

Playing with ptrace() for fun and profit

Injection de code sous GNU/Linux

Nicolas Bareil

`nicolas.bareil@eads.net`

EADS Corporate Research Center - DCR/STI/C
SSI Lab

SSTIC 2006



Il était une fois. . .

Sous UNIX, ptrace() est le seul moyen de debuggage.

- User-space,
- Interface rigide et minimaliste,
- Privilège root non nécessaire,
- Élégant ;

D'après la page de manuel de SunOS

ptrace() est unique et mystérieux.

Plan

- 1 `ptrace()`
- 2 Injection de code
- 3 Applications pratiques

man ptrace

Prototype

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

Trois modes de traçage :

- Mode pas à pas,
- Par appel système,
- Traçage passif ;

Actions classiques, documentées

Requête	Signification
PTRACE_TRACEME PTRACE_ATTACH PTRACE_DETACH	Attachement à un processus
PTRACE_PEEKTEXT PTRACE_PEEKDATA PTRACE_PEEKUSR	Lecture mémoire
PTRACE_POKETEXT PTRACE_POKEUSR PTRACE_POKEUSR	Écriture mémoire
PTRACE_GETREGS PTRACE_SETREGS	Lecture des registres Écriture des registres

Manipulation des signaux

Signaux & ptrace()

Un processus tracé est stoppé à la réception de chaque signal.
Pour le traçeur, l'arrêt semble être dû à un SIGTRAP.

L'option `PTRACE_GETSIGINFO` permet d'en connaître plus sur la raison de la notification.

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

Manipulation des signaux

Signaux & ptrace()

Un processus tracé est stoppé à la réception de chaque signal.
Pour le traçeur, l'arrêt semble être dû à un SIGTRAP.

L'option `PTRACE_GETSIGINFO` permet d'en connaître plus sur la raison de la notification.

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

Manipulation des signaux

Signaux & ptrace()

Un processus tracé est stoppé à la réception de chaque signal.
Pour le traçeur, l'arrêt semble être dû à un SIGTRAP.

L'option `PTRACE_GETSIGINFO` permet d'en connaître plus sur la raison de la notification.

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


Protection anti-pttrace()

```
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();  
    }  
}
```

Anti-pttrace()

Protection basée sur le fait qu'un debugger classique ne peut pas différencier les signaux envoyés par le noyau ou par l'utilisateur.

Anti-anti-pttrace()

Deux moyens de déterminer l'origine du SIGTRAP :

Manuel, fastidieux, non-portable

- Mode pas-à-pas ?
⇒ bit single-step du processeur,
- Arrêter par point d'arrêt matériel ?
⇒ registres de debuggage ;
- Dans un appel système ?

Élégant, portable, classe quoi

Utiliser `ptrace(PTRACE_GETSIGINFO, pid, NULL, &sig)`.
⇒ `sig.si_code == SI_USER` ?

Options de traçage : suivi des enfants

Problème de `fork()`

Solution basique :

- 1 À l'appel à `fork()`, on surveille le code de retour,
⇒ on récupère ainsi le PID du fils
- 2 On s'attache au nouveau processus,
- 3 On se met à le tracer ;

Pwned ! Race condition detected !

Le *scheduler* peut laisser le fils exécuter des instructions avant de rendre la main au traceur.

Options de traçage : suivi des enfants

Problème de `fork()`

Solution basique :

- ➊ À l'appel à `fork()`, on surveille le code de retour,
⇒ on récupère ainsi le PID du fils
- ➋ On s'attache au nouveau processus,
- ➌ On se met à le tracer ;

Pwned ! Race condition detected !

Le *scheduler* peut laisser le fils exécuter des instructions avant de rendre la main au traceur.

Follow me !

Les options `PTRACE_O_TRACEFORK` & Co servent à régler ce problème :

- Attachement automatique au fils,
- Le noyau met le fils en état `STOPPED` avant même qu'il soit déclaré `RUNNABLE`.

```
ptrace(PTRACE_SETOPTIONS, pid  
      , NULL, PTRACE_O_TRACESYSGOOD);
```

Accès à l'espace d'adressage

Lecture d'un mot mémoire

```
errno = 0;  
ret = ptrace(PTRACE_PEEKTEXT, pid, targetaddr, NULL);  
  
if (errno && ret == -1) {  
    perror("ptrace_peektext()");  
    return 1;  
}
```

Lecture de plusieurs octets

```
char buf[BUFMAX];  
int fd = open("/proc/pid/mem", O_RDONLY);  
pread(fd, buf, BUFMAX, offset);
```

Accès à l'espace d'adressage

Lecture d'un mot mémoire

```
errno = 0;  
ret = ptrace(PTRACE_PEEKTEXT, pid, targetaddr, NULL);  
  
if (errno && ret == -1) {  
    perror("ptrace_peektext()");  
    return 1;  
}
```

Lecture de plusieurs octets

```
char buf[BUFMAX];  
int fd = open("/proc/pid/mem", O_RDONLY);  
pread(fd, buf, BUFMAX, offset);
```

Plan

- 1 ptrace()
- 2 Injection de code
- 3 Applications pratiques

Où injecter les instructions ?

Objectifs

- Discrétion,
- Stabilité,
- Portabilité ;

Les candidats sont :

- La pile,
- Padding des sections ELF,
- N'importe où ;

Injection n'importe où

On écrit directement sur les instructions pointées par `eip`.

- 1 Sauvegarde des octets d'`eip`,
- 2 Écrasement par nos instructions,
- 3 Redémarrage du processus (`PTRACE_CONT`),
- 4 Restauration des anciennes instructions ;

Papa, réveille toi !

```
kill (SIGTRAP, getpid ());
```

Injection n'importe où

On écrit directement sur les instructions pointées par `eip`.

- 1 Sauvegarde des octets d'`eip`,
- 2 Écrasement par nos instructions,
- 3 Redémarrage du processus (`PTRACE_CONT`),
- 4 Restauration des anciennes instructions ;

Papa, réveille toi !

```
kill (SIGTRAP, getpid ());
```

Injection dans la pile

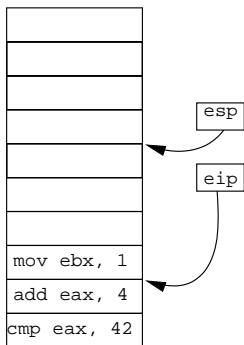


FIG.: Avant l'injection

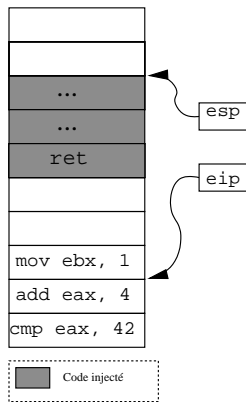
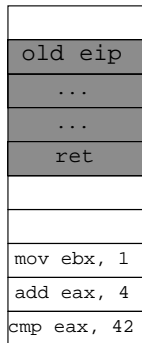


FIG.: Après injection

Injection dans la pile




eip

- eip est sauvegardé sur la pile,
- eip pointe sur *esp,
- Le shellcode se termine par un return
⇒ Retour aux instructions normales ;

Précautions

⇒ La pile doit être exécutable !

 Code injecté

Interruption d'un appel système

L'injection de code dans un processus interrompu dans un appel système peut poser des problèmes en fonction de leurs natures :

- Non-interruptibles,
- Interruptibles, pour les appels système lent
 - Redémarrable manuellement (code de retour égal à EINT),
 - Redémarrable automatiquement par le noyau ;

Sh*t happens...

Le redémarrage automatique est le cas le plus problématique car l'injecteur n'a aucun contrôle sur le noyau.

⇒ Décrémenter d'eip de 2 octets.

Interruption d'un appel système

L'injection de code dans un processus interrompu dans un appel système peut poser des problèmes en fonction de leurs natures :

- Non-interruptibles,
- Interruptibles, pour les appels système lent
 - Redémarrable manuellement (code de retour égal à EINT),
 - Redémarrable automatiquement par le noyau ;

Sh*t happens...

Le redémarrage automatique est le cas le plus problématique car l'injecteur n'a aucun contrôle sur le noyau.

⇒ Décrémenter d'eip de 2 octets.

Shellcode : Le bon, la brute et le truand

Le truand

Toujours précéder votre shellcode de deux octets inertes (NOP) et faire pointer eip sur $\&(\text{shellcode}+2)$.

La brute

Faire les mêmes vérifications que le noyau avant d'injecter.
⇒ Vérifier `orig_eax` et `eax`.

Le bon

Utiliser l'option adéquate de `ptrace()` : `PTRACE_O_SYSGOOD`.

Oasis is good !

```
man PTRACE_O_TRACESYSGOOD
```

L'option `PTRACE_O_SYSGOOD` modifie `si_code` de la structure `siginfo_t` pour indiquer l'interruption d'un appel système.

```
ptrace(PTRACE_SETOPTIONS, pid
      , NULL, PTRACE_O_TRACESYSGOOD);
...
siginfo_t sig;
ptrace(PTRACE_GETSIGINFO, pid, NULL, &sig);

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

Plan

- 1 ptrace()
- 2 Injection de code
- 3 Applications pratiques

Technique de l'oracle & Skype

Oracle

Jetez une question dans un puits et la réponse est renvoyée.

Téléphone compatible Skype ?

Le chiffrement des paquets est réalisé par une fonction trop compliquée à inverser ?

Plutôt que de réécrire la fonction, utilisez-la ! À l'aide de `ptrace()`, manipulez `eip` afin de n'exécuter que la fonction de chiffrement.

⇒ Au retour de la routine, vous avez la réponse !

Voir présentation de P. Biondi & F. Desclaux à BlackHatEurope06.

Protection anti-reverse engineering

Première protection anti-debugging ?

Un processus ne peut-être tracé que par **un seul** debugger à la fois.
→ Auto-traçage pour empêcher tout attachement ultérieur

Réponse des analystes

Émulation de l'appel à `ptrace()` pour qu'il n'échoue jamais.

Réponse à la réponse des analystes

Lancement de deux processus qui se traçent mutuellement avec un dialogue permanent pour s'assurer de la survie de l'autre.

Protection anti-reverse engineering

Première protection anti-debugging ?

Un processus ne peut-être tracé que par **un seul** debugger à la fois.
→ Auto-traçage pour empêcher tout attachement ultérieur

Réponse des analystes

Émulation de l'appel à `ptrace()` pour qu'il n'échoue jamais.

Réponse à la réponse des analystes

Lancement de deux processus qui se traçent mutuellement avec un dialogue permanent pour s'assurer de la survie de l'autre.

Protection anti-reverse engineering

Première protection anti-debugging ?

Un processus ne peut-être tracé que par **un seul** debugger à la fois.
→ Auto-traçage pour empêcher tout attachement ultérieur

Réponse des analystes

Émulation de l'appel à `ptrace()` pour qu'il n'échoue jamais.

Réponse à la réponse des analystes

Lancement de deux processus qui se traçent mutuellement avec un dialogue permanent pour s'assurer de la survie de l'autre.

Évasion d'environnement chrooté

ch[ouc]root ?

Simple restriction de la racine du système de fichier d'un processus.
Contact extérieur possible : mémoire partagée, signaux. . .

Pwned !

Si on peut envoyer des signaux à des processus, on peut les tracer !
⇒ Injection d'un shellcode permettant de s'évader de la cage.

Si /proc n'est pas disponible, le bruteforçage de tous les PID est nécessaire.

Évasion d'environnement chrooté

ch[ouc]root ?

Simple restriction de la racine du système de fichier d'un processus.
Contact extérieur possible : mémoire partagée, signaux. . .

Pwned !

Si on peut envoyer des signaux à des processus, on peut les tracer !
⇒ Injection d'un shellcode permettant de s'évader de la cage.

Si /proc n'est pas disponible, le bruteforçage de tous les PID est nécessaire.

Évasion d'environnement chrooté

ch[ouc]root ?

Simple restriction de la racine du système de fichier d'un processus.
Contact extérieur possible : mémoire partagée, signaux. . .

Pwned !

Si on peut envoyer des signaux à des processus, on peut les tracer !
⇒ Injection d'un shellcode permettant de s'évader de la cage.

Si /proc n'est pas disponible, le bruteforçage de tous les PID est nécessaire.

Firewall applicatif

Définition

ACLs basées sur l'application.

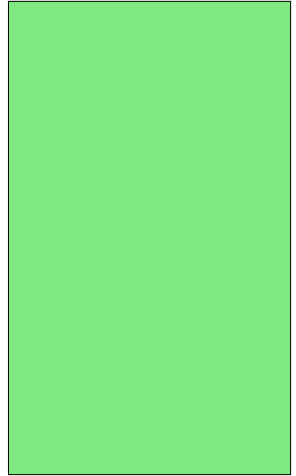
⇒ Application autorisée == tremplin

Objectifs d'un programme malicieux :

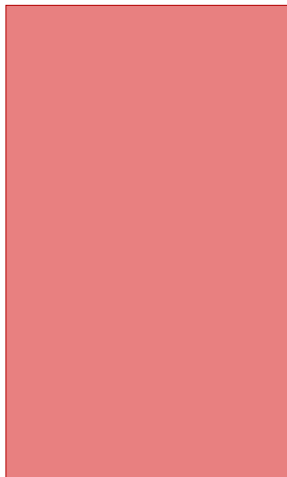
- Injection d'un `connect()` dans Mozilla,
- Récupération du descripteur de fichier,
- Transfert du descripteur de fichier au processus malicieux ;



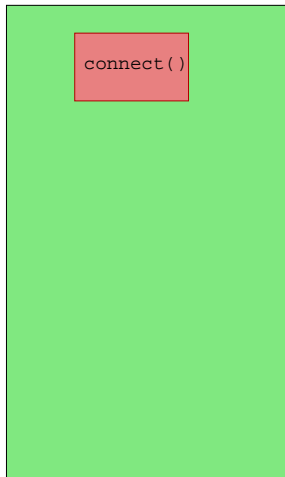
Processus malicieux



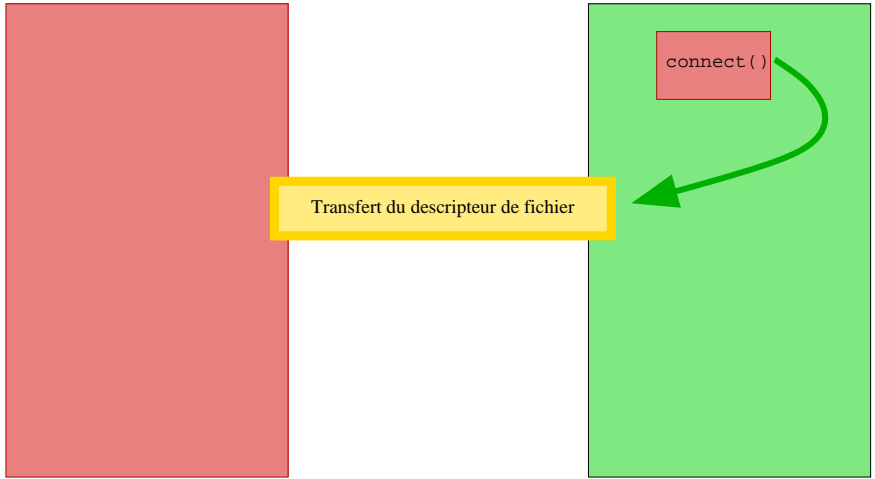
Mozilla



Processus malicieux



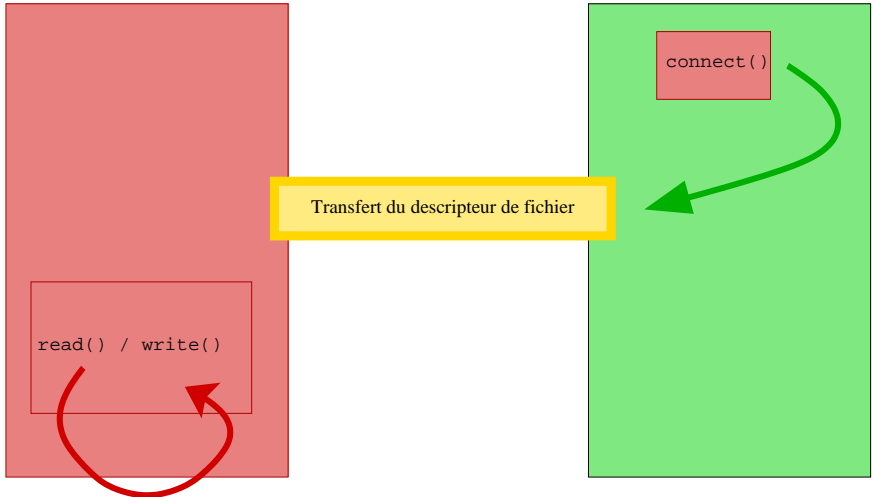
Mozilla



Processus malicieux

Mozilla





Processus malicieux

Mozilla



Contrainte

Figure de style imposée

Réussir à faire passer toutes les applications à travers ce « tunnel ».

Interception des bibliothèques dynamiques

```
$ LD_PRELOAD="/home/moi/lib/liberte.so"  
$ export LD_PRELOAD  
$ wget http://slashdot.org/
```

Surcharge de connect()

```
int connect(int sockfd, struct sockaddr *serv_addr
            , socklen_t addrlen)
{
    pid_t pid = atoi(getenv("PTRACE_PWNED"));

    socket(PF_UNIX, SOCK_STREAM, 0);
    ...
    if (fork() == 0) {
        inject_shellcode(pid, serv_addr);
        exit(0);
    }
    ...
    fd = receive_fd(unixfd);
    dup2(fd, sockfd);

    return sockfd;
}
```


Fonctionnalité méconnue

Transfert de descripteur de fichier

Il est possible de transmettre un descripteur de fichier entre deux processus indépendants à travers une socket UNIX.

```
struct cmsghdr *ch;  
struct msg_hdr msg;  
char ancillary [ CMSG_SPACE( sizeof( fd ) ) ];
```

```
ch = CMSG_FIRSTHDR(&msg);  
ch->cmsg_level = SOL_SOCKET;  
ch->cmsg_type = SCM_RIGHTS;  
*(int *) CMSG_DATA(ch) = fd;
```

```
sendmsg( sockfd , &msg , 0 );
```

Shellcode

Le shellcode :

- 1 Créé une socket UNIX vers le traceur,
- 2 Initie la connexion vers le serveur désiré,
- 3 Et envoie le descripteur de fichier.

Ne gère pas encore l'UDP.

Conclusion

ptrace() et ses limites

- Fonctions limitées,
- Portabilité impossible,
- Bugs historiques ;

L'avenir

L'avenir se tourne vers les dtrace-like ?

- Solution complètement kernel-space,
- Haut niveau,
- Scriptable ;

Des questions ?

Merci de votre attention !
Des questions ?