Modification des appels systèmes du noyau Linux : manipulation de la mémoire du noyau

J.-M Friedt, 8 septembre 2016

Résumé : le noyau est chargé d'ordonnancer les processus du système et garantir la cohérence des accès aux ressources au travers de pilotes. Ainsi, il a tous les pouvoirs sur le matériel, y compris sur la mémoire et les séquences d'instructions qui y sont lues pour être exécutées. Ces caractéristiques font du noyau une cible parfaite pour attaquer un système, que ce soit pour cacher à l'administrateur des traces d'une intrusion ou intercepter des opérations en cours d'exécution par les utilisateurs du système informatique attaqué. Nous présentons quelques méthodes de détournement des appels système, par modification des adresses contenant les fonctions ou par modification de ces fonctions elles mêmes.

Mots clés : noyau Linux, rootkit, redirection, appels systèmes.

1 Introduction

Un logiciel est conçu pour effectuer certaines opérations, qui peuvent ou non convenir à nos intentions. Lorsque les sources du logiciel sont disponibles, rien n'est plus simple pour répondre à nos besoins : nous modifions les sources, recompilons le logiciel, et l'utilisateur est satisfait. Cependant, ils existe des cas où les sources ne sont pas disponibles : logiciels propriétaires disponibles uniquement sous forme de binaire ou, une fois l'accès à un ordinateur sur lequel nous ne sommes pas administrateur acquis, noyau du système d'exploitation. Dans le cas particulier d'ordinateurs sous GNU/Linux, il se peut que nous ne désirions pas laisser de trace de notre passage, ou compliquer la tâche de l'administrateur cherchant à connaître la liste des utilisateurs et des fichiers sur son système. Le détournement des appels systèmes (`man syscalls`), en particulier tels que mis en œuvre dans les *rootkits*, répond à ces besoins. Plus intéressant que les mauvaises intentions de cacher un accès illégal à un ordinateur, manipuler le fichier binaire (exécutable [1] ou noyau) est l'opportunité de mieux comprendre le fonctionnement du système d'exploitation contrôlant notre environnement de travail, et d'en appréhender les forces et les faiblesses.

Presque tous les ouvrages contenant les mots "Linux" et "security" [2, 3] discutent exclusivement des aspects d'administration – comment éviter une intrusion, depuis l'extérieur, d'un utilisateur non-autorisé (firewall et protection réseau) ou comment éviter le gain de privilèges par un utilisateur autorisé qui n'est pas administrateur. Peu d'ouvrages discutent de la gestion du noyau et des appels systèmes qu'il fournit, puis de la façon de les manipuler [4, 5]. Par ailleurs, la majorité des considérations sur ces sujets gravite autour des architectures compatibles Intel 32 ou 64 bits, mais la popularité d'Android [1] (basé sur un noyau Linux) et des plate-formes conçues autour de processeurs ARM [6], nous conduisent à appréhender les problèmes de détournement des appels système du noyau sur ces architectures. Tous les programmes exposés dans ce document ont été testés sur un ordinateur personnel basé sur un processeur Intel I5 exécutant Debian/GNU Linux sid (noyau 4.6 x86_64) avec une compilation au moyen de `gcc` 6.1, ainsi que sur carte Olinuxino-A13-micro basée sur un processeur Allwinner A13 exécutant un système construit par `buildroot` [2] proposant Linux 4.4.2 et les outils associés dont `gcc` 4.9.3 en cross-compilateur pour ARM HF [3]. Les programmes proposés dans cette prose sont disponibles à `http://jmfriedt.free.fr/lm_kernel.tar.gz`. Tous les messages issus du noyau sont affichés par `dmesg` : on notera que parfois l'uptime de l'ordinateur était très court. En effet, toute erreur dans un module noyau (par exemple accès à une zone mémoire non-allouée) se traduira par une corruption du noyau et, tôt ou tard, un redémarrage du système. On évitera donc de tester ces programmes sur un ordinateur aux fonctionnalités critiques ...

Les concepts modernes de sécurité informatique, en particulier tels que implémentés sur les systèmes compatibles unix, tient en la séparation des droits. L'objectif est de restreindre au maximum les droits des utilisateurs, exposés au quotidien aux attaques virales [7], trojan et autres worm[8], et laisser un peu plus de droits à l'administrateur. En fin de compte, cette distribution des droits est toujours dévolue à un superviseur, qui dans le cas de Linux est pris en charge par le noyau monolithique qui donne son nom au système d'exploitation. Agir au niveau du noyau nous donne donc tous les pouvoir sur le système informatique, mais surtout du point de vue du développeur sur systèmes embarqués et microcontrôleurs, nous redonne tous les droits d'accès aux ressources sans être bridé par un noyau qui nous surveille. En particulier, cette approche nous rappelle que tout processeur, du petit ARM7 aux gros multicœurs modernes qui équipent nos ordinateurs personnels, se résume en une unité arithmétique et logique qui reçoit ses instructions d'un gros tas d'octets qu'est la mémoire. Puisque la grande majorité des architectures

1. *Malware hits millions of Android phones*, à `http://www.bbc.com/news/technology-36744925`
2. `https://github.com/trabucayre/buildroot`
3. Les cas où l'architecture du processeur induit des différences de code source sont gérés en testant la constante `__ARMEL__` indicatrice d'une cross-compilation à destination de ARM, tel que nous en informe `arm-buildroot-linux-uclibcgnueabihf-gcc -dM -E - < /dev/null | grep ARM`

actuellement en usage – à l'exception des architectures AVR d'Atmel et quelques petits Microchip conçus suivant l'architecture Harvard séparant mémoire d'instructions et mémoire de données – mélangent instructions et données selon les précepts de Von Neuman, nous serons en droit de modifier (en manipulant des données) les instructions exécutées par le processeur.

Les développeurs au niveau du noyau connaissent bien cette problématique, et quelques tentatives de protéger les pages mémoire au moyen de l'unité de gestion de mémoire (MMU – *Memory Management Unit*) [9] essaient de nous brider, mais étant toujours maîtres absolus du système informatique au niveau du noyau, il nous suffira de désactiver ces protections pour laisser libre cours à nos activités. Dans les lignes qui vont suivre, nous verrons comment modifier la fonction appelée lors d'un appel système, ou modifier le contenu de cette fonction. Mais avant tout, qu'est-ce qu'un appel système ?

2 Aspect historique : INT10, INT21 et leurs amis

Historiquement, les appels systèmes étaient gérés par des interruptions logicielles, une façon d'interrompre l'exécution séquentielle d'un programme pour sauter dans une fonction définie par le système.

```
IBM Personal Computer MACRO Assembler  Version 2.00                1-16
ORGS ----- 04/21/86  COMPATIBILITY MODULE                          04-21-86


                        PAGE
                        ;-----  VECTOR TABLE

                        ;-      ORG     0FEF3H
1551 IEF3               ;       ORG     01EF3H
1552 IEF3               VECTOR_TABLE    LABEL   WORD            ; VECTOR TABLE VALUES FOR POST TESTS
1553 IEF3 1EA5 R                DW      OFFSET TIMER_INT        ; INT 08H - HARDWARE TIMER 0            IRQ  0
1554 IEF5 0987 R                DW      OFFSET KB_INT           ; INT 09H - KEYBOARD                    IRQ  1
1555 IEF7 0000 E                DW      OFFSET  DTI             ; INT 0AH - SLAVE INTERRUPT INPUT
1556 IEF9 0000 E                DW      OFFSET  DII             ; INT 0BH -                             IRQ  3
1557 IEFB 0000 E                DW      OFFSET  DII             ; INT 0CH -                             IRQ  4
1558 IEFD 0000 E                DW      OFFSET  DII             ; INT 0DH -                             IRQ  5
1559 IEFF 0F57 R                DW      OFFSET DISK_INT         ; INT 0EH - DISKETTE                    IRQ  6
1560 IF01 0000 E                DW      OFFSET  DII             ; INT 0FH -                             IRQ  7
1561
1562                    ;-----  SOFTWARE INTERRUPTS  ( BIOS CALLS AND POINTERS )
1563
1564 IF03 1065 R                DW      OFFSET VIDEO_IO         ; INT 10H -- VIDEO DISPLAY
1565 IF05 184D R                DW      OFFSET EQUIPMENT        ; INT 11H -- GET EQUIPMENT FLAG WORD
1566 IF07 1841 R                DW      OFFSET MEMORY_SIZE_DET  ; INT 12H -- GET REAL MODE MEMORY SIZE
1567 IF09 0C59 R                DW      OFFSET DISKETTE_IO      ; INT 13H -- DISKETTE
1568 IF0B 0739 R                DW      OFFSET RS232_IO         ; INT 14H -- COMMUNICATION ADAPTER
1569 IF0D 1859 R                DW      OFFSET CASSETTE_IO      ; INT 15H -- EXPANDED BIOS FUNCTION CALL
1570 IF0F 082E R                DW      OFFSET KEYBOARD_IO      ; INT 16H -- KEYBOARD INPUT
1571 IF11 0FD2 R                DW      OFFSET PRINTER_TO       ; INT 17H -- PRINTER OUTPUT
1572 IF13 0000                  DW      00000H                  ; INT 18H -- 0F600H INSERTED FOR BASIC
1573 IF15 06F2 R                DW      OFFSET BOOT_STRAP       ; INT 19H -- BOOT FROM SYSTEM MEDIA
1574 IF17 1E6E R                DW      OFFSET TIME_OF_DAY      ; INT 1AH -- TIME_OF DAY
1575 IF19 1F53 R                DW      OFFSET DUMMY_RETURN     ; INT 1BH -- KEYBOARD BREAK ADDRESS
1576 IF1B 1F53 R                DW      OFFSET DUMMY_RETURN     ; INT 1CH -- TIMER BREAK ADDRESS
1577 IF1D 10A4 R                DW      OFFSET VIDEO_PARMS      ; INT 1DH -- VIDEO PARAMETERS
1578 IF1F 0FC7 R                DW      OFFSET DISK_BASE        ; INT 1EH -- DISKETTE PARAMETERS
1579 IF21 0000                  DW      00000H                  ; INT 1FH -- POINTER TO VIDEO EXTENSION
1580
1581 IF23               SLAVE_VECTOR_TABLE      LABEL WORD      ; ( INTERRUPT 70H THRU 7FH )
1582
1583 IF23 0000 E                DW      OFFSET RTC_INT          ; INT 70H - REAL TIME CLOCK             IRQ  8
1584 IF25 0000 E                DW      OFFSET RE_DIRECT        ; INT 71H - REDIRECT TO INT 0AH         IRQ  9
1585 IF27 0000 E                DW      OFFSET  DTI             ; INT 72H -                             IRQ 10
1586 IF29 0000 E                DW      OFFSET  DII             ; INT 73H -                             IRQ 11
1587 IF2B 0000 E                DW      OFFSET  DII             ; INT 74H -                             IRQ 12
1588 IF2D 0000 E                DW      OFFSET INT_287          ; INT 75H -  -MATH COPROCESSOR          IRQ 13
1589 IF2F 0000 E                DW      OFFSET  DIT             ; INT 76H -  -FIXED DISK                IRQ 14
1590 IF31 0000 E                DW      OFFSET  DII             ; INT 77H -                             IRQ 15
1591
1592                    ;-----  DUMMY INTERRUPT HANDLER
1593
1594                    ;-      ORG     0FF53H
1595 IF53               ;       ORG     01F53H
1596
1597 = IF53             DUMMY_RETURN    EQU     $               ; BIOS DUMMY (NULL) INTERRUPT RETURN
1598
1599 IF53 CF                      IRET
1600
1601                    ;-----  PRINT SCREEN
1602
1603                    ;-      ORG     0FF54H
1604 IF54               ;       ORG     01F54H
1605 = IF54             PRINT_SCREEN    EQU     $
1606 IF54 E9 0000 E             JMP     PRINT_SCREEN_I          ; VECTOR ON TO MOVED BIOS CODE
1607                    .LIST                                   ; TUTOR
1608                    ;---------------------------------------
1609                    ;                                       :
1610                    ;       POWER ON RESET VECTOR           :
1611                    ;                                       :
1612                    ;---------------------------------------
1613                    ;-      ORG     0FFF0H
1614 1FF0               ;       ORG     01FF0H
1615
1616                    ;-----  POWER ON RESET
1617
1618 1FF0               P_O_R   LABEL   FAR                     ; POWER ON RESTART EXECUTION LOCATION
1619
1620 1FF0 EA                    DB      0EAH                    ; HARD CODE FAR JUMP TO SET
1621 1FF1 005B R                 DW      OFFSET  RESET           ; OFFSET
1622 1FF3 F000                   DW      0F000H                  ;   SEGMENT
1623
1624 1FF5 30 34 2F 32 31 2F     DB      '04/21/86'              ; RELEASE MARKER
1625       38 36
1626
1627 1FFE                       ORG     01FFEH
1628 1FFE FC                    DB      MODEL_BYTE              ; THIS PC'S ID ( MODEL BYTE )
1629
1630 1FFF               CODE    ENDS                            ; CHECKSUM AT LAST LOCATION
1631                            END
```

Figure 1: Scan de la documentation du PC AT d'IBM à base de 80286, incluant le listing du BIOS et notamment la liste des interruptions, incluant les interruptions logicielles faisant office d'appels système. Une copie du document complet est disponible à `http://bitsavers.trailing-edge.com/pdf/ibm/pc/at/1502494_PC_AT_Technical_Reference_Mar84.pdf`

Sur les ordinateurs à base de processeurs Intel et compatibles, certains services sont fournis par le BIOS (en mode réel), accessibles par une série d'interruptions déclenchées par l'instruction `int`, avec par exemple l'interruption 0x10 pour les affichages sur écran, 0x13 pour les accès aux disques ou 0x16 pour accéder au clavier. Le choix du service est fourni dans le registre AX, et l'argument dans BX. Au dessus du BIOS, DOS fournit ses propres appels système par l'interruption logicielle 0x21 (instruction assembleur `int 0x21`) après avoir chargé, toujours en mode réel, dans le registre AX le numéro du service et dans BX l'argument. On notera que bien avant l'avènement des BIOS propriéraires, l'ordinateur à base de 80286 commercialisé par IBM était fourni avec un manuel contenant non seulement les schémas électroniques de la carte mère et de ses périphériques, mais aussi les codes sources du BIOS et l'implémentation des services qu'il fournit ! (Fig. 1). Sous DOS, le concepts d'interception des appels systèmes faisait partie de la programmation en *Terminate and Stay Resident* (TSR), dans laquelle le système d'exploitation devait ne pas libérer les ressources occupées par un programme mais les conserver en mémoire pour y accéder sous interruption logicielle : `support.microsoft.com/en-us/kb/28568` propose un exemple sous MS-DOS d'interception des appels systèmes de gestion des répertoires.. Cette approche se retrouve aujourd'hui, de façon considérablement enrichie puisque tous les appels systèmes qui définissent le respect de la norme POSIX [4] doivent être accessibles : dans le noyau Linux, la structure `syscall_table` fournit la liste des appels systèmes et la position (adresse) en mémoire de l'implémentation de chaque fonction. Cette table est ensuite utilisée pour générer les appels systèmes – donc en renseignant le registre EAX (mode protégé du processeur x86) et en appelant l'interruption 0x80, tel que nous le constatons en consultant les sources du noyau à `linux-4.4.2/arch/x86/ia32/ia32_signal.c`

3 Modification de l'adresse de la fonction appelée

Notre objectif est de modifier les appels systèmes. Nous pouvons dans un premier temps nous demander quel appel système est appelé au cours de quelle opération, pour évaluer l'intérêt de la tâche.

4. `http://pubs.opengroup.org/onlinepubs/009695399/idx/functions.html`

`strace` (*trace system calls and signals*) est l'outil du shell qui informe l'utilisateur des appels systèmes sollicités lors de l'exécution d'un programme. Ainsi, lors de `strace ls`, nous constatons qu'une multitude d'appels à `open` sont nécessaires pour charger toutes les bibliothèques suite à l'exécution de `ls` par `execve`. Les affichages à l'écran à proprement parler se font par `write`, et finalement les ressources sollicitées sont relâchées par `close`. Nous constatons donc que la manipulation de ces appels systèmes – `open`, `close`, `read` ou `write` – doit se faire avec le plus grand soin, au risque de complètement détruire les fonctionnalités du système d'exploitation. Avant de nous attaquer au problème de rediriger les appels systèmes, nous allons faire un petit détour par les pointeurs de fonction, pour nous rappeler comment rediriger un appel d'une fonction à une autre, et comment un appel de fonction est géré au niveau de l'assembleur.

Architectures Harvard et Von Neumann : l'hypothèse que nous ferons tout au cours de cette étude est qu'une instruction chargée de manipuler une donnée est capable de modifier une instruction qui sera exécutée par le processeur. Cette hypothèse n'est pas évidente, et valable uniquement sur une architecture de type Von Neumann. À la genèse du développement des ordinateurs, alors que les processeurs tels que l'IBM ASCC installé à Harvard, étaient tellement lents qu'il fallait plusieurs secondes pour effectuer une opération arithmétique, un gain de temps consistait à chercher les instructions dans une mémoire, et les données dans une autre mémoire séparée : les deux bus d'accès à ces ressources sont séparés et accessibles simultanément, au lieu d'un accès séquentiel comme dans l'architecture Von Neumann où données et instructions occupent la même mémoire. Aujourd'hui, les seuls processeurs à encore être conçus en architecture Harvard sont les AVR de Atmel et les PIC de la gamme Microchip. Ce point n'est pas anodin et a des conséquences sur la capacité d'un compilateur à optimiser l'utilisation des ressources. Un cas classique concerne les polices de caractères : tableaux volumineux de données qui ne sont accédées qu'en lectures, les tableaux définissant les polices de caractères sont typiquement préfixées de l'attribut `const` pour indiquer à `gcc` de placer les données en mémoire non-volatile, généralement disponible en excès, contrairement à la mémoire volatile. Ainsi, `const police[95*8]=0;` pour définir les 95 caractères ASCII affichables se traduit sur (architecture Von Neuman) MSP430 par (`msp430-size` après avoir compilé par `msp430-gcc`) une occupation de 1630 octets de flash (section `text`) et 2 octets de RAM (sections `bss` et `data`). Au contraire sur AVR, `avr-size` nous indique que la même compilation se traduit par 38 octets de text et 1520 octets de data : un tel programme tient tout juste dans un Atmega16U4 alors que ses 16 KB de flash restent innoccupés. Sur ordinateur personnel, ces problèmes de séparation de données et instructions ne devraient pas survenir, bien que la gestion des caches entre la mémoire généraliste et les processeurs puisse devenir un problème : en cas de dysfonctionnement des exemples proposés, on pourra tenter d'invalider les caches pour forcer le processeur à recharger les instructions en mémoire [10].

3.1 En espace utilisateur

Pour appréhender l'approche consistant à modifier l'adresse de la fonction appelée [5] il est bon de se remémorer le concept de pointeur de fonction. Appeler une fonction revient à empiler les arguments, sauter (instruction assembleur `call`) à l'adresse de la fonction chargée du traitement, puis revenir au programme principal en dépilant la valeur du *program counter* qui avait été empilée au moment de l'appel de la subroutine. C'est à cette dernière étape que nous pouvons éventuellement induire un saut dans une fonction tierce en ayant corrompu la pile dans la fonction appelée dans l'attaque classique du *buffer overflow*. Ici nous nous intéressons à modifier l'adresse de la fonction appelée. Dans l'exemple ci-dessous `ma_func` est un pointeur vers une fonction qui n'est pas déclarée initialement.

```
#include <stdio.h>
int toto(int x) {return (x+1);}
int tata(int x) {return (x+2);}

int main()
{int (*ma_func)(int);
 ma_func = &toto; printf("%d\n",ma_func(1));
 ma_func = &tata; printf("%d\n",ma_func(1));
 return 0;
}
```

Ayant déclaré le pointeur de fonction mais sans l'avoir initialisé, appeler `ma_func(argument)` se traduit évidemment par une erreur de segment (`segmentation fault`) puisque nous induisons un saut vers un segment mémoire qui n'est pas attribué au programme en cours d'exécution. Nous pouvons alors assigner le pointeur `ma_func(argument)` à l'adresse d'une fonction du programme, ici `toto` puis `tata`, qui se traduira bien par l'effet recherché, ici un résultat de 2 puis de 3. Nous serons donc capables de rediriger un appel en modifiant l'argument de l'instruction `call` lors du saut au sous-programme. Cette méthode est parfois utilisée lorsqu'une même fonction se décline de différentes façons selon les périphériques auxquels elle s'applique : nous avons par exemple rencontré ce cas dans l'implémentation libre du bus Modbus pour microcontrôleur disponible à `http://www.freemodbus.org/`. Dans ce programme, une méthode

5. `http://www.gilgalab.com.br/hacking/programming/linux/2013/01/11/Hooking-Linux-3-syscalls/`

commune à toutes les implémentations d'un protocole, par exemple `write()` ou `open()`, est définie, et les diverses implémentations sont déclinées pour les diverses implémentations matérielles en faisant pointer chaque fonction vers l'implémentation appropriée. Dans freemodbus, la fonction `eMBInit()` de `mb.c` initialise les pointeurs de fonctions selon le protocole de communication utilisé. Si au lieu d'assigner statiquement à la compilation la fonction appelée nous désirons l'assigner dynamiquement au cours de l'exécution, l'adressage indirect permet d'appeler une adresse fournie dans un registre. Dans l'assembleur AVR, plus facile à lire que l'assembleur x86, cette différence s'observe entre l'appel direct et par pointeur de fonction à `toto()` :

```
#include <stdio.h>
int toto(int x) {return (x+1);}

int main()
{int (*ma_func)(int);
 printf("%d\n",toto(1));                          // appel direct a toto
 ma_func = &toto; printf("%d\n",ma_func(1)); // passage par pointeur de fonction
 return 0;
}
```

que nous compilons avec l'option de déverminage `-g` pour conserver les symboles :

```
int toto(int x) {return (x+1);}
  ea:   cf 93           push    r28
[...]
  fc:   01 96           adiw    r24, 0x01       ; 1
[...]
 104:   cf 91           pop     r28
 106:   08 95           ret

int main()
{[...]
 printf("%d\n",toto(1));
 130:   81 e0           ldi     r24, 0x01       ; 1
 132:   90 e0           ldi     r25, 0x00       ; 0
 134:   0e 94 75 00     call    0xea    ; 0xea <toto>
[...]
 ma_func = &toto; printf("%d\n",ma_func(1));
[...]
 168:   f9 01           movw    r30, r18
 16a:   09 95           icall
[...]
}
```

Le code assembleur est issu de `avr-objdump -dSt a.out` Dans ce cas, le registre Z contient l'adresse de la fonction appelée, tel que décrit dans la liste des mnémoniques de l'assembleur AVR [6]. Z est la concaténation de R31 et R30, d'où notre intérêt pour la manipulation de R30 juste avant l'appel à `icall`.

3.2 En espace noyau

Ayant compris que l'adresse mémoire contenant la fonction peut être modifiée, il devient presque trivial d'attribuer une nouvelle destination à un appel système. Nous avons vu que la liste des appels systèmes sollicités par une commande shell est affichée par `strace`. Plutôt que risquer de casser notre système en manipulant un appel critique, nous allons considérer une fonction qui ne sert presque à rien mais dont l'effet est spectaculaire, la création de répertoire. Lorsque nous lançons `strace mkdir toto` nous obtenons notamment

```
mkdir("toto", 0777)                     = 0
```

Nous constatons que l'appel système `mkdir` est utilisé pour créer le répertoire : c'est lui que nous allons détourner.

L'approche consistant à modifier le pointeur vers la fonction implémentant un appel système est tellement évidente que les développeurs du noyau Linux essaient de brider un peu notre capacité à modifier ces appels systèmes en n'exportant pas le symbole de la table contenant les adresses des appels systèmes [7] : ne connaissant pas l'adresse où se trouve la liste des appels systèmes, il devrait nous être impossible de la modifier. Il faut donc d'abord identifier l'emplacement de la table des symboles en mémoire [8], avant d'avoir le droit de la modifier [9].

6. http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_ICALL.html
7. https://bbs.archlinux.org/viewtopic.php?id=139406
8. turbochaos.blogspot.fr/2013/09/linux-rootkits-101-1-of-3.html ou gadgetweb.de/linux/40-how-to-hijacking-the-syscall-table-on
9. https://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/

Afin de trouver l'emplacement de la table des appels systèmes, il devrait suffire de balayer la mémoire à la recherche de l'adresse de l'appel système que nous cherchons à modifier. La méthode n'est pas garantie car il se pourrait qu'un emplacement mémoire contienne par hasard les valeurs correspondant à l'adresse d'un appel système, mais les chances sont faibles et cette approche fonctionne dans la pratique. Cependant, tous les appels systèmes ne sont pas exportés vers le noyau : nous devons en trouver un afin de connaître l'adresse à rechercher en mémoire. La liste des appels système se trouve dans `include/linux/syscalls.h` des sources du noyau : tous les appels systèmes commencent par `sys_`. Un développeur naïf pourrait se dire que voulant manipuler `mkdir`, il nous suffit de trouver l'adresse de l'appel système `sys_mkdir` et de trouver son emplacement en mémoire. Cette tentative échoue au moment de l'édition de lien lors de la compilation du noyau, avec un message nous informant que le symbole `sys_mkdir` n'est pas connu. En effet, l'appel système est implémenté, mais le symbole n'est pas exporté vers les autres modules du noyau. Nous devons donc compléter la recherche pour identifier la liste des symboles exportés qui correspondent à nos besoins : dans les souces du noyau (ici 4.4.2),

```
linux-4.4.2$ grep -r EXPORT_SYMBOL * | grep \(sys_
[...]
fs/open.c:EXPORT_SYMBOL(sys_close);
kernel/time/time.c:EXPORT_SYMBOL(sys_tz);
```

nous indique que seul `sys_close` est exporté. Nous allons donc rechercher l'occurence de ce symbole, en sachant qu'ensuite la table est arrangée dans le même ordre que la liste des appels définie dans `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
[...]
#define __NR_mkdir 83
```

Il y a donc 80 emplacements mémoire d'écart entre l'adresse de l'appel système `close` et `mkdir` dans la table des appels systèmes. Le module suivant effectue la recherche dans la mémoire allouée au noyau, qui commence à l'adresse définie par la constante `PAGE_OFFSET` tel que décrit dans `linux-4.4.2/Documentation/x86/x86_64/mm.txt` : il s'agit de la constante ajoutée à l'adresse de la mémoire physique pour définir l'adresse virtuelle de la mémoire dédiée au noyau (`https://linux-mm.org/VirtualMemory`).

```c
#include <linux/module.h>
#include <linux/syscalls.h>

static unsigned long* cherche_table(void)
{unsigned long int offset = PAGE_OFFSET;    // debut du kernel en RAM
 unsigned long *sct;
 printk(KERN_INFO "PAGE_OFFSET=%lx\n",PAGE_OFFSET);
 while (offset < ULLONG_MAX)      // recherche ds la memoire allouee au noyau
  {sct = (unsigned long *)offset;
   if (*sct == (unsigned long)&sys_close)  // cherche l'@ de sys_close
      return sct;                          // on a trouve' l'@ de debut
   offset += sizeof(void *);
  }
 return NULL;
}

static int __init module_start(void)
{unsigned long *addr_table;
 addr_table = cherche_table();
 printk(KERN_INFO "table: %lx\n",(unsigned long)addr_table);
 printk(KERN_INFO "NR close: %d\n",__NR_close);
 printk(KERN_INFO "NR mkdir: %d => %lx",__NR_mkdir,addr_table[__NR_mkdir-__NR_close]);
 printk(KERN_INFO "*void: %ld\n",sizeof(void*));
 return 0;
}

static void __exit module_end(void) {}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

La fonction `cherche_table()` balaie la mémoire du noyau à partir de `PAGE_OFFSET` jusqu'à trouver un emplacement contenant l'adresse de l'appel système `sys_close` : nous avons vu auparavant avec les pointeurs de fonctions que l'adresse de la fonction s'obtient en préfixant son nom par `&`. Si le contenu de l'emplacement pointé par `sct` contient l'adresse `&sys_close`, il est probable que nous ayons trouvé la table des appels systèmes. Nous renvoyons donc cette adresse à la fonction initialisant le pilote `module_start()` et y affichons l'emplacement de la table qui doit correspondre à l'emplacement de `sys_call_table`, le

contenu de l'emplacement mémoire que nous avons trouvé et le contenu de l'emplacement 80 cases mémoire plus loin, puisque `__NR_mkdir-__NR_close` vaut 80. Chaque emplacement mémoire contient un adresse, donc de taille `sizeof(void*)`, ou 8 octets sur une architecture 64 bits.

Le chargement de ce module dans un noyau 4.6 (Debian sid à la date de rédaction de cette prose) sur une architecture 64 bits compatible Intel (x86_64) se traduit par les messages

```
[100266.370251] PAGE_OFFSET=ffff880000000000
[100266.433568] table: ffff880016001f8
[100266.433570] NR close: 3
[100266.433571] NR mkdir: 83 => ffffffff812020d0
[100266.433572] *void: 8
```

qui indiquent l'adresse de début de la zone mémoire allouée au noyau (constante `PAGE_OFFSET`), l'emplacement de la table des symboles identifiée par la recherche de l'adresse de l'appel système `sys_close()`, et le contenu de l'emplacement mémoire qui se trouve 80 cases après l'adresse que nous avons ainsi identifié. Nous validons la cohérence de ces informations avec celles fournies par le noyau dans `/boot/System.map` ou dans `/proc/kallsyms`. En effet,

```
/boot/System.map-4.6.0-1-amd64: ffffffff816001e0 R sys_call_table
/proc/kallsyms                : ffffffff816001e0 R sys_call_table
```

qui indique que la table des appels système se trouve à l'adresse `0x816001e0`, soit 24 octets avant l'adresse que nous avons identifié, ce qui est cohérent avec l'indice de l'appel système (3) multiplié par la taille de chaque case mémoire (8 octets). Par ailleurs, nous vérifions que l'emplacement mémoire que nous avons identifié comme contenant l'appel système `mkdir` contient la bonne information : `/proc/kallsyms` nous informe que

```
ffffffff812020d0 T sys_mkdir
```

qui est bien l'adresse trouvée par le pilote noyau.

Nous voilà donc convaincus de la capacité à identifier l'emplacement de la table des appels systèmes en mémoire et l'adresse de chaque fonction appelant ces appels système. La dernière subtilité pour arriver à nos fins tient encore à la MMU, qui interdit d'écrire dans la page allouée aux appels systèmes (tel qu'indiqué par l'attribut R dans `/proc/kallsyms`). Nous débloquons cet accès en informant la MMU de désactiver la protection en manipulant le bit WP du registre CR0 [10] (fonctions à évidemment retirer lors des tests sur architecture ARM) :

```c
#include <linux/module.h>
#include <linux/syscalls.h>

unsigned long *addr_table;    // global pour passer a exit()
#ifndef __ARMEL__
unsigned long original_cr0;  // global pour passer a exit()
#endif

// cf http://www.csee.umbc.edu/courses/undergraduate/CMSC421/fall02/burt/projects/→
    ↪howto_add_systemcall.html
asmlinkage        // passage de parametres par la pile et non par les registres
long (*ref_sys_mkdir)(const char __user*,int); // proto (depend de la fonction)

asmlinkage
long mon_mkdir(const char __user *pnam, int mode)
{printk(KERN_INFO "intercept: %s:%x\n", pnam,mode); return 0;}

static unsigned long* cherche_table(void)
{[...]}                       // recherche de la table des appels sys

static int __init module_start(void)
{
 addr_table = cherche_table();
 [...]                        // affichage des informations
#ifndef __ARMEL__
 original_cr0 = read_cr0();  // passe la page des appels sys en ecriture
 write_cr0(original_cr0 & ~0x00010000); // bit 16 est WP: on met \'a 0
#endif
 ref_sys_mkdir = (void *)addr_table[__NR_mkdir-__NR_close];
 addr_table[__NR_mkdir-__NR_close] = (unsigned long)mon_mkdir;
#ifndef __ARMEL__
 write_cr0(original_cr0);     // repasse la page des appels en lecture seule
#endif
 return 0;
}
```

10. `https://en.wikipedia.org/wiki/Control_register`

```
static void __exit module_end(void)
{
#ifndef __ARMEL__
 write_cr0(original_cr0 & ~0x00010000);  // remet la fonction originale
#endif
 addr_table[__NR_mkdir-__NR_close] = (unsigned long)ref_sys_mkdir;
#ifndef __ARMEL__
 write_cr0(original_cr0);
#endif
 printk(KERN_INFO "bye\n");
}
```

Les messages au chargement du pilote restent les mêmes, mais cette fois lorsque nous tentons de créer un répertoire par `mkdir toto` après chargement du module, nous recevons le message additionnel

```
[103669.045129] intercept: toto:1ff
```

qui indique que nous avons bien intercepté l'appel système. Évidemment, en l'état, un utilisateur du système se rendra immédiatement compte de l'interception de l'appel système puisque le répertoire n'est pas créé, et nous pouvons cacher les traces de l'interception en appelant tout de même la fonction d'origine pour effectuer le travail :

```
asmlinkage
long new_sys_mkdir(const char __user *pnam, int mode)
{long ret=ref_sys_mkdir(pnam, mode); // cette fois on cree le repertoire
 printk(KERN_INFO "intercept %ld: %s:%x\n", getuid(),pnam,mode);
 return 0;
}
```

Nous avons donc démontré la capacité à intercepter les appels systèmes, donner l'impression que l'appel a abouti tout en manipulant éventuellement le résultat. Cette approche est classiquement utilisée dans les *rootkits* pour cacher les traces de l'accès : le meilleur moyen de cacher ses traces est de faire croire au système d'exploitation que les fichiers n'existent pas, en interceptant `getdents` qui est la fonction utilisée par `ls` pour afficher le contenu d'un répertoire, et retirer les entrées des fichiers ou répertoires que nous ne voulons pas afficher :

```
# strace ls /tmp/ |& grep -A2 tmp
[...]
stat("/tmp/", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=740, ...}) = 0
open("/tmp/", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFDIR|S_ISVTX|0777, st_size=740, ...}) = 0
getdents(3, /* 37 entries */, 32768)    = 1368
[...]
```

4 Modification du contenu de l'emplacement mémoire

Nous avons vu comment modifier la fonction appelée lors d'un appel système. Une alternative à cette approche consiste à modifier le contenu de la fonction appelée, par exemple en écrasant le contenu de la mémoire pointé par la table des appels systèmes. Cette approche est celle classiquement implémentée par les virus [11, 12, 13], qui écrasent le début du fichier infecté. Dans le meilleur des cas, le bout de code écrasé est déplacé en fin d'exécutable pour y sauter une fois l'infection achevée et donner l'impression à l'utilisateur que son programme est toujours fonctionnel. Dans le cas du noyau Linux, nous verrons que l'excellente optimisation de `gcc` rend cette approche peu intuitive car des fonctions que nous pourrions croire autonomes d'après le code source sont insérées directement dans le flux d'exécution (*inline*) voir disparaissent complètement car leur résultat est connu du compilateur. Nous allons néanmoins tenter de trouver quelques exemples pour illustrer cette approche.

4.1 En espace utilisateur

Rappelons une fois de plus que pour une mémoire d'ordinateur, le concept de donnée ou d'instruction n'existe pas : la RAM est un tas d'octets, que l'on peut soit exécuter si la valeur correspond à un opcode définissant une opération de l'unité arithmétique et logique, soit traiter comme un argument d'une opération arithmétique ou logique. Ainsi, rien n'interdit d'écrire une valeur dans un emplacement mémoire, pour ensuite l'exécuter (concept que les défenseurs des langages protégeant la mémoire tel que Java interdisent, mais qui au contraire fera ici toute la beauté du C et de la manipulation de la mémoire au travers des pointeurs).

Dans l'exemple ci-dessous, nous abordons deux cas : écraser une fonction avec le contenu d'une autre fonction en mémoire, ou placer une fonction dans la pile puis y sauter pour en exécuter le contenu. Ces approches étant des sources classiques d'attaques, Linux tente désormais de s'en protéger en interdisant l'écriture dans les pages mémoire contenant du code – bridant par la même occasion le code automodifiable – ou l'exécution des instructions placées sur la pile. Nous devons donc débloquer ces fonctions

au moyen de `mprotect()` afin d'autoriser l'écriture et l'exécution sur ces plages de mémoire. Les deux options du programme, `function_overwrite` ou `stack_overwrite`, sont exclusives et activent l'une des deux approches proposées ci-dessus :

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

#define function_overwrite  // ecrase fonction() avec fonction_vide()
#undef  stack_overwrite     // ecrase un tableau de la pile avec fonction_vide appel\'→
    ↪ee par ma_func

int i=0;
void fonction(){printf("fonction\n");} // la fonction originale
void fonction_vide() {i=i+1;}          // la fonction de remplacement

int main()
{char c[128];
 int pagesize;
 void (*ma_func)(void);
 int longueur=(int)&main-(int)&fonction_vide;
 printf("%d %x %x %d %d\n",i,&fonction,&fonction_vide,(int)&fonction_vide-(int)&→
     ↪fonction,longueur);
 fonction();
 pagesize = sysconf(_SC_PAGE_SIZE);      // mprotect doit etre aligne'
#ifdef function_overwrite
 mprotect((void*)((int)&fonction&~(pagesize-1)),(int)&main-(int)&fonction,PROT_EXEC|→
     ↪PROT_READ|PROT_WRITE);
 memcpy(&fonction,&fonction_vide,longueur); // ecrase l'ancienne fonction
#else
 // l'execution d'un code sur la pile est aussi possible par gcc -z execstack
 mprotect((void*)((int)c&~(pagesize-1)),longueur,PROT_EXEC|PROT_READ|PROT_WRITE);
 memcpy(c,&fonction_vide,longueur);          // place la nouvelle fonction sur la pile
 ma_func=(void *)c; ma_func();
#endif

 fonction();
 printf("i=%d\n",i);
}
```

Nous constatons qu'effectivement l'appel à `fonction()` en début de programme se traduit par l'affichage du message "fonction", alors qu'après la manipulation, ce même appel, en fin de programme, se traduit par l'absence de message mais la modification de la variable `i`. Par ailleur, l'absence de distinction entre donnée et instruction est illustée par la fonction `ma_func()` qui pointe vers le tableau d'octets `c` dans lequel nous avons copié la séquence d'opcodes `fonction_vide()`.

4.2 En espace noyau

Ayant introduit le concept en espace utilisateur, pouvons nous le transposer au noyau ? Ici encore, le gestionnaire de mémoire va nous poser soucis, en interdisant l'accès aux pages mémoires inutilisées. Il semble évident que manipuler le contenu de la mémoire rend le système excessivement vulnérable, et depuis longtemps des protections ont été mises en place pour éviter l'accès à l'ensemble de la mémoire par un utilisateur [14], même possédant les droits d'administration : `http://lwn.net/Articles/267427/` décrit les limitations mises en place sur `/dev/mem` dès 2008. Un pilote noyau n'est évidemment pas assujetti à ces restrictions, et pourra librement sonder la mémoire tant que la MMU l'y autorise.

4.3 Écraser une fonction par une autre

Dans l'exemple ci-dessous, la `fonction1` va être écrasée par `fonction2`. Pour rendre l'exemple simple à comprendre, ces deux fonctions sont excessivement simples – tellement simples que `gcc` veut absolument les optimiser en précalculant le résultat à la compilation. Nous sommes obligés de forcer `gcc` à oublier ses ambitions d'optimisation en lui interdisant de déplacer les fonctions dans le code appelant (`noinline`) et en lui interdisant de faire d'hypothèse sur une valeur connue de la variable fournie en argument (`volatile`).

```
#include <linux/module.h>

static noinline int fonction1(volatile int i)
   {// printk(KERN_INFO "fonction1");
    return(i+1);}
static noinline int fonction2(volatile int i) { return(i+2);}
static noinline int fonction3(volatile int i) { return(i+3);}

void difference(char *c1,char *c2,int l) // verifie la diff entre deux zones memoire
```

```
{int k,f=0;
 for (k=0;k<l;k++)
     if (c1[k]!=c2[k])
        {f++;
         printk(KERN_INFO "diff %lx: %hhx %hhx",
                (unsigned long)(c1+k),(char)c1[k],(char)c2[k]);
        }
 if (f==0) printk(KERN_INFO "no difference");
}

void overwrite(void)
{char *sct,l;
 char *c1=(char*)&fonction1,*c2=(char*)&fonction2,*c3=(char*)&fonction3;
#ifndef __ARMEL__
 unsigned long original_cr0;
 original_cr0 = read_cr0();              // autorise ecriture
 write_cr0(original_cr0 & ~0x00010000);
#endif
 sct = (char*) fonction1;
 printk(KERN_INFO "sct=%lx\n",(unsigned long)sct);
 printk(KERN_INFO "fonction2=%lx\n",(long)c2);
 printk(KERN_INFO "fonction3=%lx\n",(long)c3);
 l=(unsigned long)c3-(unsigned long)c2; // longueur de fonction2

 printk(KERN_INFO "longueur=%x\n",(unsigned int)l);
 difference((char*)c1,(char*)c2,l);
 printk(KERN_INFO "res avant fonction1(1)=%d",fonction1(1));

 // ecrase fonction1 avec fonction2
 memcpy((void*)c1,(void*)c2,((unsigned long)c3-(unsigned long)c2));

 difference((char*)c1,(char*)c2,l);
 printk(KERN_INFO "res apres fonction1(1)=%d",fonction1(1));
 printk(KERN_INFO "the end\n");
#ifndef __ARMEL__
 write_cr0(original_cr0);          // interdit ecriture
#endif
}

static int __init module_start(void) {overwrite();return(0);}

static void __exit module_end(void) {}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

qui se traduit par

```
[117421.948195] sct=ffffffffc0a71000
[117421.948198] fonction2=ffffffffc0a71020
[117421.948199] fonction3=ffffffffc0a71040
[117421.948199] longueur=20
[117421.948200] diff ffffffffc0a71017: 1 2
[117421.948201] res avant fonction1(1)=2
[117421.948202] no difference
[117421.948203] res apres fonction1(1)=3
[117421.948203] the end
```

Une alternative pour éviter les optimisation abusives de `gcc`, observables par `objdump -dSt` du fichier `.ko` généré, est de faire appel à des fonctions plus complexes, faisant par exemple intervenir `printk`. Nous constatons bien que le premier appel à `fonction1(1)` renvoie 2, comme prévu. Nous observons la différence entre `fonction1` et `fonction2` qui est une différence sur l'argument de la somme, de 1 à 2, puis l'abence de différence après avoir écrasé `fonction1`. Finalement, le second appel à `fonction1` renvoie 3, comme c'eut été le cas si nous avions appelé `fonction2`. L'écrasement de `fonction1` par `fonction2` est démontrée.

4.4 Identifier l'emplacement d'une fonction

Nous sommes certes capables d'écraser une fonction dans le noyau, mais le problème tient maintenant à savoir où se trouve la fonction que nous désirons écraser. Comme auparavant, nous pouvons tenter une recherche sur l'ensemble de la mémoire. Dans l'exemple ci-dessous, nous utilisons une signature unique au module testé – la présence de la chaîne de caractères `qwertyuiop` qui a peu de chances de se trouver en mémoire par ailleurs – pour trouver l'emplacement de notre pilote.

```
#include <linux/module.h>

long cherche(unsigned long offset)
```

```
{unsigned char sct,sctp1,sctp2,sctp3,sctp4,sctp5;
 char s[15]="qwertyuiop\0";
 printk(KERN_INFO "%s\n",s);

 while (offset < ULLONG_MAX)      // recherche ds la memoire allouee au noyau
   {sct  = *(unsigned char*)offset;
    sctp1= *(unsigned char*)(offset+1);
    sctp2= *(unsigned char*)(offset+2);
    sctp3= *(unsigned char*)(offset+3);
    sctp4= *(unsigned char*)(offset+4);
    sctp5= *(unsigned char*)(offset+5);
    if ((sct=='q') && (sctp1=='w') && (sctp2=='e') && (sctp3=='r'))
    // if ((sct=='j') && (sctp1=='m') && (sctp2=='f'))
       {printk(KERN_INFO "-> %lx: %x %x %x %x %x %x\n",(unsigned long)offset,sct,sctp1,→
           ↪sctp2,sctp3,sctp4,sctp5);
        return(offset);
       }
    offset++;
   }
 return offset;
}

static int __init module_start(void)
{printk(KERN_INFO "%lx\n",PAGE_OFFSET);
 printk(KERN_INFO "cherche: %lx\n",(unsigned long)&cherche);
 cherche(PAGE_OFFSET);
 cherche((unsigned long)(&cherche));
return(0);}

static void __exit module_end(void) {}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

```
[118418.835824] ffff880000000000
[118418.835827] cherche: ffffffffc0a71000
[118418.835828] qwertyuiop
[118419.318922] -> ffff8800000f4af1: 71 77 65 72 74 79
[118419.318924] qwertyuiop
[118419.318929] -> ffffffffc0a7205e: 71 77 65 72 74 79
```

Nous constatons donc que la signature que nous avons inséré dans notre pilote se retrouve en plusieurs endroits dans la mémoire allouée au pilote, soit en début de mémoire si la recherche commence à `PAGE_OFFSET`, soit bien plus loin si nous commençons la recherche à l'adresse de la première fonction du module. La première occurence correspond probablement à la zone allouée pour stocker les variables du noyau et non au module lui-même.

4.5 Modifier le résultat d'une fonction

Ici nous avons introduit manuellement la signature, mais serions nous capable de modifier le résultat d'une opération en modifiant la séquence d'opcodes ? Le cas est très similaire au précédent puisque données et opcodes sont manipulés indifféremment par le processeur :

```
#include <linux/module.h>

noinline int adieu(int);

long cherche(void)
{unsigned char sct,sctp1,sctp2,sctp3,sctp4;
 unsigned long int offset = (unsigned long)&cherche; // 0xffffffffc0000000 debut du →
     ↪kernel en RAM
 printk(KERN_INFO "%lx\n",PAGE_OFFSET);
 while (offset < ULLONG_MAX)      // recherche ds la memoire allouee au noyau
   {if ((offset%0x1000000)==0x0)
        printk(KERN_INFO "ok : %lx\n",(unsigned long)offset);
    sct  = *(unsigned char*)offset;
    sctp1= *(unsigned char*)(offset+1);
    sctp2= *(unsigned char*)(offset+2);
    sctp3= *(unsigned char*)(offset+3);
    sctp4= *(unsigned char*)(offset+4);
#ifndef __ARMEL__
 if ((sct==0x8d) && (sctp1==0x87) && (sctp2==0x9a) && (sctp3==0x02) && (sctp4==0x00))
#else
 if ((sct==0xe2) && (sctp1==0x80) && (sctp2==0x0f) && (sctp3==0xa6) && (sctp4==0xe2))
#endif
       {printk(KERN_INFO "fonction cherche @ %lx\n",(unsigned long)&cherche);
        printk(KERN_INFO "code offset %lx\n",(unsigned long)offset);
```

```
        return(offset);
       }
    offset++;
   }
 return offset;
}

static int __init module_start(void)
{unsigned long offset;
#ifndef __ARMEL__
 unsigned long original_cr0;
 char s[4]={0x8d,0x87,0x2b,0x02};    // nouvelle sequence d'opcodes x86
#else
 char s[4]={0x8a,0x0f,0x80,0xe2};    // nouvelle sequence d'opcodes ARM
// initialement e2800fa6 pour add r0, r0, #664 ; 0x298
#endif
 offset=cherche();
#ifndef __ARMEL__
 original_cr0 = read_cr0();          // passe la page des appels sys en ecriture
 write_cr0(original_cr0 & ~0x00010000);
#endif
 memcpy((char*)offset,s,4);
#ifndef __ARMEL__
 write_cr0(original_cr0);           // passe la page des appels en lecture seule
#endif
 return(0);
}

noinline int adieu(int k) {return(k+666);}

static void __exit module_end(void)
{printk(KERN_INFO "sortie du module : %d\n",adieu(0));}  // doit renvoyer 0+666

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

Si nous commentons les fonctions de modification du contenu de la fonction `adieu()`, quitter le pilote se traduit bien par l'affichage de `666`, qui est le résultat attendu de l'appel à `adieu(0);`. Comment avons nous effectué la recherche de la séquence d'opcodes et les modifications à faire ? `objdump -dSt module.ko` (ou pour la version ARM, `arm-buildroot-linux-uclibcgnueabihf-objdump`) fournit la séquence de mnémoniques et d'opcodes associés à un exécutable désassemblé. Pour intel, nous trouvons

```
a0:   e8 00 00 00 00          callq  a5 <adieu+0x5>
a5:   8d 87 9a 02 00 00       lea    0x29a(%rdi),%eax
ab:   c3                      retq
```

où nous retrouvons bien 0x29a=666 (que nous remplaçons par 0x22b=555), tandis que pour ARM nous observons

```
cc:   e2800fa6        add     r0, r0, #664    ; 0x298
d0:   e2800002        add     r0, r0, #2
```

(ARM découpe son argument en 0xa6=0x298/4 – car pos=0x0f – suivi de 2 – car pos=0x00 – afin de manipuler des arguments sur 32 bits par morceaux de 12 bits [11]. Si nous voulons renvoyer 555, alors nous devons modifier les deux séquences d'opcodes, de e2800fa6 en e2800f8a pour `add r0, r0, #552`, et e2800002 en e2800003 pour `add r0, r0, #3`). Les séquences d'opcodes qui nous intéressent sont donc dans le premier cas {0x8d,0x87,0x9a,0x02,0x00} et dans le second cas {0xe2,0x80,0x0f,0xa6,0xe2}.

11. http ://www.peter-cockerell.net/aalp/html/ch-3.html

Organisation de la mémoire contenant les instructions : on notera que, alors que le tableau `s` qui contient la nouvelle séquence d'opcodes présente des données dans le même ordre que la sortie de `objdump` sur x86, nous avons inversé la séquence d'octets pour ARM. Bien que les deux architectures soient *little endian* – notre configuration de `buildroot` sélectionne `BR2_ENDIAN="LITTLE"` – et placent donc l'octet de poids le plus faible à l'adresse la plus faible (donc "à l'envers" si on lit de gauche à droite), la séquence des instructions dans une architecture CISC telle que Intel x86(_64) – instructions et arguments de taille variable – se lit octet par octet en incrémentant les adresses. Ainsi, les octets successifs que nous plaçons en mémoire suivent la séquence fournie par `objdump`. Au contraire, une architecture RISC comme ARM aligne nécessairement ses instructions (incluant l'argument) sur des multiples de 4-octets, et les instructions – de taille fixe de 4 octets – sont agencées elles aussi en little endian. On s'en convaincra avec le petit exemple ci-dessous (exécuté en utilisateur) :

```
#include <stdio.h>
int main()
{short s=0x1234;
 char *c=(char*)&s;
 unsigned long offset=(unsigned long)&main,k;
 printf("%hhx %hhx\n",*c,*(c+1));
 for (k=0;k<16;k++) printf("%hhx ",*(char*)(offset+k));
 printf("\n");
}
```

qui renvoie d'une part l'organisation d'un mot de 16 bits en mémoire (on vérifie ainsi être sur des architectures *little endian*), puis les premiers opcodes de la fonction `main`. Sur Intel x86_64, nous obtenons

```
34 12
55 48 89 e5 48 83 ec 20 66 c7 45 e6 34 12 48 8d
```

tandis que `objdump -d endian.pc | grep -A5 \<main\>` nous dit

```
0000000000400576 <main>:
  400576:       55                      push   %rbp
  400577:       48 89 e5                mov    %rsp,%rbp
  40057a:       48 83 ec 20             sub    $0x20,%rsp
  40057e:       66 c7 45 e6 34 12       movw   $0x1234,-0x1a(%rbp)
  400584:       48 8d 45 e6             lea    -0x1a(%rbp),%rax
```

Les séquences d'opcodes sont les mêmes, et nous constatons bien la taille variable des instructions et de leurs arguments, signature d'une architecture CISC. Au-constraire sur processeur A13 (cœur Cortex A8 d'architecture ARM v7 [15]), nous observons

```
# ./endian.arm
34 12
0 48 2d e9 4 b0 8d e2 10 d0 4d e2 34 32 1 e3
```

qui propose une présentation en *little endian* – donc commençant par les arguments et finissant par l'instruction – des opcodes fournis par `objdump` sous forme d'instruction suivie des arguments (lecture naturelle pour un développeur) par

```
$ arm-buildroot-linux-uclibcgnueabihf-objdump -d endian.arm | grep -A4 \<main\>
000084ac <main>:
    84ac:       e92d4800        push    {fp, lr}
    84b0:       e28db004        add     fp, sp, #4
    84b4:       e24dd010        sub     sp, sp, #16
    84b8:       e3013234        movw    r3, #4660       ; 0x1234
```

Nous constatons au chargement puis déchargement de ce module les messages suivants par `dmesg` :

```
[24254.596117] ffff880000000000
[24254.596120] fonction cherche @ ffffffffc09f2000
[24254.596121] code offset ffffffffc09f20a5
[24262.343786] sortie du module : 555
```

qui prouve bien que les instruction de `adieu()` ont été modifiées pour renvoyer 555 au lieu de 666.

4.6 Modifier le résultat d'une fonction du noyau

Cependant, ce cas est trivial car le symbole recherché se trouve dans le même module que celui chargé de modifier le contenu de la mémoire. Nous n'avons pas réussi à balayer de façon systématique toute la mémoire du noyau car nous nous heurtons à la protection mémoire de la MMU qui interdit d'accéder à des pages non-allouées [5] : des outils tels que sont `fmem` [12] ou LiME [13] [16] y parviennent, mais au prix de nombreux tests trop longs à expliciter ici.

Cependant, avons nous réellement besoin de scanner toute la mémoire pour trouver le code associé à une fonction implémentée par un autre pilote ou par le noyau ? De la même façon que nous avions cherché la table des appels systèmes en partant de l'appel système `sys_close()`, nous allons limiter

12. `http://hysteria.cz/niekt0/`
13. `https://github.com/504ensicsLabs/LiME`

notre recherche à la zone mémoire que nous supposons contenir la fonction attaquée. Comme avec les appels systèmes, un point un peu délicat tient au fait que chaque module n'exporte pas ses symboles (noter que par ailleurs presque toutes les définitions de fonctions des modules du noyau sont préfixées de `static`, donc avec une portée locale au fichier source uniquement). Illustrons ce concept en modifiant le contenu de `/proc/cpuinfo` par modification de la zone mémoire appelée lors de l'affichage du contenu de ce pseudo-fichier :

1. dans les sources du noyau, nous constatons que `/proc/cpuinfo` est rempli par les fonctions de `linux-4.4.2/arch/x86/kernel/cpu/proc.c` et en particulier la fonction `show_cpu_info()` qui affiche la fréquence en MHz. Cependant, aucun symbole des fonctions de ce fichier source n'est exporté,
2. nous constatons dans `/proc/kallsyms` que `show_cpu_info` se trouve en mémoire juste apres `x86_match_cpu`, qui lui est un symbole exporté : celà est cohérent avec le fait que le code source de cette nouvelle fonction se trouve dans le même répertoire

```
# grep -A2 x86_match_cpu /proc/kallsyms | head -3
ffffffff8103e5d0 T x86_match_cpu
ffffffff8103e670 t c_stop
ffffffff8103e680 t show_cpuinfo
```

3. nous trouvons le prototype de `x86_match_cpu` dans `linux-4.4.2/arch/x86/include/asm/cpu_device_id.h` qui permettra donc la compilation de notre pilote qui recherchera l'emplacement en mémoire des fonctions que nous désirons manipuler,
4. finalement, partant d'un point de départ d'une page mémoire allouée au noyau, nous recherchons la séquence d'opcodes, ou dans notre cas la chaîne de caractères représentative de la manipulation à effectuer en mémoire, et écrasons le contenu de cette zone mémoire avec notre nouveau message.

Initialement, le fichier `/proc/cpuinfo` nous informe des performances de nos processeurs par les messages

```
# grep Hz /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1750.835
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1899.421
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1813.601
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
cpu MHz         : 1898.914
```

Nous chargeons le module dont le code source est

```
#include <linux/module.h>
#include <asm/cpu_device_id.h> // prototype de x86_match_cpu

long cherche(void)
{unsigned char sct,sctp1,sctp2,sctp3,sctp4;
 unsigned long int offset = (unsigned long)&x86_match_cpu; // fonction proche de l'→
    ↪appel qu'on va modifier
 printk(KERN_INFO "%lx\n",offset);
 while (offset < ULLONG_MAX)      // recherche ds la memoire allouee au noyau
   {sct  = *(unsigned char*)offset;
    sctp1= *(unsigned char*)(offset+1);
    sctp2= *(unsigned char*)(offset+2);
    sctp3= *(unsigned char*)(offset+3);
    sctp4= *(unsigned char*)(offset+4);
 if ((sct=='c') && (sctp1=='p') && (sctp2=='u') && (sctp3==' ') && (sctp4=='M'))
      {printk(KERN_INFO "code offset %lx\n",(unsigned long)offset);
       printk(KERN_INFO "... %hhx %hhx %hhx\n",(unsigned char)sctp4,*(unsigned char*)(→
           ↪offset+5),*(unsigned char*)(offset+6));
       return(offset);
      }
    offset++;
   }
 return offset;
}

static int __init module_start(void)
{unsigned long offset;
 unsigned long original_cr0;
 char s[16]="toto THz\t: 42 \0";  // nouveau message a afficher
 offset=cherche();
 original_cr0 = read_cr0();          // passe la page des appels sys en ecriture
 write_cr0(original_cr0 & ~0x00010000);
 memcpy((char*)offset,s,16);
 write_cr0(original_cr0);           // passe la page des appels en lecture seule
 return(0);
}
```

```
static void __exit module_end(void) {printk(KERN_INFO "sortie du module");}

module_init(module_start);
module_exit(module_end);
MODULE_LICENSE("GPL");
```

pour désormais obtenir

```
# insmod 3mymod.ko
# grep Hz /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
model name      : Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
toto THz        : 42  cache size        : 3072 KB
```

Nous voici donc en possession d'un ordinateur qui ne fonctionne plus sur des "cpu" mais sur des "toto", cadencés à 42 THz. On prendra soin de ne pas recharger une seconde fois ce pilote, puisque maintenant la chaîne recherchée "cpu M" n'existe plus dans `/proc/cpuinfo` et le balayage de la mémoire heurtera immanquablement une page non-allouée au noyau, se traduisant par un accès illégal et un pilote qui ne pourra plus être détaché du noyau.

Au chargement du pilote, `dmesg` nous informe des adresses des divers emplacements qui nous intéressent :

```
[  175.403604] ffffffff8103e5d0
[  175.416773] code offset ffffffff817cd383
[  175.416777] ... 4d 48 7a
```

en commençant par l'emplacement de la fonction `x86_match_cpu` puis de la chaîne de caractères recherchée. Ces informations sont cohérentes une fois de plus avec `/proc/kallsyms`. Nous avons donc démontré notre capacité à intervenir sur la zone mémoire allouée à une partie du noyau autre que notre propre module. Le passage de la manipulation des chaînes de caractères affichées au contenu de la mémoire (variables) ou instructions n'est plus qu'une question d'analyse du code désassemblé du noyau pour identifier la séquence d'opcodes ou l'emplacement de la variable à modifier.

Cette approche a par exemple été utilisée, il y a maintenant fort longtemps, par l'auteur pour débloquer quelques fonctions du logiciel propriétaire Framework de contrôle de potentiostat PC3-300 de Gamry : accéder à certaines fonctionnalités se traduisait par un message d'erreur. Une fois le logiciel désassemblé, l'adresse du message d'erreur est recherchée, puis l'instruction qui prend en argument cette adresse. Nous pouvons supposer que le bout de code qui fait appel au message d'erreur est aussi celui chargé d'autoriser l'accès à ces fonctionnalité. La condition donnant l'autorisation étant identifiée, il suffit de modifier l'opcode de saut avec une condition (`je` : *jump if equal*) par son opposé (`jne` : *jump if not equal*) pour débloquer la fonctionnalité du logiciel. Cette approche est d'autant plus valide à l'époque actuelle où de nombreux instruments scientifiques sont commercialisés avec toutes les fonctions matérielles installées et le fabriquant demande un paiement additionnel pour simplement débloquer la fonctionnalité logicielle.

5 Conclusion

Nous avons présenté quelques méthodes de modification de la mémoire d'un système informatique exécutant GNU/Linux en insérant un module chargé soit de trouver les appels systèmes et rediriger les appels vers nos propres fonctions, soit d'écraser les fonctions d'origine pour les remplacer avec nos propres instructions. Au delà de la capacité à modifier le fonctionnement de systèmes fournis uniquement sous forme de binaire, est-ce que ces modifications peuvent importer au commun des utilisateurs ? Il nous semble qu'avec la prolifération des systèmes embarqués sous GNU/Linux (box de connexion internet, récepteurs GPS, disques durs) avec l'absence du respect des concepts de base de sécurité (absence de compte utilisateur avec seule une connexion root, mots de passe en dur dans l'image flashée sur le système embarqué [17, 18]), une compréhension des méthodes de corruption du noyau peut être utile. La mode des objets connectés, ou IoT, va probablement encore empirer notre exposition aux attaques en éliminant la couche physique protégeant l'accès au périphérique – les liaisons radiofréquences étant accessibles à tout interlocuteur à proximité du périphérique – et il n'y a aucun doute que les attaques se multiplieront dans cette direction, avec la capacité à cacher ses traces selon certains des mécanismes discutés ici. Y a-t-il une perspective d'amélioration de la protection contre ces attaques ? Nous plaçant au plus bas niveau du système d'exploitation, où seule la MMU (matériel) handicape nos capacités d'action, il est peu probable qu'une solution stable existe. Les diverses versions de Linux palient à quelques méthodes disponibles sur internet qu'un utilisateur qui se contente de copier sans comprendre ne pourra re-appliquer, mais nous avons vu ici toutes les étapes pour reprendre étape par étape la démarche d'identification de la zone mémoire à modifier, information qui est nécessairement accessible si le système d'exploitation doit pouvoir

fournir le service pour lequel il est conçu. [19] discute de la sécurité amenée par la redistribution aléatoire des accès mémoire, qui rompt au moins partiellement l'hypothèse de proximité des appels systèmes ou des structures de données exportées par rapport aux emplacements recherchés. La principale limitation citée par [20] est la nécessité de reproduire un environnement local duppliquant la configuration du noyau attaqué sur le système distant : tant qu'un noyau de `kernel.org` est utilisé, l'obtention de la version (`uname -a`) et de la configuration (`/boot/config*`) devraient rendre cette étape faisable.

Remerciements

Je remercie S. Guinot (association Sequanux, Besançon) et F. Tronel (CentraleSupélec/Inria, Rennes) pour avoir répondu à mes questions et orienté mes recherches au cours de cette étude.

Références

[1] M.H. Ligh, A. Case, J. Levy & A. Walters, *The Art of Memory Forensics – Detecting Malware and Threats in Windows, Linux, and Mac Memory*, Wiley (2014)

[2] D.J. Barrett, R.E. Silverman & R.G. Byrnes, *Linux Security Cookbook*, O'Reilly (2003)

[3] R.J. Hontanon, *Linux Security*, Sybex (2001)

[4] R. ONeill, *Learning Linux Binary Analysis*, Packt Publishing (2016)

[5] madsys, *Finding hidden kernel modules (the extrem way)*, Phrack **61** (2003)

[6] D.-H You, *Android platform based linux kernel rootkit*. Phrack **68** (2011), et 6th IEEE International Conference on Malicious and Unwanted Software (MALWARE) (2011)

[7] Silvio Cesare, *Unix viruses* à `https://www.win.tue.nl/~aeb/linux/hh/virus/unix-viruses.txt`

[8] B. Hatch, J. Lee & G. Kurtz, *Hacking Linux exposed : Linux security secrets & solutions*, Osborne/McGraw-Hill (2001)

[9] sd & devik, *Linux on-the-fly kernel patching without LKM*, Phrack **58** (2001)

[10] Ce point a rapidement été abordé, naïvement, en annexe C de la thèse de l'auteur disponible à `https://hal.archives-ouvertes.fr/tel-00509641/document` dans le contexte d'un coprocesseur Z80 sur carte ISA de PC.

[11] M.A. Ludwig, *The Little Black Book of Computer Viruses – Volume 1 : the basic technology* American Eagle Publications (1990), disponible à `http://www.cin.ufpe.br/~mwsa/arquivos/THE_LITTLE_BLACK__BOOK_OF_C.PDF` – le lecteur se souviendra peut être qu'à la sortie de cet ouvrage – *Naissance d'un virus* (Addison & Wesley) en 1993, nombre de clients – y compris l'auteur – accompagnaient l'achat de cet ouvrage d'une introduction à l'assembleur x86, avec rupture de stock de *L'assembleur facile* aux éditions Marabout (1989) !

[12] M.A. Ludwig, *The Little Black Book of Computer Viruses – Computer Viruses, Artificial Life and Evolution* (1993)

[13] M.A. Ludwig, *The Giant Black Book of Computer Viruses* (1995)

[14] A. Lineberry, *Malicious Code Injection via /dev/mem*, BlackHat Europe (2009)

[15] *ARMv7-M Architecture Reference Manual* (2010) à `http://www.pjrc.com/teensy/beta/DDI0403D_arm_architecture_v7m_reference_manual.pdf`, l'accès à la version officielle sur `http://infocenter.arm.com` nécessitant de s'enregistrer : section *A3.3.1 Control of endianness in ARMv7-M*, nous apprenons que "The endianness setting only applies to data accesses. Instruction fetches are always little endian."

[16] P. Wächter & M. Gruhn, *Practicability study of Android volatile memory forensic research*, IEEE Workshop on Information Forensics and Security (WIFS), 2015

[17] C. Heffner, *Exploiting Network Surveillance Cameras Like a Hollywood Hacker*, Black Hat (2013), disponible à `https://www.youtube.com/watch?v=B8DjTcANBx0`

[18] GTVHacker, *Hack All The Things : 20 Devices in 45 Minutes*, Defcon 22 (2014), disponible à `https://www.youtube.com/watch?v=h5PRvBpLuJs`

[19] D.M Stanley, D. Xu & E.H. Spafford, *Improved kernel security through memory layout randomization* IEEE 32nd International Performance Computing and Communications Conference (IPCCC), 2013

[20] J. Stüttgen, & M. Cohen, *Robust Linux memory acquisition with minimal target impact*, Digital Investigation **11** (2014), pp.S112–S119 à `http://www.sciencedirect.com/science/article/pii/S174228761400019X`