Playing with `ptrace()` for fun and profit

Injection de code sous Linux

Nicolas Bareil

EADS CRC DCR/STI/C
`Nicolas.Bareil@eads.net`

Résumé `ptrace()` est un appel système peu documenté et obscure dont l'objectif est de permettre le debugging de programmes. Néanmoins, cet appel système présente des fonctionnalités intéressantes susceptibles d'être utilisés à des fins malveillantes.

Ainsi, un accès total en lecture **et** écriture sur toute la mémoire virtuelle d'un processus (ainsi que ses registres) est fourni pour une application utilisant `ptrace()`. Qui plus est, cette prise de contrôle requiert les mêmes privilèges que l'envoi d'un signal sur un autre processus.

Dans cette partie, nous présentons les fonctionnalités de `ptrace()` et leur utilisation, en abordant les problèmes d'injection de code comme le placement des instructions dans l'espace d'adressage du processus et l'interruption des appels systèmes. Enfin, nous présentons un panel d'applications de `ptrace()`.

Nous nous attacherons à ne décrire que le comportement d'un noyau Linux (`2.6`) sur architecture `x86`.

1 Présentation de `ptrace()`

Brièvement, `ptrace()` permet d'accéder en lecture/écriture à tout l'espace d'adressage d'un processus, c'est à dire aussi bien les données (`.text`, `.data`, etc.) que les structures de contrôle comme les registres du processeur. Toutes les fonctionnalités de `ptrace()` sont réalisées à partir d'un appel système unique, tout le travail de *dispatching* est réalisé en espace noyau (*kernel space*).

Dans cette section, nous présentons d'abord le fonctionnement, puis les fonctionnalités de `ptrace()`. Enfin, nous illustrons comme l'utiliser dans vos applications.

1.1 Fonctionnement général

Avant de pouvoir tracer un processus, il est nécessaire de s'attacher à lui, les permissions requises sont les mêmes que celles nécessaires à l'envoi d'un signal à un processus, sauf en cas de modifications du noyau comme cela est fait dans le patch *grsecurity*. Cela signifie qu'il n'est pas nécessaire d'être `root` et que tout utilisateur peut *ptracer* ses propres processus (si le processus ne tourne pas avec le bit `suid` bien sûr). Cette caractéristique peut être intéressante dans le cadre de *daemons* lancés sous l'utilisateur `nobody`.

Lors d'un attachement, les droits d'accès sont vérifiés et le traceur devient le père du processus tracé. Malgré cette adoption, le vrai père est mémorisé et recevra tout de même le signal `SIGCHLD` à la mort de son fils. Afin d'interférer au minimum, les appels à `getppid()` du fils renverront le `PID` du vrai père. Dans la suite de cet article, par souci de cohérence avec l'implémentation du noyau, nous désignerons le processus tracé comme l'enfant du traceur (qui sera donc le père).

Le parent peut laisser s'exécuter le processus et ne reprendre la main qu'à la réception d'un signal ou il peut être interrompu à chaque instruction en utilisant le mode pas à pas (*singlestep*) ou lors d'un appel système :

- dans le mode de traçage instruction par instruction, on utilise le bit *Trap flag* (`TF`) directement utilisé par le processeur qui lèvera une interruption de debugging à chaque instruction exécutée,
- dans le mode de surveillance des appels système, le noyau va lui-même interrompre le processus à l'entrée et à la sortie d'un *syscall* afin de pouvoir contrôler les arguments et le résultat de la fonction. Cette fonctionnalité intervient au cœur de la routine d'accès `syscall_entry` qui vérifie immédiatement si le processus courant est tracé ou non.
  Si c'est le cas, elle passe la main à la routine `syscall_trace_entry` qui va s'occuper de prévenir le traceur, rétablir le processus et enfin exécuter l'appel système dès que le père aura rendu la main. Lorsque le *syscall* est terminé, la routine `syscall_exit_work` va notifier le père de la même façon qu'à l'entrée.
  Ces routines, écrites en assembleur, sont disponibles en annexe 3 ou dans les sources du noyau Linux (`arch/i386/kernel/entry.S`).

Portabilité Malgré sa relative omniprésence dans les systèmes UNIX, il est naïf de penser qu'on pourrait faire tourner notre code ailleurs que sur une machine spécifique (la notre en général). `ptrace()` est intimement lié au noyau, nous ne sommes donc pas assuré de pouvoir lancer une application écrite pour une version (majeure) différente du noyau : la mémoire virtuelle vue par un processus peut avoir été complètement remaniée, les alignements, la taille d'un mot peut changer, la sémantique de certains signaux modifiée, etc.

De même, entre différentes architectures, il est difficile d'assurer une quelconque compatibilité : les registres changent, leurs tailles, la capacité du processeur a accédé à des adresses mémoires non alignées, etc.

1.2 Prototype

Voici la définition de la fonction :

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request \
          , pid_t pid, void *addr, void *data);
```

Prototype La variable `request` peut recevoir les valeurs suivantes :
- `PTRACE_{TRACEME,ATTACH,DETACH}`, pour s'attacher et se détacher d'un processus,
- `PTRACE_PEEK{USR,DATA,TEXT}`, lecture de la mémoire,
- `PTRACE_POKE{USR,DATA,TEXT}`, écriture dans la mémoire,
- `PTRACE_{SET,GET}REGS`, manipulation des registres,
- `PTRACE_{SYSCALL,SINGLESTEP,CONT}`, mode de traçage ;

Avec GNU libc, `ptrace()` a la particularité d'être une fonction à arguments variables ce qui permet d'écrire de manière simplifiée certaines requêtes. Néanmoins, lors de de l'utilisation du prototype raccourcie, le comportement n'était pas forcément celui expecté.

Regardons maintenant les valeurs de `request` qui ne sont pas documentées (lors de l'écriture de cet article, un *patch* était en cours de soumission).

Manipulation des signaux Les requêtes `PTRACE_{GET,SET}SIGINFO` permettent respectivement de récupérer et modifier les signaux de l'enfant. La structure `siginfo_t` donnée en paramètre contient de nombreuses informations sur l'émetteur (PID, UID), le numéro de signal ainsi que des éléments spécifiques au signal : `SIGSEGV` fournit l'adresse qui a généré la faute par exemple.

Un processus tracé est arrêté à chaque réception d'un signal (à l'exception de `SIGKILL`), en fait, il est interrompu avant même qu'il en prenne connaissance puisque c'est le père qui reprend la main afin de pouvoir décider de la suite. Il peut ainsi laisser le signal parvenir au fils, le bloquer ou en envoyer un autre à la place.

Cette possibilité offre un moyen de détecter les protections anti-`ptrace()` basées sur l'envoi de signaux `SIGTRAP` [5] comme cela est illustré ci-dessous :

```
int stayalive;

void trapcatch(int i) {
    stayalive = 1;
}

int main(void) {
  ...
  stayalive = 1;
  signal(SIGTRAP, trapcatch);

  while (stayalive) {
    stayalive = 0;
    kill(getpid(), SIGTRAP);

    do_the_work();
  }
  ...
}
```

La « protection » repose sur le fait qu'un debugger basique (ou l'application du reverser qui utilise `ptrace()`) ne peut pas faire de distinction entre le signal `SIGTRAP` émis par un breakpoint ou en mode pas à pas et le signal qu'un processus s'envoie à lui-même. Ainsi, si le processus est tracé, tous les `SIGTRAP` seront captés par le père et le fils ne les recevra jamais, ce qui nous fera sortir de la boucle dans l'exemple montré.

Le *GNU Debugger* (gdb) est désormais capable de relayer ces signaux lorsqu'il voit que le `SIGTRAP` n'a pas été généré par un point d'arrêt (*breakpoint*) ou par un appel système. Par contre, il ne réussit toujours pas à détecter le subterfuge lorsqu'il est en mode pas-à-pas.

Il y a pourtant deux techniques possibles : si nous sommes en mode *singlestep* et que le `SIGTRAP` est généré à cause de lui, le bit `BS` sera à 1 dans le registre de debug `dr6`, 0 sinon. À la réception d'un `SIGTRAP` , il convient donc de regarder ce bit et de vérifier que nous ne sommes pas sur un point d'arrêt.

La deuxième solution est d'utiliser l'option `GETSIGINFO` afin de récupérer les informations au sujet du signal en attente, des dernières sont accessibles sous la forme d'une structure `siginfo`.

```
typedef struct siginfo {
 int si_signo;    /* numéro de signal */
 int si_errno;
 int si_code;     /* provenance: user? kernel?*/
 ...
} siginfo_t;
```

Les signaux envoyés par des utilisateurs sont remarquables par la valeur `SI_USER` dans `si_code`, nous permettant de voir qu'on doit relayer ce signal car il ne nous est pas destiné.

Options supplémentaires Le comportement de processus tracé peut être modifié en manipulant des options de `ptrace()`. Elles sont utilisées de cette façon :

```
int options = PTRACE_O_TRACEFORK           \
            | PTRACE_O_TRACESYSGOOD | ... ;
ptrace(PTRACE_SETOPTIONS, pid, NULL, options);
```

Les options de suivi de processus `PTRACE_O_TRACE{VFORK,FORK,CLONE,VFORKDONE}` permettent d'activer le traçage de tous les enfants qui seraient créés par le processus fils.

La surveillance des appels à `fork()` du fils ne suffit pas car elle est vulnérable à une *race condition*, c'est-à-dire qu'entre le moment où le père exécute une instruction après le `fork()` et l'examen des registres pour récupérer le `PID` du nouveau fils, nous ne savons pas si le fils a été exécuté par le *scheduler* !

Grâce aux options précédentes, le noyau arrête lui-même les fils avant même qu'ils ne puissent être mis en état `RUNNABLE`. Il est tout de même nécessaire au debugger de s'attacher à ces nouveaux processus.

Récupération du PID Après un `fork()` (ou équivalent : `vfork()`, `clone()`, etc.), le traceur doit récupérer le `PID` du nouveau fils. Il est possible de regarder directement la valeur dans le registre de résultat (`eax`) du fils ou bien d'utiliser une requête `PTRACE_GETEVENTMSG` qui se limite à copier la variable noyau `child->ptrace_message` en espace utilisateur.

Cette variable n'est modifiée que dans deux cas, à la création et à la destruction d'un processus tracé, respectivement pour y affecter le nouveau PID ou le code de retour (*exit code*).

L'interruption d'un appel système nous intéressera plus particulièrement dans la section suivante mais l'option `PTRACE_O_TRACESYSGOOD` nous aidera à détecter cet état car le membre `si_code` de la structure `siginfo_t` (récupérable via une requête `PTRACE_GETSIGINFO`) aura la valeur 0x80 d'ajoutée.

Accès à l'espace d'adressage Cette fonctionnalité est bien connue donc nous ne nous y attacherons pas. Il est ainsi possible de lire et écrire dans tout l'espace d'adressage du processus de cette façon :

```
/* lecture d'un mot depuis targetaddr */
errno = 0;
ret = ptrace(PTRACE_PEEKTEXT, pid, targetaddr, NULL);

if (errno && ret == -1) {
    perror("ptrace_peektext() ");
    return 1;
}
```

Sous Linux, le segment `TEXT` et `DATA` sont les mêmes donc `PTRACE_PEEKTEXT` et `PTRACE_PEEKDATA` sont synonymes.

Lorsque le père a la main sur le processus tracé, le fils est en état `STOPPED` après un changement de contexte, toutes ses informations volatiles (comme les registres) sont poussées dans l'espace d'adressage du processus et sont accessibles directement par des requêtes `PEEKUSR`.

Le tableau 1, en annexe 1 présente les adresses pour accéder aux registres avec les options `PTRACE_PEEKUSR` et `PTRACE_POKEUSR`.

Néanmoins, la méthode la plus élégante reste d'utiliser `ptrace()` avec `PTRACE_GETREGS` qui va copier beaucoup plus qu'un unique mot mémoire.

Lecture et modification des registres processeur En plus de copier tous les registres d'un coup, ils vont être rangé dans une structure `user` décrite ci-dessous et disponible dans `linux/user.h` :

```
struct user {
   struct user_regs_struct regs;
   struct user_i387_struct i387; /* Math register */
```

```
   ... /* Text, data and stack segment size (pages) */
   unsigned long start_code;     /* text's address */
   unsigned long start_stack;    /* stack's address */
   ...
   struct user_i387_struct* u_fpstate;
   char u_comm[32]; /* user command */
   int u_debugreg[8];
};

struct user_regs_struct {
    long ebx, ecx, edx, esi, edi, ebp, eax;
    unsigned short ds, __ds, es, __es;
    unsigned short fs, __fs, gs, __gs;
    long orig_eax, eip;
    unsigned short cs, __cs;
    long eflags, esp;
    unsigned short ss, __ss;
};
```

L'exemple suivant montre un *debugger* simpliste, il trace le processus en mode pas à pas et à chaque cycle, vérifie que le registre `eip` ne pointe pas sur l'adresse 0x41414141.

```
struct user luser;
long ret;
...
while (1) {
  ptrace(PTRACE_SINGLESTEP, child, NULL, NULL);
  ...
  errno = 0;
  ret = ptrace(PTRACE_GETREGS, child, NULL, &regs);

  if (ret == -1 && errno != 0) {
     perror("ptrace(getregs) ");
     return 1;
  }
  if (luser.regs.eip == 0x41414141)
    printf("Bingo!\n");
}
```

Maintenant que nous avons vu les bases de `ptrace()`, nous allons pouvoir montrer comment l'utiliser.

1.3 Utilisation de `ptrace()` dans vos applications

Lorsque vous allez vous plonger dans l'écriture d'un traceur, vous allez vite vous rendre compte que la documentation est pauvre, la seule documentation

autoritaire reste la page de manuel et... les sources du noyau. Pourtant, tout ce qu'il y a à savoir est écrit mais retrouver l'information noyée dans une masse de détail est difficile.

Par exemple, la vérification du code de retour de `ptrace()` peut vous jouer des tours lorsque vous utilisez `PTRACE_POKETEXT`. La fonction vous renvoi le contenu de la mémoire à l'adresse spécifiée, mais ce contenu peut être égale à la valeur valide -1, valeur qui est en général associée à un échec de l'appel système. La solution correcte est de vérifier la variable `errno` de cette façon :

```
#include <errno.h>
#include <sys/ptrace.h>

errno = 0;
ret = ptrace(PTRACE_POKETEXT, pid, ...);

if (errno && ret == -1) {
    perror("ptrace_poketext() ");
    return 1;
}
```

Il est temps d'utiliser la fonction ! S'amuser à programmer son propre debugger, c'est bien, injecter du code (c'est-à-dire détourner le flux d'instructions d'un processus pour lui faire exécuter un autre) dans un processus, c'est encore mieux. L'injection de code via `ptrace()` peut se réaliser par beaucoup de chemins en fonction de nos besoins : est-ce qu'on remplace le code existant ? Est-ce qu'on doit exécuter nos instructions puis rétablir le processus comme si rien ne s'était passé ? Doit-on faire tourner le code en parallèle ? La section suivante tente de discuter de ces problématiques.

Où placer les instructions à exécuter ? Afin d'être exécuté, le code injecté (que nous appellerons *shellcode*) doit être présent dans la mémoire du processus victime, mais où ? Si nous tenons à être le plus discret, aucune modification de variable ou effet de bord ne doit avoir lieu. Étudions maintenant les endroits où écrire le code :

La pile est un milieu pratique puisque de la même manière qu'un appel de fonction (au sens assembleur), un espace peut être réservé en créant une *stack frame* spécifique. Une fois le *shellcode* exécuté, on décrémente le pointeur de pile correctement et le code disparaît.

Cette technique est intéressante mais ne fonctionnera pas sur les systèmes rendant la pile non-éxecutable (comme *PaX* ou *execschield*). La figure 1 schématise la mémoire du processus en précisant où pointe `eip` et le haut de la pile (`esp`). À ce moment, nous nous attachons à ce processus, décrémentons nous même la pile et y injectons notre code.

Entre ces deux clichés, le fils n'a pas eu la main, seul le père a fait ces manipulations. Avant de rendre la main au fils, il faut détourner le flux, c'est-à-dire pointer le registre `eip` vers le haut de la pile, pour exécuter le code injecté.

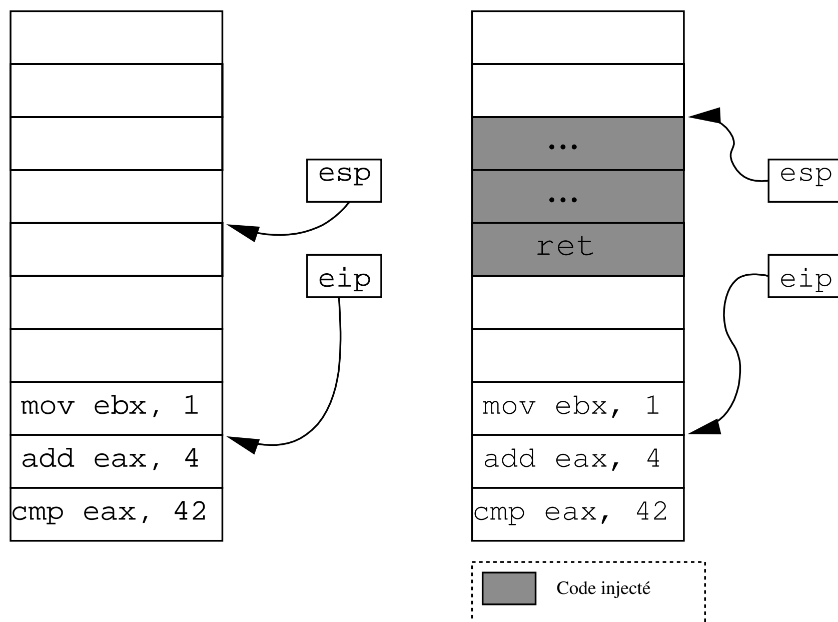

Fig. 1. Avant l'injection **Fig. 2.** Après injection

Celui-ci va être écrit comme une fonction. Ainsi, à l'entrée du shellcode, `eip` aura été préalablement poussé. Lorsque le *shellcode* sera terminé, l'instruction `ret` ramènera le flot d'exécution à l'endroit originale.

Notre code doit donc se présenter comme une classique fonction assembleur :

```
push ebp
mov ebp, esp
; pusha?
; ...
; payload
; ...
; popa?
mov esp, ebp
pop ebp
ret
```

L'appel vers la pile va être réalisé en simulant un `call esp`. On ne peut que simuler cet appel de fonction en modifiant nous même `eip` et `esp` puisqu'où est-ce qu'on pourrait injecter les deux octets de l'instruction `call` ? Écrire à l'adresse pointée par `eip` détruirait le contexte du processus qui ne pourrait plus continuer normalement. De même, si on injecte à l'adresse `eip - 2`, il y a de fort risque d'écraser une instruction faisant partie d'une boucle.

L'instruction `call esp` pousse sur la pile la valeur d'`eip` et écrase ce registre par la valeur ici contenue dans `esp`. En utilisant deux instructions `ptrace()` (l'une simulant l'empilement d'`eip` et l'autre pour modifier `eip` et `esp`), nous pouvons arriver à ce résultat comme le montre la figure 3.

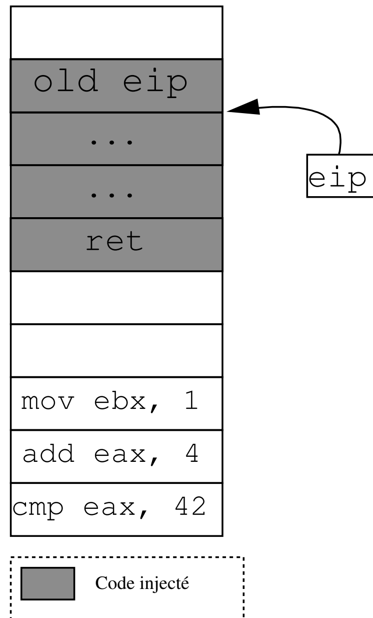

Fig. 3. Injection terminée

Le format d'exécutable ELF[2] utilise des sections qui, lorsqu'elles sont mappées en mémoire sont alignées avec la taille d'une page mémoire, entraînant alors du padding (bourrage). Milieu plus intéressant que la pile car s'il y a du padding pour la section `TEXT` (là où sont stockées les instructions), cet espace mémoire est bien sûr déclaré avec les droits d'exécution.

Cette technique peut également permettre de contourner des applications qui vérifieraient l'intégrité de leur code sans pour autant vérifier les données contenues dans le padding.

La méthode brutale consiste à écrire notre code à l'adresse pointée par `eip`, exécuter notre code en continuant le processus et à la fin de l'exécution de notre code, restaurer tout ce qui doit l'être.

Cette solution a le mérite d'être la plus furtive puisque notre code n'apparaîtra jamais pendant l'exécution des instructions originales. Toutefois, la phase de restauration est la plus problématique puisqu'il faut détecter la fin de notre code ! L'injecteur peut soit faire du pas à pas et repérer la dernière instruction, soit faire continuer (`PTRACE_CONT`) le code et attendre que le code réveille l'injecteur.

L'injecteur peut être réveillé par la livraison d'un signal SIGTRAP au processus lui-même, donnant ainsi la main au père qui pourra recouvrir ses traces.

```
/* papa, réveille toi et nettoie moi ! */
kill(SIGTRAP, getpid());
```

Interruption d'appel système Il peut être nécessaire d'aider le fils à reprendre la main, en particulier si le code a été injecté pendant un appel système (par exemple `read`, `write`, `wait`, etc.).

Sur architecture `x86`, à l'entrée d'un appel système, le noyau pousse `eax` sur la pile[1] (`eax` contenant le numéro de *syscall* demandé). Ensuite, l'appel système est exécuté et à sa fin, met sa valeur de retour dans ce même registre `eax`.

Les appels systèmes considérés lents sont interruptibles, à la réception d'un signal, le noyau va arrêter l'appel système pour exécuter le handler. Puis, après traitement du signal, deux cas possibles :

- L'appel système est automatiquement redémarré,
- Le redémarrage doit être manuel (le code de retour de l'appel système échoue avec `errno` égal à `EINT` ;

Pour le redémarrage automatique, le noyau rétablit `eax` en utilisant sa copie locale contenue dans la pile, notre fameux `orig_eax`, puis décrémente `eip` de deux octets, soit la taille de l'instruction `int 0x80`.

Pour détecter l'interruption d'un appel système, nous allons mimer le noyau Linux et regarder nos registres : `eax` doit valoir -1 et `orig_eax` doit contenir un numéro d'appel système correct.

Une deuxième méthode, qui a le mérite d'être portable sur toutes les architectures, est d'utiliser l'option `PTRACE_O_SYSGOOD` qui va ajouter 0x80 à `si_code` accessible via une requête `PTRACE_GETSIGINFO`.

```
/* au début du debugging, on donne l'option */
ptrace(PTRACE_SETOPTIONS, pid
      , NULL, PTRACE_O_TRACESYSGOOD);
...

/* puis lorsqu'on a interrompu le fils, on
 * vérifie qu'on n'était pas dans un syscall
 */
siginfo_t sig;
ptrace(PTRACE_GETSIGINFO, pid, NULL, &sig);

if (sig.si_code & 0x80)
   printf("was in a syscall\n");
```

L'interruption d'un appel système pose problème lors de l'injection de code, car le noyau va essayer de redémarrer le *syscall* en soustrayant deux octets à `eip` lors de la reprise du processus, votre shellcode devra donc commencer par

[1] Cette valeur sur la pile représente le registre virtuel `orig_eax`

deux octets d'instructions inertes comme `NOP`. `eip` sera réglé de façon à exécuter `shellcode+2` (après les `NOPs`), si le processus était dans un appel système, le noyau fera la soustraction et commencera par exécuter les `NOP` puis votre code.

Conclusion Nous avons terminé de présenter `ptrace()` (réponse à la question « comment s'en servir ? »), la section suivante va s'intéresser aux applications concrètes.

2 Application pratique

2.1 Actions légitimes

Nettoyer sa table de processus Un processus terminé doit être « confirmé » par son père, en d'autre terme, il faut que le parent fasse un `wait()` pour prendre connaissance de la mort de son fils.

Si ce n'est pas fait, le fils est placé en état *zombie* et reste dans la table des processus. Pour commencer facilement avec `ptrace()` nous allons injecter dans le père un appel à `wait()` en utilisant la méthode brutale : on va écrire les octets pointés par `eip`. Le code injecté est résumable en deux lignes :

```
waitpid(-1, 0, WNOHANG);
/* pour réveiller l'injecteur */
kill(SIGTRAP, getpid());
```

L'injecteur se résume aux lignes suivantes :

```
int status;
struct user luser;
unsigned char olddata[LENSHELLCODE];

ptrace(PTRACE_ATTACH, pid, NULL, 0);
wait(&status);
ptrace(PTRACE_GETREGS, pid, NULL, &luser);

/* récupére les données pointées par %eip et les
 * sauvegarde avant de les écraser par notre shellcode
 */
injectercode(luser.regs.eip, olddata);

ptrace(PTRACE_CONT, pid, NULL, 0);
wait(&status);

restauration(luser.regs.eip, olddata, LENSHELLCODE)
ptrace(PTRACE_DETACH, pid, NULL, 0);
```

Plus fréquent, vous avez lancé une application qui va tourner longtemps (un transfert de fichier, un long calcul, etc.) alors que vous étiez sur un point de montage que vous aimeriez bien démonter.

Si votre application n'utilise aucun fichier de ce point de montage, il peut-être intéressant de modifier son répertoire courant en injectant un appel à `chdir()`. Ces deux petits exemples présentaient les cas simples de simple injection de code. Intéressant nous maintenant à l'utilisation de binaires considérés hostiles.

Fakebust *fakebust* est un logiciel développé par Michal Zalewski afin de pouvoir lancer des binaires inconnus (hostiles) sans avoir à utiliser une machine virtuelle ou à avoir à effectuer une longue analyse statique.

fakebust est basé uniquement sur `ptrace()` en suivant un processus à l'aide de `PTRACE_SYSCALL`. Cela signifie que le processus tracé est uniquement interrompu à l'entrée et à la sortie d'un appel système, ainsi qu'à la réception d'un signal.

À l'entrée d'un des *syscalls* considérés comme dangereux (par exemple `open`, `socket`, `unlink`, etc), *fakebust* va autoriser les appels systèmes au cas par cas en interrogeant l'utilisateur s'il doit exécuter l'appel système, le refuser ou le simuler.

Cet outil est pratique lors de l'analyse dynamique de binaire (tel qu'un soi-disant 0day sur OpenSSH) si vous n'avez pas à faire de *reverse engineering*, dans ce cas, vous allez devoir trouvez d'autres outils. Lorsque des protections anti-debugging sont mises en œuvre, le « simple » GNU Debugger peut se révéler inadapté.

Par exemple, la pose de point d'arrêt logiciel (le fait de générer une interruption de code 3 par l'opcode `0xCC`) va modifier l'intégrité d'un processus et les possibles protections le détecteront. Cette problématique a été soulevé lors du reversing de Skype par Fabrice Desclaux et Philippe Biondi[1]. En effet, des centaines de contrôles (aux alentours de trois cents) d'intégrité sont réalisés en permanence afin de transformer la vie du reverser en un véritable cauchemar.

Reverser Skype Leur premier problème était de contourner les contrôles d'intégrité qui interviennent partout, la pose de point d'arrêt logiciel entraîne la modification de l'espace d'adressage et est donc détecté par ces *checksummers* qui, en réponse, transforment les structures de code en un pur aléa.

L'objectif était, si possible, de ne pas modifier/supprimer du code ; Afin d'arriver à ce résultat, une fois la localisation des checksums faite, il ne restait qu'à placer des points d'arrêts matériel (alors indétectable par Skype) à l'entrée et à la sortie des fonctions. Or, sur x86, on ne dispose que de quatre points d'arrêt matériel, quantité insuffisante pour surveiller les centaines de *checksummers*.

Une solution est donc d'utiliser des points d'arrêt logiciel (qui sont illimités), mais qui ont l'inconvénient d'être détectés par les vérificateurs d'intégrité. On voudrait ainsi utiliser les avantages de chaque méthode, c'est ainsi que Philippe Biondi a eu l'idée de lancer deux processus de Skype en parallèle, l'un truffé de *software breakpoints* sur chaque checksummer et l'autre de point d'arrêt matériel.

Lorsque le premier processus est interrompu à l'entrée d'une fonction, un point d'arrêt matériel est placé à au même endroit dans le code du processus jumeau, ainsi qu'à la fin de la fonction, puis on démarre l'exécution du processus jumeau. Lorsque ce dernier a calculé une somme d'intégrité correcte, on récupère ce résultat à l'aide de notre deuxième point d'arrêt matériel et on le transmet au premier processus qui passera alors avec succès le test d'intégrité.

`ptrace()` a permit de tout faire très simplement :

- Pose de points d'arrêt logiciel (`PTRACE_POKETEXT` avec l'opcode `CC` injecté),
- Exécution du processus (`PTRACE_CONT`),
- À l'arrivée dans une fonction de contrôle d'intégrité, on pose un *breakpoint* matériel (`PTRACE_\-SETREGS`) au même endroit dans l'autre processus et on le laisse s'exécuter (`PTRACE_CONT`),
- Dès que le résultat est calculé, on le récupère (`PTRACE_PEEKDATA`) et l'injectons dans le premier processus (`PTRACE_POKEDATA`);

Skype a été écrit avec la volonté manifeste de ne pas permettre un *reverse engineering* facile. Fabrice Desclaux s'est ainsi cassé les dents sur une fonction d'une taille monstrueuse et obscurcie au maximum, celle-ci a pour rôle de chiffrer les paquets sortants et entrants (un dérivé de RC4). Si vous souhaitez développer un téléphone compatible avec Skype, vous êtes alors obligé d'implémenter exactement cette fonction.

La technique de l'Oracle consiste à jeter dans un puits une question et de recevoir la réponse en retour. C'est exactement ce qui a été fait dans Skypy (un client Skype un petit peu particulier) avec le binaire Skype comme oracle.

À l'aide de `ptrace`, on s'attache à une instance de Skype et en modifiant `eip`, on ne s'en sert que pour lui donner notre paquet à chiffrer/déchiffrer et le résultat ressort immédiatement.

Cette technique aurait même pu être utilisée pour calculer les sommes d'intégrité précédentes.

Virtualisation de système *UserModeLinux* est une implémentation du noyau Linux tournant en espace utilisateur. On arrive ainsi à une virtualisation de système d'exploitation en conservant des performances correctes.

Toute l'implémentation repose sur l'utilisation de `ptrace()`, en effet, l'objectif initial était de faire tourner le maximum d'instructions nativement et passer par une couche d'abstraction lorsque des instructions privilégiées sont exécutées telles que les appels systèmes.

Au démarrage, la machine virtuelle créée un *thread* qui va servir à tracer tous les processus de la machine virtuelle. Après attachement, les processus sont continués avec `PTRACE_SYSCALL` qui va permettre de stopper les processus de la même façon que *fakebust*, l'exécution réelle de l'appel système va être réaliser en transférant le contexte du processus vers celui du noyau virtuel qui exécutera lui même le *syscall*[2].

[2] C'est l'ancien moyen de réaliser cette opération, voir la documentation d'UML pour plus d'information

Réaction à un incident Grâce à `ptrace()`, nous devenons omniscient, nous sommes en mesure d'inspecter chaque processus dans ses moindres détails. Ce fait est particulièrement intéressant lorsque vous arrivez sur une machine compromise avec des programmes user-space qui tournent.

Si l'hypothèse de départ est que le noyau n'a pas été modifié ou qu'il ne s'intéresse pas à `ptrace()`, nous allons ainsi pouvoir inspecter les processus suspects comme les *backdoors*, redirecteur de ports, clients IRC, etc.

Bien que les pirates utilisent désormais des moyens de communication sécurisés (chiffrement SSL, *OpenSSH*), il est toujours possible de voir ce qui transit en surveillant les appels systèmes `read()`, `write()`, `send()`, `recv()`, etc.

Nous pouvons également détourner et retourner des *rootkits* noyaux comme le populaire Suckit. Pour ce dernier, l'accès au seul pirate est limité par un mot de passe qui autorise ou non l'utilisation du module noyau.

Comme cela a été montré par Frédéric Raynal à *EusecWest*[3], le binaire de contrôle est chiffré en RC4 avec une graine de 64 octets placée en fin de fichier avec la configuration (contenant le mot de passe hashé).

À l'exécution, le binaire est complètement déchiffré en mémoire puis tente d'authentifier l'utilisateur avec la demande d'un mot de passe. Celui-ci est hashé et comparé à celui stocké en mémoire.

Si vous faîtes l'autopsie à chaud d'une machine piratée par *suckit*, vous aimeriez prendre le contrôle du *rootkit* afin d'accéder à tous les fichiers et processus cachés par exemple.

La première étape va donc être de récupérer le binaire déchiffré, l'exécuter, puis trouver le *hash* du mot de passe. Ensuite, modifier le binaire afin de lui mettre le hash du mot de passe pour que la comparaison soit vraie et que vous soyez identifié.

2.2 Outil de sécurité

Protection anti-*reverse engineering* Sous GNU/Linux, les protections anti-reversing se sont longtemps limitées à modifier les sections ELF et empêcher l'utilisation d'un debugger ou plus généralement de `ptrace()`.

La technique la plus populaire et facile à mettre en œuvre est d'utiliser une limitation interne du noyau : un processus ne peut-être tracé que par un unique debugger.

Ainsi, les applications tentent, soit aléatoirement, soit à l'initialisation du programme, de s'attacher à elle-même (`PTRACE_TRACEME`). En cas d'échec, `errno` contiendra `EPERM`, une hypothèse très probable à ce code de retour est que le processus est déjà tracé, déclenchant alors un cataclysme pour ennuyer l'analyste.

La réponse à cet auto-traçage a conduit les *reverse engineers* à modifier le code du binaire afin de remplacer les appel à `ptrace()` par des `NOP`. Mais cela implique d'avoir également à déjouer les vérifications d'intégrité si nécessaire.

L'émulation permet de ne pas modifier le binaire : elle exécute réellement `ptrace()` (qui va certes échoué), mais avant de rendre la main au processus, le

registre représentant la valeur de retour (`eax` sur `x86`) va être modifié avec un code de retour valide.

Comme toutes les protections, le jeu du chat et de la souris a alors commencé entre les *reversers* et les programmeurs avec la création de fils se traçant mutuellement. Afin d'éviter l'émulation des appels systèmes, un dialogue est réalisé en permanence entre les processus pour s'assurer qu'ils sont encore vivant.

Évasion d'environnement `chrooté` Une bonne pratique des administrateurs est de `chrooter` chaque service, c'est à dire que pour un processus donné, sa racine du système de fichier va être déplacé dans un répertoire beaucoup plus restreint. Sur un noyau Linux non protégé, l'évasion d'un environnement `chrooté` est triviale puisque l'accès au reste du noyau n'est pas limitée. Plusieurs vecteurs permettent ainsi de s'échapper ou nuire au système : mémoire partagée, signaux, double chrootage, etc.

À l'intérieur d'un processus `chrooté`, la vision des processus extérieurs n'a pas été impactée : il est toujours possible de leur envoyer un signal... et donc d'utiliser `ptrace()` !

Les conséquences sont alors désastreuses si le service prisonnier tourne sous les privilèges `root` puisqu'il va être possible de tracer tous les processus du système.

Néanmoins, un environnement limité ne dispose pas (du moins c'est à espérer) de `/proc`, récupérer le `PID` d'un processus va alors nécessiter d'essayer tout l'espace de `PID` (1 à 65 535). Bien que le processus `init` existe toujours avec le `PID 1`, on ne peut pas l'utiliser puisque c'est le seul processus qui ne peut pas être tracé (limitation faite par le noyau).

Un article publié dans *Phrack* [4] présente des exemples de *shellcode* permettant de s'évader d'un `chrootage`.

Patchage à la volée La mise à jour du code d'un processus en exécution est un sujet appliqué principalement au noyau puisqu'aucun redémarrage n'est demandé, cette contrainte est beaucoup moins forte pour le code en espace utilisateur que l'on préfère, avec raison, patcher au niveau du binaire et relancer le service derrière.

Néanmoins, un attaquant recherche à causer le moins d'effet de bord possible et la relance d'un service peut-être rapidement détecté. Si le noyau est protégé, par des *secure levels*, ou si la mémoire du noyau (`/dev/kmem`) est inaccessible en écriture, il peut être intéressant de pousser notre *rootkit* à l'intérieur d'autres processus stratégiques comme `sshd`, `bind`, `X`, `syslogd`, etc.

Ce *patchage* peut être réalisé pour installer des *backdoors* mais également afin de servir à rendre aveugle certains logiciels comme les *Host IDS*.

Propagation de codes malicieux L'injection de code dans les environnements *Microsoft Windows* est un sujet connu et beaucoup de recherches (des deux couleurs du chapeau) ont ainsi été faites.

De nombreuses techniques pour contourner les logiciels de sécurité locaux ont vu le jour : les anti-virus se sont tous vu attaqués afin d'être désactivés par l'infection en cours, idem pour les *host intrusion detection system*.

De même, les limites des *firewalls* personnels ont été prouvées puisque les connexions sont autorisées par application. L'injection d'un *shellcode* dans cette application autorisée ouvre alors les portes du réseau.

Avec l'arrivée sous Linux de firewalls similaires (ou même de l'option `--cmd-owner` de Netfilter) à ceux qu'on peut trouver sous Windows, les mêmes problèmes ressurgissent comme nous le montre la prochaine et dernière section.

2.3 Cas pratique : contournement d'un firewall applicatif

Implémentation générale Dans cette démonstration, il a été utilisé le système *NuFW* basé sur Linux/Netfilter qui est un *firewall* réseau utilisant une authentification des paquets par utilisateur, par `GID`, par application, par système d'exploitation, etc.

Nous avons configuré NuFW avec la configuration suivante :

```
# iptables -P FORWARD DROP
# iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
# iptables -A FORWARD -i eth0 -o eth1 -p tcp --dport 80 -j QUEUE
# cat /etc/nufw/acls.nufw
[webapp]
decision=1
gid=1000
proto=6
SrcIP=0.0.0.0/0
SrcPort=1024-65536
DstIP=0.0.0.0/0
DstPort=80
app=/usr/bin/firefox
OS=Linux
```

Ici, seul Mozilla Firefox a le droit de sortir en TCP sur le port 80. C'est cette application qui servira de tremplin à notre processus ayant besoin de se connecter à un serveur sur le port 80.

Étant donné que Firefox tourne sous notre identité et que NuFW ne prend pas en compte les possibilités offertes par `ptrace()`, il est possible de s'attacher au navigateur et créer nos propres connexions à partir de lui.

Une solution est de concevoir toutes nos applications (malicieuses) afin d'utiliser ce mécanisme : faire toutes les opérations tout en restant attaché à *firefox* avec `ptrace()`. Cela fonctionne, mais reste peu pratique.

La meilleure solution serait de ne pas avoir à modifier nos binaires, tout en bénéficiant de `ptrace()`. Comment ?

Linux, ainsi que beaucoup d'UNIX, sont capables de transmettre des descripteurs de fichier entre différents processus totalement indépendant. Cette fonctionnalité est parfaite dans notre cas, on laisse à notre injecteur le soin de faire

un `connect()` depuis *firefox*, puis une fois le descripteur obtenu, on le passe à notre application.

Pour que cela soit transparent, nous pouvons utiliser le mécanisme de « surcharge » offert par les bibliothèques dynamiques sous Linux. La fonction `connect()` va être surchargé pour s'attacher au navigateur, y injecter le code de connexion puis transmettre le descripteur de fichier. Le binaire original n'aura subit alors aucune modification (en contrepartie, il y aura juste la création inutile d'une socket) et obtiendra un descripteur parfaitement valide.

Transfert de descripteur de fichier entre deux processus indépendants Aucune documentation n'existe pour décrire la possibilité de tranférer deux descripteurs de fichier, seules quelques lignes dans la page de manuel de `cmsg` donnent un rapide exemple.

En résumé, il faut créer une socket UNIX entre les deux processus et envoyer des messages via `sendmsg()` et `recvmsg()`. En pratique, cela demande beaucoup plus de doigté puisqu'il va falloir débusquer les particularités (bugs ?) du noyau Linux sur cette fonctionnalité très bien cachée. Les annexes 3 et 3 contiennent respectivement le code C réalisant l'envoi et la réception de descripteur de fichier.

Code à injecter Notre *shellcode* réalisant tout le travail de création de socket est assez volumineux puisque sans optimisation, il est long de 360 octets. Il est d'ailleurs à prendre garde de vérifier que la taille de votre code soit un multiple d'un mot machine (32 bits sur `x86`) puisque `ptrace()` a une granularité très limitée : il ne peut lire et écrire atomiquement qu'un mot à la fois. Pensez donc à ajouter des `NOP` pour combler.

Le code source assembleur (assemblable par *nasm*) sera disponible sur le site du SSTIC prochainement. Son code est assez lourd puisque tout le travail consiste à remplir des structures C imbriquées.

3 Conclusion

Nous avons vu que les possibilités de `ptrace()` ne se limitaient pas seulement au simple debugging mais qu'il avait un champ d'application beaucoup plus large : évasion d'environnements *chrootés*, contournement d'HIPS ou firewall, aide au *reverse engineering*, etc.

Toutes les techniques connues sous Microsoft Windows sont reproductibles partiellement sous Linux, comme la récupération des mots de passe en mémoire par exemple. `ptrace()` souffre néanmoins de beaucoup de limitations, un souffle nouveau pourrait voir le jour avec l'introduction d'une nouvelle interface, cette fois tournant en kernel-space et essayant de se rapprocher aux fonctionnalités de DTrace sous Solaris : Systemtrap.

Références

1. Desclaux F. et Biondi P. (2006), *Silver Needle in the Skype*, http://blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf
2. TISCommittee (1995), *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, http://refspecs.freestandards.org/elf/elf.pdf
3. Raynal, F. (2006) Malicious crypto, http://eusecwest.com/slides06/esw06-raynal.pdf.
4. Anonymous (2002), *Building ptrace injecting shellcodes*, Phrack 59, http://www.phrack.org/phrack/59/p59-0x0c.txt
5. Cesare S. (1999) Linux anti-debugging techniques (fooling the debugger), http://vx.netlux.org/lib/vsc04.html

Routine d'entrée dans un appel système

```
syscall_trace_entry:
    movl $-ENOSYS,EAX(%esp)
    movl %esp, %eax
    xorl %edx,%edx
    call do_syscall_trace
    cmpl $0, %eax
    jne resume_userspace

    movl ORIG_EAX(%esp), %eax
    cmpl $(nr_syscalls), %eax
    jnae syscall_call
    jmp syscall_exit
```

Adresses mémoire pour accéder aux registres

Ces décalages correspondent à la valeur de `addr` dans la requête `ptrace()` suivante :

```
  ptrace(PTRACE_POKEUSR, pid, addr, data);
```

Envoi de descripteur de fichier

```
void send_fd(int sockfd, int fd)
{
  struct cmsghdr *ch;
  struct msghdr msg;
  struct iovec iov;
  char ancillary[CMSG_SPACE(sizeof(fd))];
  char tmp = '\0';
```

Décalage	Registre mémoire
0x0	`%ebx`
0x4	`%ecx`
0x8	`%edx`
0xC	`%esi`
0x10	`%edi`
0x14	`%ebp`
0x18	`%eax`
0x1C	`%dans`
0x20	`%es`
0x24	`orig_eax`
0x28	`%eip`
0x2C	`%cs`
0x30	`%eflags`
0x34	`%oldesp`
0x38	`%oldss`

Tab. 1. Adresses mémoire des registres

```
  memset(&msg, 0, sizeof(msg));
  iov.iov_base = &tmp;
  iov.iov_len  = 1;
  msg.msg_iov = &iov;
  msg.msg_iovlen = 1;
  msg.msg_control = &ancillary;
  msg.msg_controllen = sizeof(ancillary);

  ch = CMSG_FIRSTHDR(&msg);
  if (! ch) {
    printf("cmsg_firsthdr failed");
  }
  ch->cmsg_level = SOL_SOCKET;
  ch->cmsg_len   = CMSG_LEN(sizeof(fd));
  ch->cmsg_type  = SCM_RIGHTS;
  *(int *) CMSG_DATA(ch) = fd;

  if ( sendmsg(sockfd, &msg, 0) < 0) {
    perror("sendmsg()");
  }
}
```

Réception d'un descripteur de fichier

```
 int receive_fd(int sockfd)
{
```

```
  struct cmsghdr *ch;
  struct msghdr  msg;
  struct iovec iov;
  char ancillary[CMSG_SPACE(sizeof(int))];
  char tmp;
  int fd = -1;

  memset(&msg, 0, sizeof(msg));

  iov.iov_base = &tmp;
  iov.iov_len  =  1;
  msg.msg_iov = &iov;
  msg.msg_iovlen = 1;
  msg.msg_control = &ancillary;
  msg.msg_controllen = CMSG_LEN(sizeof(ancillary));

  fd = recvmsg(sockfd, &msg, 0);

  if (fd > 0) {
    /* recvmsg() succedeed */
    ch = CMSG_FIRSTHDR(&msg);

    if (!ch && ch->cmsg_type != SCM_RIGHTS) {
      fprintf(stderr, "cmsg_type is not good... (was %d)\n"
              , ch->cmsg_type);
    } else {
      memcpy(&fd, CMSG_DATA(ch), sizeof(fd));
    }
  } else {
    perror("recvmsg()");
  }
  return fd;
}
```