous. De nombreux changements ont été réalisés au niveau du chargeur dynamique pour rendre possible ce mappage séparé, où les bibliothèques et l'ELF principal ne sont plus chargés d'un bloc.
OpenBSD utilise également un ASLR similaire à celui de PaX, mais plus limité. Voici le résultat de [PAXTEST] :

```
Anonymous mapping randomisation test : 20 bits (guessed)
Heap randomisation test (ET_EXEC) : No randomisation
Main executable randomisation (ET_EXEC) : No randomisation
Shared library randomisation test : 16 bits (guessed)
Stack randomisation test : 15 bits (guessed)
```

Comme Exec-Shield, OpenBSD n'offre pas d'équivalent aux restrictions MPROTECT de PaX. De ce fait, le processus (et donc potentiellement l'attaquant !) est autorisé à rendre une page exécutable. Nous avons également pu remarquer à l'aide de [DTDUMPER] que plusieurs entrées dans la LDT et la GDT sont des segments de code parfaitement valides pour exécuter du code n'importe où dans la partie utilisateur de l'espace d'adressage (les limites 0x1FFFFFFF correspondent à la limite de 512 Mo alors que les limites 0xCFBFDFFF couvrent tout l'espace utilisateur).
Cela constitue un énorme trou de sécurité puisqu'il suffit de réaliser un branchement inter-segment (far call, far jump, far ret,...) afin d'exécuter du code dans une zone censée être non exécutable ! Voici les segments de code utilisables :

```
DT Dumper
julien@cr0.org
GDT size: 0xFFFF (8192 entries), GDT LA: 0xE7B25000
LDT's selector is 0x18 (entry #3 in GDT)
TSS's selector is 0x168 (entry #45 in GDT)
IDT size: 0x7FF (256 entries), IDT LA: 0xD05CEF60
CS: 0x1F DS: 0x27 SS: 0x27 ES: 0x27
Dumping GDT (0xE7B25000)
[...]
(#004) 0x23 00CCFB000000FBFD Code D/B=32bts AVL=0 Present DPL=3 TYPE= cRA
BASE=0x00000000 LIMIT=0xCFBFDFFF
(#005) 0x2B 00C1FB000000FFFF Code D/B=32bts AVL=0 Present DPL=3 TYPE= cRA
BASE=0x00000000 LIMIT=0x1FFFFFFF
[...]
Dumping LDT (0xD05CEEC0)
(#002) 0x17 00CCFB000000FBFD Code D/B=32bts AVL=0 Present DPL=3 TYPE= cRA
BASE=0x00000000 LIMIT=0xCFBFDFFF
(#003) 0x1F 00C1FB000000FFFF Code D/B=32bts AVL=0 Present DPL=3 TYPE= cRA
BASE=0x00000000 LIMIT=0x1FFFFFFF
[...]
```

1 Comme dit mon ami Yoann Guillot, il manque give_me_a_root_shell().

Par exemple, dans le cas d’un débordement de tampon sur la pile, W^X ne sert absolument à rien car il suffit d’effectuer un far ret vers le segment de code illimité. Par chance (pour les pirates), l’opcode de far ret (0xCB ou 0xCA) ne prend qu’un octet, il est donc très simple de le trouver dans des zones exécutables à adresse fixe (afin de ne pas avoir à déjouer l’ASLR) : par exemple la section .text de l’exécutable principal (OpenBSD ne propose pas de système d’exécutables relogeables ET_DYN).

Il suffit donc en cas de stack overflow de préparer la pile : le descripteur 0x23 (ou 0x17), l’adresse de retour classique (qui peut être l’adresse du shellcode si on la connaît ou quelque chose de plus compliqué..), l’adresse d’un octet 0xCB. Le code ci-dessous [RETF_DEMO] simule cette situation (en construisant la pile à l’aide de ~~push~~ au lieu d’un overflow) avec un shellcode sur la pile et un retour sur un ~~jmp esp~~ :

```
; This would be in our target program
fret db 0xCB
runstack:
jmp esp ; obviously we execute code on the stack
; Entry point
global _start
_start:
; This is our payload
push 0xFEEB ; shellcode (this is jmp -2)
push 0x17 ; our segment selector
push runstack ; this is the classic return address
; don’t expect to have a jmp esp in real-life though ;)
push fret ; finding a static offset with 0xCB in standard ELF’s .text
; is very easy
; This is the standard ret after we control the stack
ret
```

Quand la fonction vulnérable retourne, l’exécution reprend sur un far ret, ce qui a pour conséquence de retourner sur l’adresse ~~runstack~~ tout en chargeant le sélecteur de segment sur la pile (0x17) dans CS. ~~runstack~~ peut être l’adresse de tout code menant à l’exécution de notre shellcode (par exemple directement l’adresse du shellcode, si on la connaît !). Notons toutefois que pour le cas précis des débordements sur la pile, un autre dispositif de prévention est utilisé dans OpenBSD : Propolice [SSP].

Comparaison

Nous avons vu plusieurs techniques qui essaient d’empêcher l’injection de code arbitraire dans un processus. Nous avons vu que parmi celles-ci, l’approche de PaX est la seule qui permette de réellement prouver qu’il n’est

pas possible d’injecter du code (en dehors du ~~mmap()~~ PROT_EXEC d’un fichier, à gérer via un système de contrôle d’accès).
L’approche de PaX consiste à ne pas faire d’hypothèse sur le degré de contrôle de l’attaquant et d’empêcher le processus lui-même d’injecter du nouveau code à l’exécution, c’est une application du principe de moindre privilège.
Dans PaX empêcher l’attaquant de détourner le flot d’exécution du programme se fait à l’aide de la randomisation qui est une fonctionnalité à différencier complètement de la partie anti-injection de code.
PaX est particulièrement efficace lorsqu’il est utilisé au sein du framework [GRSECURITY] qui apporte certaines des fonctionnalités manquantes : anti info-leak, anti bruteforce et système de TPE (Trusted Path Executable) ou RBAC (Role Based Access Control) pour lutter contre les ~~mmap()~~ PROT_EXEC de fichiers contrôlés par l’attaquant.
À l’opposé, OpenBSD et Exec-Shield laissent à l’application le privilège de pouvoir injecter du code exécutable et s’appuient sur la randomisation pour éviter le détournement du flot d’exécution du programme (qui une fois acquis par l’attaquant peut donc être utilisé pour injecter du code exécutable).
Cette approche est surtout efficace lorsque l’attaquant a un faible contrôle du flot d’exécution (par exemple en cas de heap overflow), mais est très limitée avec un contrôle plus fort permettant d’enchaîner les appels de fonctions (par exemple en cas de débordement sur la pile). Cette approche est également limitée par l’existence de potentiels bugs permettant l’information leaking (ou features, cf. ~~/proc/pid/maps~~ en local sous Linux) et oblige à se défendre contre le bruteforce pour rester efficace. En revanche, OpenBSD comme Exec-Shield y gagnent beaucoup en simplicité du code du noyau.

Autres considérations

Protection d’un noyau

Ce thème avait déjà été évoqué à une Rump Session de SSTIC [SECULINUX]. À l’heure actuelle, lorsqu’un pirate a déjà un accès local à une machine sous GNU/Linux et désire élever ses privilèges, les failles du noyau sont souvent la meilleure option. La plupart des binaires privilégiés sont largement audités et au fil du temps de mieux en mieux sécurisés.
Au contraire, le noyau Linux est en développement constant, le code est complexe et rarement audité et de nombreuses vulnérabilités sont présentes dans ses millions de lignes de code. D’autant plus qu’en mode noyau l’erreur ne pardonne vraiment pas : plus que jamais le moindre bug peut se traduire en faille de sécurité.
Pour ajouter encore à cela, le paradigme est complètement différent lorsque l’on essaie d’exploiter un bug du noyau : on possède déjà l’exécution de code

arbitraire en mode utilisateur et l’on veut simplement augmenter ses privilèges. Cela signifie que l’on peut créer un processus dont on contrôle parfaitement l’espace adressage en mode utilisateur et que l’on peut exécuter le code de notre choix pour créer la situation où la faille pourra être exploitée. Par exemple, les déréférences de pointeurs NULL qui sont, lorsqu’on exploite un programme en mode user, difficilement exploitables pour le commun des mortels [DELALLEAU] deviennent exploitables très facilement puisqu’on contrôle la partie basse de l’espace d’adressage.
Protéger le noyau avec des méthodes classiques telles que rendre la pile noyau non exécutable, rendre son adresse aléatoire (RANDKSTACK dans PaX) ou recompiler le noyau avec [SSP] (OpenBSD) ne sert en général à rien : presque toute erreur de programmation dans le noyau est une vulnérabilité et il est difficile de les classifier.
On voit finalement très peu d’erreurs classiques telles que les débordements de tampon. De plus comme le code du noyau est le plus privilégié, il n’est pas possible d’appliquer une politique de « moindre privilège » comme le fait PaX pour les processus en mode utilisateur : on ne peut pas empêcher un noyau compromis d’effectuer certaines opérations.
Il est donc actuellement très difficile d’empêcher un attaquant d’exploiter des failles dans son noyau. La seule méthode est sans doute de recourir à un système de contrôle d’accès ou un TPE (Trusted Path Execution) pour empêcher les démons et les utilisateurs locaux d’exécuter du code qui n’a pas été préalablement validé par l’administrateur.
Là encore le changement de paradigme est lourd de conséquences : on doit sans doute veiller à interdire l’utilisation d’interpréteurs (Perl, Python, Ruby...) qui permettent plus ou moins à leur utilisateur d’exécuter du code arbitraire (en tout cas d’avoir un grand degré de contrôle du processus de l’interpréteur). Il faut également voir que dans cette situation, tout exécutable autorisé par l’administrateur devient privilégié (puisqu’il est autorisé à exécuter du code), et devient donc une cible pour l’attaquant : un bug exploitable dans /bin/ls donne à l’attaquant l’occasion d’exécuter du code arbitraire et d’exploiter une faille noyau. Cette situation est délicate, car tous ces programmes n’ont pas été conçus pour être considérés comme privilégiés et sont très rarement audités.

Le cas de Windows

Il est difficile pour un éditeur différent de Microsoft de protéger l’espace d’adressage car cela nécessite des modifications au coeur du système. Microsoft a cependant, avec Windows XP SP2 et Windows server 2003, commencé à introduire quelques éléments de prévention générique. Outre les modifications au niveau du compilateur, proches de SSP/Propolice, on peut notamment citer l’utilisation du flag NX lorsqu’il est présent afin de rendre

certaines zones de données non exécutables. Il n’y a pas cependant sous Windows de mécanisme qui correspondrait aux restrictions ~~mprotoct()~~ de PaX. Cependant, on voit apparaître depuis environ un an des solutions HIPS. Si certaines comme Ozone ou Whentrust font de l’ASLR, la plupart des logiciels de « gros » éditeurs (CISCO CSA, McAfee Entercept) sont des systèmes de contrôle d’accès. La mise aléatoire de l’espace d’adressage souffre de plusieurs limitations importantes sous Windows : outre la possibilité pour un attaquant, commune à tous les OS, de réaliser de l’information leaking, il est très difficile de reloger certaines DLL, notamment ~~ntdll.dll~~ et ~~kernel32.dll~~ à cause du processus d’amorçage de Windows. De plus, les exécutables PE n’ont pas les informations de relocation nécessaires et ne peuvent donc pas être relogés. Dans ces conditions, l’attaquant possède au moins une source fiable d’opcodes à adresses fixes (~~kernel32~~ et ~~ntdll~~ étant susceptibles d’évoluer avec les services packs), et les exploits fiables sous Windows n’utilisent depuis longtemps pas directement l’adresse présumée de la pile : une technique courante en cas de stack overflow est d’écraser une structure ~~EXCEPTION_REGISTRATION~~ d’un Frame-Based exception Handler et d’écraser le pointeurvers le handler avec une adresse contenant un ~~jmp ebx~~ (depuis Windows XP SP1, EBX ne pointe plus vers la structure ~~EXCEPTION_REGISTRATION~~, cependant un pointeur est toujours présent sur la pile à esp+8, et on peut donc utiliser des instructions telles que ~~pop xxx~~; ~~pop yyy~~; ~~rot~~ présentes dans notre PE). Windows XP SP2 vérifiant maintenant que le handler appelé est enregistré (ou qu’il n’est pas dans un module PE avec un Load
Configuration directory) ces techniques évoluent de plus en plus. Les systèmes de contrôle d’accès (appelés souvent systèmes de détection comportementale sous Windows), sont utilisés de manière classique pour confiner un processus contrôlé par l’attaquant, mais aussi pour empêcher l’appel d’un NSS (Native System Service, les appels système sous Windows) ou d’une fonction d’une bibliothèque depuis des zones jugées anormales comme la pile.
Le paradigme dans lequel travaillent ces logiciels est que l’attaquant a déjà l’exécution de code arbitraire et qu’il faut essayer de le détecter le plus tôt possible. Pour cela, des hooks sont réalisés au niveau de certaines bibliothèques et des NSS et une analyse sont réalisés afin de déterminer si la situation est saine : par exemple ces produits peuvent vérifier que l’adresse de retour n’est pas située dans la pile.

L’exécution de code arbitraire, même sans pouvoir appeler de fonction de bibliothèques ou de NSS permet déjà de faire énormément de choses.

- On peut obtenir beaucoup d’informations sur l’espace d’adressage du processus exploité en utilisant le TEB et le PEB que l’on peut retrouver via le sélecteur de segment fs et un offset fixe (ce qui rend la randomisation du TEB/PEB (par rapport au segment ds) réalisée par

Windows XP SP2 inutile une fois l'exécution de code arbitraire acquise par l'attaquant). Ces informations peuvent ensuite être exploitées :

- On peut recopier son shellcode dans une zone mémoire jugée « saine » (en général, cela nécessite un peu de travail car une zone mémoire disponible en écriture ne devrait pas être jugée saine).
- On peut mettre à profit du code existant dans l'espace d'adressage afin qu'il réalise les appels de fonctions pour nous, tout en réalisant un « lifting » approprié de la pile pour simuler une situation normale. On peut, si l'on sait d'avance quel HIPS on veut contourner « suivre les hooks » pour retrouver la fonction originale. Ces techniques sont toujours utilisables, mais sont plus ou moins complexes à mettre en oeuvre selon la qualité de l'heuristique du produit à contourner. Il faut également noter que les hooks de bibliothèques en mode user peuvent être contournés en appelant directement les NSS (pour rester générique, on peut utiliser NTDLL.DLL pour en retrouver les numéros).

Remerciements
Pipacs et Bradley Spengler pour leur sympathie et pour contribuer grandement à la sécurité sous Linux. Laurent Butti, Yoann Guillot, Raphaël Rigo et Philippe Biondi pour leur relecture.

~~Retrouvez cet article dans : **Misc 23**~~

Posté par (La rédaction) | Signature : Julien Tinnès | Article paru dans


Laissez une réponse

Vous devez avoir ouvert une session pour écrire un commentaire.

« Précédent Aller au contenu »
Identifiez-vous
Inscription
S'abonner à UNIX Garden

Articles de 1ère page

- Git, les mains dans le cambouis
- PostgreSQL 8.3 : quoi de neuf ?
- Introduction à Ruby on Rails

- Linux Pratique HS N°17 - Mars/Avril 2009 - Chez votre marchand de journaux
- 88 miles à l'heure !
- Développement et mise en place d'un démon Unix
- Calculer ses rendus Blender en cluster ou comment faire sa propre « render farm » avec DrQueue
- Technologie rootkit sous Linux/Unix
- CMake : la relève dans la construction de projets
- Des petits sondages pour améliorer nos magazines

Actuellement en kiosque : 

Catégories

- Administration réseau
- Administration système
- Agenda-Interview
- Audio-vidéo
- Bureautique
- Comprendre
- Distribution
- Embarqué
- Environnement de bureau
- Graphisme
- Jeux
- Matériel
- News
- Programmation
- Réfléchir
- Sécurité
- Utilitaires
- Web

Articles secondaires

30/10/2008
Google Gears : les services de Google offline

Lancé à l’occasion du Google Developer Day 2007 (le 31 mai dernier), Google Gears est une extension open source pour Firefox et Internet Explorer permettant de continuer à accéder à des services et applications Google, même si l’on est déconnecté.…

Voir l'article...
7/8/2008
Trois questions à...

Alexis Nikichine, développeur chez IDM, la société qui a conçu l’interface et le moteur de recherche de l’EHM.…

Voir l'article...
11/7/2008
Protéger une page avec un mot de passe

En général, le problème n’est pas de protéger une page, mais de protéger le répertoire qui la contient. Avec Apache, vous pouvez mettre un fichier

~~.htaccess~~ dans le répertoire à protéger....

Voir l'article...
6/7/2008
hypermail : Conversion mbox vers HTML

Comment conserver tous vos échanges de mails, ou du moins, tous vos mails reçus depuis des années ? mbox, maildir, texte... les formats ne manquent pas. ...

Voir l'article...
6/7/2008
iozone3 : Benchmark de disque

En fonction de l'utilisation de votre système, et dans bien des cas, les performances des disques et des systèmes de fichiers sont très importantes....

Voir l'article...
1/7/2008
Augmentez le trafic sur votre blog !

Google Blog Search (http://blogsearch.google.fr/) est un moteur de recherche consacré aux blogs, l'un des nombreux services proposés par la célèbre firme californienne....

Voir l'article...

- **GNU/Linux Magazine**

  - GNU/Linux Magazine N°113 - Février 2009 - Chez votre marchand de journaux
  - Édito : GNU/Linux Magazine 113
  - Un petit sondage pour améliorer nos magazines
  - GNU/Linux Magazine HS N°40 - Janvier/Février 2009 - Chez votre marchand de journaux
  - Edito : GNU/Linux Magazine HS 40

- **GNU/Linux Pratique**

  - Linux Pratique HS N°17 - Mars/Avril 2009 - Chez votre marchand de journaux
  - Édito : Linux Pratique HS N°17
  - Linux Pratique HS 17 - Communiqué de presse

- Linux Pratique Essentiel N°6 - Février/Mars 2009 - Chez votre marchand de journaux
- Édito : Linux Pratique Essentiel N°6

MISC Magazine

- Un petit sondage pour améliorer nos magazines
- MISC N°41 : La cybercriminalité ...ou quand le net se met au crime organisé - Janvier/Février 2009 - Chez votre marchand de journaux
- Édito : Misc 41
- MISC 41 - Communiqué de presse
- Les Éditions Diamond adhèrent à l’APRIL !

© 2007 - 2009 UNIX Garden. Tous droits réservés .