Sujet informatique embarquée – examen M1

J.-M Friedt, 21 juin 2016

Tous les documents sont autorisés, aucune connexion numérique vers le réseau, radiofréquence ou filaire, n'est autorisée.

La sécurité d'un système Unix, et de Linux en particulier, tient en la séparation stricte des accès aux ressources entre les divers utilisateurs et l'administrateur : il s'agit d'une règle de conception fondamentale quelquesoit l'architecture matérielle sur laquelle s'exécute le noyau. Nous avons vu que pour accéder au matériel, le noyau a tous les droits, encore supérieurs à ceux de l'administrateur qui ne peut qu'émettre des commandes agissant depuis l'espace utilisateur : par conséquent, un pilote modifiant les fonctionnalités du noyau donne à un utilisateur mal intentionné des pouvoir supérieurs à ceux de l'administrateur, pour peu qu'il arrive à insérer son module noyau. À titre d'exemple, les *rootkit* font une utilisation intensive des modifications d'appels systèmes transitant par le noyau pour cacher leur activité, à l'insu de l'administrateur. Nous nous proposons d'explorer une approche fonctionnelle sur noyaux 3.x et 4.x de Linux pour rediriger des fonctions sollicitées lors des appels systèmes.

Afin de modifier le comportement d'un appel système, deux approches sont possibles : soit le contenu de la fonction est écrasée avec le nouvelle séquence d'instructions, soit l'adresse de la fonction appelée est modifiée pour pointer vers la nouvelle séquence d'instructions. La première approche a longuement été utilisée dans l'implémentation des virus sous MS-DOS – à une époque où aucun superviseur (noyau) n'implémentait d'appels systèmes logiciels – tandis que la seconde est utilisée pour rediriger les appels systèmes sous GNU/Linux par exemple dans les outils visant à cacher à l'administrateur certaines fonctions de son système d'exploitation corrompu par un utilisateur qui désire rester invisible.

Lors de la compilation *sans optimisation* d'un code en C, la séquence d'instructions en assembleur suit de façon séquentielle la séquence des instructions en C.

1. Proposer un petit (3 lignes doivent suffire) programme qui contient une fonction `toto()` et dont la fonction principale affiche l'*emplacement* de la fonction `toto()` en mémoire. Quelle méthode du langage C indique l'emplacement en mémoire d'une structure ?
2. Proposer une méthode – sous certaines hypothèses de comportement du compilateur en l'absence d'optimisation que nous vérifierons – pour trouver la longueur de la fonction (*i.e.* la taille mémoire occupée par la fonction).
3. Démontrer votre capacité à écraser le contenu de la fonction `toto` par le contenu de la fonction `tata()` qu'on s'efforcera de construire moins complexe (nécessitant moins d'opcodes) que `toto()`. À titre d'exemple, `toto()` pourrait incrémenter de une unité une variable passée en argument, tandis que `tata()` incrémenterait de deux unités. Démontrer le bon fonctionnement après compilation sous GNU/Linux et exécution en espace utilisateur.
4. On notera que si on n'y prend pas soin, une page mémoire contenant un code exécutable n'est pas en écriture, et le code de la question précédente se traduit par une erreur de segmentation : il faut pour ce faire
   `mprotect((void*)page_debut),longueur,PROT_EXEC|PROT_READ|PROT_WRITE);`
   Comment calculer l'adresse de début de la page contenant l'adresse de la fonction, calculée à la première question, connaissant la taille de la page mémoire `int pagesize=sysconf(_SC_PAGE_SIZE);`.
5. Imaginer un mécanisme permettant d'écraser une séquence d'instructions, voir une fonction, dans un programme interagissant avec un utilisateur (donc dans lequel nous n'avons pas accès au code source mais uniquement la capacité à rentrer une réponse à une requête du programme, par exemple par `scanf()`).

On notera que cette dernière attaque a été tellement utilisée que Linux implémente maintenant une protection qui interdit d'exécuter le contenu de la pile : afin de démontrer expérimentalement ce dernier concept (que nous ne demandons pas ici), il faut penser à rendre la pile exécutable par `mprotect(page_debut,longueur,PROT_EXEC|PROT_READ|PROT_WRITE);`

Le comportement d'un appel système est similaire à celui que nous avons vu lorsqu'un appel POSIX est transmis à un pilote pour faire appel aux entrées de la structure de données `file_operations` : un pointeur de fonction indique quelle fonction sera appelée pour répondre à la requête. L'ensemble des appels systèmes accessibles depuis l'espace utilisateur est décrit dans `/usr/include/i386-linux-gnu/bits/syscall.h`. Ainsi, contrairement au cas des modules noyau qui fournissent chacun leur structure de donnée pour leurs appels de fonctions, le noyau Linux propose une table qui inclut tous les appels de fonctions : cette table est décrite dans `/usr/include/asm-generic/unistd.h`

6. en observant la séquence des appels systèmes en préfixant une commande `mkdir` par l'instruction `strace`, quel est l'appel système spécifique à cette opération qui est sollicité lors de la création d'un répertoire sous GNU/Linux ? Quel est l'indice de cet appel système dans la table des appels systèmes ?

Une méthode classique [1] pour cacher un fichier ou un répertoire, voir des actions mal intentionnées, à l'administrateur système d'un ordinateur sous GNU/Linux, est d'insérer un module qui intercepte les appels systèmes et en modifie le comportement pour ne pas répondre aux sollicitations correspondantes. À titre d'exemple, le module noyau disponible à `http://jmfriedt.free.fr/module_M1.c` – que l'on trouve aussi sur chaque ordinateur dans `/home/jmfriedt/module_M1.c` – intercepte un appel système et le détourne de sa fonction d'origine. Quelques subtilités tiennent en ce que l'emplacement de la table contenant les pointeurs de fonctions n'est plus exportée par le noyau vers les modules. Nous devons donc chercher dans la mémoire du noyau où se trouve cette table. Le noyau Linux est stocké à la fin de l'espace d'adressage virtuel, entre l'adresse `PAGE_OFFSET` et la fin de l'espace d'adressage virtuel [2]. Une recherche exhaustive permet de trouver l'emplacement du pointeur vers une des fonctions correspondant à un appel système connu, par exemple dans notre cas `close` (fonction `aquire_sys_call_table`),

1. `turbochaos.blogspot.fr/2013/09/linux-rootkits-101-1-of-3.html` ou `gadgetweb.de/linux/40-how-to-hijacking-the-syscall-table-on`
2. `http://www.arm.linux.org.uk/developer/memory.txt`

```
unsigned long int offset = PAGE_OFFSET; // debut du kernel en RAM
unsigned long **sct;
while (offset < ULLONG_MAX) {
    sct = (unsigned long **)offset;
    if (sct[__NR_close] == (unsigned long *) sys_close) return sct;
    offset += sizeof(void *);
}
```

7. démontrer la capacité à compiler le module sur carte Olinuxino A13 et à le charger en mémoire. On se rappellera que la méthode pour compiler les modules consiste à appeler le makefile qui se trouve dans les sources du noyau exécuté sur la plateforme, en passant les variables `ARCH` pour préciser l'architecture du processeur sur lequel nous travaillons, `-C` pour préciser où se trouvent les sources, et en précisant que nous désirons compléter la méthode `modules`. Fournir les message affiché lors du chargement.
8. Ayant identifié l'emplacement de la table des fonctions correspondant aux appels systèmes, nous redirigeons une de ces entrées vers nos propres fonctions. Constater l'effet sur l'appel système qui a été intercepté (quel est-il ? comment démontrer l'interception de cet appel ?)
9. modifier le programme (fonction `new_sys_mkdir()` pour que malgré l'interception de l'appel système, l'utilisateur (ou l'administrateur) ait tout de même l'impression que son système fonctionne normalement, et ne puisse pas se rendre compte de la supercherie.

Nous constatons donc dans cet exercice qu'une compréhension détaillée du fonctionnement d'un microprocesseur, en particulier au niveau du noyau qui interagit avec le processeur comme le ferait un programme en C sur microcontrôleur, avec toutes les autorisations de manipulation des registres et de la mémoire, permet d'en manipuler à volonté les fonctionnalités et en particulier de retirer tout espoir de sécurité du système.

Réponses

1.
```
int toto(int x) {return (x+1);}
int tata(int x) {return (x+2);}
int main() { printf("%x\n",(unsigned int)&toto);}
```
Nous utilisons le *pointeur* vers la fonction (`&toto`) pour en identifier l'emplacement.

2.
```
int toto(int x) {return (x+1);}
int tata(int x) {return (x+2);}
int main() { printf("%x\n",(unsigned int)&tata-(unsigned int)&toto);}
```
sous hypothèse que le compilateur n'optimise pas le code en inversant par exemple l'ordre des fonctions ou en insérant le code directement dans la fonction principale sans faire appel à une fonction.
On se convainc que le résultat (longueur 0x0b=11 octets) est correct par `objdump -dSt fichier.elf` :

```
080483eb <toto>:
int toto(int x) {return (x+1);}
 80483eb:       55                      push   %ebp
 80483ec:       89 e5                   mov    %esp,%ebp
 80483ee:       8b 45 08                mov    0x8(%ebp),%eax
 80483f1:       83 c0 01                add    $0x1,%eax
 80483f4:       5d                      pop    %ebp
 80483f5:       c3                      ret
```

3.
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int i=0;
void toto() {i+=1;} // la fonction originale
void tata() {i+=2;} // la fonction de remplacement

int main()
{int longueur=(int)&main-(int)&tata;
 printf("%d %x %x %d %d\n",i,&toto,&tata,(int)&tata-(int)&toto,longueur);
 toto();
 printf("i=%d\n",i);
 memcpy(&toto,&tata,longueur); // ecrase l'ancienne fonction
 toto();
 printf("i=%d\n",i);
}
```
se traduit par un accès illégal à un segment mémoire (Segmentation Fault)

4.
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h> // mprotect
#include <unistd.h>   // sysconf

int i=0;
void toto() {i+=1;} // la fonction originale
void tata() {i+=2;} // la fonction de remplacement

int main()
{int pagesize;
 int longueur=(int)&main-(int)&tata;
 printf("%d %x %x %d %d\n",i,&toto,&tata,(int)&tata-(int)&toto,longueur);
 toto();
 printf("i=%d\n",i);
 pagesize = sysconf(_SC_PAGE_SIZE);      // mprotect doit etre aligne'
 mprotect((void*)((int)&toto&~(pagesize-1)),(int)&main-(int)&toto,PROT_EXEC|PROT_READ|PROT_WRITE);
 memcpy(&toto,&tata,longueur); // ecrase l'ancienne fonction
 toto();
 printf("i=%d\n",i);
}
```
effectue l'opération attendue (renvoie 1 puis 3, soit un incrément de 1 puis de 2). Le calcul de la page contenant l'adresse de la fonction s'obtient par le masque `&~(pagesize-1)`

5. le mécanisme classique est le buffer overflow dans lequel un utilisateur entre une chaîne de caractères plus longueur que la taille du tampon capable d'accueillir ces données. De cette façon, la pile est corrompue par les données qui dépassent du tableau alloué en mémoire, permettant à un utilisateur d'injecter du code dans le programme.
6. `strace mkdir toto` nous indique que l'appel système `mkdir("toto", 0777)` est appelé : c'est cet appel système que nous redirigerons pour intercepter la création de répertoires par un utilisateur (ou l'administrateur)
7. au chargement, le pilote fournit l'adresse à laquelle est stocké le noyau : c0000000
8. une fois le pilote chargé, l'appel système `mkdir` est intercepté et nous retournons une valeur nulle sans avoir créé le répertoire : il n'est plus possible de créer un répertoire. Par ailleurs, le pilote nous informe, dans les messages du système (`dmesg`), de l'interception de la requête de l'appel système (`intercept`), du nom du répertoire qui aurait dû être créé, et ses permissions.

9. la capacité de création de répertoire est simplement restaurée en appelant la fonction originale de création du répertoire – que nous avions pris soin de mémoriser dans **`ref_sys_mkdir`**, lors de l'appel de notre fonction de remplacement :

```
long new_sys_mkdir(const char __user *pnam, int mode)
{long ret=ref_sys_mkdir(pnam, mode);
 printk(KERN_INFO "intercept: %s:%x\n", pnam,mode);
 return 0;
}
```