UberLogger : un observatoire niveau noyau pour la lutte informatique défensive

Ion Alberdi, Jean Gabès, and Emilien Le Jamtel

ENSEIRB
Talence, France,
http://uberlogger.rstack.org

Dans l'étude des grands prédateurs du monde animal, deux méthodes existent : soit le naturaliste se déplace sur le lieu de vie de l'animal, soit il recrée ce lieu de vie dans une réserve naturelle et analyse sans risque ses habitudes et ses méthodes de chasse. Dans le domaine de la lutte informatique défensive, ces deux approches sont également valables. Soit les scientifiques vont sur le terrain des pirates, soit ils recréent des conditions propices à la « vie » des pirates dans un environnement sécurisé dans le but de les étudier. Cette deuxième approche peut se résumer simplement : les pots à miel (« *honeypot* »). Le dispositif que nous allons présenter permet de créer un système de « *honeypots* » utilisant les particularités des plates-formes, comme par exemple *UML*[1]. Ce dispositif peut également entrer dans le cadre du « *forensics* » permettant de déterminer ainsi si une machine a été compromise et est encore utilisée par un pirate, ou bien dans celui d'une machine « *sandbox* » permettant de tester des programmes non sûrs.

1 But de UberLogger : *honeypots*, *forensics* et *sandbox*

1.1 Les pots à miel

Ils constituent des machines « sacrifiées » dont le but premier est de se faire compromettre. Elles sont rigoureusement observées et leur utilisation ne doit pas permettre au pirate, une fois le contrôle pris, de rebondir vers une autre machine qui ne serait pas un pot à miel. Un réseau peut être créé avec un ensemble de pots à miel. Un des intérêts de notre dispositif est la possibilité de simuler un tel réseau au sein d'une unique machine physique comme nous le verrons par la suite.

1.2 Forensics

La discipline du « *forensics* » consiste à décider si une machine a été compromise et si elle est encore utilisée. Les contraintes liées à cette activité peuvent être de différentes natures suivant la machine à « autopsier » :

[1] « User Mode Linux » permet de faire tourner un noyau *GNU/Linux* en tant que processus de l'espace utilisateur, pour plus de détail se référer à [1].

- machine dont la disponibilité n'est pas une contrainte forte : dans ce cas ci, la machine peut être arrêtée, mise à part du réseau, et des méthodes classiques (ex : *Knoppix*, ...) peuvent permettre de découvrir les changements effectués par le pirate sur la machine.
- machine dont la disponibilité est primordiale : impossible dans ce cas de faire le test hors-ligne, ni de redémarrer la machine. Dans ce cas, il faut un dispositif capable de se charger/décharger dynamiquement et permettant d'observer finement le comportement du système afin de déterminer si la machine est vraiment compromise ou non. Il doit être le plus furtif possible pour ne pas éveiller les soupçons d'un éventuel pirate. Le dispositif dont nous allons parler peut servir donc dans ce cas.

1.3 SandBox

Dans le cadre du « *sandbox* », une machine est utilisée afin d'étudier le comportement d'un logiciel non sûr. On peut par exemple étudier les interactions du programme avec les fichiers du système ou les appels aux ressources réseaux. Le dispositif peut alors s'apparenter au logiciel `strace` mais avec les informations enregistrées dans une base de données. Ceci permet d'effectuer plus simplement une analyse poussée des résultats. Le dispositif est tout de même soumis à des contraintes de furtivité car il existe des logiciels intelligents qui cherchent à detecter la présence d'un traceur sur la machine et modifient alors leur comportement.

1.4 Actions du dispositif

Le but étant d'obtenir des informations sur les activités du pirate ayant pris le contrôle de la machine, il faut trouver un moyen de capturer et d'enregistrer ces données sans éveiller ses soupçons. Le pirate étant supposé avoir les droits d'administrateur, il n'est pas envisageable de placer le dispositif de capture dans l'espace utilisateur. Cacher un processus de manière totalement discrète en se restreignant à cet espace est difficile voire illusoire.

Le dispositif doit donc se placer au niveau du noyau. Une autre contrainte de sa localisation vient de l'utilisation dans le cadre du « *forensics* ». En effet, dans ce cas, la mise en place ne doit pas nécessiter le redémarrage de la machine. Il est donc souhaitable de pouvoir mettre le programme sous la forme d'un module si on le souhaite.

La position dans le noyau doit aussi faire l'objet d'attentions toutes particulières. Il ne faut pas situer le dispositif trop bas dans le noyau car, dans ce cas, il est plus ardu de séparer les informations utiles de la masse d'information créée par le passage des différentes couches du noyau. Ainsi, il faut se placer dans une couche haute du noyau. Une partie de la solution réside dans les appels systèmes.

1.5 Le rôle des appels systèmes

Ils se situent au meilleur endroit pour capter les informations de l'utilisateur. Ils constituent l'interface d'accès aux ressources du noyau pour l'utilisateur. Ils

sont bien évidemment dans l'espace noyau et ils en constituent la partie la plus haute, proposant à l'utilisateur les seuls points d'accès vers ce dernier.

La technique de capture des appels systèmes est très appréciée des pirates qui l'utilisent pour masquer leurs activités par modification de leurs comportements. Prenons par exemple l'appel *getdents* sur *GNU/Linux* qui permet de lister un répertoire. En le modifiant le pirate peut ainsi empêcher l'administrateur de voir les fichiers déposés sur la machine sans son accord.

1.6 Les informations retenues par le choix des appels

Le but du programme étant de surveiller l'activité des pirates ainsi que leurs méthodes pour prendre le contrôle des machines, le choix des appels systèmes à enregistrer est primordial. Une méthode consisterait à tous les capturer. Cependant cette méthode génère des logs à analyser d'une taille gargantuesque et il est presque impossible d'en extraire les informations les plus pertinentes.

Le programme doit donc capter suffisamment d'appels système pour pouvoir tracer les activités du pirate sans pour autant générer trop d'informations. De plus il semble intéressant de proposer la possibilité de choisir les appels à capturer à l'administrateur. La capture et le traitement des informations ne doivent pas être perceptibles par le pirate pour des raisons évidentes de furtivité.

Les principaux appels systèmes. Voici les appels que nous avons jugés les plus intéressants :

- *open*, *read* : permettent de savoir quels fichiers[2] sont lus par le pirate et de savoir quelles informations il a réussi à obtenir.
- *mmap, readv, pread* : sont les solutions utilisées par les pirates pour ne pas utiliser l'appel read et ainsi éviter un point de contrôle.
- *write* : permet de savoir quelle modification le pirate a effectué sur les fichiers[3] du système.
- *fork*, *execve* : permettent de savoir quels sont les programmes qui sont exécutés par le pirate. Ainsi, combinés avec *read*, il est possible de récupérer les outils récupérés par le pirate et de savoir s'il les a utilisés. L'appel *execve* permet entre autres de récupérer tous les arguments d'une fonction (toute la ligne de commande par exemple), l'appel *fork* permet quant à lui d'obtenir une meilleure traçabilité pour connaître les liens de parenté entre les différents processus.
- *chmod*, *chown* : permettent de déterminer les modifications effectuées par le pirate concernant les permissions des fichiers du système.
- *init_module* : permet de déterminer quels modules ont été chargés par le pirate.
- *getdents/getdents64* : permettent d'obtenir le nom des répertoires listés par le pirate.

[2] au sens *Unix*
[3] au sens *Unix* ici aussi

- *setuid* : permet de savoir quels changements de droits d'utilisateurs les programmes du pirate ont effectués.
- *ioctl* : permet de savoir quelles sont les modifications sur les périphériques qui ont été effectuées par le pirate.

Certains de ces appels possèdent des variantes qui pourraient être utilisées par les pirates pour passer outre le dispositif. Ainsi, une capture des variantes des appels est également nécessaire comme par exemple *readv* pour *read*.

Les appels systèmes secondaires. L'administrateur a la possibilité d'enregistrer également d'autres appels systèmes qui ne sont pas jugés prioritaires mais qui apportent tout de même quelques informations utiles. Ces appels systèmes sont les suivants : *create_module, delete_module, chroot, capset, capget, clone, query_module, chdir, kill.*

De plus l'appel système *socketcall* peut être enregistré dans le cas des appels à : *accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, recvfrom, recvmsg, send, sendmsg, socket, socketpair, sendto, shutdown, setsockopt.*

Le nombre d'appels systèmes secondaires étant relativement important, ils ne seront pas tous décrits. Ils constituent des appels classiques permettant d'obtenir des informations sur le système et son environnement. On note la possibilité d'enregistrer l'appel système *socketcall* lors d'appels à *bind* ou *connect* par exemple. Ainsi l'administrateur pourra savoir si le pirate ouvre un port sur la machine ou s'il se connecte à une autre machine.

Limites sur la quantité d'information envoyées. Lorsque la quantité d'informations collectée et envoyée est trop volumineuse, les performances de la machine ou du réseau chutent visiblement. Ce problème est exploité par les pirates pour « *fingerprinter* » un *honeypot*. Par exemple la *dd-attack* consiste à envoyer une requète ping à la passerelle, à lancer une copie de `/dev/zero` dans `/dev/null` et de renvoyer une requète ping à la passerelle. une différence remarquable entre les deux ping implique alors la présence d'un dispositif capturant des appels systèmes et les envoyant sur le réseau.

Une première solution consiste à essayer de détecter qu'une information est intéressante et qu'elle ne résulte pas d'une opération qui a pour but d'inonder le dispositif d'informations inutiles. Il est cependant difficile de faire ce tri, car il faudrait pouvoir détecter toutes les différentes opérations inutiles qu'un pirate peut faire : on peut facilement détecter une copie de `/dev/zero`, mais si le pirate fait une copie de `/dev/random`...

Nous avons choisi d'utiliser la solution alternative suivante : lorsque le flot de données envoyées est trop important (le débit limite permettant de rester furtif étant laissé à l'appréciation de l'administrateur), les paquets qui arrivent en trop sont capturés aléatoirement. Plus précisément, un compteur de débit mesure régulièrement le débit courant et lorsque celui-ci devient trop important, on passe en mode rejet (et on revient en mode normal lorsque ce débit est acceptable). Dans ce cas, soit T une période de temps et max_donnee le nombre

d'octets que nous pouvons envoyer sans dépasser le débit maximum, on peut alors par exemple décider d'envoyer un paquet sur la période T de la facon suivante :

```
/*Initialisation au début de la période*/
donnee_restante = max_donnee;

/*Durant les appels de la période*/
si donnee_restante <=0 fin;
octet_a_envoyer = nb_octet_logge_par_appel();
decision = pile_face();/*Di oui ou non aléatoirement*/
si decision
 max_donnee= max_donnee - octet_a_envoyer;   envoyer_information();
sinon fin;
```

Ainsi le nombre de paquets envoyés durant cette période n'est pas loin de *max_donnee* ce qui nous assure d'avoir un débit proche de *max_donnee*. Certes, il y a des paquets intéressants qui seront perdus, cependant le pirate ne pourra jamais avoir la certitude que ses techniques ne seront pas dévoilées car il ne peut pas prévoir quels paquets seront choisis. De plus, il ne pourra plus utiliser d'attaque semblable au *dd-attack* pour détecter un « *honeypot* ». Nous pouvons enfin n'appliquer ce filtre que sur les appels gourmands en information : à savoir : *read, pread, readve* et *mmap* et garder toutes les informations sur les autres.

Informations communes aux captures. À chaque capture d'appel système, le moment auquel a été effectué l'appel (secondes et micro-secondes), le *pid* du processus, l'*uid* et l'*euid* de l'appelant, les *capabilities* (*effective, inheritable et permitted*) et la valeur de retour sont enregistrés ainsi que des informations spécifiques à chaque appel système. Toutes ces informations forment ainsi un paquet qui est transmis à un démon par une méthode que l'on verra par la suite, pour y être analysées et insérées dans une base de données.

ma

2 Méthode de capture d'information et furtivité

Il existe au moins deux façons de modifier le comportement du noyau pour récupérer les informations intéressantes. Cette étude n'a pour l'instant été menée que pour *GNU/Linux*, et ne s'applique donc pas forcément aux autres systèmes.

2.1 Méthode statique

L'ensemble des méthodes utilisées passe par une modification des sources du noyau et une recompilation de ce dernier. Voici deux façons de capturer les informations venant des appels systèmes :

1. modification du code des appels systèmes,
2. modification de l'handler des appels systèmes. Le processus consiste ici à détecter quel appel système est demandé et à récupérer les informations nécessaires avant d'appeler l'appel système. Ce handler est dépendant de l'architecture (voir `/usr/src/linux/arch/<arch_choisie>/entry.S`).

Ces différentes modifications ont l'inconvénient d'être facilement détectables.

Pour tous les exemples qui sont fournis dans cette partie, le code de *sys_read* (`linux/fs/read_write.c`) a été modifié de cette manière :

```
void coucou(void){
}

asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count)
{
ssize_t ret;
struct file * file;
ret = -EBADF;
file = fget(fd);
coucou();
        ......
```

Si la version décompressée du noyau est présente sur la machine, il suffit de regarder dans les symboles définis dans l'exécutable s'il n'y a pas de fonctions non classiques :

```
#nm vmlinux | grep coucou
c012fb18 T coucou
```

Cet argument suppose quelques conditions :

1. Le *vmlinux* est disponible,
2. L'image du noyau possède les symboles nous trahissant.

La première condition n'est pas forcément vérifiée, en effet un administrateur peut effacer le *vmlinux* et ne laisser que la version compressée du noyau, cependant vu que `lilo` ou `grub` décompressent cette image au démarrage, cette action est faisable[4].

Il en est de même pour la seconde condition car il est possible d'enlever ces symboles à l'aide d'un programme tel que `strip` [5] ou sinon de programmer les modifications sans fonctions ni variables globales ou statiques. Cet argument permet de résoudre le problème des symboles.

[4] nous n'avons pas testé cette affirmation car il ne nous a pas semblé rentable de passer du temps sur cet aspect car ce n'est pas le seul argument contre la solution patch.

[5] l'utilisation de l'option -N est plus que recommandée pour ne pas enlever tous les symboles car sinon le pirate se posera des questions en voyant un noyau sans symbole et le compilateur ne pourra pas éditer les liens des modules utilisant les symboles enlevés.

Si la technique précédente ne fonctionne pas, il reste la possibilité de désassembler le code des fonctions modifiées[6].

Pour l'exemple, le symbole coucou a été supprimé du binaire :

```
#strip -N coucou vmlinux
```

Pour détecter la modification il suffit d'utiliser *gdb* pour désassembler la fonction soupçonnée :

```
#gdb vmlinux
(gdb)disas sys_read
0xc012fb1c <sys_read+0>:        push   %ebp
0xc012fb1d <sys_read+1>:        push o  %edi
0xc012fb1e <sys_read+2>:        push   %esi
0xc012fb1f <sys_read+3>:        push   %ebx
0xc012fb20 <sys_read+4>:        mov    0x14(%esp),%eax
0xc012fb24 <sys_read+8>:        mov    0x1c(%esp),%ebp
0xc012fb28 <sys_read+12>:       call   0xc01307ac <fget>
0xc012fb2d <sys_read+17>:       mov    %eax,%esi
0xc012fb2f <sys_read+19>:       call   0xc012fb18 <-appel à coucou
0xc012fb34 <sys_read+24>:       test   %esi,%esi
.....
```

et de comparer le code obtenu avec le code classique de la fonction.

```
#gdb vmlinux
(gdb)disas sys_read
0xc012fb18 <sys_read+0>:        push   %ebp
0xc012fb19 <sys_read+1>:        push   %edi
0xc012fb1a <sys_read+2>:        push   %esi
0xc012fb1b <sys_read+3>:        push   %ebx
0xc012fb1c <sys_read+4>:        mov    0x14(%esp),%eax
0xc012fb20 <sys_read+8>:        mov    0x1c(%esp),%ebp
0xc012fb24 <sys_read+12>:       call   0xc01307a0 <fget>
0xc012fb29 <sys_read+17>:       test   %eax,%eax <-pas d'appel
                                               à coucou
0xc012fb2b <sys_read+19>:       mov    $0xfffffff7,%edi
0xc012fb30 <sys_read+24>:       mov    %eax,%esi
.....
```

Si le symbole définissant l'appel système modifié a été enlevé du binaire, *gdb* ne pourra le désassembler.

```
#strip vmlinux
```

Cependant, l'adresse de la fonction se trouve dans le *System.map*, et une fois l'adresse récupérée, le code assembleur est obtenu de cette manière :

[6] Une méthode pour désassembler une image compressée est présentée dans [13].

```
#grep sys_read System.map
c0124e54 T sys_readahead
c012fb1c T sys_read
c012ff4c T sys_readv
c0135c80 T sys_readlink
#gdb vmlinux
(gdb) disas 0xc012fb1c 0xc012fb3c
Dump of assembler code from 0xc012fb1c to 0xc012fb3c:
0xc012fb1c:     push   %ebp
0xc012fb1d:     push   %edi
0xc012fb1e:     push   %esi
0xc012fb1f:     push   %ebx
0xc012fb20:     mov    0x14(%esp),%eax
0xc012fb24:     mov    0x1c(%esp),%ebp
0xc012fb28:     call   0xc01307ac
0xc012fb2d:     mov    %eax,%esi
0xc012fb2f:     call   0xc012fb18 <-appel à coucou
0xc012fb34:     test   %esi,%esi
0xc012fb36:     mov    $0xfffffff7,%edi
0xc012fb3b:     je     0xc012fbeb
End of assembler dump.
```

L'adresse de fin de lecture n'a pas besoin d'être connue car il suffit de tâtonner différentes valeurs pour ça.

Le fichier System.map a été utilisé dans cet exemple pour trouver l'adresse du code de *sys_read*, cependant on peut se passer de ce fichier en analysant `/dev/kmem` [7].

Finalement ces différents arguments nous ont paru suffisamment convaincants pour abandonner les méthodes statiques.

2.2 Méthode dynamique

Il s'agit ici de modifier le comportement du noyau durant son exécution. Les méthodes dynamiques ont pour avantage de ne pas laisser de trace sur le binaire *vmlinux*. De plus, la méthode est idéale pour l'utilisation en forensics : dans le cas d'un soupçon sur la contamination d'une machine, il n'est pas question de réinstaller un nouveau système avant de savoir si une compromission s'est réellement produite. L'administrateur pourra alors charger le dispositif sur la machine et capter de manière discrète les opérations effectuées sur le système. Si la machine n'est pas contaminée, alors ce dernier pourra décharger le dispositif sans avoir causé de dommage au système.

Deux méthodes dynamiques principales peuvent être énumérées :

[7] nous n'allons pas détailler ici cette méthode. Pour plus de renseignements se reporter à [5].

1. module : il faut pour cela que le noyau soit compilé avec le support *LKM* (Loadable Kernel Module),
2. accès à `/dev/kmem` : `/dev/kmem` est un périphérique virtuel qui présente l'image de la mémoire du noyau. Le rootkit SucKIT utilise cette méthode qui est cependant dépendante de l'architecture. Les problèmes relatifs à `/dev/kmem` sont énumérés plus loin dans l'article.

L'insertion de module par `/dev/kmem` est une action compliquée, dépendante de l'architecture, et les implémentations connues à ce jour ne fonctionnent pas de manière systématique[8].

Pour cela la solution *LKM* a été préférée. Même si cette méthode semble a priori moins furtive, des techniques permettent de remédier à ce problème. Il faut dans un premier temps masquer le module en lui même.

Masquage de la présence du module. Dans le cas où le programme est présent en mémoire sous forme d'un module, il ne faut pas que le pirate s'aperçoive de la présence de celui-ci. Différentes méthodes sont proposées.

- **Modification de la liste des modules :** Les références des modules sont sauvegardées (dans *GNU/Linux*) sous forme d'une liste chaînée. Ainsi un module peut se « dés-inscrire » de cette liste et le pirate ne verra pas avec des moyens habituels [9] la présence du module. Ceci est illustré par la figure 1. Cependant, des méthodes existent pour passer outre en cherchant directement dans l'espace d'adressage du noyau des occurrences de structure de module (ceci peut être empêché par l'utilisation de *capabilities*).

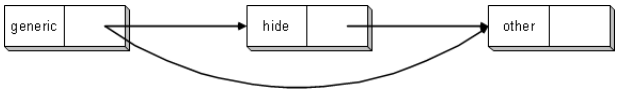

Fig. 1. Fonctionnement du saut de module

- **Solution par « fusion » de modules :**
  Les modules se présentant sous forme de fichiers *ELF (Executable and Linkable Format)*, une solution (c.f [3]) consiste à « fusionner » deux modules. Une fois les deux objets liés, en modifiant les entrées des segments *.symtab* et *.strtab* il est possible de changer les références des fonctions et de provoquer ainsi le chargement du module légitime et de celui dont on souhaite cacher la présence lors de l'initialisation du module global. La modification des références provoque le changement suivant au sein de l'objet :
  – init_module → dumm_module (appelé par hide_module),

[8] Comme nous pouvons le voir dans l'article, le développeur de SucKIT [5] le reconnaît lui méme : « Disadvantages :Non-portable, i386-linux specific,Buggy as hell ».

[9] par *lsmod* par exemple.

– hide_module → init_module.

On obtient ainsi les apparences et les fonctionnalités du module légitime ainsi que les instructions provenant du programme que l'on souhaite cacher dans un seul module. Cette méthode fonctionne sous différents systèmes :
– *GNU/Linux*,
– *NetBSD*,
– *Solaris*.
et est illustrée dans le schéma 2.

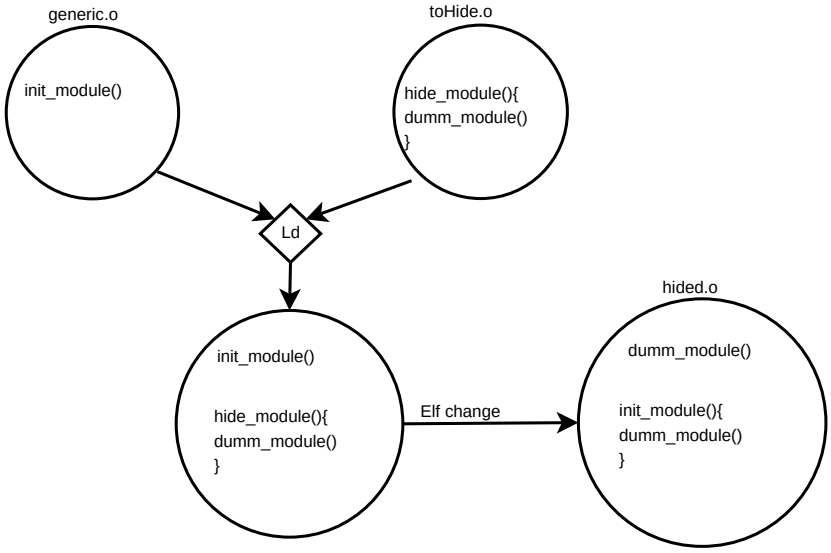

Fig. 2. Fonctionnement de la fusion de module

Une fois ce premier choix fait, plusieurs méthodes s'offrent à nous :

jump : Il s'agit ici d'écraser le code des appels systèmes existants en y mettant un Jump pour rediriger l'exécution vers nos nouvelles fonctions qui ont du coup le devoir fastidieux de réimplementer entièrement les appels systèmes.

modification de la table des appels systèmes : Les adresses des appels systèmes sont toutes stockées dans une table appelée *sys_call_table* et définie dans `linux/arch/<arch>/entry.S`, l'« handler » de l'interruption 80 appelle les fonctions depuis la valeur stockée dans ce tableau. Cette technique consiste à récupérer l'adresse de l'appel que nous voulons capturer à l'index qui lui correspond, ce qui nous permet de le réutiliser, et de remplacer cette adresse par la fonction que nous avons définie. C'est cette technique que nous avons choisie.

Cette technique a pour avantage de ne pas modifier l'appel système que nous voulons détourner. Cependant la table des appels systèmes, elle, est modifiée. Deux

sources d'informations permettent de détecter cette modification : `/proc/kcore` et `/dev/kmem`.

Masquage des modifications effectuées. Le premier périphérique trahissant la modification est utilisé avec *gdb*. Pour cela il faut préalablement lire l'adresse de la table des appels systèmes dans le *System.map* :

```
#grep sys_read System.map
c0124e54 T sys_readahead
c012fb18 T sys_read
c012ff40 T sys_readv
c0135c74 T sys_readlink

#grep sys_call_table System.map
c0206349 R __kstrtab_sys_call_table
c020bec8 R __ksymtab_sys_call_table
c020e2f0 D sys_call_table

#gdb vmlinux /proc/kcore
(gdb)x/255 0xc020e2f0
0xc020e2f0 <sys_call_table>:    0xc011d690      0xc0116db0
                                0xc01057d0      0xf9cf30d0 <-pas
                                                        sys_read
0xc020e300 <sys_call_table+16>: 0xc012fc04      0xc012f650
                                0xc012f740      0xc01171a0
.....
```

Il suffit de regarder les valeurs stockées dans le tableau et de comparer avec celles attendues. En regardant l'indice auquel la valeur a changé, on déduit l'appel système détourné (voir *arch/asm-<arch>/unistd.h* pour la correspondance indice appel système.) Pour `/dev/kmem` il suffit de lire ce fichier à l'offset : $@table\_appel\_systeme + indice\_appel * 4$ (l'unité est l'octet).

Pour pallier ce problème, il suffit de détecter une lecture dans un endroit compromettant et de remplacer dans le buffer de lecture la valeur compromettante par la valeur attendue. Cette technique a été implementée pour `/dev/kmem` et pour l'appel système *sys_read*. L'identification de la lecture sur `/dev/kmem` se fait en trouvant le majeur et le mineur du fichier lu (ces valeurs sont définies dans *linux/Documentation/devices.txt*), ensuite le buffer du read est modifié si nécessaire.

```
inline ssize_t new_read(unsigned int fd, char *buf, size_t count) {
  ssize_t r;
  struct file *file;
  struct inode *dinode;
  unsigned long read_pos=(unsigned long)(sct+__NR_read),end_offset,current_pos;
  size_t n_octet = (size_t)sizeof(unsigned long);
```

```
  unsigned int major,minor;

  /*We get the file structure of the file descriptor*/
  file = current->files->fd[fd];
  /*We get the current offset of the file*/
  current_pos = (unsigned long)file->f_pos;

  /*We call the original read*/
  r = original_read(fd, buf, count);

  /*We check if an error occured*/
  if( r > 0 ){

    /*We get the inode of the file*/
    dinode = file->f_dentry->d_inode;

    major = (unsigned int)MAJOR(dinode->i_rdev);
    minor = (unsigned int)MINOR(dinode->i_rdev);
    /*We get the major and the minor of the file*/
    if ( major == 1 && minor == 2 ) {
      /*We are in kmem (see linux/Documentation/devices.txt*/
      /*We see how far the read has gone*/
      end_offset = current_pos + r;

      /*We check if we have work to do*/
      if ( current_pos <= read_pos <= end_offset ){
/*We have to work!*/
size_t difference = (size_t)(end_offset - read_pos + 1);
size_t n_to_write = (n_octet < difference) ? n_octet : difference;
unsigned int where_start = (unsigned int)read_pos - current_pos;
/*We erase the value that betrays us in the buffer*/
copy_to_user((void*)(buf+where_start),(void*)(&original_read),
                    n_to_write);
      }
    }
  }
  return r;
}
```

Il nous reste à redéfinir les autres appels qui peuvent être utilisés pour lire cette valeur (à savoir *pread,readv et mmap*) et à implémenter la même fourberie sur `/proc/kcore`.

3 Robustesse du dispositif

Le premier besoin d'un « *honeypot* » est de recueillir un maximum d'informations pertinentes, le second est la furtivité du dispositif tandis que le troisième est d'empêcher autant que possible la désactivation du dispositif.

Il est supposé dans cette partie que l'assaillant a réussi à avoir un accès root sur la machine, de plus seul le cas *GNU/Linux* sera traité.

Le noyau Linux offre la possibilité de partitionner le pouvoir absolu de root. Ces différents droits s'appellent « *capabilities* » et sont définis dans

`include/linux/capability.h`

des zones critiques du noyau sont encerclées par des blocs du type qui offrent donc un moyen de gérer plus finement les privilèges.

`extrait de linux/kernel/sys.c`

```
asmlinkage long sys_reboot(int magic1, int magic2, unsigned int cmd,
                           void * arg)
{
char buffer[256];

/* We only trust the superuser with rebooting the system. */
if (!capable(CAP_SYS_BOOT))
return -EPERM;
....
```

Pour désactiver le dispositif, le pirate doit trouver un moyen pour accéder au noyau et désactiver les fonctions de log. Une première possibilité qui s'offre à lui est d'insérer un module qui redirige les appels systèmes vers ses fonctions. La solution consiste à enlever CAP_SYS_MODULE lors de l'insertion du dispositif, ce qui empêchera d'enlever et de rajouter des modules à tous les utilisateurs du système.

Le pirate a une autre possibilité bien plus sauvage qui consiste à redémarrer la machine car le noyau ne chargera pas automatiquement le module, ou bien de compiler un nouveau noyau et de démarrer sur celui ci. Pour cela CAP_SYS_BOOT est aussi enlevé.

Il lui reste encore une possibilité qui est utilisée par le rootkit *SucKIT* qui s'insère dans le noyau par `/dev/kmem`, il faut donc empêcher l'écriture dans `/dev/kmem`. Enlever la « *capability* » CAP_SYS_RAWIO offre une solution à ce problème.

On peut a priori se dire qu'une fois ces mesures prises en compte le système est robuste. Cependant, deux arguments montrent que cette idée est fausse.

3.1 Failles dans le noyau permettant d'avoir l'accès root avec les « *capabilities* » que l'on a enlevé

Il se peut qu'une faille permette au pirate de retrouver ces « *capabilities* » et de désactiver notre dispositif. La solution proposée par notre dispositif est de

vérifier que le processus courant ne dispose pas des « *capabilities* » enlevées à chaque appel d'un appel système détourné. Le cas échéant une alarme est envoyée au serveur du dispositif qui selon le choix de l'administrateur peut par exemple décider de couper l'alimentation de cette machine. Cette technique consistant à tuer une machine à laquelle on ne fait plus confiance, est utilisée dans les « *clusters* » pour tuer un noeud qui pourrait endommager un système de fichier partagé entre les noeuds se nomme : *STONITH* (Shoot the other node in the head). Des périphériques permettant d'implémenter cette technique sont disponibles sur le marché, pour plus de détail voir [14]. La technique *STHITH* (Shoot the « *honeypot* » in the head) s'avère efficace dans ce cas.

3.2 /dev/kmem par *mmap*

Si les entrées/sorties, sur `/dev/kmem` sans CAP_SYS_RAWIO sont impossibles, il est possible de modifier la mémoire du noyau en utilisant l'appel système *mmap* sur `/dev/kmem`. Ce problème a été identifié par l'équipe de *grsecurity*[10] qui n'a pas encore donné de solution convaincante car des programmes comme *XFree* ont besoin d'éxécuter *mmap* sur `/dev/kmem`. Une solution consiste à empêcher une telle éxécution en modifiant par exemple l'appel système *mmap*, mais présente deux inconvénients :

1. si le « *honeypot* » a un serveur X (cependant un « *honeypot* » n'a a priori pas besoin de serveur X) ou un programme nécessitant cette fonctionnalité, la machine ne pourra pas tourner convenablement,
2. furtivité, les serveurs qui empêchent cet accès à `/dev/kmem` ne sont pas courants, un pirate qui essaye en vain cette méthode pourrait se poser des questions.

Nous ne pouvons pas proposer de meilleure solution que de déléguer ce problème à l'administrateur du dispositif.

4 Méthode d'envoi des informations

4.1 Cas d'un réseau de honeypots sur une machine

Le premier but du programme était de pouvoir implémenter un réseau de « *honeypots* » sur une seule machine. La solution consiste dans ce cas à lancer n *UML* connectés sur un même réseau local et d'enregistrer l'activité de chacune de ces fausses machines dans une base de données située sur l'« *host* » en créant n canaux de communication *UML* $\rightarrow$ Host. Le dispositif est installé en tant que module caché dans chaque *UML*. Comme nous l'avons vu précédemment, cette solution nous parait la plus adaptée pour la création de *honeypots*. Une telle communication est possible depuis le noyau de l'*UML* qui a accès au système de fichiers de l'« *host* » à l'aide de fonctions spécifiques à *UML* : *os_open_file* et *os_write_file* implémentées pour la fonctionnalité de *tty_logging* proposée par

[10] voir `http://www.sikurezza.org/mlangel/msg00063.html`.

UML afin d'enregistrer les activités sur les *tty*. À l'aide de ce mécanisme chaque *UML* écrit sur un pipe nommé qui lui est assigné sur le système de fichier de l'« *host* » et un processus démon sur l'« *host* » se charge de récupérer les données et de les envoyer à la base de données par un mécanisme décrit par la suite. Le point important dans ce mécanisme est le fait que la seule communication critique au vue de l'*UML* concerne celle écrivant dans le pipe nommé. Or le système sur l'*UML* ne peut pas observer cette communication qui est donc totalement discrète.

Le plus grand avantage de cette configuration est le coût d'un tel dispositif : pour la création d'un réseau de n honeypots, n machines sont économisées ($n-1$ « *honeypots* » + la machine hébergeant la base de données).

Le premier inconvénient qui vient à l'esprit est la charge que la machine en question va devoir supporter. Il semble plus raisonnable de stocker la base de données sur une autre machine afin d'alléger sa charge. Cette architecture peut être observée sur la figure 3.

La furtivité du dispositif. La furtivité du dispositif est basée sur deux principaux éléments :

- **le fait que le pirate ne puisse pas accéder à l'« *host* »** ce qui implique dans un premier lieu de faire tourner l'« *host* » sur un noyau patché afin que les *UML* fonctionnent en *skas* mode (mode qui implémente la séparation des espaces utilisateurs et espace noyau de l'uml) car par défaut *UML* fonctionne avec un seul espace d'adressage ce qui permet à un processus en espace utilisateur d'écrire dans la mémoire du kernel de l'uml qui est dans l'espace utilisateur du « *host* » et offre donc la possibilité d'atteindre le système de fichier de l'« *host* » par exemple.
- **le pirate ne doit pas pouvoir « *fingerprinter* » un *UML*.** En effet, le système *UML* n'est pas très répandu sur les systèmes en production et ainsi, la détection de cette particularité par le pirate pourrait lui paraître suspecte. Si la première condition paraît réalisable, cette condition est beaucoup plus difficile à implémenter comme le montrent les divers travaux effectués dans le domaine.

L'architecture du système par *UML* se trouve dans la figure 3. Ce dispositif permet de créer un réseau de « *honeypots* » de faible coût mais sa qualité dépend de paramètres incertains comme la capacité du pirate à « *fingerprinter* » les *UML* et prendre le contrôle de l'host. Des dispositifs plus classiques permettent de créer un réseau de « *honeypots* » plus robuste mais avec des canaux de communication moins discrets.

4.2 Cas non *UML*

Dans le cas d'une configuration plus classique, chaque « *honeypot* » est une machine physique. Les cas du « *forensics* » et des « *sandboxes* » s'apparentent au cas où le réseau est constitué d'une unique machine. Le dispositif doit ainsi envoyer les données vers un processus démon situé sur une machine distante afin

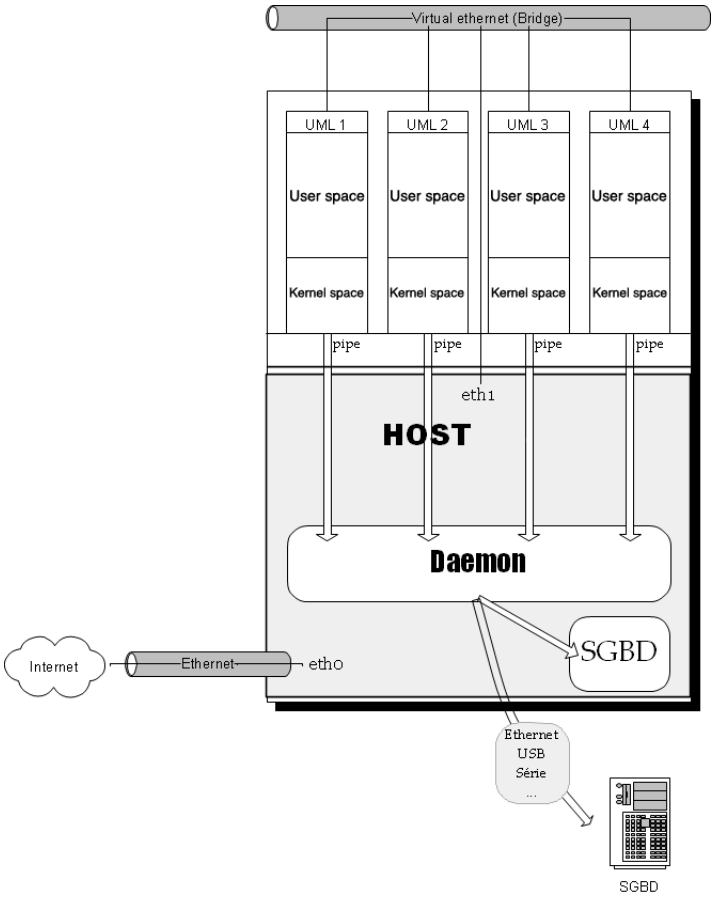

Fig. 3. Fonctionnement dans le cas *UML*

de les traiter et de les enregistrer dans une base de données. Différents supports sont possibles pour la communication.

Les supports de communication. Plusieurs canaux sont possibles pour l'envoi de paquet :

- **communication par ethernet :** la liaison est directe entre la machine et le démon/base de données. L'inconvénient ici est que le pirate à de nombreux outils à disposition pour « *sniffer* » le trafic véhiculant les informations qui trahissent le dispositif et ainsi de le détecter mais des moyens étudiés par la suite permettent de mettre en défaut ces méthodes.
  Différentes solutions sont possibles quant à l'architecture du réseau de capture :
  1. Une solution proposée par l'outil « *HoneyWall* » propose de mettre en place une machine capturant les informations envoyées par le « *honeypot* » de manière discrète. Le pirate est relié au réseau de pot à miels par cette machine qui fait office de « bridge » . De plus, le système permet de limiter et d'analyser le flux (voir le site de Snort inline [9]) réseau sortant. Les risques de rebonds du pirate à partir du « *honeypot* » sont donc diminués. Ce système a pour inconvénient de pouvoir donner accès au dispositif démon/base de données si le pirate arrive à prendre le contrôle de la passerelle.
  2. Une autre solution proposée repose sur le fait que la machine qui enregistre les données se situe derrière un « TAP » (Test Access Point) (voir le paper sur Sebek BSD [10]) ce qui permet d'avoir une furtivité accrue et un risque limité de compromission du serveur, celui-ci ne pouvant répondre. Cette technique seule ne permet pas cependant de limiter le flux sortant et peut permettre ainsi au pirate de rebondir à partir des « *honeypots* » .

  Ainsi, une solution convenable repose dans l'utilisation de ces deux techniques :
  - une passerelle permet de filtrer les flux sortant des « *honeypots* » et ainsi de limiter les risques de rebonds,
  - la machine capturant les informations des « *honeypots* » se situe derrière un « TAP » ce qui limite ses risques de compromission.

  Cette architecture peut être vue dans la figure 4.
- **communication par port série ou USB :** ici le pirate ne dispose pas d'autant d'outils pour sniffer la communication que par l'ethernet. L'inconvénient est que ici le débit du port série est insuffisant pour transporter la somme d'informations fournie par le dispositif. Il peut cependant s'avérer suffisant lorsque l'administrateur souhaite capter un nombre limité d'appels systèmes sur une seule machine.
  En ce qui concerne le port USB, cette solution permet en utilisant une connexion de type *bulk transfert* d'avoir un débit relativement élevé.
  La furtivité de ces deux méthodes repose principalement sur le fait que ces canaux sont moins répandus que l'*Ethernet* pour l'envoi d'informa-

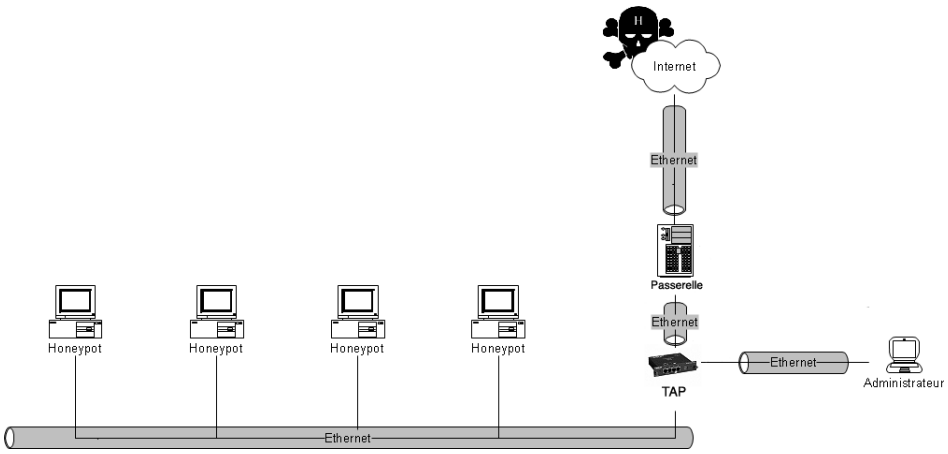

Fig. 4. Architecture classique

tion. Cependant, il n'existe pas encore autant de méthodes pour cacher les communications sur ces canaux que par *Ethernet*.

Le choix de la solution Ethernet. Le dispositif étudié ici repose ainsi sur l'utilisation du protocole ethernet. Cette solution paraît plus raisonnable car ce protocole a été conçu pour les réseaux contrairement à l'*USB*. Dans ce cas, il faudrait n câbles et ainsi n points d'entrée sur la machine distante si on n'utilise pas de concentrateur. Cependant, la solution *Ethernet* pose des problèmes en ce qui concerne la furtivité des échanges. En effet, sans contre mesures, le pirate pourrait les voir.

Solutions de dissimulations.

- Modification de la pile réseau du noyau. Une solution envisageable est proposée par *Sebek*. Elle consiste à modifier la pile *Ethernet* des noyaux des machines sur le réseau des « *honeypots* » pour qu'elles ne prennent pas en compte des paquets ayant un champ spécial (*Magic Number*). Cette solution est efficace et a déjà fait ses preuves. Contrairement à *Sebek*, le dispositif envoi les informations sur des trames *Ethernet* pures car il est destiné à une étude « locale » des informations et non pas sur une étude « distante » (réseaux distants). Ainsi, l'utilisation des couches IP et supérieures n'est pas utile dans ce cas. Cette solution est ainsi la première à être mise en oeuvre pour ce dispositif dans le cas de réseaux classiques d'« *honeypots* ».
- Utilisation de *cover channel*. Une autre solution, non encore implémentée, consiste en la dissimulation d'informations dans des communications qui ne seraient pas perçues par le pirate comme inhabituelles. Un exemple est la dissimulation d'informations au sein de paquets provenant du protocole de *spanning-tree* ou *DNS* grâce à des techniques de canaux cachés. Cette technique a cependant pour désavantage que les considérations de débit

et de furtivité sont contradictoires. Cette solution peut être envisageable pour le transport d'un nombre très restreints d'appels systèmes.

5 Stockage, traitement et analyse des informations

Après avoir vu les différents procédés mis en place pour récupérer les informations de manière discrète, il est nécessaire de mettre à disposition de l'administrateur des outils permettant d'utiliser les données ainsi capturées. La première chose à mettre en place est donc de stocker ces informations de manière intelligente, puis de définir des outils permettant de tirer de cette grande quantité de données les choses intéressantes.

5.1 Stockage des informations

Le premier choix à faire pour le stockage des données est le système de gestion de base de données à utiliser. MySql est adapté pour plusieurs raisons :

- son utilisation est très répandue, et donc implémentée sur la majorité des systèmes d'exploitation,
- son utilisation est simple, grâce à des outils puissants d'administration,
- sa licence est open source, et compatible avec la nôtre.

Une fois ce choix fait, il ne reste plus qu'à construire la base de données. La solution la plus simple est de définir une table pour chaque appel système. Ce choix permet alors d'exécuter rapidement des requêtes sur la base de donnée (autant en écriture qu'en lecture). Certes, un système moins gourmand en espace disque peut être mis en place, mais il semble préférable de favoriser la simplicité des requêtes (et donc la performance). À noter tout de même une particularité pour l'appel système *execve* qui possède un nombre variable d'arguments : en effet, une table supplémentaire est créée, prenant une entrée pour chaque argument de chaque appel *execve*.

5.2 Conception et cadre d'utilisation de l'interface

Une fois les informations stockées dans la base, de multiples solutions existent pour les afficher. Pour les mêmes raisons que le choix du système de gestion de base de données, l'utilisation de scripts *PHP* semble la solution la plus appropriée. En effet, ce langage est très répandu et simple à implémenter. De plus, il s'associe à la perfection avec *MySQL*. L'affichage des informations recueillies permet donc à l'administrateur de repérer les activités louches.

Ainsi, dans le cas d'un « *honeypot* » , il peut détecter une utilisation non-prédéfinies lors de l'installation du système. Il pourra donc en retirer des informations sur les méthodes et les outils utilisés par l'intrus.

Dans le cas d'une utilisation en *Forensics*, l'administrateur pourra comparer l'activité de la machine avec des enregistrements effectués lors d'un fonctionnement normal.

Dans le cas *Sandbox*, il apparaît évident que ces informations permettront de tester la réelle activité des applications installées.

5.3 Analyse des résultats

Devant la quantité non-négligeable d'informations recueillies par *Uberlogger*, il semble nécessaire de mettre à disposition de l'administrateur des outils lui permettant d'interpréter et de trier ce flot d'information. Le langage *PHP* permet de répondre efficacement à ce besoin à l'aide de scripts interagissant avec les données de la base.

Plusieurs outils sont actuellement disponibles, et de futurs outils sont en cours de développement.

La première nécessité était de permettre à l'administrateur de trier les résultats selon ses besoins, que ce soit par type d'appels système, par ordre chronologique ou par « *honeypot* » (dans le cas d'un réseau de « *honeypot* »).

Un simple tri n'étant pas suffisant pour repérer les activités louches réalisées sur un système comme l'envoi de « *Shellcode* » par exemple, une fonction de recherche dans les champs stockés dans la base est en place. Cet outil permet de choisir d'abord sur quel appel effectuer la recherche, puis sur quel champ en particulier, cette fonctionnalité est illustrée dans la figure 5.

Quand cela peut s'averer utile, le contenu de certains champs est également affiché en Héxadécimal, comme illustré dans la figure 6.

Dans la prochaine version de l'outil, il sera possible d'effectuer des recherches croisées entre plusieurs appels système (pour repérer toutes les actions effectuées par un utilisateur sans avoir à mettre en œuvre plusieurs recherches par exemple). Une autre fonctionnalité à mettre en place sera de permettre une recherche dans un intervalle de temps prédéfini.

Une piste qui nous a paru intéressante à explorer est l'analyse statistique des résultats. Une augmentation dans l'utilisation de certains appels système par exemple permettra en effet de prouver la présence d'une activité inhabituelle et donc une intrusion. Un script permet actuellement d'afficher un graphique représentant la quantité d'appels utilisées sur un intervalle donné voir 7 , et il sera intéressant de mettre à disposition de l'administrateur un moyen de comparer ces résultats avec une activité considérée normale sur le système. Une alerte pourrait alors être mise en place dans le cas où l'activité sort de son cadre habituel.

Ces outils ne sont bien sûr pas encore suffisants pour permettre une surveillance poussée de l'activité des systèmes, mais d'autres viendront prochaine-

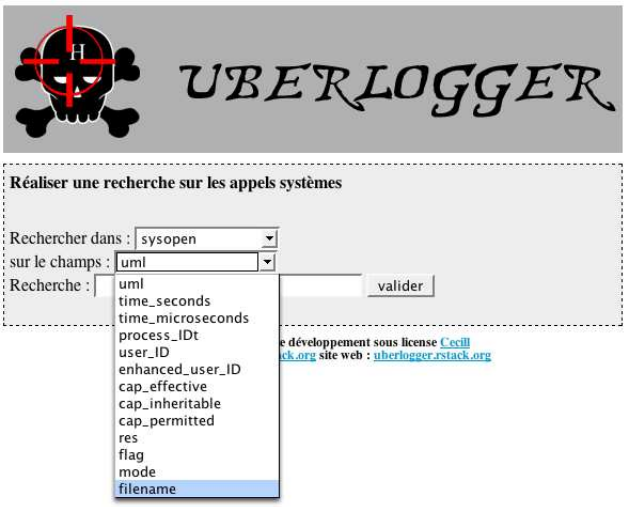

Fig. 5. Page de recherche

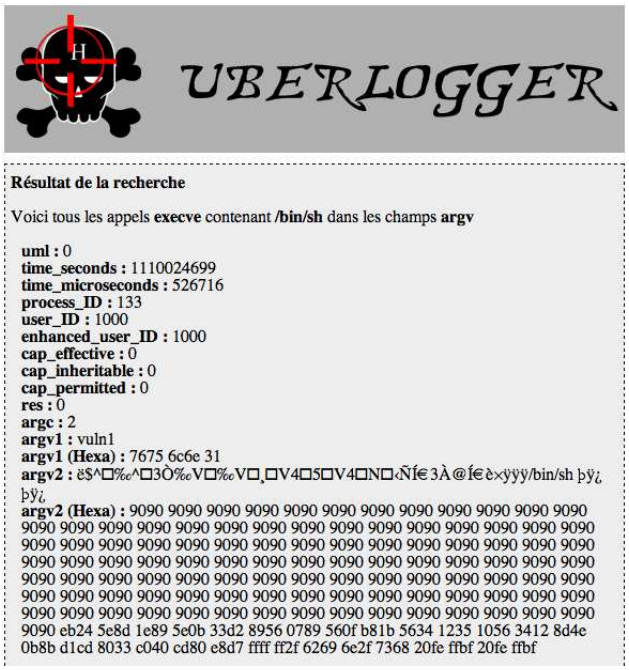

Fig. 6. Résultats de la recherche de `/bin/sh`

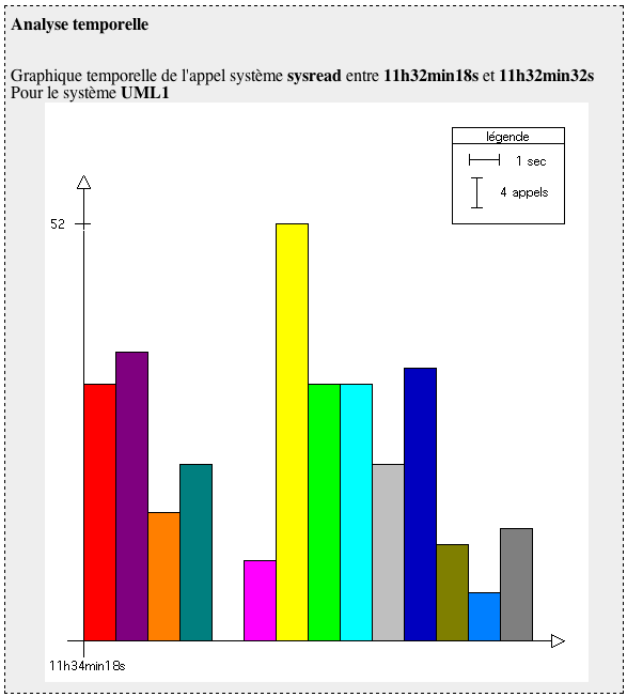

Fig. 7. Graphique temporel

ment étayer ce panel de fonctions. Nous avons notamment pensé à des outils permettant de tracer l'activité d'un utilisateur sur un diagramme.

5.4 Gestion de la taille de la base de données

La taille des données récoltées est très grande suivant les appels systèmes captés. Cette taille peut être contraignante dans un premier temps pour l'espace disque du système hébergeant la base de données et dans un second temps pour l'administrateur qui doit analyser une somme considérable d'information.

Les données trop anciennes (de l'ordre de plusieurs semaines) ne sont pas nécessaires si l'administrateur observe régulièrement son parc d'« *honeypots* » afin de savoir si l'un d'eux a été compromis. En effet, une fois les données intéressantes de l'attaque analysées et extraites par l'administrateur, ces dernières ne sont plus nécessaires sous forme brute. Il en est de même pour les données résultantes d'une activité « saine » du « *honeypot* ». Cependant, ces données sont utiles dans le cadre de l'analyse statistique de l'activité. Le dispositif peut ainsi à intervalles réguliers (de l'ordre de plusieurs semaines) procéder à une agrégation des données anciennes. Ce système permet d'avoir une sauvegarde des tendances du système permettant une analyse statistique et de réduire grandement la taille de la base car seules les informations récentes ou de tendances générales sont sauvegardées.

6 Comparaison aux solutions existantes

6.1 Sebek

Sebek 2.1.7 est une solution applicable au même domaine de la lutte informatique défensive. Elle permet de logger l'appel système *read* (plus *open*, *fork* et *socket* dans la version *3Beta*) ainsi que le nom du programme exécuté sur la ligne de commande. Cependant, elle fonctionne sous *UML* de la même façon que sur un Linux normal, c'est à dire par une communication par *UDP/IP/Ethernet*. Ainsi notre solution permet de logger plus d'appels systèmes et d'exploiter les spécificités de *UML* pour rendre la communication moins visible. Dans le cas du Linux normal notre solution repose sur l'utilisation du protocole *Ethernet* pur. Cette méthode est plus légère mais n'est pas facilement routable pour être analysée à grande distance.

6.2 Honeyd

On peut également noter le cas de *Honeyd 1.0* qui permet de créer un réseau virtuel simulé au sein d'une machine. L'attaquant se voit alors confronté à un ensemble de scripts simulant des machines physiques. Notre solution permet de créer un réseau de machines UML au sein d'une machine et ainsi le pirate est confronté aux véritables applications. Cependant, un réseau d'une taille relativement élevée peut être simulé par *Honeyd* alors que notre solution repose sur des UML et est donc plus coûteuse en termes de performance.

7 État du projet

7.1 État actuel du projet

Le projet est relativement récent (moins de six mois à l'écriture de cet article) mais est déjà bien avancé en ce qui concerne l'*UML*. En effet, sur cette plate-forme, le système est opérationnel. Le portage sur *Linux* n'est pas encore complet en ce qui concerne l'envoi par *ethernet*. Le portage sous *FreeBSD* (effectué par une autre équipe) permet l'envoi d'informations sur *ethernet* mais ne capte pas encore tous les appels systèmes. Les différentes techniques de masquage de modules sont opérationnelles. L'interface permet à l'heure actuelle de faire des recherches dans les champs des appels systèmes.

7.2 Développements proches du projet

Dans un futur proche, le portage sur *Linux* sera fini et l'envoi par *ethernet* sera totalement discret. De plus, la méthode du masquage de la présence de la capture des appels systèmes sera réalisé pour *read, readv, pread* et *mmap*. Le portage sur *FreeBSD* permettra la capture des même appels systèmes que pour *Linux*. La protection par rapport aux attaques du type *dd-attack* sera pleinement opérationnelle.

L'interface permettra d'analyser statistiquement les données récoltées et le système permettant de contrôler la taille de la base de données sera implémenté.

7.3 Futur du dispositif

Dans un futur plus éloigné, les techniques utilisant les canaux cachés pourront être utilisés afin d'étendre les possibilités offertes à l'utilisateur. Dans le même ordre d'idée, la méthode d'envoi par *USB* sera testée. Le portage sur les noyaux *Linux* 2.6 sera également étudié, le programme ne fonctionnant à l'heure actuelle que sur les noyaux *Linux* 2.4.

Conclusion

Nous avons ainsi vu une conception possible pour un composant d'un « *honeypot* ». Cet outil permet la mise en place d'un réseau de pots à miel au sein d'une machine unique ce qui permet de réduire le coût de cette infrastructure. Il permet également une mise en place plus classique avec l'utilisation de plusieurs machines physiques. Ainsi, suivant la volonté et le temps de l'administrateur, de multiples appels systèmes sont enregistrables et permettent d'augmenter la possibilité d'obtenir des informations sur les actes et les outils des pirates.

Le dispositif peut également être utilisé dans le cadre du « *forensics* » sur les systèmes à fortes contraintes de disponibilités. L'administrateur peut observer et détecter les activités non prévues du système et déterminer si la machine est compromise ou non.

Une autre utilisation possible de cet outil est dans le cadre des « *sandbox* ». En effet, si l'administrateur souhaite tester un autre outil pour vérifier les réels accès que celui-ci fait au système, il peut utiliser notre dispositif pour cela. Il pourra voir alors si des accès aux fichiers systèmes sont effectués ou si le logiciel tente de se connecter à un serveur distant.

En ce qui concerne la licence du logiciel, c'est la `Cecill` qui a été choisie car elle permet de s'intégrer parfaitement dans le cadre légal français et de permettre son développement en tant que logiciel libre.

Cet outil pourrait dans l'avenir évoluer de la lutte informatique défensive vers une lutte plus active en détectant des comportements anormaux et en agissant en fonction au sein même du noyau.

Par Alberdi Ion, Gabès Jean, Le Jamtel Emilien, nous remercions tout particulièrement Laurent Oudot et Pierre Lalet pour leur soutien tout au long du projet, ainsi que Florian Haradji, Caroline Chauvin, Sébastien Mondet et Fédéric Moulins pour leurs conseils avisés.

Plus d'informations ainsi que les sources peuvent être trouvées sur le site du projet :
`http://uberlogger.rstack.org/` ainsi que sur la mailing list `uberlogger@rstack.org`

Références

1. Jeff Dikes, *Home page User Mode Linux*, `http://user-mode-linux.sourceforge.net/`
2. Fred Raynal, *Root-kit et intégrité*, `http://www.miscmag.com/articles/full-page.php3?page=104`
3. Truff, *Techniques de fusion de modules*, `http://www.phrack.org/show.php?p=61`
4. *Detecting Kernel Compromison with gdb*, `http://www.securityfocus.com/infocus/1811`
5. sd <sd@sf.cz> and devik <devik@cdi.cz>, *Linux on-the-fly kernel patching without LKM*, `http://www.phrack.org/phrack/58/p58-0x07`
6. *Kernel Newbies*, `http://kernelnewbies.org`
7. The Honey Net project, `http://project.honeynet.org`
8. Home page de mysql, `http://www.mysql.com`
9. Home page de snort inline, `http://snort-inline.sourceforge.net/`
10. Sebek BSD paper, `http://honeynet.droids-corp.org/download/doc/sebek-bsd.pdf`
11. Home page de PHP, `http://www.php.net`
12. Daniel P.Bovet and Marco Cesati, *Understanding the Linux kernel*, 2nd Edition, O'Reilly Ed.

13. *Méthode pour débugger un noyau compressé*, `http://groups.google.fr/groups?q=+how+to+decompress+vmlinuz\&hl=fr\&lr=\\\&client=firefox\&rls=org.mozilla:en-US:unofficial\&selm=1CC2b-58s-1"%"40gated-at.bofh.it\&rnum=4`
14. Description de la méthode STONITH, `http://linux-ha.org/stonith/`