Protection contre les injections de code

Mélanie Echeverria & Solenne Lefranc

3 décembre 2007

Table des matières

Introduction

Aujoud'hui l'intrusion sur un systéme local ou à distance reste dû en grandre parti à la vulnérabilité de type applicatif. Les buffers overflows sont principalement lié à cette vulnérabilité. Dans cet article, nous allons présenter des techniques de protection contre les injections de code c'est à dire contre l'exécution de code arbitraire sur une machine exécutant un programme. Dans un premier temps, nous feront quelques rappels de base sur les processeurs et la gestion de la mémoire sous Linux, puis nous présenterons les protections dite des "canaris" et des Nx bit. Nous commencerons par quelques rappels sur les buffers overflows afin de mieux comprendre comment un attaquant peut prendre possession d'un système.

1 Rappels de base

1.1 Les buffers Overflows

Le buffer Overflow est une technique qui peut être utilisée pour violer la politique de sécurité d'un système. Celle-ci consiste à exploiter une faille dans une application pour exécuter un code arbitraire qui compromettra la cible. Pour cela, il faut écrire dans un buffer[1] plus de données qu'il ne peut en contenir, afin d'écraser certaines parties du code de l'application et d'injecter des données permettant d'exploiter la faille.
Pour mieux comprendre, nous allons prendre un exemple. Le but de l'exemple suivant est de modifier la valeur affichée par la console lors de l'exécution du code :

bas de pile
main
i
foo
eip
ebp
buffer
haut de pile

```
#include < stdio.h>
  int foo(int a,char b,float c,double d){
      char buffer[8] ;
      int *ret ;
      ret=buffer+12 ;
      (*ret)+=0 ;
      return 0 ;
  }

  int main(){
      int i ;
      foo(1,'a',2.5,2.777777) ;
      i=1 ;
      printf("%d\n",i) ;
      return 0 ;
  }
```

Normalement ce programme affiche la valeur de *i*. Pour arriver à notre objectif, nous allons le faire en 2 étapes :

1. changer la valeur de *i* en modifiant l'adresse pointée par *buffer*.
2. changer l'adresse de retour de la fonction *"foo"* , afin de la positionner sur le *printf*

Le programme final sera :

```
#include < stdio.h>
  int foo(int a,char b,float c,double d){
      char buffer[8] ;
      int *ret ;
      /* 1ere étape :
      52 = différence entre l'adresse du buffer et celle de i
      ce qui positionne le buffer sur i*/
      ret=buffer+52 ; i
      (*ret)=5 ; //changement de la valeur de i
      /* 2eme étape :
      16 = différence entre l'adresse de retour de la fonction foo et celle du buffer
      7 = différence entre l'adresse de retour de la fonction foo et celle du printf */
      ret=buffer+16 ;
      (*ret)+=7 ;
      return 0 ;
  }

  int main(){
```

[1] Un buffer est une zone mémoire temporaire d'une application

```
    int i;
    foo(1,'a',2.5,2.777777);
    i=1;
    printf("%d\n",i);
    return 0;
}
```

Les buffers Overflows peuvent être un danger pour le système puisqu'un attaquant a alors la possibilité dans prendre le contrôle. Nous étudierons donc des techniques de protection.

1.2 Architectures des processeurs

1.2.1 Architecture Nermann

Cette architecture se divise en 4 parties matérielles :

1. unité arithmétique, logique et traitement
   son rôle est d'effectuer les opérations de base
2. unité de contrôle
   séquençage des opérations
3. mémoire
   - mémoire volatile
     programmes et données en fonctionnement
   - mémoire permanente
     programmes et données de base
4. entrée-sortie
   communication avec le monde extérieur

La séparation entre l'espace de stockage et le processeur est implicite.

1.2.2 Architecture Harvard

Cette architecture est l'opposé de celle vu précédement, la séparation est physique entre les données et la mémoire du programme.
L'accès à chacunes des 2 zones se fait par deux bus distincts. L'avantage de cette architecture est qu'elle peut effectuer simultanément des traitements sur les données et sur les instructions à exécuter.

1.2.3 Architecture x86

L'architecture x86 repose sur la limitation matérielle pour protéger les pages mémoires (non marquées en lecture seule donc exécutables). Si l'application est mal codée alors nous permettons de passer du code en exécution qui sera écrit dans ces pages, ainsi exécutables par diverses techniques (call, REL, sauts,...).

1.2.4 Architecture SPARC de Sun

Cette architecture n'est plus utilisée. Nous allons juste la présenter. Elle est organisée de la manière suivante :

1. d'une unité entière :
   Elle implémente 136 registres de 32 bits utilisés via des mécanismes de fenêtrage et lit une instruction dans le cache. Si celle-ci est flottante alors elle abandonne l'instruction.

2. d'une unité flottante :
   Elle implémente 32 registres de 32bits et comprend une pile permettant à l'instruction d'attendre la fin de son exécution.
3. d'un cache unifié d'instructions et de données
   Il est virtuellement adressé, possède une capacité de 64Ko. Le temps d'accès à celui-ci est de 1 cycle en cas de succès et 18 sinon.
4. d'une unité de gestion de la mémoire
   L'espace d'adressage de la mémoire est linéaire et paginé. La taille d'une page est de 4ko. L'architecture supporte jusqu'à 256 contextes[2] *simultanément.*

1.2.5 Architecture SPARC64

Cette architecture est constituée de 2 composants logiques et 4 composants mémoires qui sont responsables des fonctionnalités suivantes :

1. une unité centrale
2. 2 caches externes pour les instructions et les données
3. une unité de gestion de la mémoire

1.3 L'espace d'adressage sous linux

Il existe deux types de mémoire sous linux : la première est la mémoire réelle du système, appelée *mémoire physique*, et la seconde, *la mémoire virtuelle*. Cette dernière est divisée en 2 parties qui sont organisées :

- d'une partie pour l'espace utilisateur allant de l'adresse `0x00000000` jusqu'à `0xbfffffff`
- d'une partie pour le noyau allant de `0xc0000000` à `0xffffffff`

Chaque processus dispose d'une adresse virtuelle unique, chaque adresse virtuelle est formée de blocs d'octets contigus appelés *pages*. Une *région mémoire* est définie à chaque appel de fonction dans un processus. Elle contient les données nécessaires concernant les droits d'accès (lecture, écriture ou exécution), les protections de pages. Chaque caractéristique de ces régions peut être retrouvée grâce au fichier maps.

2 Protection via la méthode des "canaris"

Le principe de base des canaris est le plus communément utilisé aujourd'hui. L'erreur d'implémentation du code n'étant pas toujours prévisible, la méthode de protection des canaris est directement ajoutée lors de la compilation du code. Celle-ci se fait grâce à l'ajout de valeurs connues sur la pile entre un buffer et des paramètres du code pour prévenir des buffers overflows.

Le terme de canari fait référence à l'emploi des canaris dans les mines de Houille. En effet, les canaris ont la faculté de détecter les gaz toxiques plus tôt que les mineurs ; ils servaient ainsi d'alerte biologique.

[2] un contexte est un espace virtuel alloué à un processus

Il existe 2 types de canaris qui préviennent les buffers overflows : les canaris aléatoires et les canaris XOR-aléatoires.

2.1 Les canaris aléatoires

Le canari aléatoire génère un nombre aléatoire lors de l'initialisation du programme. Ce nombre aléatoire est stocké dans une variable globale qui n'est connue que par le code de vérification du canari. Afin d'affiner la sécurité, il est stocké entre des pages[3] de mémoire non allouées de telle façon qu'il soit impossible de localiser sa position en parcourant la mémoire au hasard.

Il est facile de déjouer le canari dans le cas où l'attaquant connaît sa position ou obtient les droits de lecture sur la pile.

2.2 Les canaris XOR-aléatoires

Le canari XOR-aléatoire est basé sur le même principe que le canari aléatoire à la différence que le nombre généré au début du programme dépendra également des variables du code à protéger. Si le canari ou les données à protéger sont modifiés alors la valeur du canari est erronée ; nous détectons aussitôt l'intrusion de code.

Il est plus difficile de le déjouer car l'attaquant doit connaître le canari, l'algorithme et les données protégées pour reproduire le canari attaquant sans être détecté.

2.3 Les implémentations

2.3.1 *StackGuard*

La première protection de pile utilisant les canaris a été *StackGuard*, en 1997. Il s'agit d'une extension à GCC. Cette protection se présente sous forme de *patch* mais n'a jamais été inclue dans le code du compilateur. Il a été inventé par Crispin Cowan et mis en application comme canari zéro pour GCC 2.7.2.2 par Aaron Grier puis vérifié par Peat Bakke. Perry Wagle en a assuré la maintenance pour le projet d'Immunix, et a implémenté les canaris de type Random et Random XOR. La version 3.x de GCC n'offre aucune protection de pile officielle et la protection adoptée ci-dessous a été adaptée pour GCC 4.1.

2.3.2 GCC Stack-Smashing Protection : *ProPolice*

Ecrit par Hiroaki Etoh d'IBM, le *Stack-Smashing Protector* ou SSP connu sous le nom de *ProPolice*, est une amélioration de *StackGuard*. Son nom se réfère directement au mot "propolis" qui est une résine utilisée par les abeilles afin de combler les trous et défendre la ruche en réduisant son entrée.

Elle est distribuée sous forme de patch pour GCC 3 et sous forme intégrée dans GCC 4. *ProPolice* diffère de *StackGuard* en plusieurs points, comme par

[3] Une page correspond à une partie mémoire sélectionnée

exemple, en protégeant tous les registres sauvés dans le prologue de la fonction et en réordonnant les variables de celle-ci.

Le patch *ProPolice* est activé dans *OpenBSD*, *Ubuntu* depuis la version 6.10, *DragonFlyBSD* et *IPCop Linux*. Il est également disponible dans *NetBSD*, *Debian* et *Gentoo* mais est désactivé par défaut.

ProPolice est activé par défaut sous *OpenBSD*, et peut être désactivé par le flag **-fno-stack-protector**. La SSP peut être activée par défaut en ajoutant le flag de **-fstack-protector** pour ce qui concerne la protection de string, et **-fstack-protector-all** pour tout autre type de protection.

2.3.3 StackGhost (protection hardware)

Ce patch a été écrit par Mike Frantzen pour le noyau OpenBSD et plus particulièrement pour les architectures SPARC et SPARC64 de Sun MicroSystem. Il détecte les modifications des pointeurs de retour (c.f : 2.4) et protège d'une manière automatique et transparente toutes les applications. Une perte de temps d'exécution est à prendre en compte.

La technique de StackGhost se représente sous 3 parties :
- sauvegarder les registres et l'adresse de la valeur de retour avant l'appel de fonction.
- lancer l'appel de fonction
- restaurer les registres après l'appel de fonction

Lors de la regénération des registres, le processeur crée un piège dans le noyau. Celui-ci sauve les anciens fenêtres de registres[4] dans la pile utilisateur.

Ensuite vient la restauration des registres, si la fenêtre est sauvée dans la pile alors le processeur récupère la fenêtre. Il initialise le piège puis remet la fenêtre sauvegardée dans le registre courant. Le piège opère donc avant et après l'appel de fonction.

La protection de StackGhost utilise la méthode du XOR cookie. Celle-ci consiste à "prendre un empreinte" de l'adresse de la valeur de retour avant la régénération et vérifie que c'est la même après. Si ce n'est pas le cas alors le canari bloque la regénération.

2.4 Les limites de la protection

Même si les protections par canaris sont difficiles à contourner, elles n'évitent pas pour autant certaines attaques. Elles ne peuvent, par exemple, pas protéger des buffers overflows dans le tas. De plus si la fonction utilise un pointeur sur fonction en variable locale, ce pointeur ne sera pas protégé contre un éventuel écrasement (cf : Fig. 1).

Une solution est de placer les buffers potentiellement dangereux juste au dessus du canari, avec les autres données ainsi elles ne pourront être modifiées.

[4] *une fenêtre de registre* est une ensemble de registre créé pendant l'appel de focntion. Le processeur a un nombre limite de fenêtre s'élévant à 6 ou 7 fenêtres.

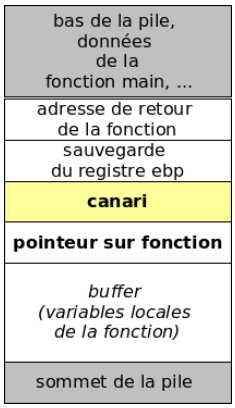

FIG. 1 – Exemple de pointeur sur fonction non protégé

Propolice et visual C++ réorganisent la pile autant que possible mais ne peuvent pas réorganiser les structures, l'ordre des membres étant commun à tout le programme. Le programmeur devra alors tenir compte des risques en créant des variables.

Enfin, ces techniques sont connues des experts et rendent difficile l'exploitation de la faille mais ne l'empêchent pas pour autant car il existe de nombreuses techniques de contournement.

3 NX bit

Cette technologie utilisée par les architectures Neuman et Harvard, permettent de séparer les différents types de données stockées (les jeux d'instruction processeur, les codes, les données) en isolant certains secteurs de la mémoire. Si la mémoire possède l'attribut Nx alors il sera impossible de lancer le processus. Elle empêche certains exploits de sécurité tels que les débordements de tampon.

Le Nx bit se référe au 63^{eme} bit sur une entrée de 64 bits dans le tableau de page[5], si celui-ci est positionné à 1 aucune exécution n'est possible. Inversement, s'il est positionné à 0.

Dans les anciennes technologies matérielles (comme les anciens X86), des logiciels (Exec Shield) existaient pour émuler le principe du Nx bit. Si un système d'exploitation était muni d'un de ces logiciels, il avait la capacité d'empêcher l'injection et l'exécution de code comme par exemple le ver Sasser[6].

[5]Structure de données employée par un système d'exploitation pour faire le lien entre l'adresse virtuelle (unique pour les processus) et l'adresse physique.

[6]attaque le service LSASS - *Local Sécurity Autority Subsystem Servive* qui est fondée sur une certaine partie mémoire pour rendre le code affichable ou exécutable.

3.1 La protection de l'espace d'adressage

Cette technologie utilise la technique de la protection de l'espace d'adressage -*" executable space protection "*. Elle permet de marquer une zone mémoire comme non exécutable : si un utilisateur (bienveillant ou non) tente d'exécuter un code dans cette zone mémoire alors une erreur se produira.

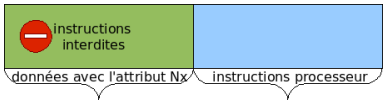

FIG. 2 – Principe de la protection de l'espace d'adressage

3.2 Implémentations du Nx bits dans les systèmes d'exploitation

- Mac os : implémentation du Nx bits dans l'unité centrale soutenue pas Apple

- Linux : les distributions Ubuntu, OpenSuse, ..., ne permettent pas l'option HIGHMEM64, qui est exigée pour accéder au NX bit en mode à 32 bits, dans le noyau par défaut. Le mode PAE nécessaire au Nx bit n'est pas compatible lors de l'initialisation avec le pré-Pentium Pro (qui inclut Pentium MMX) et les processeurs Pentium M.

- solaris : la protection de NX est automatiquement permise par défaut sur les processeurs x86 qui supportent ce dispositif.

- windows : sur XP Service Pack 2, Server 2003 SP1 et Windows Vista, il est possible utiliser la technologie Nx bit.

3.3 OpenWall non exécutable

Ce patch est le premier patch de linux, développé par Sun Designer en 1998, permettant de rendre difficile l'injection de code mais fournit également d'autres protections comme par exemple limiter l'accès d'un utilisateur uniquement à ses propres processus dans */proc*.

La technique Openwall pour rendre la pile non exécutable est de poser une limite au descripteur de segment, référencé par le registre cs[7]. Cette limite est inférieure à l'adresse linéaire de la pile utilisateur. Il n'y a pas de changement pour les autres descripteurs de segment.

Dans le cas où le processeur essaie d'exécuter du code sur la pile, la limite posée est dépassée. En effet, lors d'une exécution de code le registre cs est utilisé et le descripteur de segment est référencé par celui-ci. Ce dépassement provoque

[7] registre qui contient le numéro de segment.

une exception et tue le processus. Dans l'autre cas, les registres utilisés sont différents du registre cs et il n'y a pas de changement dans le déroulement du processus.

Il existe des limites à cette protection notamment il est possible d'injecter du code dans la pile. Grâce à la technique de l'armure ACSII, Solar Designer a voulu remédier à ces limites.

Afin d'empêcher l'injection d'une adresse de retour cible via les chaines de caractères, l'armure ACSII fait en sorte que les adresses des bibliothèques aient toujours un zéro. Cette méthode est efficace seulement s'il est difficile pour l'attaquant d'injecter des zéros.

Il existe maintenant des méthodes plus efficace que nous présenterons par la suite.

3.4 Exec-shield

3.4.1 Présentation

Ce patch de sécurité développé par Ingo Molnar utilise la fonctionnalité du Nx bit sur les processeurs x86 afin de protéger des buffers overflows et des heaps overflows. Un seul bit permet de déterminer si la permission est d'écriture ou d'exécution, ce qui rend le contrôle d'exécution impossible. Le patch Exec-shield garde une zone de la pile pour l'exécution. Cette zone va de 0 à l'adresse la plus élevée possible. Les demandes de mémoire exécutable se font par l'appel de fonctions comme mmap[8] ou encore mprotect[9].

En plus d'émuler le NX bit, Exec shield utilise l'armure ASCII pour y placer le bit PROT_EXEC. Tout cela pour maintenir les données exécutables le plus bas possible dans la pile afin d'avoir une zone exécutable la plus petite possible et pour rendre difficile *l'attaque return-to-libc*. En appelant la fonction mprotect avec l'argument PROT_EXEC qui permet d'exécuter le code en zone mémoire, l'intrus peut facilement exécuter du code sur une autre machine. C'est que ce Exec-shield essaie d'empêcher.

3.4.2 Utilisation

Tout code a besoin de savoir s'il est correctement exécutable ou non, c'est la compilation avec GNU et l'éditeur de lien qui le permet. Des commandes existent afin de le savoir, par exemple pour la commande cp :

```
readelf -l /bin/cp | fgrep STACK
GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4
```

Cette commande nous permet de voir que la commande cp n'a pas besoin de pile exécutable (de la 2^{eme} à la dernière colonne), l'avant dernière colonne nous

[8]La fonction mmap demande la projection en mémoire d'une taille en octets donnée. Cette adresse n'est qu'une préférence, généralement 0. La véritable adresse où l'objet est projeté est renvoyée par la fonction mmap.
Elle établit ou supprime une projection en mémoire des fichiers ou des périphériques.

[9]mprotect, contrôle les autorisations d'accès à une partie de la mémoire.

indique spécifiquement qu'elle a besoin que la pile soit en lecture /écriture(RW). Si nous avions trouvé RWX alors la commande ou le code aurait besoin d'une pile exécutable. Une fois le binaire créé, il est possible de rendre la pile exécutable ou non de la façon suivante :

execstack -s *nomDeLObjet*

Cette commande permet d'ajouter un flag *PT_GNU_STACK* dans l'en-tête du programme qui décrit si la pile a besoin d'être exécutable ou non. Grâce au compilateur, par exempe GCC, la détection de la nécessité d'une pile exécutable est faite automatiquement.

La configuration de Exec-shield se fait via les fichiers *"/proc/sys/kernel/exec-shield"* en écrivant 0, 1 ou 2 à l'intérieur :

0 Exec Shield est totalement désactivé.

1 Le noyau suit les intructions du flag PT_GNU_STACK lorsqu'il vérifie les permissions. Si le binaire vérifié n'a pas ce flag alors la pile exécutable est créée.

2 Cette option est quasiment identique à la précédente à l'exception du fait que tous les binaires qui n'ont pas le flag PT_GNU_STACK seront exécutés en dehors de la pile.

Cependant des imperfections existent dans Exec-shield : l'une d'elle est rectifiable via la méthode de randomisation de l'espace adressage. En effet, si à chaque chargement d'un processus, l'adresse de la pile se fait toujours sur la même page alors cela est une faille puisqu'il est possible de la trouver. En utilisant la randomisation de l'espace d'adressage, nous pouvons réduire cette possibilité. C'est grâce au fichier *"/proc/sys/kernel/exec-shield-randomize"* qui permet d'activer ou de désactiver la randomisation de la stack.

3.5 Pax

Pax émule la technologie Nx bit. Il se présente sous la forme un patch pour le noyau linux qui évolue depuis sa création en 2000. Au départ, Pax est plus la preuve du concept plutôt qu'un patch de sécurité. Son évolution s'est faite en introduisant de nouvelles fonctionnalités comme l'ASLR ("Address Space Layout Randomization") ou encore le VMA mirroring.
Contrairement aux canaris, Pax n'empêche pas directement les débordements de tampon mais évite l'exécution de code injecté par ceux-ci. Utiliser la méthode des canaris (par exemple avec StackGuard) et simultanément Pax est un risque engendrant le blocage du système d'exploitation.
Le patch Pax offre les protections :

- rendre les pages mémoires non exécutables
- interdire la modification des permissions des pages mémoires
- rendre aléatoire les adresses à chaque exécution d'un programme

3.5.1 Protection de l'espace d'adressage

Comme nous l'avons vu dans le rappel sur l'espace d'adressage (cf 1.3), les régions mémoire ont trois types de permission possibles. Or les processeurs

IA-32 n'ont que deux protections possibles (read ou write et utilisateur ou super-utilisateur). Cette différence va poser problème puisqu'il est alors impossible de séparer les droits des pages en lecture/écriture des droits des pages en exécution. Le rôle de la protection de l'espace d'adressage de Pax est de remédier à ce problème.

Grâce au flag PROT_EXEC que l'on définit dans les permissions de la page, il est impossible d'avoir les permissions d'écriture et d'exécution simultanément. Cette restriction se fait via la fonction mprotect(). Si la restriction mprotect() de Pax est activé, il n'est théoriquement pas possible d'exécuter des pages mémoire.

Il existe d'autres techniques permettant de mettre en oeuvre la fonctionnalité des pages non-exécutables :
- PAGEEXEC
- SEGMEXEC

Dans le cas où le processeur ne supporte pas de marquer une page comme non exécutable, PaX utilise la séparation des TLB[10]. Les PAGEEXEC fonctionnent de la manière suivante :
- un flag superviseur est mis sur les pages non-exécutables.
- si une page non-exécutable est utilisée en mode utilisateur, alors PaX termine le processus.
- l'utilisateur a la permission d'accéder à la page si celle-ci est demandée sans être exécutée.

Actuellement PAGEEXEC utilise deux techniques, la pagination et la segmentation. Cela permet une plus grande efficacité au niveau de la gestion des zones mémoire.

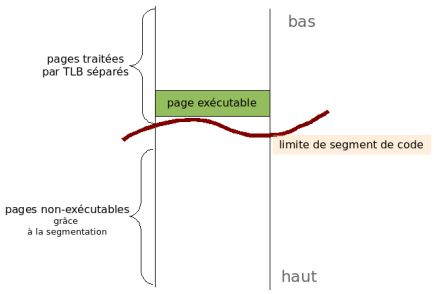

FIG. 3 – Combinaison de la pagination et la segmentation dans PAGEEXEC

[10] *Translation Lookaside Buffers* : référence des adresses physiques dans un tableau ce qui accélère la traduction adresses linéaires en adresses physiques.

Pour ce qui concerne SEGMEXEC, elle s'appuie sur le *VMA mirroring* et permet d'obtenir des pages non exécutables grâce à la segmentation. Le principe de base est de séparer les zones d'écriture des codes et des librairies. Chacune des zones est respectivement appelée segment de données et segment de code. *VMA mirroring* permet d'accéder aux pages de segment de code dans les segments de données. Ceci rend exécutables les pages contenues dans les segments de code contrairement aux pages segment de données.
Au final, nous avons une pile non-exécutable mais le prix à payer est le passage de 1,5 Go à 3 Go pour la taille d'un processus dans l'espace d'adressage. De plus son implémentation reste assez complexe.

3.5.2 L'ASLR - "Address Space Layout Randomization"

Le principe de l'ASLR est simple : il suffit de randomiser la pile et le reste du code en mémoire afin que l'attaquant ne puisse localiser l'adresse nécessaire à l'exécution du code malveillant. Appliquer cette technique reste assez coûteux en terme de performances.
La randomisation de l'espace d'adressage se décompose en 3 parties :

1. **randomisation de la pile utilisateur** :
   Les bits 4 à 27 sont randomisés par RANDUSTACK grâce à la fonction de *fs/exec.c* qui est responsable de la création des piles lors de l'appel système *execve()*.
2. **randomisation des zones mappés (sans adresse fixe)** :
   La fonction RANDMMAP permet de rendre aléatoire les adresses des bibliothèques, du tas (seulement si *libc* utilise la fonction *mmap()*), des requêtes manuelles de mappage. La fonction *arch_get_unmapped_area()* (de *mm/mmap.c*) permet de rechercher une zone mémoire libre et RANDMMAP randomise les bits 12 à 27. Pour plus de précisions, si le tas est géré par *mmap()*, c'est alors une randomisation normale. Dans l'autre cas il est géré par *brk()*, et Pax fournit une faible randomisation du tas.
3. **randomisation de l'exécutable principal** :
   C'est une méthode très coûteuse qui randomise les exécutables de type ET_EXEC. Cette randomisation était utilisée seulement par PaX, mais a été retirée pour le noyau Linux 2.6.

3.5.3 Utilisation

Le patch Pax est disponible sur le site officiel de Pax[11]. Voici les options que l'on peut configurer en fonction de la sécurité que l'on veut apporter à notre système :

- Enforce non-executable pages
- Paging based non-exucatable pages
- Address Space Layout Randomization
- Randomize kernel stack base
- Randomize user stack base
- Randomize mmap() base

[11] http ://pax.grsecurity.net/

Grâce à toutes les techniques employées par Pax, il n'est pas possible de contourner les barrières contre l'injection de code. Pax reste le patch le plus sûr contre l'injection de code en empêchant l'attaquant de détourner l'exécution d'un programme. Si nous ajoutons au patch d'autres techniques de sécurité contenu dans le path *grsecurity* alors nous augmentons encore plus la sécurité de notre système.

Conclusion

Nous avons donc présenté deux techniques de protection contre des injections de code. Pour le cas des canaris, le protection est faite pour détecter l'injection de code à la base, et pour le cas du Nx bits de protéger contre les conséquences des buffers overflows : l'exécution de code arbitraire. Mais il apparaît que cette dernière protection est beaucoup plus sûre, la protection par canaris étant contournable.

Références

[1] *DLFP :Exec Shield : protection contre les débordements de tampons.* `http://linuxfr.org/2003/05/04/12324.html`, November 2007.

[2] *Openwall Project - Information Security sofware for open environments.* `http://www.openwall.com`, November 2007.

[3] *Pax.* `http://en.wikipedia.org/wiki/Pax`, November 2007.

[4] *Stack-smashing protection.* `http://en.wikipedia.org/wiki/Stack-smashing_protection`, September 2007.

[5] Vianney Devreese, Iannis Formet, Jeremie Baldy, and Lionel Flandrin. *Les protections contre les attaques par corruption de la pile.* May 2007.

[6] Samuel Dralet. *Exec-Shield ... Il y a des jours comme ça.* `http://www.lexfo.fr/blog/index.php/2007/06/16/21-exec-shield-il-y-a-des-jours-comme-ca`, June 2007.

[7] Ulrich Drepper. *Security Enhancement in Red Hat Enterprise Linux.* September 2005.

[8] Denis Ducamp. *Durcissement du noyau Linux.* `http://www.hsc.fr/ressources/presentations/durcissement`, 2001.

[9] Frédéric Raynal and Samuel Dralet. *Protections contre l'exploitation des débordements de buffers - Les patches kernel.* `http://www.cgsecurity.org/Articles/2-MISC/Protections-2/index.html`, June 2003.

[10] Julien Tinnès. *Protection de l'espace d'adressage : état de l'art sous Linux et OpenBSD. Misc*, (23), January 2006.