Travail d'études
Licence Informatique
2003-2004

Les techniques de protection du logiciel

Julien BURLE Térence SCALABRE

Encadré par Mr Pierre CRESCENZO

Sommaire

Comment un développeur peut essayer de protéger efficacement les secrets de son programme ? Intéressons nous de plus prés à l'intérêt de protéger son code et aux différentes techniques mise en oeuvre dans le langage C sous un environnement UNIX.

Chapitre I : Quelques raisons de protéger le code et le point de vue légal

Doit on protéger ses code sources et pour quelles raisons ? Sommes-nous légalement protégé contre l'utilisation du code source de notre logiciel ?

Il existe différents points de vue sur le fait ou non de protéger son code. De plus les raisons varient celon les créateurs de logiciels (grandes entreprises, petit programmeur etc...). Alors que les points de vue diffèrent, les idées sont arrêtées, essayons de présenter les arguments en faveur ou contre la protection du logiciel au niveau juridique mais aussi au niveau du développement.

1 - Pourquoi protéger son code ?

Cela peut empêcher l'utilisation du code source de notre programme dans le but d'en tirer profit, où d'accéder aux technologies mise en ouvre dans le logiciel. C'est aussi un bon moyen de s'assurer qu'un client ne fasse pas de mise à jour sans autorisations. D'autre part cela permet d' empêcher des personnes mal intentionnées de trouver des failles par une analyse du code source et de s'en servir contre le programme.
Par exemple : les créateurs du ver Blaster se sont servis d'une mise à jour de Windows, pour repérer une faille dans le système d'exploitation et ainsi déployer leur virus.

Mais cela présente aussi de nombreux inconvénients. Par exemple ces techniques empêchent la plupart du temps quiconque d'améliorer le programme (ce qui peut être fait gratuitement). De plus laisser son code visible par tous peut aussi être un bon moyen de se faire connaître. Si le code est bon, les personnes qui le verront seront s'en souvenir. Enfin protéger ses programmes coûtent très cher, en temps et en argent. Cet investissement pourrait être mieux utiliser pour l'amélioration du programme et de ces performances.

2 - L'aspect juridique

La loi Française nous protège dans la plupart des cas contre le piratage et l'espionnage industriel. Un logiciel est protégé de la décompilation par le droit commun d'une part (art 323-1 du nouveau code pénal) sur le fait de pénétrer dans un système, élément de la propriété d'autrui. Et d'autre part, le logiciel trouve une protection dans le droit d'auteur. Celui-ci interdit en effet la décompilation en France. L'œuvre logiciel est alors considérée comme une œuvre de l'esprit. Effectivement, le programme d'ordinateur n'étant pas lisible d'emblée; pour accéder aux éléments de fond, on doit le traduire dans un langage compréhensible (ce qui revient à le décompiler), et donc toucher à sa forme. Or le droit d'auteur interdit les actes de reproduction et de traduction auxquels la décompilation s'assimile. Cette forme de protection devient une arme absolue contre la décompilation.

La décompilation est toutefois tolérée sous certaines conditions par la Directive de mai 1991 (Dir con. CE n°91-250, 14 mai 1991, article 6, JOCE 17 mai 1991, n°L.122, p.42) et la loi française en reprend les dispositions dans l'article L 122-6 du code de propriété intellectuelle. Le but avoué de la décompilation est d'assurer l'interopérabilité, c'est-à-dire l'articulation des logiciels les uns avec les autres. Des conditions restrictives, protectrices du droit des auteurs de logiciels originaux sont posés par les textes :

- celui qui décompile doit être fondé à utiliser le logiciel
- il ne doit pas avoir les informations par une autre voie
- la décompilation est strictement limitée au nécessaire
- l'information acquise ne sert qu'à la réalisation de l'interopérabilité
- on ne peut écrire un nouveau logiciel à partir d'un logiciel décompilé

Le conseil européen a proposé le 18 mai 2004, une directive sur la brevetabilité des inventions mise en oeuvre par ordinateur. Ce qui rendrai le droit d'auteur du logiciel inutile. La brevetabilité des logiciels pourrait contraindre les acteurs du logiciel libre à des dépôts de brevets défensifs, qui détournent leur capacité d'investissement vers des procédures contre-productives. Une entreprise pourrait ainsi déposer un brevet sur un type de fichier, un algorithme ou un logiciel. Ce qui à long terme pourrait causer la mort du logiciel libre.

Mais peu importe les raisons de le faire, financières, personnel ou par orgueil. Les techniques de protections existent. A nous de savoir si il faut où non les utiliser. Et donc de savoir comment les utiliser.

Chapitre II : Techniques de protection du code source

Nous pouvons distinguer plusieurs types de protections : dans un premier temps des techniques qui vont empêcher le rétro-ingénieur(ou « reverser » en anglais) d'utiliser des débogueurs, ces protections sont efficaces pour ralentir la compréhension du code, dans un deuxième temps des méthodes pour protéger le code en lui même soit par des modifications de son code assembleur, soit dans un cadre plus général par des techniques autorisant l'exécution du code compilé sous certaines conditions.

1 – Détecter l'appel système ptrace

L'appel système **ptrace** sert, sous UNIX, à déboguer les processus. Il permet de tracer un processus et de lui faire faire ce que l'on souhaite : changer sa mémoire, modifier ses registres... L'utilisation primordiale de cette fonction est l'implémentation de points d'arrêt pour le débogage (Ce syscall est utilisé par tous les débogueurs sous UNIX comme gdb, strace...) On peut alors exécuter le code, instructions par instructions ce qui est extrêmement utile au reverser. Un processus peut donc prendre des mesures anti-ptrace, afin de ne pas être tracé. Cette technique utilise le fait qu'un processus ne peut pas être tracé plusieurs fois et il peut se tracer lui même à l'aide de la requête TRACEME. Nous pouvons donc vérifier si notre programme est débogué ou pas.

Voici les détails sur ptrace :

```
synopsis
        #include <sys/ptrace.h>
        int ptrace(int request, int pid, int addr, int data);
valeur renvoyée
        ptrace renvoie 0 s'il réussit, ou −1 s'il échoue, auquel
cas errno contient le code d'erreur.
```

Si un ptrace est en cours et que l'on tente un deuxième ptrace dessus, le deuxième va échouer. Il va donc suffire au processus d'essayer de se tracer et de vérifier la valeur de retour de ptrace : si elle est égale à -1, il est alors tracé, sinon tout va bien.

Voici un code illustrant cette technique :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>

int main(void) {
     if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
           printf("DEBUGGING... Bye\n");
           return 1;
     }
     printf("Hello\n");
     return 0;
}
```

Testons le code :

```
burlej@var > gcc -o test_ptrace test_ptrace.c
burlej@var > test_ptrace
Hello
burlej@var > gdb -q test_ptrace
(gdb) run
Starting program: /u/linfo/burlej/te/test/test_ptrace
DEBUGGING... Bye
Program exited with code 01.
```

La protection fonctionne, on notera que l'on peut appliquer la même méthode à la fonction strace.
Dans cette exemple nous envoyons explicitement un message d'erreur sur stdout, pour plus d'efficacité, nous pouvons n'émettre aucun message d'erreur.
Cette méthode est très simple à réaliser mais un reverser chevronné pourra par exemple supprimer le TRACEME ou commencer le débogage une fois le test passé. Il est aussi possible d'intervenir au niveau du kernel pour modifier le comportement de ptrace.

2 – Détecter les points d'arrêt (breakpoints)

Un breakpoint sert, dans les débogueurs à faire une pause ou à stopper l'exécution d'un processus à un instant donné. Pour observer, changer un registre ou un emplacement de la mémoire, un reverser va poser des breakpoints. Une technique de protection passe alors par la détection de ces breakpoints dont l'instruction assembleur est **int3** (l'opcode correspondant est 0xCC). Lorsqu'un débogueur met un breakpoint à une certaine adresse, il sauve l'opcode courant pour mettre à la place 0xCC. Quand un code exécute l'instruction int3, il reçoit un signal SIGTRAP.
Pour se protéger des breakpoints, on va donc scanner notre code à la recherche de l'opcode 0xCC.
Voici un exemple mettant en oeuvre cette technique :

```
#include <stdio.h>
#include <stdlib.h>

void foo(void) {
  printf("Salut\n");
}

int main(void) {
  unsigned long *i;
  /* la boucle cherche des breakpoints dans la fonction foo() */
  for(i = (unsigned long *)foo; i < (unsigned long *)main; i++) {
    if(((*i & 0xff) == 0xcc) ||
       ((*i & 0xff00) == 0xCC) ||
       ((*i & 0xff0000) == 0xCC) ||
       ((*i & 0xff000000) == 0xCC)) {
      /* si un opcode est 0xCC */
      printf("Breakpoint detecte \n");
      /* on quitte le programme */
      exit(EXIT_FAILURE);
    }
    foo();
    return 0;
  }
}
```

On compile:

```
burlej@charon ~/te/test> gcc -o test_bp test_bp.c
burlej@charon ~/te/test> test_bp
Salut
burlej@charon ~/te/test> gdb -q test_bp
(gdb) b foo   # on place un breakpoint au debut de la fonction foo
Breakpoint 1 at 0x804835c
(gdb) run
Starting program: /u/linfo/burlej/te/test/test_bp
Breakpoint detecte

Program exited with code 01.
```

Cette protection fonctionne. Le seule problème de cette technique est qu'un programme peut contenir l'opcode 0xCC sans que ce soit forcément un breakpoint, nous provoquons alors un arrêt involontaire. Nous pouvons cependant prévoir des exceptions ou faire en sorte que les fonctions sensibles que l'on contrôle n'en contiennent pas.

3 - Poser des faux breakpoints

Cette méthode s'appuie toujours sur l'utilisation des breakpoints comme la précédente. Comme nous avons vu lorsqu'un programme rencontre un opcode 0xCC, il reçoit un signal SIGTRAP. Lorsque le débogueur voit un 0xCC, il met en pause le programme et attends l'instruction de continuer. Le comportement par défaut d'un processus lorsqu'il reçoit un SIGTRAP est de quitter. Cette technique consiste donc à changer ce comportement par défaut grâce à la fonction `signal` appliquée à SIGTRAP.

Voici un exemple :

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void sighandler(int signal) {
  /* si le processus n'est pas tracé on arrive ici */
  printf("Le pass est \"salut\"\n");
  exit(0);
}

int main(void) {
  /* on place le sighandler */
  signal(SIGTRAP,sighandler);
  /* on place un faux breakpoint */
  __asm__("int3");
  /* le debogueur arrive ici */
  printf("pris au piege\n");
  return EXIT_FAILURE;
}
```

L'execution normale du programme nous donne comme résultat :

```
burlej@tamanrasset ~/te/test> gcc -o test_fauxbp test_fauxbp.c
burlej@tamanrasset ~/te/test> test_fauxbp
Le pass est "salut"
```

Avec gdb :

```
burlej@tamanrasset ~/te/test> gdb -q test_fauxbp
(gdb) run
Starting program: /u/linfo/burlej/te/test/test_fauxbp

Program received signal SIGTRAP, Trace/breakpoint trap.
0x080483d3 in main ()
(gdb) cont
Continuing.
pris au piege

Program exited with code 01.
(gdb) quit
```

Lors de l'exécution normale du code, la fonction __asm__("int3"); (le breakpoint) va lancer le sighandler. Si le code est débogué , il va continuer avec les instructions suivant l'int3. Comme auparavant il est préférable de ne pas mettre de signal d'erreur (nous les avons placés pour l'exemple).

Ces trois techniques interviennent au niveau de l'utilisation des appels système ptrace, strace utilisées par les débogueurs, ou sur le principe de fonctionnement des débogueurs (Nous avons vu alors la gestion des points d'arrêt comme protection). Voyons maintenant des techniques se basant sur d'autres types de protections visant a gêner ou empêcher le désassemblage

4 - Protection par code checksum

Voici une protection élégante du code, elle se base sur le code original du programme (au niveau assembleur) et vérifie par un algorithme que celui ci n'ait pas été modifié. Cette protection gène considérablement le reverser qui en modifiant le code binaire pour simplifier le code (en enlevant des techniques de protections connues qu'il rencontre par exemple) va empêcher le programme de fonctionner. L'algorithme va donc autoriser le programme à s'exécuter que si le nombre d'opcodes n'a pas changé.

Voici un programme en assembleur, il contient une partie que nous allons ensuite essayer de protéger efficacement (la partie `coool`), tout d'abord voyons comment un reverser peut facilement sauté sur cette partie.

```
.globl main
main:
xorl %eax,%eax
cmp $1,%al
je coool

pascoool:
/* exit(0); */
exit:
     xorl %eax,%eax
     xorl %ebx,%ebx
     inc %eax
     int $0x80

coool:
     xorl %eax,%eax
     movb $4, %al        // 4 = SYS_write
     xorl %ebx,%ebx
     inc %ebx            // 1 = STDOUT
     movl $chaine,%ecx   // l adresse de Gagne
     xorl %edx,%edx
     mov $8,%edx         // on ecrit 8 octets
     int $0x80           // on execute le syscall write
     jmp exit
chaine:
.string "Gagne !\n"
```

Pour l'instant, ce programme quitte tout de suite à cause de l'instruction `cmp $1,%al` suivi de `je coool` (`je` fais un saut à l'adresse indiquée en cas d'égalité). Modifions maintenant le binaire grâce au programme hexedit (un éditeur hexadécimal) pour que le programme saute tout le temps sur `coool`.

Nous remplaçons :

```
8048336 : 3c 01 cmp $0x1,%al
8048338 : 74 07 je 8048341 <coool>
```

Par :

```
8048336 : 3c 01 cmp $0x0,%al
8048338 : 74 07 je 8048341 <coool>
```

Un seul octet est modifié dans le binaire : au lieu de comparer %al avec 1 on le compare avec 0 (pour avoir une égalité en résultat de la comparaison et ainsi sauter sur l'instruction `coool` qui nous intéresse).

```
burlej@rome ~/te/test> test_checksum
Gagne !
```

Pour contrer ce genre d'attaque (on modifie une valeur à comparer pour sauter vers une autre instruction) nous allons utiliser la technique de protection par code checksum. Nous faisons la somme de tous les opcodes contenus dans le main jusqu'à `coool`, on obtient le résultat 0xfef, soit 4079.

Voici le nouveau code :

```
.globl main
main:
jmp checksum
rett:
xorl %eax,%eax
cmp $1,%a1
je coool

pascoool:
exit:
[...]
coool:
[...]
     jmp exited

checksum:
     xorl %ebx,%ebx
     mov $checksum,%ecx
     sub $main,%ecx // boucle de main jusqu'à checksum
     mov $main,%esi // on part de main
boucle:
     lodsb
     add %eax,%ebx  // la somme dans ebx
     loop boucle
     cmpl $4079,%ebx
     jne baaad      // si la somme est fausse
     jmp rett
baaad:
     // message d'erreur
     [...]
     jmp exit
[...]
```

Si maintenant nous changoens une partie du code avec hexedit, par exemple le `cmp $1, %al` en `cmp $0,%al`, la somme de tout le code va changer (elle est décrémentée de 1).

```
burlej@rome ~/te/test/> checksum             #normal
burlej@rome ~/te/test/> hexedit checksum
burlej@rome ~/te/test/> checksum             #modifié
Bad!
```

Le checksum a été alerté. Notons que ce type de protection doit être étroitement liée au partie vitale du programme pour plus d'efficacité (On peut faire un checksum sur une plus petite partie du programme et non sur le code entier). Enfin si le checksum est faux nous pouvons directement planté le PC pour gêner le reverser. (la séquence d'instruction F0 0F C7 C8 peut être utilisée pour planter un Pentium).

5 – Faux désassemblage (false disassembly)

Cette technique permet de fausser les résultats du désassemblage du code. Nous allons pour cela rajouter dans notre code les opcodes d'un jump vers des instructions que nous voulons exécuter, et entre les opcodes du jump et ces instructions mettre de faux opcodes destinés à faire perdre l'alignement du code désassemblé. Le but est de brouiller le code pour compliquer la tache du reverser. Voici un exemple en assembleur pour illustrer cette technique (en ASM syntaxe AT&T). Ce code lit une chaîne sur l'entrée standard et la compare avec le mot de passe “pass”. “Good” est écrit sur la sortie standard si le mot de passe est bon, et “Bad” est écrit si il est mauvais.

```
.globl main
main:
        xorl %eax,%eax
        movb $3,%al             // 3 = SYS_read
        xorl %ebx,%ebx          // 0 = STDIN
        movl %esp,%ecx          // %ecx = buffer
        mov $4, %edx            // 4 octets sont lus
        int $0x80               // exécute le syscall read
        cmpl $0x73736170, (%ecx)    // compare la chaîne de caractère rentrée avec «pass»
        je good         // si les chaînes sont égales on saute vers good
bad:                    // sinon on est en bad
        xorl %eax,%eax
        movb $4,%al             // 4 = SYS_write
        xorl %ebx,%ebx
        inc %ebx                // 1 = STDOUT
        pushl $0xa646162        // on met «bad» dans la pile
        movl %esp,%ecx          // on place dans %ecx l'adresse de bad
        xorl %edx,%edx
        mov $4,%edx             // on écrit 4 octets
        int $0x80               // et on exécute le syscall write
        jmp exit
good:
        xorl %eax,%eax
        movb $4,%al             // 4 = SYS_write
        xorl %ebx,%ebx
        inc %ebx                // 1 = STDOUT
        movl $cool,%ecx         // l'adresse de «good» ce retrouve dans %ecx
        xorl %edx,%edx
        mov $7,%edx             // on écrit 7 octets
        int $0x80               // on exécute le syscall write
exit:
        xorl %eax,%eax
        inc %eax                // 1 = SYS_exit
        xorl %ebx,%ebx          // exit(0);
        int $0x80               // exécute le syscall exit
cool:
.string "Good !\n"
```

On compile ce programme et on le teste :

```
burlej@achille ~/te/test> gcc false_diss.s -o false_diss
burlej@achille ~/te/test> echo pass | false_diss
Good !
burlej@achille ~/te/test> echo salu | false_diss
bad
burlej@achille ~/te/test>
```

On désassemble le code pour examiner le main:

```
burlej@achille ~/te/test> objdump -d false_diss

false_diss:     format de fichier elf32-i386

[...]
080482f4 <main>:
 80482f4:       31 c0                   xor    %eax,%eax
 80482f6:       b0 03                   mov    $0x3,%al
 80482f8:       31 db                   xor    %ebx,%ebx
 80482fa:       89 e1                   mov    %esp,%ecx
 80482fc:       ba 04 00 00 00          mov    $0x4,%edx
 8048301:       cd 80                   int    $0x80
 8048303:       81 39 70 61 73 73       cmpl   $0x73736170,(%ecx)
 8048309:       74 19                   je     8048324 <good>
[...]
```

C'est dans la fonction main que se trouve la partie la plus intéressante pour un cracker. Nous effectuons la comparaison entre la chaîne rentrée et «pass» dans ce main. Le code n'étant pas protégé, il est très facile pour un reverser de trouver le mot de passe en analysant la ligne :

```
cmpl   $0x73736170,(%ecx)
```

`$0x73736170` équivaut à «pass».

Maintenant, nous allons essayer de protéger la fonction main grâce à la technique de faux désassemblage. Voici le main après modifications :

```
main:
     xorl %eax,%eax
     movb $3,%al
     xorl %ebx,%ebx
     movl %esp,%ecx
     mov $4, %edx
     int $0x80
.ascii "\xEB\x01\xE8"
     cmpl $0x73736170, (%ecx)
     je good
```

La ligne ajoutée avant la comparaison n'affecte pas le code qui marche comme avant, cependant nous obtenons un code désassemblé différent :

```
080482f4 <main>:
 80482f4:  31 c0               xor    %eax,%eax
 80482f6:  b0 03               mov    $0x3,%al
 80482f8:  31 db               xor    %ebx,%ebx
 80482fa:  89 e1               mov    %esp,%ecx
 80482fc:  ba 04 00 00 00      mov    $0x4,%edx
 8048301:  cd 80               int    $0x80
 8048303:  eb 01               jmp    8048306 <main+0x12>
 8048305:  e8 81 39 70 61      call   6974bc8b <_end+0x6170278f>
 804830a:  73 73               jae    804837f
<__libc_csu_fini+0x3>
 804830c:  74 19               je     8048327 <good>
```

Après le `int $0x80`, le code est totalement différent par rapport au précédent. La ligne `.ascii "\xEB\x01\xE8"` ajoute les opcodes d'un `jmp +1 (\xEB\x01)`, puis un opcode de début de call `(\xE8)` (On saute par dessus, donc). Le désassembleur va prendre le `\xE8` comme le début d'un call (ce qui n'est pas le cas), et va prendre les 4 octets suivant comme l'adresse du call :

```
8048305:  e8 81 39 70 61       call   6974bc8b
```

L'alignement est donc perdu et ne va être retrouvé que plusieurs lignes après (la dernière instruction du main est la bonne `je 8048327 <good>`). Grâce à cette technique le code désassemblé semble incompréhensible à première vue.
Même si cette technique peut être déjouer elle est utile combiner à d'autres pour camoufler du code (par exemple lorsque l'on implémente la méthode « anti-ptrace »).

6 – Polymorphisme

Cette technique assez lourde pour le programme peut être très utile pour décourager un cracker (ce qui en soit peut être considérer comme une protection). L'idée est que, pour arriver à un résultat voulu (par exemple placer la valeur 1 dans le registre eax) nous pouvons passer par beaucoup de tergiversations. Ces modifications vont compliquer la compréhension du code.

Par exemple :

```
movl $1,%eax
```

peut être remplacé par :

```
push $1
popl %eax
```

mais aussi par :

```
xorl %eax,%eax
add $0x13,%eax
and %eax,%eax
dec %eax
sub $0x11,%eax
```

ou encore par :

```
xorl %eax,%eax
cdq                 // équivaut à un xor %edx,%edx
leal 1(%edx),%eax   // place 1 dans eax
movl %fs,%ebx       // le registre fs contient toujours 0
add %ebx,%eax       // équivaut à add $0,%eax
```

Le reverser va devoir comprendre le but de toutes ces instructions, qui peuvent parfois le dérouter. L'inconvénient de cette méthode est qu'elle ralentit le code. On peut aussi mêler des instructions inutiles et sans effets afin de faire ressembler le code à du « false disassembly ».

7 – Strip

L'outil strip sous UNIX permet d'enlever toutes les références aux symboles dans un binaire afin de gagner de la place. Ces symboles sont utiles au linking et au débogage mais n'apportent rien au fonctionnement normal du programme. Au désassemblage, un binaire ainsi traité ne contiendra plus de références aux noms de fonctions (même pas le main). Cette technique gène le débogage et le désassemblage puisqu'il n'est plus aussi aisé de mettre un point d'arrêt sur une fonction, que l'on doit alors désigner par son adresse plutôt que par son nom.

Nous pouvons regarder les effets de strip sur ce petit programme :

```
#include <stdio.h>

void fonct1(void) {
  printf("la fac !\n");
}

int main(void) {
  printf("Salut ");
  fonct1();
  return 0;
}
```

Voici quelques tests,

```
burlej@lima ~/te/test> gcc -o test_strip test_strip.c
burlej@lima ~/te/test> test_strip
Salut la fac !
burlej@lima ~/te/test> strip -o test_strip2 test_strip
burlej@lima ~/te/test> test_strip2
Salut la fac !
```

Pour l'instant les deux exécutables sont les mêmes, mais analysons leur code désassemblé

```
burlej@lima ~/te/test> objdump -d test_strip

test_strip:     format de fichier elf32-i386

Déassemblage de la section .init:

08048230 <_init>:
 8048230:       55                      push   %ebp
[...]

Déassemblage de la section .text:

08048278 <_start>:
 8048278:       31 ed                   xor    %ebp,%ebp
[...]
08048328 <fonct1>:
[...]
08048340 <main>:
[...]
```

Dans l'exécutable normal nous retrouvons les références au main et à la fonction 1.

```
burlej@lima ~/te/test> objdump -d test_strip2

test_strip2:     format de fichier elf32-i386
[...]
08048278 <.text>:
 8048278:       31 ed                   xor    %ebp,%ebp
[...]
```

Dans la version modifié par strip, nous ne trouvons plus aucune de ces références.

8 – Les Obfuscateurs

Cette méthode consiste à compliquer le code source en altérant sa lisibilité. Le but est de rendre le code source illisible, lors d'une décompilation éventuelle.
Un obfuscateur transforme les noms de variables et de fonctions. Par exemple le nom de la fonction "int main()" deviendra "c e()", une lettre "H" deviendra "\x48" et les noms de variables seront moins explicites, comme par exemple "MAX_COUNT" pourra devenir "b" ou "1235".

Nous avons utilisé un programme : COBF (The C/C++ Sourcecode Obfuscator/Shroude), qui nous a permis de transformer un programme simple:

```
#include <stdio.h>
#ifdef unix
#define MAX_COUNT 10
#else
#define MAX_COUNT 20
#endif

int main()
{
  int i;
  for (i = 0; i < MAX_COUNT; ++i)
        printf("Hello %d!\n", i);
  return 0;
}
```

qui devient:

```
#include<stdio.h>
#include"cobf.h" /* Ceci est ajouter par le programme */
#ifdef unix
#define b 10
#else
#define b 20
#endif
c e(){c a;d(a=0;a<b;++a)f("\x48\x65\x6c\x6c\x6f\x20\x25\x64\x21\n",a)
;g 0;}
```

le fichier "cobf.h" contient:

```
#define c int
#define e main
#define d for
#define f printf
#define g return
```

Cette technique bien que trés utilisée, surtout en Java, n'est pas très puissante. Il suffit d'un peu de temps et de courage pour récupérer le programme source. La complexité pour retrouver tous les types de variables, fonctions et nom de variables augmentent si le programme est plus long. Même si cette technique est très simple, il est nécessaire de la connaître pour savoir la contrer (elle s'inscrit dans les techniques pouvant décourager le reverser).

9 – Cryptage

Une méthode célèbre pour rendre le « reverse » plus difficile consiste à crypter la section TEXT (le code) directement dans le binaire. Une fois cryptée, un petit bout de code va décrypter toute la section afin que le programme s'exécute normalement. Cette méthode résiste à un désassemblage directe, elle ne ralentit pas le programme car le décryptage ne s'effectue que durant le lancement de l'exécutable.

10 – Conseils utiles lors de l'implémentation de protection

Ces conseils sont utiles principalement pour les programmes dans une version d'évaluation qu'un reverser va essayer de cracker (il recherchera alors un mot de passe ou une fonction manquante dans la version d'évaluation).

- La première astuce consiste à ne jamais laisser un mot de passe codé en clair, cela rendrait la tache du reverser vraiment trop facile.
- - Ensuite nous devons à tout prix éviter un test du type

```
if( verifier_mot_de_passe() == vrai)
      continuer();
else
      arréter();
```

- Une fonction de vérification de clé ne doit pas permettre de déchiffrer son comportement afin de créer des générateurs de clé. On pourra pour cela utiliser un RTF (Rotate through Carry to the Left) ou un RCR (Rotate through Carry to the Right) sur une valeur associée à la clé. En effet il est difficile d'effectuer l'opération inverse.
- Dans les zones sensibles placer beaucoup de saut conditionnels rend le reverse pénible surtout si certaines protections s'activent et plantent le PC dans les cas les plus extrêmes.
- Utilisez des sections critiques comme « nombre magique » pour manipuler le code. Par exemple, utilisez une portion de code pour « XORer » une partie de la clé (cette clé sera associée au fonctionnement du programme). Cela rend impossible la modification de ces portions de code. Le but est de rendre alors la gestion des clés dynamiques avec le code et en particulier les sections que le reverser tentera de modifier.
- Lors de l'utilisation de techniques contre les débogueurs, nous pouvons désactiver le clavier avant d'entrer dans une section critique et le réactiver ensuite, lorsque le reverser placera un point d'arrêt dans une de ces sections il ne pourra plus utiliser son clavier pour lancer une nouvelle commande notamment celle pour continuer l'exécution avec le débogueur. Il sera alors obligé de repérer où la désactivation s'opère et ensuite redémarrer pour continuer.

Les techniques présentées sont assez basiques. La plupart des protections industrielles utilisent des méthodes plus élaborées qui mettent en application les principes de protection que nous avons évoquées. Cependant l'idée reste la même. En réfléchissant, nous pouvons trouver beaucoup d'idées de protections, de manières à compliquer la vie des « reverser ». Cependant il nous semble illusoire de penser mettre au point une technique inviolable, une protection pourra seulement retarder le reverser déterminé.

Après l'éludes de ces techniques nous pensons qu'un programmeur a tout intérêt de livrer son code source. D'une part car il ne perdra pas de temps en effectuant beaucoup de protections qui au final risque d'être contournées par les reverser les plus expérimentés (on a noté au cours de nos recherches les mines d'informations accessibles par les sites de cracker diffusant des listes de programmes crackés et la façon de contourner leur propres protections). D'autre part en rendant libre le code source, chaque programmeur peut corriger des bugs, des failles de sécurité qui au final renforceront le programme. Les groupes de programmation open-source en sont les meilleurs exemples.

Quelques définitions:

Interopérabilité : Capacité qu'ont deux systèmes à se comprendre l'un l'autre et à fonctionner en synergie. Contraire: incompatibilité.

Rétro-ingénierie ou ingénierie inverse (reverse-engineering) : reconstituer les sources ou le modèle d'un système d'information existant. Il consiste à désassembler, décompiler un logiciel existant afin de déterminer comment il a été conçu. Ceci correspond au mécanisme inverse du développement.

Code d'opération (opcode) : dans un programme assembleur, les instructions à exécuter sont stockées dans la mémoire centrale, codées avec des valeurs entières d'octets: les opcodes.

Point d'arrêt (breakpoint) : lors du débogage, permet d arrêter l'exécution du programme à une ligne donnée.

Bibliographie

http://vx.netlux.org/lib/vsc04.html : pages contenant quelques protections de base avec exemple

http://www-ensimag.imag.fr/cours/Systeme/documents/reverse.engineering/reverse.engineering.rtf
site très intéressant présentant d'une part les protections pouvant être utilisées, ainsi que des astuces, l'auteur s'attache cependant beaucoup à décrire les techniques de « reverse enginering » plutôt que les protections.

http://in.fortunecity.com/skyscraper/browser/12/htoprote.html : quelques informations intéressantes sur les techniques de protection par gestion des breakpoints.

http://www.acm.uiuc.edu/sigmil/RevEng/index.html : une présentation générale du « reverse enginering »

http://home.arcor.de/bernhard.baier : site traitant les ofuscateurs (COBF)

http://www.droit-ntic.com/

http://www.avocats-publishing.com/ nos deux principales ressources pour l'aspect juridique

http://www.hower.org/Kudzu/Articles/Piracy/index.html la page web de kudzu, un programmeur donnant son avis sur les besoins de protéger son logiciel. Très interessant à lire pour les idées développées.

http://www.woodmann.com/crackz/Miscpapers.htm : Diverses informations sur certaines protections

The Hackademy journal : Articles sur diverses techniques de protection.

Programmez : Article sur les protections java, principalement les obfuscateurs.