Analyse dynamique depuis l'espace noyau avec Kolumbo

Julien Desfossez, Justine Dieppedale et Gabriel Girard
jd(@)revolutionlinux.com

Revolution Linux, Sherbrooke, Québec, Canada,
Université de Sherbrooke, Département des sciences,
Sherbrooke, Québec, Canada

Résumé La plupart des logiciels malveillants à l'heure actuelle détectent les debuggers traditionnels, ainsi lors de l'analyse, ils changent de comportement ce qui rend la tâche beaucoup plus complexe. Nous allons présenter dans cette partie les fonctionnalités mises en place dans le module noyau Kolumbo pour simplifier l'analyse de tels logiciels. Quatre approches sont prévues afin de laisser le plus de choix à l'analyste : le suivi des appels systèmes et des arguments, la récupération du contenu de la mémoire virtuelle, l'insertion de pseudo points d'arrêts et le contournement d'un système anti-analyse basé sur `ptrace`. Le module a été conçu pour avoir un impact minimal sur le système et pour limiter les risques de détection par le programme cible, toutefois il n'est pas conçu pour analyser des programmes ayant un accès au noyau.

1 Étude des systèmes anti-analyse

La plupart des malwares actuels implémentent des systèmes de protections contre l'analyse. Ces systèmes ont pour but de complexifier la tâche des chercheurs. On peut diviser l'analyse d'un programme en deux catégories : l'analyse statique qui consiste à étudier le programme sous sa forme binaire, et l'analyse dynamique qui consiste à étudier le comportement d'un programme lors de son exécution.

Nous allons présenter ici les techniques les plus couramment utilisées pour protéger des programmes binaires sous Linux.

1.1 Protections contre l'analyse statique

Chiffrement du code binaire :
Le chiffrement de binaire à l'aide d'encodeurs (*packers*) est une des techniques les plus utilisées. Le principe est le suivant :

- le binaire original est chiffré ;
- un lanceur se charge de déchiffrer le code en mémoire ;
- le pointeur d'exécution est redirigé vers le code déchiffré ;
- le code du lanceur en mémoire est effacé.

Ainsi le code binaire du programme n'est jamais stocké dans un fichier sur le disque. Cette technique permet en plus de se protéger contre l'analyse anti-virus de base qui consiste à comparer la somme de contrôle (*checksum*) d'un programme avec une liste pré-établie.

Il existe plusieurs variantes de cette méthode de protection, comme par exemple le déchiffrement au fur et à mesure du programme.

1.2 Protections contre l'analyse dynamique

Détection de points d'arrêts :
Dans les debuggers habituels, un point d'arrêt (*breakpoint*) est défini en remplaçant l'instruction à l'adresse choisie par *int3* (0xCC). Lors de l'exécution de cette instruction, le programme s'arrête et le contrôle est donné au processus parent : le debugger. Celui-ci est alors en charge de remettre en place l'instruction originale, modifier le compteur d'instruction et continuer l'exécution normale.

La méthode pour détecter la présence d'un point d'arrêt consiste à chercher l'instruction 0xCC dans le code ou faire un *checksum* de la page de code. Le code suivante présente une implémentation de cette technique :

```
void foo()
{
    printf("Hello\n");
}

int main()
{
    if ((*(volatile unsigned *)((unsigned)foo + 3) & 0xff) == 0xcc) {
        printf("BREAKPOINT\n");
        exit(1);
    }
    foo();
    return 0;
}
```

Dans Kolumbo, il est possible de déterminer le moment d'insertion du point d'arrêt dans le code. Ainsi, nous pouvons attendre que le test d'intégrité soit passé avant d'insérer notre modification. Ceci requiert une connaissance préalable du programme et un test régulier de l'intégrité du code risque de détecter la modification. Donc la fonctionnalité de point d'arrêt dans Kolumbo est à utiliser de manière prudente.

Mise en place de faux points d'arrêts :
L'instruction *int3* sert également à lancer le signal SIGTRAP. Pour rendre l'analyse dynamique plus compliquée, il est possible d'utiliser cette instruction et la traiter à l'aide d'un gestionnaire de signaux pour que l'exécution continue normalement.

Lors de l'exécution, un debugger normal va interpréter cette instruction comme un point d'arrêt au lieu de laisser le programme s'exécuter. Ainsi, le programme peut détecter que le gestionnaire de signal n'a jamais été appelé et conclure qu'un debugger a interrompu le fonctionnement du programme. L'exemple de code suivant détermine la présence d'un debugger en modifiant la valeur d'un entier dans la gestion du signal SIGTRAP.

```
static volatile int traced = 1;

void sig_handler(int signo) {
    traced = 0;
}

void test_trap() {
    __asm__ __volatile__ ("int3\n\t");
}

int main() {
    signal(SIGTRAP, sig_handler);
    test_trap();
    if (traced) {
        printf("Debugger detected - Signal\n");
        return 1;
    }
    return 0;
}
```

Kolumbo n'utilise pas l'instruction *int3* pour mettre en place des points d'arrêts, donc ce test n'a aucun impact sur celui-ci. Le mécanisme utilisé est un pseudo appel système. C'est-à-dire une interruption 0x80 sans paramètres réels pour forcer le programme à passer en espace noyau. Ainsi les problèmes de signaux sont éliminés par contre un basculement dans l'espace noyau supplémentaire survient ce qui peut causer d'autres problèmes.

Détection de debuggers basés sur *ptrace* :
Ptrace est un appel système utilisé par les debuggers *user-space* traditionnels pour s'attacher à un processus. Un processus ne peut être tracé que par un seul autre processus. La technique pour détecter la présence d'un debugger consiste à essayer de se tracer soi-même (`PTRACE_TRACEME`). Si l'appel échoue, alors le programme peut déterminer qu'un autre processus le surveille. L'extrait de code suivant réalise cette opération.

```
int main() {
    // Ptrace check
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
        printf("Debugger detected - Ptrace\n");
```

```
            return 1;
        }
    }
```

Étant donné que l'appel à la fonction *ptrace()* déclenche un appel système et que Kolumbo fonctionne dans l'espace noyau, nous avons implémenté la possibilité de modifier le code de retour si nécessaire. Ceci nous permet d'utiliser les outils habituels tels que *gdb* ou *strace* pour étudier le comportement d'un programme. À l'heure actuelle, la valeur renvoyée est 0 si un processus fait un appel à *ptrace()* alors qu'il est déjà sous observation. Cette technique peut être contournée si le programme s'attend à recevoir une autre valeur (par exemple si il se trace lui-même plusieurs fois ou s'il démarre un autre processus pour le tracer).

timing check :
Une autre méthode de détection de debugger consiste à mesurer le temps passé à différents emplacements pendant l'exécution. Un écart trop grand par rapport au temps prévu à l'origine peut indiquer la présence d'un debugger. Ces tests peuvent se faire en utilisant l'instruction *rdtsc*, ce compteur est une variable de 64 bits incrémentée à chaque tick CPU. Il permet ainsi d'avoir une mesure très précise sur le temps passé dans un bloc de code. Toutefois, pour mettre en place ce test, il est nécessaire de calibrer parfaitement le seuil afin d'éviter les faux positifs.

2 Outils existants

L'analyse de programmes depuis le noyau n'est pas un domaine nouveau, il existe déjà des outils qui réalisent cette tâche. Dans cette partie nous allons faire un tour d'horizon rapide des solutions existantes et les comparer avec Kolumbo. Cette liste n'est pas exhaustive mais la plupart des outils dans ce domaine ont tendance à rester dans l'ombre et logiciels étudiés ici sont les plus documentés.

2.1 Helikaon Debugger

Ce module noyau crée par Jason Raber (Riverside Research Institute) a été présenté lors de la conférence REcon 2008 à Montréal. Il s'agît d'un projet visant à tracer les appels systèmes faits par un logiciel et à récupérer le contenu des registres à des endroits précis pendant l'exécution. Son objectif est d'être invisible pour le programme en cours d'analyse. La récupération du contenu des registres en dehors des appels systèmes se fait en "déroutant" le flot d'exécution du programme pour le faire passer dans une fonction prévue à cet effet. Malheureusement ce projet n'est pas disponible en dehors de chez RRI.

2.2 Eresi

Eresi est une plateforme multi-architecture dédiée pour l'analyse de binaires. Il permet de faire de l'analyse statique et dynamique grâce à un langage de script. Eresi a déjà été présenté lors de plusieurs conférences incluant SSTIC[1].

Les composants *Ke2dbg* et *Kernsh* sont les parties qui interagissent avec l'espace noyau. Avec ceux-ci, il est possible d'étudier le noyau Linux en cours d'exécution, de le modifier, le debugger, injecter du code C compilé et détourner des fonctions du noyau. Ainsi, il est possible d'analyser n'importe quel programme depuis l'espace noyau en utilisant le langage ERESI. Ce logiciel est donc très intéressant pour l'analyse de binaires, mais nécessite une bonne maîtrise du langage et de ses possibilités.

2.3 rr0d

Rr0d est un debugger ring 0, il fonctionne sous Linux, BSD et Windows et offre la possibilité d'analyser n'importe quel binaire. Seul un module est nécessaire pour faire fonctionner ce logiciel. Ensuite une belle interface nous permet d'interagir avec le système.

2.4 SystemTap

SystemTap est un logiciel développé pour simplifier la prise d'information dans le noyau Linux. Il est conçu principalement pour les développeurs, le but principal est d'aider au diagnostic des problèmes de performance ou des problèmes fonctionnels. Il fonctionne soit par la ligne de commande, soit par un langage de script développé pour l'occasion. À l'heure actuelle, il est principalement prévu pour fonctionner avec du code s'exécutant dans l'espace noyau. Ainsi, il est très utile pour étudier les pilotes. Pour tracer des applications utilisateurs avec SystemTap, il est nécessaire d'appliquer les patchs utrace[2].

2.5 Uberlogger

Uberlogger est un module noyau conçu pour enregistrer l'exécution de certains appels systèmes sur une machine. Pour réaliser cette tâche, il surcharge les appels choisis dans la table des appels systèmes puis rajoute ses fonctionnalités permettant de maintenir un historique dans une base de données. Ensuite via une interface web, il est possible de consulter ces données. Ce projet a été principalement adapté pour les `honeypots` et les analyses `forensic`. Malheureusement ce projet n'a pas eu de mise à jour depuis plus de trois ans.

[1] http ://www.eresi-project.org/wiki/EresiArticles
[2] http ://people.redhat.com/roland/utrace/

2.6 Linux Trace Toolkit

Linux Trace Toolkit (LTT-ng) est un patch pour le noyau Linux conçu afin d'instrumenter le noyau Linux pour avoir des mesures précises à différents emplacements du noyau. Il est principalement développé pour diagnostiquer les problèmes de performances sur les systèmes parallèles ou temps réel. Il est fourni avec une interface de visualisation (LTTV) permettant de retracer tous les évènements qui se sont produits sur la machine. Les principes fondamentaux de ce projet sont la performance et la stabilité car il est prévu pour être installé sur des systèmes en production.

Ce projet est très intéressant pour l'analyse d'un serveur Linux, cependant l'infrastructure à mettre en place est assez lourde et les informations concernant un programme malveillant se limitent à la liste des appels systèmes exécutés par celui-ci ce qui est intéressant mais pas suffisant.

3 Fonctionnement de Kolumbo

Le fonctionnement de base de Kolumbo repose sur le suivi et l'interception des appels systèmes. Un appel système permet à un programme s'exécutant dans l'espace utilisateur de faire des appels au noyau pour que ce dernier réalise des opérations nécessitant un plus haut niveau de privilèges. Parmi ces opérations, on retrouve l'accès à la mémoire, à des fichiers, à des périphériques, etc.

Il existe deux mécanismes pour réaliser des appels systèmes sur l'architecture x86 : le premier est basé sur l'interruption logicielle 0x80 et le second utilise le mécanisme *sysenter/sysexit*. L'approche *sysenter* intégrée depuis le Pentium II apporte beaucoup de vitesse au mécanisme basé sur l'interruption 0x80. À l'heure actuelle, Kolumbo supporte uniquement le premier système, donc nous allons détailler son fonctionnement.

3.1 Gestion des appels systèmes

Un appel système est déclenché en plaçant le numéro d'appel dans le registre *eax* et les paramètres dans les registres suivants, ensuite l'appel à l'interruption 0x80 fait basculer le système en mode noyau pour traiter l'appel système. L'exemple suivant montre comment appeler les appels systèmes *write* et *exit*.

```
section .text       ;section declaration
    global _start   ; entry point

_start:
    ;write our string to stdout
    mov     edx,len ;third argument: message length
```

```
        mov     ecx,msg ;second argument: pointer to message to write
        mov     ebx,1   ;first argument: file handle (stdout)
        mov     eax,4   ;system call number (sys_write)
        int     0x80    ;call kernel

        ;and exit
        mov     ebx,0   ;first syscall argument: exit code
        mov     eax,1   ;system call number (sys_exit)
        int     0x80    ;call kernel

    section .data       ;section declaration
        msg     db      "Hello, world!",0xa ;our string
        len     equ     $ - msg ;length of our string
```

3.2 Interception de l'interruption 0x80

Les interruptions et les exceptions sous Linux sont des évènements qui permettent de modifier l'ordre des instructions à exécuter. Ces évènements correspondent à des signaux électriques générés par des composants matériels à l'intérieur ou à l'extérieur du processeur. Il faut distinguer deux types d'interruptions : les interruptions matérielles générées par les périphériques d'entrées/sorties et les interruptions logicielles générées soit par des erreurs de programmation (division par zéro, erreur de segmentation, etc...) ou par des situations exceptionnelles telles que des fautes de pages ou des appels systèmes.

Pour les besoins de Kolumbo, il est nécessaire de connaître tous les appels systèmes exécutés par un programme pendant la durée de son analyse. La technique ici requiert de remplacer le gestionnaire de l'interruption 0x80 par le nôtre pour récupérer le contrôle sur le système lors d'un appel système. Nous allons donc voir comment sont gérées les interruptions sous Linux et comment remplacer le gestionnaire.

Fonctionnement des interruptions sous Linux :
Lorsqu'une interruption est déclenchée, le noyau consulte une table permettant de trouver la fonction responsable de celle-ci. Cette table s'appelle *Interrupt Descriptor Table* (IDT). Le format de la table IDT est le suivant : chaque entrée correspond à un vecteur d'interruption ou d'exception qui contient un descripteur de 8 octets. Le registre CPU *idtr* permet de localiser la table IDT n'importe où en mémoire. Il contient l'adresse physique de la table ainsi que sa taille maximale (limite). Il est initialisé au démarrage avec l'instruction assembleur *lidt* (*Load IDT Register*).

Repérage du gestionnaire de l'interruption 0x80 :
Afin de remplacer le gestionnaire de l'interruption 0x80, il est nécessaire de trouver son emplacement en mémoire. Grâce au registre *idtr*, on peut retrouver la table IDT,

par contre il est nécessaire de calculer manuellement des décalages pour trouver le gestionnaire pour l'interruption 0x80.

L'instruction assembleur *sidt* (*Store IDT Register*) retourne le contenu du registre *idtr* dans une variable de 6 octets : 4 octets pour l'adresse de base de la table IDT et 2 octets pour la taille maximale de cette table (*limit-field*). Comme chaque descripteur fait 8 octets, il est nécessaire de faire un décalage à partir de l'adresse de base pour trouver le descripteur de l'interruption qui nous intéresse. Dans le cas de l'interruption 0x80, il faut décaler de 1024 (8 fois 0x80). À partir de ce moment, on récupère une structure de type *idt_descriptor* qui nous permet de trouver l'adresse de la fonction.

L'extrait de code suivant réalise exactement ces opérations :

```
void *get_system_call(void)
{
        unsigned char idtr[6];
        unsigned long base;
        struct idt_descriptor desc;

        asm ("sidt %0" : "=m" (idtr));
        base = *((unsigned long *) &idtr[2]);
        memcpy(&desc, (void *) (base + (0x80*8)), sizeof(desc));
        return((void *) ((desc.off_high << 16) + desc.off_low));
}
```

Remplacement du gestionnaire de l'interruption 0x80 :
Maintenant que l'adresse du gestionnaire de l'interruption 0x80 est trouvée, voyons comment intégrer notre code dans celui-ci. L'objectif ici est de remplacer dans la fonction *system_call* l'appel aux fonctions *syscall_trace_entry* et *sys_call_table(,%eax,4)* par un appel vers notre fonction. Le code ci-dessous provient des sources du noyau Linux. Il s'agit du code source du gestionnaire de l'interruption 0x80 (*system_call*). On y retrouve l'appel aux deux fonctions que nous voulons remplacer.

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_THREAD_INFO(%ebp)
                                        # system call tracing in operation
        testb $(_TIF_SYSCALL_TRACE|_TIF_SYSCALL_AUDIT),TI_flags(%ebp)
        jnz syscall_trace_entry
        cmpl $(nr_syscalls), %eax
        jae syscall_badsys
syscall_call:
        call *sys_call_table(,%eax,4)
        movl %eax,EAX(%esp)             # store the return value
syscall_exit:
        cli
        movl TI_flags(%ebp), %ecx
        testw $_TIF_ALLWORK_MASK, %cx   # current->work
```

```
        jne syscall_exit_work
restore_all:
        RESTORE_ALL
```

Si on regarde ce code compilé, on obtient la sortie suivante :

```
c0103d3c:   50                      push   %eax
c0103d3d:   fc                      cld
c0103d3e:   06                      push   %es
c0103d3f:   1e                      push   %ds
c0103d40:   50                      push   %eax
c0103d41:   55                      push   %ebp
c0103d42:   57                      push   %edi
c0103d43:   56                      push   %esi
c0103d44:   52                      push   %edx
c0103d45:   51                      push   %ecx
c0103d46:   53                      push   %ebx
c0103d47:   ba 7b 00 00 00          mov    $0x7b,%edx
c0103d4c:   8e da                   movl   %edx,%ds
c0103d4e:   8e c2                   movl   %edx,%es
c0103d50:   bd 00 e0 ff ff          mov    $0xffffe000,%ebp
c0103d55:   21 e5                   and    %esp,%ebp
c0103d57:   66 f7 45 08 c1 01       testw  $0x1c1,0x8(%ebp)
c0103d5d:   0f 85 c1 00 00 00       jne    c0103e24
c0103d63:   3d 37 01 00 00          cmp    $0x137,%eax   <-- ICI
c0103d68:   0f 83 2a 01 00 00       jae    c0103e98      <-- ICI

c0103d6e :  # syscall_call
c0103d6e:   ff 14 85 c0 f4 2c c0    call   *0xc02cf4c0(,%eax,4)
c0103d75:   89 44 24 18             mov    %eax,0x18(%esp)

c0103d79 :  # syscall_exit
c0103d79:   fa                      cli
c0103d7a:   8b 4d 08                mov    0x8(%ebp),%ecx
c0103d7d:   66 f7 c1 ff fe          test   $0xfeff,%cx
c0103d82:   0f 85 cc 00 00 00       jne    c0103e54

c0103d88 :  # restore_all
c0103d88:   8b 44 24 30             mov    0x30(%esp),%eax
```

À partir de cette information et de l'adresse de base, il est possible de calculer l'emplacement exact de l'appel aux deux fonctions qui nous intéressent. L'idée ici est de remplacer la comparaison avant le saut (*jae*) par un saut vers notre fonction. Pour ce faire, on parcoure le code à la recherche de l'instruction *0x0f 0x83* (car il s'agit d'une instruction unique dans cette portion de code) et on recule de 5. Une fois au bon endroit on remplace l'instruction *cmp* par un *push* (opcode *0x68*), on insère l'adresse de notre fonction et l'instruction *ret* (opcode *0xc3*). Ceci aura pour effet d'appeler notre fonction avec un impact minimal sur le système (seulement un *push*). Bien sûr lors de ces étapes, on sauvegarde les adresses et les portions de code modifiées pour pouvoir les restaurer plus tard.

L'extrait de code suivant réalise exactement les étapes décrites ci-dessus :

```
void set_idt_handler(void *system_call, void *hooked_idt)
{
        unsigned char *p;
        unsigned long *p2;
        p = (unsigned char *) system_call;

        while (!((*p == 0x0f) && (*(p+1) == 0x83)))
                p++;
        p -= 5;
        *p++ = 0x68;
        p2 = (unsigned long *) p;
        *p2++ = (unsigned long) (hooked_idt);

        p = (unsigned char *) p2;
        *p = 0xc3;
```

On applique ce principe de recherche d'*opcodes* pour remplacer l'appel à la fonction *syscall_trace_entry*. À partir de ce moment, lorsqu'un appel système est exécuté, la fonction passée en paramètre (*new_idt*) est exécutée. Notre fonction est maintenant responsable de vérifier que le numéro d'appel système (stocké dans *eax*) est inférieur à *NR_syscalls* (0x137) car nous avons remplacé cette vérification par manque de place. Si le test passe, notre fonction en charge de traiter les appels systèmes (*hook*) va être appelée. Sinon, on appelle la fonction *syscall_exit* (dont l'adresse a été trouvée avec une procédure similaire à la précédente).

La fonction *new_idt* est une fonction en assembleur qui vérifie que le numéro d'appel système est possible grâce à une comparaison à la variable globale *NR_syscalls*. Il faut bien noter ici que le moindre changement d'un registre rend le système hors d'usage d'où le choix de l'assembleur dans ces parties critiques. Un simple if en C modifie les registres ce qui provoque inévitablement un *kernel panic*.

```
void new_idt(void)
{
        __asm__ __volatile__
        (
        "cmp %0, %%eax\n"
                "jae badsyscall\n"
                "jmp hook\n"

                "badsyscall:\n"
                    "jmp dire_exit\n"

        : : "i" (NR_syscalls) // 0x137
        );
}
```

Pour terminer, la fonction *hook* fait le lien entre l'appel système et les fonctions originales. Ceci permet d'intercepter un appel système avant qu'il ait lieu. Dans le cadre de ce module, l'utilité principale de *hook* est d'afficher l'appel système en cours (et ses paramètres) s'il a été déclenché par un processus que nous désirons suivre.

Cette fonction nous permet également de gérer notre système de points d'arrêts, mais cette fonctionnalité est détaillée plus loin.

Restauration du gestionnaire de l'interruption 0x80 original :
Lorsque le module est déchargé, il est impératif que le système retourne dans un état normal. Donc, nous devons absolument remettre le gestionnaire des appels systèmes dans l'état où nous l'avons trouvé. À l'étape précédente, nous avons sauvegardé les adresses et les données modifiées, donc il suffit de les restaurer lors du déchargement du module (fonction *exit_lkm*).

3.3 Interception des appels systèmes

Afin d'implémenter des fonctionnalités, comme le suivi de processus fils, il est nécessaire d'intercepter certains appels systèmes. Tel que nous avons vu au chapitre précédent, lorsqu'un appel système est déclenché, le système consulte la table des appels systèmes (*syscall_table*) avec le numéro de l'appel (stocké dans *eax*) pour trouver la bonne fonction à appeler. Pour intercepter un appel système, il faut donc écrire dans cette table à l'index *eax*, l'adresse de la fonction à appeler, puis préparer la fonction à recevoir les bons paramètres, faire le traitement et renvoyer le bon type de données.

Le problème principal est de trouver l'adresse de cette table, car depuis les noyaux 2.6, le symbole correspondant n'est plus exporté. Une des techniques est de consulter le fichier *System.map* généré lors de la compilation du noyau, mais ceci implique une action manuelle. De plus, ce fichier n'est pas nécessaire au fonctionnement de Linux, donc il risque de ne pas être présent sur le système. Pour résoudre ce problème, nous avons utilisé une méthode provenant du rootkit *enyelkm*[3] pour repérer dynamiquement l'adresse de cette table en se basant sur la table IDT.

Repérage dynamique de la table des appels système :
Comme étudié dans la section concernant le remplacement du gestionnaire de l'interruption 0x80, il est possible d'utiliser l'adresse de la table IDT pour se retrouver dans le code responsable de la gestion des appels systèmes. En utilisant une technique similaire, nous allons pouvoir trouver l'adresse de la table des appels systèmes car elle est utilisée dans la même partie de code :

```
syscall_call:
        call *sys_call_table(,%eax,4)
```

[3] http ://www.enye-sec.org

```
c0103d6e : # syscall_call
c0103d6e : ff 14 85 c0 f4 2c c0      call   *0xc02cf4c0(,%eax,4)
```

Dans cet affichage, on voit que l'adresse que nous cherchons est 0xc02cf4c0. Pour trouver cette adresse, nous allons chercher la suite d'instructions 0xff, 0x14 et 0x85. Le code responsable de cette tâche se situe dans la fonction *get_sys_call_table*.

```
void *get_sys_call_table(void *system_call)
{
        unsigned char *p;
        unsigned long s_c_t;
        p = (unsigned char *) system_call;
        // On cherche la sequence 0xff 0x14 0x85
        while (!((*p == 0xff) && (*(p+1) == 0x14) &&
                                 (*(p+2) == 0x85)))
                p++;

        dire_call = (unsigned long) p;

        p += 3;
        s_c_t = *((unsigned long *) p);

        p += 4;
        after_call = (unsigned long) p;

        /* 0xfa : cli */
        while (*p != 0xfa)
                p++;
        dire_exit = (unsigned long) p;
        return((void *) s_c_t);
}
```

Surcharge de quelques appels systèmes :
Une fois la table des appels systèmes repérée, l'interception d'un appel système consiste à inscrire une fonction à l'adresse *sys_call_table[eax]*. Nous avons surchargé quelques appels systèmes afin d'ajouter des fonctionnalités nécessaires pour notre module.

En guise d'exemple étudions le cas de l'appel système *fork*. Nous l'avons intercepté (avec *clone* et *vfork*) dans le but de récupérer les numéros de processus fils crées par le processus courant si celui-ci est sous observation. Lorsque l'appel système *fork* est déclenché, la fonction originale est appelée avec les mêmes paramètres, le code de retour est sauvegardé, un traitement local est effectué et la valeur de retour originale est renvoyée. Il faut noter ici l'utilisation de *asmlinkage*, étant donné que les paramètres des appels systèmes sont passés sur la *stack*.

```
asmlinkage int (* zold_sys_fork) (struct pt_regs regs);
```

```
asmlinkage int znew_sys_fork (struct pt_regs regs) {
        int ret;
        ret = zold_sys_fork(regs);
        if(search_pid(current->pid) != -1)
                insert_pid(ret);
        return ret;
}

static void syscall_hooking(void) {
        zold_sys_fork = (void *) (sys_call_table[__NR_fork]);
        sys_call_table[__NR_fork] = (void *)znew_sys_fork;
}


static void syscall_unhooking(void) {
        sys_call_table[__NR_fork] = (void *)zold_sys_fork;
}
```

À partir de ce moment, l'appel à *fork* va déclencher un appel à la fonction *znew_sys_fork* qui lui-même va appeler la fonction originale après avoir fait son traitement. Lors du déchargement du module, il faudra appeler la fonction *syscall_unhooking* pour remettre la table des appels systèmes dans son état original.

Maintenant que nous avons vu les manipulations que réalise Kolumbo au niveau des appels systèmes et des interruptions, voyons comment il accède à la mémoire d'un programme.

3.4 Mémoire virtuelle sous Linux

Un processus, pendant son exécution, travaille uniquement avec des adresses virtuelles. Ceci a plusieurs avantages tels qu'une bonne isolation entre les différents processus et la possibilité d'attribuer un espace contigu de mémoire même si le système ne dispose pas suffisamment de mémoire physique. À l'intérieur d'un programme, il n'y a aucun problème pour travailler avec des adresses virtuelles, cependant le processeur n'a pas la notion d'espace mémoire virtuel, il travaille avec des adresses physiques. La correspondance entre mémoire virtuelle et mémoire physique est réalisée dans le noyau par des tables de pages.

Linux fonctionne avec un système de table de pages à trois niveaux. Ainsi, les systèmes 64 bits ont suffisamment d'espace pour adresser la totalité de l'espace disponible. Le premier niveau se nomme *Page Global Directory* (PGD). Il est représenté par le type de données *pgd_t* et permet d'accéder au deuxième niveau nommé *Page Middle Directory* (PMD) représenté par le type de donnée *pmd_t*. Enfin, le dernier niveau permettant d'accéder à la page physique se nomme *Page Table Entry* (PTE) et il est représenté par le type de donnée *pte_t*.

Ce qui est intéressant de noter ici, est que chaque processus possède ses propres pages de tables. Ainsi, le champ `mm_struct->pgd` d'une tâche (*task_struct*) fait référence au PGD du processus et non à celui du système complet. Avant d'accéder à une page, il est également nécessaire de vérifier qu'elle est disponible en mémoire centrale ou de déclencher une faute de page pour corriger la situation. Dans le noyau Linux, la fonction *get_user_pages* nous permet d'accéder à une page en mémoire physique à partir d'un processus, d'une adresse de départ et d'une taille. Cette fonction réalise la traduction d'adresses virtuelles en adresses physiques en suivant les trois niveaux et s'assure que la page est disponible. Elle crée un tableau de pointeurs vers des pages et retourne le nombre de pages récupérées. À partir de ce moment, il est possible de travailler sur ces pages depuis le noyau.

3.5 Segmentation de la mémoire d'un processus

Lorsqu'un processus est chargé en mémoire, son espace d'adressage linéaire est divisé en différentes zones appelées segments. Ces segments sont déterminés en fonction de leur utilité. Il existe six segments principaux : Text, Data, gvar, BSS, Heap et Stack.

Ces six segments sont en réalité chargés dans trois zones mémoires distinctes : *text*, *data* et *stack*. Le segment *text* contient le code exécutable du programme. Le segment *data* contient les données initialisées du programme, les données non-initialisées et les variables globales (zone *BSS* de l'exécutable) ainsi que le *heap*. La structure *mm_struct* accessible à partir de l'attribut *mm* d'une structure de tâche (*task_struct*) permet d'accéder à ces différents segments.

Le fichier *maps*, disponible par processus dans le pseudo-système de fichier */proc*, donne la liste des segments mémoires d'un processus. Par exemple, pour le processus *cat*, on y retrouve :

```
08048000-0804f000 r-xp 00000000 08:01 103586     /bin/cat
0804f000-08050000 rw-p 00006000 08:01 103586     /bin/cat
08050000-08071000 rw-p 08050000 00:00 0          [heap]
b7d60000-b7d9f000 r--p 00000000 08:01 33855      /usr/lib/locale/en_CA.utf8/
    LC_CTYPE
...
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
```

Dans l'ordre, on retrouve :

- le segment *text* : facile à repérer, car il a les droits de lecture/exécution et il est lié au fichier exécutable ;
- le segment *data* : droits de lecture/écriture et lien avec le fichier ;
- le segment *heap* ;

– un ou plusieurs segments contenant les bibliothèques dynamiques ;
– le segment *stack*.

La lecture de ce fichier nous donne donc beaucoup d'informations, par contre il est difficile de suivre l'évolution des zones mémoires d'un processus en consultant uniquement ce fichier. La fonction suivante, affiche le début et la fin des différentes zones mémoire d'un processus (*Virtual Memory Area* : VMA) ainsi que les adresses des trois zones principales (*text*, *data* et *stack*) pendant l'exécution du programme.

```
static void print_mem(struct task_struct *task) {
        struct mm_struct *mm;
        struct vm_area_struct *vma;
        int count = 0;
        mm = task->mm;
        printk("\nThis process has %d vmas.\n", mm->map_count);

        //iteration on process virtual memory areas
        for (vma = mm->mmap ; vma ; vma = vma->vm_next) {
                printk ("Vma number %d: ", ++count);
                printk("  Starts at 0x%08lx, Ends at 0x%08lx\n",
                          vma->vm_start, vma->vm_end);
        }

        printk("Code  Segment start = 0x%lx, end = 0x%lx \n"
                 "Data  Segment start = 0x%lx, end = 0x%lx\n"
                 "Stack Segment start = 0x%lx\n\n",
                 mm->start_code, mm->end_code,
                 mm->start_data, mm->end_data,
                 mm->start_stack);
}
```

L'appel à cette fonction peut être déclenché n'importe où pendant l'exécution du programme cible et permet de voir l'évolution des zones mémoires. Par exemple, lorsqu'un programme est chiffré avec l'encodeur UPX, la routine de déchiffrement crée deux zones mémoires (appel système *mmap* ou *old_mmap*) accessibles en lecture et écriture, déchiffre les segments *text* et *data* dans ces nouvelles zones, change les permissions (appel système *mprotect*) pour que la zone *text* soit uniquement accessible en lecture et exécution puis détruit la zone ayant servi à déchiffrer le code avec l'appel système *munmap*. Avec cette information, nous connaissons avec précision les zones mémoires qui nous intéressent. L'exemple suivant illustre ce mécanisme : on exécute un programme chiffré (*hellopacked*) et on analyse les appels systèmes qu'il déclenche avec l'outil *strace*. Pour faciliter la lecture, seuls les appels et paramètres importants ont été conservés.

```
$ strace ./hellopacked
...
old_mmap(0x8048000, 50221, PROT_READ|PROT_WRITE|PROT_EXEC,) = 0x8048000  #
    creation zone text
```

```
mprotect(0x8048000, 50218, PROT_READ|PROT_EXEC) = 0 # changement des droits sur
     la zone text
old_mmap(0x8055000, 1103, PROT_READ|PROT_WRITE, ...) = 0x8055000 # creation
    zone data
mprotect(0x8055000, 1100, PROT_READ|PROT_WRITE) = 0 # changement des droits sur
     la zone data
munmap(0xc01000, 8192)                  = 0 # suppression de la zone de d{\'e}
    chiffrement
...
```

Lecture et écriture dans la mémoire d'un processus :
Maintenant que nous avons vu comment est gérée la mémoire d'un processus, voyons comment y accéder depuis le noyau. À partir du descripteur de tâche (*task_struct*), l'accès à toutes les zones de mémoire se fait en suivant une liste chaînée (élément *vm_next* de la structure *vm_area_struct*), donc les seules informations nécessaires sont le descripteur de tâche, l'adresse de début et la longueur (ce qui va déterminer si on accède à une ou plusieurs pages). La fonction *zget_page* parcoure les différents segments et recherche la zone qui contient l'adresse de départ. Une fois qu'elle est repérée, elle appelle la fonction du noyau Linux *get_user_pages* avec en paramètre l'entrée *vm_mm* de la zone mémoire concernée.

```
int zget_page(struct task_struct *task, unsigned int start,
        unsigned int end, int npages, struct page **pages, int *page_start) {
    struct vm_area_struct *vma;
    // iterate all vma of the process
    for(vma = task->mm->mmap; vma; vma = vma->vm_next) {
            // found the vma we are looking for ?
            if(start < vma->vm_end && end > vma->vm_start) {
                   *page_start = vma->vm_start;
                    // get the page(s) and return the number of pages
                    return get_user_pages(task, vma->vm_mm, start, npages,
                                                        0, 1, pages, NULL);
            }
    }
    return -1;
}
```

Ainsi, on récupère les pages qui contiennent les zones que nous cherchons. Cependant, récupérer des pages n'est pas suffisant pour pouvoir lire et écrire dans la mémoire d'un processus. En effet, Linux nous interdit d'accéder à de la mémoire physique directement, donc pour réaliser les opérations de lecture et écriture, il faut importer la zone mémoire dans l'espace mémoire du noyau avec l'instruction *kmap*. Cette fonction crée un lien entre la zone de mémoire physique et une plage d'adresses virtuelles en créant une entrée de type PTE (*Page Table Entry*) dans une table de page spéciale. De ce fait, le noyau peut accéder à la page. À la fin du traitement, il est nécessaire d'appeler la fonction *kunmap* pour effacer l'entrée PTE.

Par exemple, la fonction *zdump_region* copie une portion d'une page dans un tableau. Elle reçoit la page en paramètre, appelle *kmap*, lit les données et appelle *kunmap*.

```
void zdump_region(struct page *pages, int start, int len,
                          unsigned char *data, int offset) {
        int i;
        unsigned char *maddr = kmap(pages);
        for(i = 0; i < len; i++) {
                data[offset + i] = maddr[start + i];
        }
        kunmap(pages);
}
```

Une fois cette notion maitrisée, l'écriture dans une zone mémoire d'un processus est très simple car elle revient à écrire dans un tableau de caractères. La fonction *zchange_bytes* remplace certains octets à l'adresse passée en paramètre. Par rapport à la fonction précédente, la seule différence est l'écriture dans le tableau.

```
void zchange_bytes(struct page *pages, int start, int len,
                                        unsigned char *data) {
    int i;
    unsigned char *maddr = kmap(pages);
    for(i = 0; i < len; i++) {
        printk("changing : %02x by %02x\n", maddr[start + i], data[i]);
        maddr[start + i] = data[i];
    }
    kunmap(pages);
}
```

Maintenant que nous avons vu comment lire et écrire dans la mémoire d'un processus en cours d'exécution, voyons comment utiliser ces fonctionnalités pour récupérer un fichier au format ELF à partir de son image mémoire. Ensuite, nous verrons comment modifier la mémoire du programme pour le forcer à passer en espace noyau.

Reconstruction d'un ELF à partir de son image mémoire :
Sous Linux, le format exécutable le plus populaire est ELF (*Executable and Linking Format*). Le principal avantage par rapport à l'ancien format *a.out* est la possibilité d'utiliser des bibliothèques partagées.

La structure d'un fichier ELF est assez complexe et ne sera pas détaillée dans ce document. Cependant dans le cadre de ce projet, il est nécessaire de comprendre les notions bases de ce format. Les 80 premiers octets d'un ELF correspondent à l'en-tête de l'exécutable, on y retrouve les informations nécessaires pour comprendre l'organisation du reste du fichier. L'outil *readelf* nous permet d'afficher facilement le contenu de l'en-tête de base.

```
$ readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048130
  Start of program headers:          52 (bytes into file)
  Start of section headers:          498476 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         5
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 32
```

Il est particulièrement intéressant de noter l'adresse du point d'entrée du programme (*Entry point address*) ainsi que les adresses de départ des sections (référencées dans la section *Start of section headers*. Après l'en-tête de base, on retrouve un certain nombre de sections. Les sections correspondent aux segments expliqués dans la partie concernant la gestion de la mémoire. Au minimum, on retrouve les sections *text* et *data*.

Enfin, tout à la fin du fichier se situe la table des en-têtes de section. Cette table donne les informations sur la taille de chacune des sections, ce qui permet au chargeur du programme de placer les différents segments dans les bonnes pages mémoires.

Donc, pour construire un ELF minimal, il faut les informations suivantes :

- L'en-tête de base ;
- Les données des sections (au moins *text* et *data*) ;
- La taille et l'emplacement relatif (*offset*) de chaque section ;

Avec la fonction *zdump_region*, nous avons vu qu'il est possible de récupérer toutes ces informations directement dans la mémoire d'un programme. Dans le cas d'un programme de type ELF non chiffré, la simple copie des sections *text* et *data* est suffisante pour obtenir un programme fonctionnel. Le binaire est chargé dans son intégralité en mémoire, donc les en-têtes sont disponibles dans la mémoire du programme. Le seul travail nécessaire du côté du module est de connaître l'adresse de début et de fin de chacune des sections ce qui se fait en lisant la structure de description du programme *mm_struct* (voir partie concernant la gestion de la mémoire dans le noyau). Une autre approche est de lire l'en-tête ELF depuis le noyau et de

calculer les adresses à partir de cette informations, mais ceci implique de lire toute la mémoire du programme avant de commencer la copie.

Comme nous l'avons vu auparavant, lors du chargement d'un programme chiffré, la routine de déchiffrement crée les zones mémoires pour accueillir le code et les données du programme original puis, quand le programme est déchiffré, change le pointeur d'instruction vers la nouvelle zone de code. Ceci pose un problème pour la récupération du programme déchiffré depuis la mémoire, car les pointeurs vers le début et la fin des zones *text* et *data* sont initialisés au démarrage du programme, donc ils pointent vers les zones de la routine de déchiffrement. De plus, lors du déchiffrement la routine se sert des en-têtes de sections ELF pour placer les données au bon endroit, mais ne copie pas ces en-têtes. De ce fait, il est impossible de connaître la fin de chacune des sections. Pour l'instant, lorsqu'on exécute la copie de la mémoire d'un processus chiffré, on récupère les bonnes sections *text* et *data*. Cependant, nous sommes obligés de récupérer des pages entières qui contiennent beaucoup de zéros qui ne font pas partie du programme original. Le programme est bien déchiffré, mais il est impossible de l'exécuter. Pour reconstruire un programme fonctionnel, il est nécessaire de modifier manuellement le programme et recréer les en-têtes de sections. Dans le cas de UPX, les en-têtes de sections se trouvent en clair dans la zone en clair du binaire chiffré, donc il est possible de les récupérer, mais ce n'est pas automatique.

Insertion de points d'arrêts pendant l'exécution :
Un des concepts particulièrement intéressant de l'analyse dynamique est la possibilité d'arrêter le programme pendant son exécution afin d'obtenir des informations telles que l'état de la pile et des registres du processus. Comme nous l'avons vu auparavant, il est possible de détecter les techniques d'analyse dynamique habituelles. De ce fait, un programme peut détecter qu'un point d'arrêt est placé et changer de comportement.

Notre module se situe en espace noyau, donc pour récupérer des informations sur le programme en cours d'exécution, il est nécessaire que celui-ci entre en espace noyau.

Nous avons vu avec la fonction de trace de Kolumbo que nous pouvons afficher les registres lorsqu'un appel système se produit, donc pour récupérer l'information des registres entre deux appels systèmes, nous allons injecter un faux appel système. Ceci simulera le comportement d'un point d'arrêt : à un moment prédéfini, nous allons récupérer le contrôle sur le programme en cours d'exécution pour obtenir des informations sur celui-ci. Encore une fois, ce procédé se doit d'être absolument indétectable pour le programme.

Comme nous l'avons déjà vu, un appel système est une interruption 0x80 qui se traduit par les instructions 0xCD 0x80. Notre objectif est donc d'insérer une

interruption 0x80 pendant l'exécution du programme à un emplacement prédéterminé. Voici les étapes pour ajouter un point d'arrêt :

- Repérer l'emplacement où insérer le point d'arrêt, typiquement entre deux appels systèmes ;
- Repérer l'appel système où déclencher l'insertion du point d'arrêt, normalement l'appel juste avant l'emplacement désiré pour limiter les risques de détection ;
- Lorsque l'appel d'insertion est déclenché :
  - Sauvegarder les deux octets à partir de l'adresse du point d'arrêt ;
  - Remplacer ces deux octets par 0xCD 0x80 ;
- Lorsqu'un appel système a comme adresse de retour l'adresse du point d'arrêt plus 2 (0xCD 0x80), ceci nous indique que notre point d'arrêt a été atteint, les étapes sont alors :
  - Afficher les registres ;
  - Remettre en place les deux octets originaux ;
  - Changer l'adresse de retour à l'adresse du point d'arrêt, comme si on lui demandait de réexécuter la même instruction (c'est-à-dire deux octets en arrière) ;
  - Ne pas exécuter le traitement normal d'un appel système ;
  - Renvoyer le contrôle à l'utilisateur.

Lorsque ce faux appel système est atteint, on s'en sert pour afficher les informations qui nous intéressent, on remet le programme dans son état original et on change le pointeur d'instruction pour que l'exécution redémarre normalement. L'impact sur le processus est donc minimal et presque imperceptible pour celui-ci. Tout ce traitement est réalisé par les fonctions *add_bp* et *restore_post_bp*. Une note importante : le point d'arrêt ne peut pas être inséré n'importe où dans le code, il est impératif que l'adresse de base corresponde à l'adresse d'une instruction et non pas à l'adresse d'un paramètre. Sinon, le point d'arrêt n'aura aucun effet à part corrompre la mémoire et le fonctionnement du processus.

3.6 Suivi des processus fils : appel système fork

Jusqu'à présent, nous pouvons suivre tous les appels systèmes réalisés par un processus, récupérer le contenu de sa mémoire et ajouter des points d'arrêts à un emplacement arbitraire pour obtenir plus d'informations. Cependant, si un processus crée un processus fils grâce à l'appel système *fork*, nous perdons sa trace. Pour contourner ce problème, nous avons implémenté un système pour suivre une famille de processus.

Cycle de vie d'un processus :
Lors du démarrage d'un programme, l'appel système *fork* est déclenché pour créer un nouveau contexte pour le processus naissant. Ce contexte inclut entre autres un espace mémoire, une pile, un environnement et un numéro de processus (PID). Ensuite dans ce nouveau contexte, le pseudo appel système *execve* est déclenché pour charger le contenu du nouveau programme. Lorsque le programme a fini son exécution, il fait un appel système *exit* ou *exit_group* (pour la gestion des fils d'exécutions). Le programme peut terminer plus brutalement si un signal est émis, par exemple dans le cas d'une erreur de segmentation.

Pour notre module, il est donc impératif de connaître les nouveaux processus créés par un processus en cours d'observation, mais également savoir quand un de ces processus se termine.

Suivi au coeur du noyau d'une famille de processus :
Les appels systèmes *fork*, *vfork* et *clone* sont responsables de créer des nouveaux processus. Tous les trois retournent le numéro de processus nouvellement créé. Nous avons déjà vu comment surcharger un appel système. Pour connaître le nouveau PID nous avons utilisé la même technique :

- Interception des appels systèmes ;
- Vérification que le processus courant (`current->pid`) est un des processus sous observation ;
- Si c'est le cas :
  - On déclenche l'appel système original ;
  - On récupère le code de retour ;
  - On ajoute le nouveau PID dans notre liste de processus suivis ;
  - On renvoie comme valeur de retour le nouveau PID.
- Si le processus n'est pas un processus suivi, on exécute simplement l'appel système original en renvoyant le nouveau PID.

L'exemple ci-dessous est la version modifiée de l'appel système fork. On peut voir que l'appel système original est appelé et que le nouveau PID est sauvegardé si le processus courant est un processus en cours d'observation.

```
asmlinkage int znew_sys_fork (struct pt_regs regs) {
        int ret;
        ret = zold_sys_fork(regs);
        if(search_pid(current->pid) != -1)
                insert_pid(ret);
        return ret;
}
```

La liste de processus est maintenue dans un tableau de taille fixe. Nous avons opté pour cette approche pour sa simplicité de mise en place comparée à implémenter une liste chaînée dynamique dans le noyau. Pour le moment, nous supposons que les processus fils créés ont tous les mêmes comportements, seuls les paramètres changent.

La fin normale d'un processus est caractérisée par un appel système à *exit* ou *exit_group*, nous avons donc utilisé la même technique que précédemment : on intercepte les appels à *exit* et *exit_group* si le processus courant fait partie de notre liste, on le supprime. Un processus peut également se terminer de manière anormale suite à un signal. Dans ce cas, nous devons parcourir la table des processus actifs et regarder si le processus qui a déclenché le signal est encore présent.

3.7 Contournement d'une protection anti-traçage

Comme vu dans le premier chapitre, il existe plusieurs méthodes de détection de *debugger*. Nous avons implémenté une fonction pour contourner la détection de *debuggers* basés sur *ptrace* tels que *strace* et *gdb*. Le principe est le suivant, le programme fait un appel à *ptrace* pour se tracer lui-même. Le code de retour de cette fonction indique si l'appel à *ptrace* a fonctionné. Cet appel échoue si un autre processus est déjà en train de tracer le processus courant. Ainsi, le processus peut détecter qu'un *debugger* est déjà en train de le tracer. La fonction suivante implémente ce test.

```
// Ptrace check
if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
    printf("Debugger detected - Ptrace\n");
    return -1;
} else {
    return 0;
}
```

Si on exécute cette fonction avec *strace*, on obtient la sortie suivante :

```
# strace ./detectptrace
...
ptrace(PTRACE_TRACEME, 0, 0x1, 0)       = -1 EPERM (Operation not permitted)
write(1, "Debugger detected - Ptrace\n", 27Debugger detected - Ptrace) = 27
exit_group(-1)                           = ?
Process 30590 detached
```

On voit que le code de retour de l'appel à *ptrace* est -1, ce qui indique au programme qu'il est en train d'être analysé.

Contournement invisible de ptrace :
Encore une fois, pour contourner la fonction de détection de *ptrace*, nous allons

intercepter cet appel système et modifier la valeur de retour. Si le processus ayant déclenché l'appel système est un processus en cours d'observation, on renvoie le code de retour 1, ce qui lui signifie que l'appel a fonctionné. Ainsi, il continue son exécution normale et termine avec le code de retour 0.

```
# strace ./detectptrace
...
ptrace(PTRACE_TRACEME, 0, 0x1, 0)       = 1
exit_group(0)                           = ?
Process 30592 detached
```

Cette technique n'est pas suffisante pour contourner toutes les protections *anti-ptrace*, car si le programme cible exécute deux fois l'appel de la fonction *ptrace*, il s'attend à ce que le deuxième appel échoue ce qui n'est pas le cas en ce moment. L'implémentation des autres protections contre les systèmes de détection de *debugger* est en cours de réalisation à l'heure actuelle.

3.8 Communication entre un module noyau et un processus utilisateur

Comme nous l'avons vu à plusieurs reprise, notre module noyau a besoin d'échanger des informations avec l'espace utilisateur. Pour être plus précis, nous avons trois types de communications : l'affichage de messages depuis le noyau pour le mode trace, la récupération de l'image mémoire d'un programme et la configuration du module. Pour afficher des messages depuis le noyau, nous utilisons la fonction *printk* fournie avec le noyau. Dans cette section, nous allons voir en détail les deux autres modes de communication mis en place dans notre module.

Échange de données brutes : character device :
La récupération de l'image mémoire d'un programme impose d'échanger beaucoup de données binaires entre le noyau et l'espace utilisateur. Depuis le noyau, il est fortement déconseillé de créer des fichiers sur le système de fichier à cause du risque élevé de corruption de données. Pour contourner ce problème, la solution consiste à créer un périphérique de type caractère (*character device*), d'utiliser un programme dans l'espace utilisateur pour lire les données provenant de ce périphérique et de créer un fichier.

Du côté du noyau, il faut créer une structure de données pour stocker les informations et implémenter la fonction de lecture du périphérique. Lorsqu'un processus accède au périphérique, le pilote dans le noyau appelle cette fonction qui lui retourne les données. Pour limiter les risques de dépassement de tampon, nous avons implémenté la structure de données du tampon circulaire (*ring buffer*) avec limite.

Ainsi, il est impossible de dépasser les limites de taille, mais on empêche également l'écriture au dessus de données non lues.

Cette solution n'est pas optimale et sera remplacée par le système de fichier *relayfs* afin de garantir un meilleur niveau de performance. Ce système de fichier est conçu explicitement pour régler le problème de création de fichiers depuis l'espace noyau. Cependant dans le cadre de ce projet, la solution du périphérique de type caractère était plus appropriée étant donné le temps nécessaire pour la mettre en place.

Modification des paramètres d'un module : sysctl :
Nous avons vu que le module possède plusieurs modes de fonctionnement et qu'il requiert différents paramètres pour fonctionner. Afin d'éviter de recharger celui-ci à chaque changement de paramètre, nous avons implémenté une interface *sysctl* pour le contrôler pendant qu'il est chargé.

Cette interface utilise le pseudo-système de fichier */proc* pour communiquer. Chaque fichier dans `/proc/sys/debug/kolumbo` est lié à une fonction de lecture et d'écriture. Dés qu'une valeur est lue ou écrite, la variable correspondante dans le code est accédée. Ainsi il est possible de changer le numéro de processus à tracer en utilisant la commande `echo 444 > /proc/sys/debug/kolumbo/pid`. Dans le noyau la fonction correspondant à l'écriture dans le fichier *pid* déclenche l'arrêt des actions en cours, vide la liste des processus en cours d'observation et remet le module dans l'état initial. Tout ceci est implémenté à l'aide de variables d'état.

Sysctl possède également une API permettant d'accéder aux variables dans */proc* directement depuis un programme ou avec la commande *sysctl*. Ainsi la commande `sysctl debug.kolumbo.pid=444` aura le même effet que la commande précédente.

4 Conclusion

Kolumbo est un module noyau autonome dont le but est de faciliter l'étude des *malwares* Linux. Il est équipé de plusieurs fonctionnalités permettant de contourner les protections anti-analyse. Ce projet est encore jeune est plusieurs fonctionnalités sont en cours d'élaboration. Les deux priorités sont : ajouter le support des architectures 64 bits et la gestion de *sysenter/sysexit*.

À plus long terme, Kolumbo va devenir un module permettant aux outils déjà existants (tels que *gdb* ou *strace*) de fonctionner même pour l'analyse de binaires protégés contre l'analyse. Étant donné qu'il existe déjà de très bons outils pour l'analyse dynamique, le but est de faciliter leur utilisation même dans les cas plus complexes en ajoutant le code côté noyau rendant les protections anti-analyse inefficaces.

5 Remerciements

Je tiens à remercier Nicolas Bareil et Frédéric Tronel pour les relectures et les conseils, Justine Dieppedale pour sa collaboration dans la réalisation de ce projet et Gabriel Girard de l'Université de Sherbrooke pour m'avoir aidé à réaliser ce projet.