

Déréférencement de pointeur NULL dans le noyau Linux - Protection du noyau avec seccomp-bpf

29 novembre 2022

Ce TP vous permet de mettre en oeuvre une exploitation d'une vulnérabilité présente au sein du noyau de système d'exploitation, en l'occurrence ici Linux 5. Vous pourrez donc, à travers un exemple, comprendre ce type de vulnérabilité et mesurer les différentes conséquences que peut provoquer l'exploitation d'une telle vulnérabilité.

Travail sur machine virtuelle Téléchargez et décompressez la machine virtuelle dédiée au travail pratique.

```
$ cd
$ wget -c 'http://flash.enseeiht.fr/bemorgan/arch-secu-os.tgz'
$ tar xvzf arch-secu-os.tgz
$ cd arch-secu-os
$ ./start.sh
```

1 Création d'un module vulnérable

La vulnérabilité que vous pourrez analyser est insérée dans un module du noyau. Le module est un module de type **device character**, et il permet simplement d'obtenir, à un instant donné le nombre de processus qui s'exécutent sur le système. Vous pouvez lister les types de fichiers en mode caractère supportés par votre système à l'aide du fichier suivant :

```
$ cat /proc/devices
```

Une fois réalisé et chargé, votre module sera normalement listé parmi les modules de ce fichier.

Pour accéder aux fonctions de ce module, vous devez donc passer par un device de type **character** et utiliser les fonctions d'ouverture (**device_open**), de fermeture (**device_release**), de lecture (**device_read**) et de contrôle (**device_ioctl**). Un tel device est simplement un fichier "particulier", créé grâce à la commande **mknod** et auquel est associé un **major number**, qui fait le lien avec le module que vous allez réaliser.

Voici le code de ce module.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/cred.h>
#include <linux/sched/signal.h>
#include <asm/uaccess.h>
#include <linux/uaccess.h>

#include <linux/ioctl.h>

#include "module_device.h"

static int major;
static short int in_use = 0;

/**
 * Attention ce module ne supporte pas une utilisation concurrente !
 */

/* Structure interne */

struct proc_info {
    int nb_proc;
    int (*update)(unsigned long);
} __attribute__((packed));

static struct proc_info *proc;
static int calcul(unsigned long param) {
    int nb=0;
    struct task_struct * task;

    for_each_process(task) {
        nb++;
    }

    return nb;
}

/* Lit le nombre de processus */
static ssize_t device_read(struct file *filp, char *buffer,
    size_t length, loff_t * offset) {
```

```
int i = sizeof(int);

if (proc == NULL) {
    return 0;
}
put_user(proc->nb_proc, (int *)buffer);

return i;
}

static ssize_t device_write(struct file *filp, const char *buffer,
    size_t length, loff_t *offset) {
    unsigned long nb;
    nb = copy_from_user((int *)&(proc->nb_proc), (int *)buffer, length);
    get_user(proc->nb_proc, (int *)buffer);
    return nb;
}

/* Fonction speciales associees au fichier */
static long device_ioctl(struct file *file,
    unsigned int ioctl_num, unsigned long ioctl_param) {

    switch (ioctl_num) {
        case IOCTL_RESET:
            if (proc == NULL) {
                proc = kmalloc(sizeof(struct proc_info), GFP_KERNEL);
            }
            proc->nb_proc = 0;
            proc->update = calcul;
            break;

        case IOCTL_UPDATE:
            proc->nb_proc = proc->update(ioctl_param);
            break;
        default:
            return -1;
    }
    return 0;
}

/* Ouverture du fichier */
static int device_open(struct inode *inode, struct file *file) {
```

```
    if (in_use) {
        return -EBUSY;
    }

    in_use++;
    proc = NULL;
    try_module_get(THIS_MODULE);
    return 0;
}

/* Fermer le fichier */
static int device_release(struct inode *inode, struct file *file) {
    in_use--;
    if (proc != NULL) {
        kfree(proc);
        proc = NULL;
    }
    module_put(THIS_MODULE);

    return 0;
}

/* Structure utilisee pour enregistrer le device */
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .unlocked_ioctl = device_ioctl,
    .open = device_open,
    .release = device_release
};

/* Fonction executee au chargement */
int init_module(void) {
    // Enregistrement du device
    major = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);
    return (major < 0 ? major : 0);
}

/* Execute au echargement du module unloaded */
void cleanup_module(void) {
    if (major >= 0) {
        unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
    }
}
```

```
    if (proc!=NULL) {  
        kfree(proc);  
        proc = NULL;  
    }  
}
```

Comme vous pouvez le constater, ce module gère essentiellement une variable globale `proc` de type `struct proc_info`, qui contient deux champs : `nb_proc`, qui est le nombre de processus s'exécutant sur le système à un instant donné et `update`, un pointeur sur la fonction permettant de mettre à jour ce nombre de processus.

Ce module contient essentiellement deux fonctions, explicitées ci-après.

1.1 La fonction `device_read`

Cette fonction est appelée lorsqu'un `read` est effectué sur un fichier associé à notre module. Cette fonction doit recopier dans la variable `buffer` en espace utilisateur la valeur de la variable `nb_proc` (qui est en espace noyau). Pour cela, il est nécessaire d'utiliser la fonction `put_user`.

1.2 La fonction `device_ioctl`

La fonction `device_ioctl` du fichier permet plusieurs commandes spéciales. La commande `IOCTL_RESET` doit être appelée en premier et alloue un espace mémoire pour `proc` si ce dernier est nul. Ensuite, cette fonction réinitialise `nb_proc` à la valeur 0 et `update` à l'adresse de la fonction qui permet de mettre à jour ce nombre : la fonction `calcul`. L'allocation mémoire dans le kernel s'exécute grâce à la fonction `kmalloc(<taille>, GFP_KERNEL)`.

La commande `IOCTL_UPDATE` se contente simplement de mettre à jour `nb_proc` en invoquant la fonction `update`.

La fonction `calcul` est le coeur de notre module puisque c'est cette fonction qui permet de calculer le nombre de processus sur le système à un instant donné.

Vous pouvez vous documenter sur le code du noyau linux soit en le téléchargeant sur le site <https://www.kernel.org> ou bien en explorer sa version en ligne aux références croisées sur le site <https://elixir.bootlin.com>.

2 Travail à réaliser

Dans une première partie de ce TP nous allons commencer par utiliser correctement le module vulnérable pour ensuite l'attaquer et exploiter la vulnérabilité présente. Dans une seconde partie, nous allons mettre en pratique une mécanique de filtrage des appels avec `seccomp-bpf` pour se protéger d'interactions malveillantes de programmes corrompus par un attaquant, dans un cadre de tolérance aux intrusions. Il est proposé dans un premier temps de limiter les interactions système de

l'application utilisant notre module pour l'empêcher d'exploiter la vulnérabilité. Enfin, nous rendrons tolérantes aux intrusions une série d'applications réelles toujours à l'aide de `seccomp-bfp`.

2.1 Identification de la vulnérabilité

Vous devez simplement comprendre le code à l'aide des explications et du code fourni ci-dessus et identifier la vulnérabilité présente. De quel type est-elle ?

2.2 Test “normal” du module

Réalisez un programme qui utilise correctement ce module sans exploiter la vulnérabilité.

2.3 Exploitation de la vulnérabilité

Nous allons à présent étudier comment exploiter la vulnérabilité de différentes façons en suivant scénario d'attaque décrit dans la figure 1.

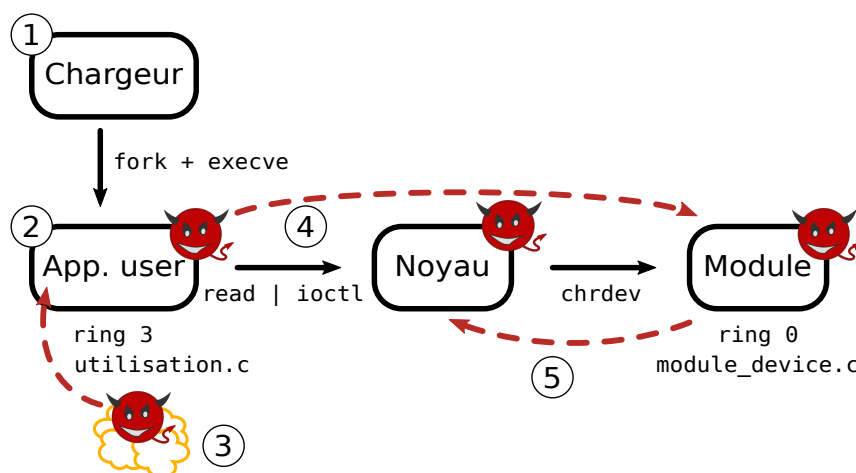


FIGURE 1 – Intrusion dans le noyau linux à l'aide du module vulnérable

2.3.1 Défaillance du module

il est possible de commencer par provoquer une défaillance du programme appelant le module. Pour cela, il suffit simplement d'exploiter la vulnérabilité. Ecrire un programme qui provoque cette défaillance.

2.3.2 Exécution d'une fonction du noyau

Dans un second temps, nous proposons de profiter de cette vulnérabilité de façon à faire exécuter à notre module un code que nous choisissons. Pour cela, nous allons

simplement essayer de détourner le module en lui faisant exécuter la fonction **panic** présente dans le noyau.

Le principe de l'exploitation est le suivant. Il suffit que le programme exploit soit capable de mapper l'adresse 0 et d'insérer à cette adresse des données correspondant à la structure **proc_info**. Ainsi, si à l'adresse 4, on insère un pointeur de fonction, cette fonction sera donc celle qui sera exécutée dans la commande **IOCTL_UPDATE**, lorsque elle est appelée alors que **proc** est nul.

On vous propose de placer à l'adresse 4, l'adresse de la fonction **panic** du noyau linux, qui provoque un arrêt complet du noyau et donc du système. Malheureusement, l'adresse de fonction n'est pas accessible directement depuis l'espace utilisateur. On peut néanmoins calculer son adresse grâce à la présence du fichier **/proc/kallsyms** sur une machine Unix.

Pour que l'exploitation fonctionne, (mapper l'adresse 0 est par défaut interdit dans notre distribution Linux), il faut modifier le fichier **/proc/sys/vm/mmap_min_addr** et insérer 0 dans ce fichier.

1. Ecrire le code de la fonction **get_kernel_sym** qui prend en paramètre un nom de fonction du noyau et qui renvoie son adresse, en tirant partie du fichier **/proc/kallsyms**.
2. A l'aide de la fonction **mmap**, mapper l'adresse 0 dans votre programme et positionner à l'adresse 4 l'adresse de la fonction **panic** du noyau. On pourra par exemple utiliser ainsi la fonction **mmap** :

```
mmap(0, 4096, PROT_READ | PROT_WRITE | PROT_EXEC,  
     MAP_ANON | MAP_PRIVATE | MAP_FIXED, 1, 0);
```

3. Exécutez ensuite la commande **IOCTL_UPDATE** et constatez le kernel panic.
4. Rebootez!!

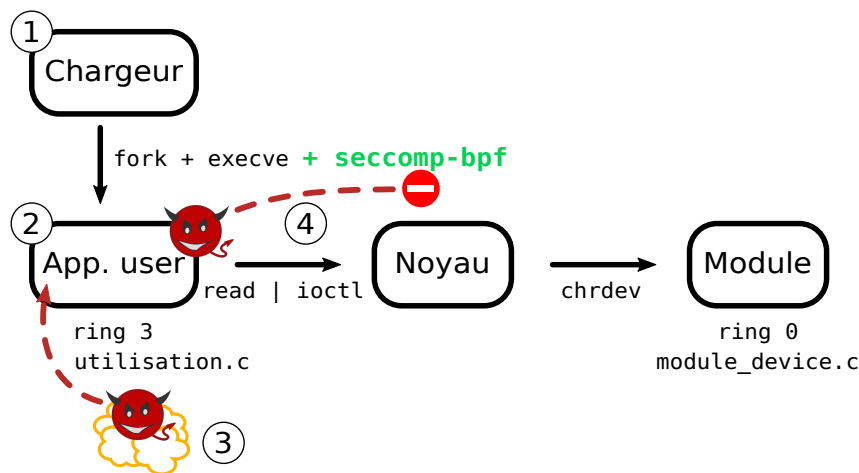
Quelles sont les mesures de protection que nous avons du désactiver pour que cette exploitation fonctionne ?

2.4 Premiers pas vers seccomp-bpf

Service de protection de **seccomp-bpf** : "je n'ai pas confiance en mes interactions utilisateur → je limite mes privilèges à ce dont j'ai strictement besoin, en espérant que cela limite les conséquences d'une intrusion".

Considérons que l'attaquant n'a pas accès à de l'exécution de code arbitraire sur notre machine, mais qu'il est en mesure de corrompre une application déjà chargée à distance. Les applications potentiellement vulnérables sont chargées et exécutées après l'application des filtres d'appels systèmes. Un attaquant pourrait compromettre le code de l'application, mais la conséquence de la compromission serait limitée étant donné le filtrage des appels systèmes (figure 2).

On vous propose maintenant de faire une première utilisation de **seccomp-pbf**, présenté dans le cours, afin de rendre l'exploitation du module vulnérable impossible,

FIGURE 2 – Tolérance aux intrusions à l'aide de `seccomp-bpf`

même si la vulnérabilité est encore présente dans le noyau. L'exploitation ci-dessus est basée sur le fait que l'attaquant peut mapper l'adresse 0. Il utilise pour cela l'appel système `mmap`. Nous allons donc utiliser `seccomp-bpf` pour contrôler l'utilisation de cet appel.

2.4.1 Rappels de cours et libseccomp

Pour rappel, `seccomp-bpf` utilise une machine virtuelle aux instructions basiques dérivées de *Berkeley Packet Filter* (BPF). Les machines virtuelles généralistes supportent un nombre limité d'instructions¹. L'application de BPF aux filtres d'appels systèmes réduit encore plus ce nombre². Voici un listing de certaines restrictions de la machine virtuelle BPF :

- l'ALU est très simple : `ADD` ; `MUL` ; `OR` ; ...
- les branchements interdisent les boucles
- la mémoire est seulement accessible indirectement : `LOAD` ; `STORE`
 - la classique contient 16 cases
 - les arguments de l'appel système sont accessibles à l'aide de la structure `struct seccomp_data` depuis un autre espace mémoire dédié (`BPF_ABS`)³.
- les scripts se terminent par 5 sorties possibles :
 - `SECCOMP_RET_ALLOW` : appel autorisé

Pour simplifier l'utilisation de `seccomp-bpf`, une bibliothèque permettant de compiler des scripts BPF a été développée. Elle s'utilise en incluant dans le code de l'application `#include <seccomp.h>`, puis principalement à l'aide de 2 fonctions :

1. <https://elixir.bootlin.com/linux/v5.4.3/source/net/core/filter.c#L941>
 2. <https://elixir.bootlin.com/linux/v5.4.3/source/kernel/seccomp.c#L174>
 3. <https://elixir.bootlin.com/linux/v5.4.3/source/include/uapi/linux/seccomp.h#L59>

```
# seccomp_init met en place le filtrage apres l'execution de cette
# fonction, plus aucun appel systeme n'est possible. Mode de
# filtrage strict
seccomp_filter_ctx scmp = seccomp_init(SCMP_ACT_KILL);

# seccomp_rule_add peut etre utilisee pour autoriser un appel
# systeme quelque soit ses parametres
seccomp_rule_add(scmp, SCMP_ACT_ALLOW, SCMP_SYS(close), 0);

# seccomp_rule_add peut aussi etre utilisee pour autoriser un appel
# systeme sous reserve que certains parametres de l'appel soient
# constraints
seccomp_rule_add(scmp, SCMP_ACT_ALLOW, SCMP_SYS(open), 1,
    SCMP_A1(SCMP_CMP_EQ, O_RDWR));

# seccomp_load compile et charge le script dans le noyau
if (seccomp_load(scmp) != 0) {
    fprintf(stderr, "Failed to load the filter in the kernel\n");
    return -1;
}
```

Un bon point d'entrée pour de la documentation sur cette bibliothèque est le manuel : `man 3 seccomp_rule_add`. Les pages de manuel de `seccomp-bpf` du noyau sont disponibles ici : `man 2 seccomp`.

Lorsque vous activez `seccomp-bpf` en mode strict et que les règles sont violées, le noyau vous indique dans le journal quelle en est la raison :

```
$ dmesg | grep syscall
[ 7492.912646] comm="seccomp-1" syscall=59 ip=0x7fcb78fe9d1b
[ 7507.122172] comm="seccomp-1" syscall=257 ip=0x7f8d6ad9d15b
```

En utilisant cette stratégie, vous pouvez apprendre le comportement du programme et autoriser au fur et à mesure ce qui est nécessaire d'autoriser.

La liste des appels systèmes est disponible à cette page : https://elixir.bootlin.com/linux/v4.0/source/arch/x86/syscalls/syscall_64.tbl

2.4.2 Travail à faire

Protégez le processus utilisation en trois étapes :

1. modifiez directement le code de `utilisation.c` pour y insérer le filtre;
2. développez un chargeur d'application `chargeur.c` qui va configurer le filtre puis exécuter `utilisation` avec `execve`;

- développez une bibliothèque partagée `filtre.so` dans laquelle le filtre sera déclaré dans un constructeur, c'est à dire exécutée avant le `main` du programme `utilisation.c`. Ainsi nous pouvons protéger `utilisation` sans modifier son code!

Montrer dans les trois cas que `seccomp-bpf` peut empêcher le programme malveillant de parvenir à ces fins.

Déclaration d'un constructeur dans un fichier C :

```
#include <stdio.h>

__attribute__((constructor))
static void installe_filtre(void) {
    printf("Constructeur filtre BPF !\n");
}
```

Compilation d'une bibliothèque partagée :

```
$ gcc -o filtre.so -shared ...
```

Chargement d'une bibliothèque et exécution d'un constructeur avant l'exécution du `main` d'un programme :

```
$ LD_LIBRARY_PATH=. LD_PRELOAD=filtre.so ./utilisation
```

Vous pouvez récupérer le code du filtre BPF compilé par `libseccomp` avec la fonction `seccomp_export_bpf`. Consultez la page de manuel et analysez la sortie de l'export à l'aide du décompilateur BPF installé sur la machine virtuelle : `seccomp-tools`⁴.

Exemple de script BPF désassemblé avec cet outil :

```
seccomp-tools disasm /tmp/out.bpf
line  CODE  JT   JF      K
=====
0000: 0x20 0x00 0x00 0x00000004  A = arch
0001: 0x15 0x00 0x13 0xc000003e  if (A != ARCH_X86_64) goto 0021
0002: 0x20 0x00 0x00 0x00000000  A = sys_number
0003: 0x35 0x00 0x01 0x40000000  if (A < 0x40000000) goto 0005
0004: 0x15 0x00 0x10 0xffffffff  if (A != 0xffffffff) goto 0021
0005: 0x15 0x0e 0x00 0x00000000  if (A == read) goto 0020
0006: 0x15 0x0d 0x00 0x00000001  if (A == write) goto 0020
0007: 0x15 0x0c 0x00 0x00000003  if (A == close) goto 0020
0008: 0x15 0x0b 0x00 0x00000005  if (A == fstat) goto 0020
0009: 0x15 0x0a 0x00 0x00000008  if (A == lseek) goto 0020
```

4. <https://github.com/david942j/seccomp-tools>

```
0010: 0x15 0x09 0x00 0x00000009  if (A == mmap) goto 0020
0011: 0x15 0x08 0x00 0x0000000a  if (A == mprotect) goto 0020
0012: 0x15 0x07 0x00 0x0000000b  if (A == munmap) goto 0020
0013: 0x15 0x06 0x00 0x0000000c  if (A == brk) goto 0020
0014: 0x15 0x05 0x00 0x00000015  if (A == access) goto 0020
0015: 0x15 0x04 0x00 0x0000003b  if (A == execve) goto 0020
0016: 0x15 0x03 0x00 0x0000009e  if (A == arch_prctl) goto 0020
0017: 0x15 0x02 0x00 0x000000e4  if (A == clock_gettime) goto 0020
0018: 0x15 0x01 0x00 0x000000e7  if (A == exit_group) goto 0020
0019: 0x15 0x00 0x01 0x00000101  if (A != openat) goto 0021
0020: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0021: 0x06 0x00 0x00 0x00000000  return KILL
```

2.5 Exemples plus complets de seccomp-bpf

Consolidez votre compréhension en réalisant deux *sandboxes* `seccomp-bpf` pour le programmes `/bin/echo` et `/usr/bin/nc`. Comparez les résultats avec un programme compilé statiquement et dynamiquement (`$ gcc -static -o ...`)

Spécification `/bin/echo` :

- Écrire uniquement sur la sortie standard
- Chaînes de caractères d'une longueur maximale de 10

Spécification `/usr/bin/nc` :

- Écrire uniquement sur la sortie standard
- Lecture uniquement sur la sortie standard

3 Conclusion

Vous pouvez maintenant aller dormir ...

4 Références

<https://ajxchapman.github.io/linux/2016/08/31/seccomp-and-seccomp-bpf.html>