

REVERSE ENGINEERING

REMERCIEMENTS

Tous mes remerciements à Serge Rouveyrol qui m'a donné le goût de la bidouille en assembleur.

Remerciements également à la +HCU et à tous ses étudiants pour tout le travail qu'ils ont réalisé et les connaissances qu'ils ont rassemblées et qu'il mettent gracieusement à la disposition du plus grand nombre.

SOMMAIRE

1	Introduction.....	5
2	Notions d'assembleur 80x86 INTEL.....	6
2.1	Représentation de la mémoire	6
2.2	Les registres.....	6
2.2.1	Registres de segments	7
2.2.2	Registres d'offset	7
2.2.3	Registre de flags.....	7
2.3	Les instructions les plus courantes	8
2.4	Appel de procédure.....	11
3	Principes généraux.....	12
3.1	Le principe du crack	12
3.2	Les outils du cracker.....	12
3.3	Les API Windows.....	12
3.4	Les services DOS et BIOS	12
3.5	Les principes de base des protections.....	13
4	Les différents types de protection.....	15
4.1	CD check.....	15
4.1.1	Vérification de la présence d'un CD	15
4.1.2	Vérification de la présence d'un fichier	16
4.2	Mots de passe (serial number).....	16
4.3	Time check	19
4.4	Clé électronique.....	19
5	Comment se protéger.....	20
5.1	Principes de base	20
5.2	Vérification de clé.....	20
5.3	Désactivation des débogueurs	21
5.4	Désactivation des désassembleurs	23
6	Conclusion	25

1 Introduction

Le reverse engineering consiste à analyser et comprendre un code inconnu (dont on n'a pas le source). Cette science peut être utilisée pour ajouter des fonctionnalités à un programme, pour comprendre le fonctionnement d'un logiciel insuffisamment documenté, pour comprendre la technologie employée par un concurrent ou, dans le cas qui nous intéresse, pour supprimer les protections des logiciels.

Les protections consistent à n'autoriser l'accès sans restriction au logiciel qu'aux utilisateurs ayant payé pour obtenir cet accès. La partie du reverse engineering consistant à supprimer les protections s'appelle le cracking et se résume à lever ces restrictions d'utilisation.

Les logiciels protégés sont généralement les sharewares et les versions d'évaluation que l'on peut trouver sur le web ou sur des CD offerts dans des magazines.

Pour les concepteurs de logiciels, la protection est un sujet critique puisque le principal vecteur de diffusion de leurs programmes est internet, et qu'internet est également le principal vecteur de diffusion des cracks (programmes chargés de supprimer les protections). Un utilisateur peut donc télécharger une version d'évaluation puis le crack correspondant et utiliser librement le logiciel sans verser de droits à l'auteur. Ceci est illégal bien entendu mais est couramment pratiqué par les particuliers un tant soit peu avertis. Cette pratique lèse les petits développeurs et permet la large diffusion des produits des grandes compagnies.

Pour réaliser des protections efficaces, il faut connaître les techniques employées par les crackers et comme les crackers ne disposent pour travailler que du code exécutable d'un programme, il faut, comme eux, connaître l'assembleur (c'est dur mais c'est comme ça). Ce rapport comportera donc un chapitre sur l'assembleur INTEL 80x86.

Ensuite je traiterai différentes méthodes de protection avec leurs points faibles et la façon de les cracker. Puis je finirai par des principes et des astuces pour décourager le cracker moyen (hélas, le cracker expérimenté est capable de venir à bout de n'importe quelle protection).

2 Notions d'assembleur 80x86 INTEL

2.1 Représentation de la mémoire

La représentation de la mémoire dans les processeurs INTEL est héritée du 8086 (processeur 16 bits) conçu en 1978. Je vais donc décrire les principes du 8086 qui restent valables (à peu près) pour tous les processeurs du 80286 au Pentium (80586).

L'unité de base du 8086 est l'octet. Ceci signifie que chaque cellule de mémoire peut contenir un nombre de 0 à 256. Chaque cellule de mémoire a une adresse. La première dans la mémoire a l'adresse 0, puis l'adresse 1, puis 3, etc.

Les registres du 8086 peuvent contenir 1 mot (2 octets), ce qui veut dire qu'un registre peut stocker et effectuer des opérations sur des nombres allant de 0 à 65535.

Un moyen d'accéder la mémoire est de placer l'adresse d'une cellule de mémoire dans un registre et de dire au processeur d'utiliser la donnée à cette adresse.

Puisqu'un octet a sa propre adresse et qu'on ne peut pas avoir un nombre supérieur à 65535 dans un registre, il est impossible d'adresser plus de 65535 octet avec un seul registre.

Intel a résolu ce problème en créant les segments. Chaque segment est long de 65535 octets. On dit au 8086 où on veut se placer dans la mémoire en lui disant dans quel segment on se trouve et à quelle adresse on se trouve dans ce segment (la position dans un segment s'appelle l'offset) : une adresse mémoire est repérée par le couple segment:offset. Les segments sont numérotés de 0 à 65535.

Par conception, on trouve un segment tous les 16 octets.

Numéro de segment	Adresse de départ
0h	0h
1h	10h
2h	20h

Les segments se recouvrent. A l'exception des 16 premiers octets, n'importe quelle adresse dans la mémoire appartient à plus d'un segment. Par exemple l'adresse 37h peut être notée 0:37 ou 1:27 ou 2:17 ou 3:7.

Au-dessus de l'adresse 65519d chaque adresse appartient à 4096 segments.

Représentation des nombres : les nombres sont stockés en commençant par les bits de poids faible (bit de poids faible dans l'adresse la plus basse).

Ex : le nombre 2D 56 A1 8E est stocké sous la forme 8E A1 56 2D.

2.2 Les registres

Les registres des 80x86 sont de deux types : les registres de segment (sur 16 bits) et les registres d'offset (sur 16 bits pour le 80286 et sur 32 bits à partir du 80386).

Il existe également un registre des indicateurs de condition (sur 16 bits pour le 80286 et sur 32 bits à partir du 80386) ou flags.

D'autres registres comme le registre de débogage sont également présents mais il ne sont pas utilisés pour la génération de code.

Les registres d'offset sont composés de registres généraux et de registres réservés à des usages plus particuliers. Certains des registres généraux sont également utilisés par des opérations particulières.

2.2.1 Registres de segments

CS : Code Segment (segment dans lequel se trouve le code en cours d'exécution)

SS : Stack Segment (segment dans lequel se trouve la pile)

DS : Data Segment (segment dans lequel se trouvent les données initialisées du programme)

ES : Extra Segment (segment à tout faire)

FS, GS : ?

2.2.2 Registres d'offset

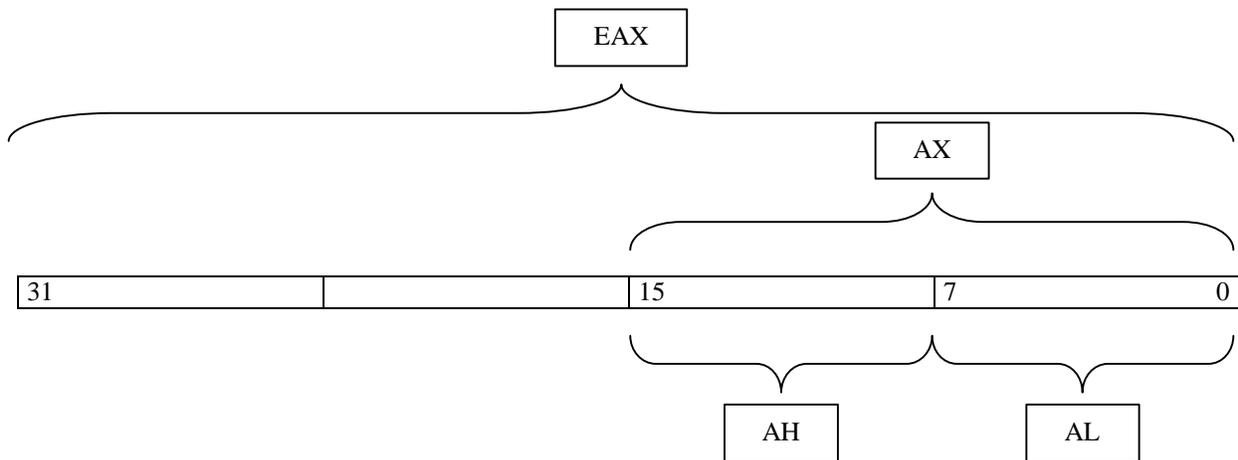
- EAX, EBX, ECX, EDX : registres généraux (ECX est utilisé comme compteur de boucle pour les opérations LOOP et REP ; EAX, EBX et EDX sont utilisés dans certaines opérations arithmétiques).
- EIP : Instruction Pointer (offset de l'instruction en cours d'exécution)
- ESP : Stack Pointer (offset de la pile)
- EBP : Base Pointer (offset de pile au début de la procédure courante)
- EDI : Destination Index (offset de destination lors de la copie de blocs d'octets)
- ESI : Source Index (offset de source lors de la copie de blocs d'octets)

Les registres ESI et EDI peuvent aussi être utilisés comme des registres généraux.

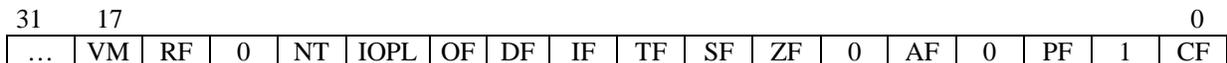
La lettre E devant chaque registre signifie Extended et indique que le registre contient 32 bits.

Les registres généraux (EAX, EBX, ECX, EDX) peuvent être décomposés en sous registres de 16 et 8 bits.

Ex : EAX se décompose en AX (qui contient les 16 bits de poids faible) lui-même se décomposant en AH (8 bits de poids fort) et AL (8 bits de poids faible).



2.2.3 Registre de flags



Les bits 18 à 32 sont réservés.

VM : Virtual Mode

RF : Resume Flag

NT : Nested Task

IOPL : Input/Output Privilege Level

OF : Overflow Flag (1 en cas de débordement arithmétique)
 DF : Direction Flag (décrémentation ou incrémentation des registres ESI et EDI lors des copies de blocs d'octets ; 1 => décrémentacion)
 IF : Interrupt Flag
 TF : Trap Flag
 SF : Sign (1 si le résultat d'une opération est négatif)
 ZF : Zero (1 si le résultat d'une opération est 0)
 AF : Auxiliary Carry
 PF : Parity
 CF : Carry (1 si l'opération produit une retenue)

2.3 Les instructions les plus courantes

D'une manière générale, pour les opérations à 2 opérandes la syntaxe est la suivante :

OP destination, source

Ex : SUB EAX, EBX => EAX ← EAX - EBX

La plupart des opérations affectent le registre de flags.

L'opérateur d'indirection est [reg]. [reg] désigne le contenu de l'adresse contenue dans le registre reg.

Ex : MOV ESI, 00045A12 => place dans ESI l'adresse 00045A12
 MOV ECX, [ESI] => place dans ECX le contenu de l'adresse 00045A12

Dans les instructions il est parfois nécessaire de préciser la taille des opérandes que l'on utilise. Pour désigner un octet on utilise le préfixe *byte ptr*, pour un mot *word ptr* et pour un double mot *dword ptr*.

Ex : MOV [ESI], byte ptr 23 => place à l'adresse contenue par ESI l'octet 23h
 MOV [ESI], word ptr 23 => place à l'adresse contenue par ESI le mot 0023h

Liste des instructions

ADD op1, op2 => op1 ← op1 + op2

AND op1, op2 => op1 ← op1 AND op2

CALL proc => (appel de procédure)

Si la procédure est déclarée FAR (i.e. elle ne se trouve pas dans le segment courant)

[ESP] ← CS

ESP ← ESP - 2

[ESP] ← EIP

ESP ← ESP - 2

CS ← segment proc

EIP ← offset proc

Si la procédure est déclarée NEAR (i.e. elle se trouve dans le segment courant)

[ESP] ← EIP

ESP ← ESP - 2

EIP ← offset proc

CMP op1, op2=> op1 - op2 et positionnement des flags en fonction du résultat

CMPSB => (CMP Strings Byte-for-byte)

compare les chaînes se trouvant aux adresses DS :ESI et ES :EDI octet par octet

[DS :ESI] - [ES :EDI]

si DF=0 alors $ESI \leftarrow ESI + 1$; $EDI \leftarrow EDI + 1$
 si DF=1 alors $ESI \leftarrow ESI - 1$; $EDI \leftarrow EDI - 1$
CMPSW => (CMP Strings Word-for-Word)
 compare les chaînes se trouvant aux adresses DS :ESI et ES :EDI mot par mot
 [DS :ESI] – [ES :EDI]
 si DF=0 alors $ESI \leftarrow ESI + 2$; $EDI \leftarrow EDI + 2$
 si DF=1 alors $ESI \leftarrow ESI - 2$; $EDI \leftarrow EDI - 2$
DEC op => $op \leftarrow op - 1$
DIV op => (division non signée) (suivant les cas EAX et EDX doivent être préalablement chargés)
 Si op est un octet
 $AX \leftarrow AX / op$
 AL ← quotient
 AH ← reste
 Si op est un mot
 $DX : AX \leftarrow DX : AX / op$
 AX ← quotient
 DX ← reste
 Si op est un double mot
 $EDX : EAX \leftarrow EDX : EAX / op$
 EAX ← quotient
 EDX ← reste
IDIV op => (division signée) cf DIV
IMUL => (multiplication signée) cf MUL
INC op => $op \leftarrow op + 1$
INT n°IT => se branche sur la procédure référencée par n°IT
 Empilage des flags
 Empilage de CS
 Chargement de CS avec le segment référencé par n°IT
 Empilage de EIP
 Chargement de EIP avec l'offset référencé par n°IT
IRET => retour d'interruption
 Dépilage de EIP → EIP
 Dépilage de CS → CS
 Dépilage des flags → flags
JMP ad => saut inconditionnel à l'adresse ad
Jxx ad => saut conditionnel à l'adresse ad (xx est la condition) (le choix se fait en testant le registre des flags qui est fonction de l'opération précédente)
 Ex : **JNZ ad**(saut si le flag ZF n'est pas positionné i.e. le résultat de l'opération précédente n'est pas égal à 0)
 JAE ad(saut si supérieur ou égal)

Les conditions

Above (CF=ZF=0), Above or Equal (CF=0), Below (CF=1), Below or Equal (CF=ZF=1 ou CF et ZF différents), Carry (CF=1), CX is Zero (CX=0), ECX is Zero (ECX=0), Equal (ZF=1), Greater (SF=OF ou ZF=0), Greater or Equal (SF=OF), Less (SF != OF), Less or Equal (SF !=OF ou ZF=1), Not Above (cf BE), Not Above or Equal (cf B), Not Below (cf AE), Not Below or Equal (cf A), No Carry (CF=0), Not Equal (ZF=0), Not Greater (cf LE), Not Greater or Equal (cf L), Not Less (cf GE), Not Less or Equal (cf G), No Overflow (OF=0), No Parity (PF=0), No Sign (SF=0), Not Zero (ZF=0), Overflow (OF=1), Parity (PF=1), Parity Even (cf P), Parity Odd (cf NP), Sign (SF=1), Zero (ZF=0).

Les conditions contenant Above ou Below concernent les nombres non signés et celles contenant Greater ou Less concernent les nombres signés.

LEA op1, op2 => (Load Effective Address)

LODSB => LOaD String Byte
AL \leftarrow byte ptr DS:[ESI]
si DF=0 alors ESI \leftarrow ESI+ 1
si DF=1 alors ESI \leftarrow ESI- 1

LODSW => LOaD String Word
AX \leftarrow word ptr DS:[ESI]
si DF=0 alors ESI \leftarrow ESI+ 2
si DF=1 alors ESI \leftarrow ESI - 2

LOOP adresse => boucle sur adresse et décrémente ECX tant que ECX !=0

LOOPE adresse => boucle sur adresse et décrémente ECX tant que ECX !=0 et ZF=1

LOOPNE adresse => boucle sur adresse et décrémente ECX tant que ECX !=0 et ZF=0

LOOPZ adresse => cf LOOPE

LOOPNZ adresse => cf LOOPNE

MOV op1, op2 => [op1] \leftarrow [op2]

MOVSb => (MOV Strings Byte-for-byte)
[ES:EDI] \leftarrow byte ptr [DS:ESI]
si DF=0 alors ESI \leftarrow ESI+ 1 ; EDI \leftarrow EDI + 1
si DF=1 alors ESI \leftarrow ESI- 1 ; EDI \leftarrow EDI - 1

MOVSW => (MOV Strings Word-for-Word)
[ES:EDI] \leftarrow word ptr [DS:ESI]
si DF=0 alors ESI \leftarrow ESI+ 2 ; EDI \leftarrow EDI + 2
si DF=1 alors ESI \leftarrow ESI- 2 ; EDI \leftarrow EDI - 2

MUL op => (multiplication non signée)
Si op est un octet
AX \leftarrow AL * op
Si op est un mot
DX:AX \leftarrow AX * op
Si op est un double mot
EDX:EAX \leftarrow EAX * op

NOP => aucun effet

NOT op => op \leftarrow NOT op

OR, XOR => cf AND

POP op => op \leftarrow [SS:ESP]

ESP \leftarrow ESP + 2

PUSH op => ESP \leftarrow ESP - 2
[SS:ESP] \leftarrow op

REP instruction => répète l'instruction de traitement de chaînes (LODxx, MOVxx, CMPxx, SCAXx) et décrémente ECX jusqu'à ce que ECX=0.

REPE instruction => (REP while Equal)

répète tant que ECX !=0 et ZF=1

REPNE instruction => (REP while Not Equal)

répète tant que ECX !=0 et ZF=0

REPZ instruction => cf REPE

REPZ instruction => cf REPNE

RET => (retour d'une procédure)

Si la procédure est déclarée FAR (i.e. elle ne se trouve pas dans le segment courant)

POP EIP

POP CS

Si la procédure est déclarée NEAR (i.e. elle se trouve dans le segment courant)

POP EIP

Si RET est suivi d'un entier n (n doit être pair)

$ESP \leftarrow ESP + n$

Charge dans op1 l'offset de op2 (op1 doit être un registre général)

SBB op1, op2 => (soustraction avec retenue) $op1 \leftarrow op1 - op2 - CF$

SCASB => (SCAN Strings Byte-for-Byte)

Cherche l'octet AL dans ES :EDI

AL – byte ptr [ES :EDI]

si DF=0 alors $EDI \leftarrow EDI + 1$

si DF=1 alors $EDI \leftarrow EDI - 1$;

SCASW => (SCAN Strings Word-for-Word)

Cherche l'octet AX dans ES :EDI

AX – word ptr [ES :EDI]

si DF=0 alors $EDI \leftarrow EDI + 2$

si DF=1 alors $EDI \leftarrow EDI - 2$

SUB op1, op2 => $op1 \leftarrow op1 - op2$

TEST op1, op2 => $op1 \text{ AND } op2$ et positionnement des flags (utilisé principalement pour tester si un opérande est nul => TEST op, op => ZF=1 si op est nul)

2.4 Appel de procédure

J'utiliserai la convention C pour l'appel de procédure:

- empilage des paramètres en ordre inverse de la spécification
- appel de la procédure
- mise à jour de la pile (dépile des paramètres) après le retour de la procédure

ex : appel d'une fonction ayant 3 paramètres de 4 octets chacun (mots doubles)

```
push param3
```

```
push param2
```

```
push param1
```

```
call proc
```

```
add esp, 0C
```

La valeur de retour éventuelle est stockée dans le registre EAX.

NB: pour les appels à l'API Windows, c'est la convention Pascal qui est appliquée, c'est-à-dire que c'est la procédure appelée qui se charge de nettoyer la pile.

3 Principes généraux

3.1 Le principe du crack

Le crack consiste à explorer un code exécutable, à localiser les protections utilisées et à modifier le code de façon à supprimer les effets de ces protections.

Ceci fait, le cracker écrit un patch qui est un exécutable qui va modifier le code du programme cible. Ce sont ces patches que l'on trouve sur le web. Les patches ne sont pas illégaux en eux-mêmes, c'est leur utilisation qui est illégale, et plus généralement toute modification d'un programme protégé par copyright.

3.2 Les outils du cracker

Le cracker a besoin de divers outils pour explorer un code étranger :

- Un désassembleur qui donne une représentation en assembleur (dead listing) du code binaire d'un exécutable et qui offre des fonctions facilitant l'exploration du code. On a donc un fichier texte.
- Un débogueur bas-niveau qui permet d'exécuter pas à pas un programme en cours d'exécution et de placer des points d'arrêt (breakpoint) sur des sections critiques du point de vue de la protection. Il permet également de modifier en temps réel le code et les paramètres du programme. C'est l'outil essentiel du cracker. La référence absolue, l'alpha et l'omega du cracker est le débogueur SoftIce de NuMega.
- Un éditeur hexadécimal qui permet de modifier en dur l'exécutable.
- Des programmes espions qui surveillent les accès disques, les accès à la base de registre, etc.
- Un cerveau

La connaissance précise de ces outils permet de trouver des astuces de programmation les rendant inefficaces : par exemple de planter le programme si SoftIce est détecté. Malheureusement la plupart de ces astuces sont connues des crackers et peuvent être déjouées. Cependant une protection capable de planter SoftIce et les désassembleurs les plus utilisés (comme Wdasm32 et IDA Pro) dispose déjà de bons atouts pour résister au cracker moyen.

3.3 Les API Windows

L'Application Programming Interface Windows est un ensemble de procédures mises à la disposition du programmeur pour lui faciliter la tâche et lui permettre d'accéder à des services système et graphiques.

Ex: MessageBox()

Le suffixe Ex dans certaines fonctions signifie Extended (nouvelle version d'une fonction déjà existante).

Le suffixe A dans certaines fonctions indique une version 32 bits de la fonction (par ex. MessageBoxA).

Par extension, j'appellerai API toute fonction faisant partie de l'API.

3.4 Les services DOS et BIOS

Les services DOS et BIOS sont l'équivalent des API Windows et sont accessibles grâce aux interruptions en passant des paramètres dans des registres précis en fonction du service appelé. Le type de service est indiqué dans le registre AH.

L'appel de ces services se fait bien sûr en assembleur.

3.5 Les principes de base des protections

La plupart des protections consistent à vérifier la validité de l'utilisation du logiciel et à réagir en conséquence. Par exemple, si le programme doit se désactiver 30 jours après son installation, celui-ci teste la date courante à chaque exécution et empêche l'exécution si le délai est dépassé.

Ceci se traduit par un code de ce type :

```
call proc_de_test          ; appel de la procédure de test
cmp  eax, valeur_correcte ; comparaison du code de retour avec la valeur correcte
jne  au_revoir            ; s'ils ne sont pas égaux, branchement sur la sortie du
                           programme
```

...

Généralement, avant la sortie réelle du programme, celui-ci affiche un message indiquant le pourquoi de la sortie (par exemple : Votre période d'évaluation a expiré). Ce type de message fait appel à des API bien connue des crackers (par ex. MessageBoxA) et peuvent être facilement breakpointées (c'est-à-dire que l'on peut indiquer au débogueur de prendre la main dès que la fonction est utilisée), ce qui permet d'accéder au cœur de la protection. Dans les chapitres concernant les différents types de protection, je listerai les différentes fonctions couramment utilisées par chaque protection pour vous permettre de localiser leurs points faibles. Une bonne protection doit éviter tant que faire se peut l'utilisation de ces fonctions.

En effet une fois que le cracker aura repéré une section de programme à éviter (comme `au_revoir`), il s'efforcera de remonter le cours du programme pour déterminer à quel endroit se fait le choix entre la section `good_guy` (tout va bien) et la section `bad_guy` (`au_revoir`) et à forcer ce choix vers `good_guy`.

Dans l'exemple précédent, il est évident que l'on ne veut pas aller à `au_revoir`. On veut donc ne pas faire de saut et continuer l'exécution normale du programme. On peut donc remplacer l'instruction `jne au_revoir` par des instructions `nop` (ne rien faire). Ainsi, quelle que soit la valeur retournée par `proc_de_test`, le programme s'exécutera normalement. Une autre solution aurait été de transformer `jne` (Jump if Not Equal) en `je` (Jump if Equal). Ainsi on ne se branche sur `au_revoir` que si `proc_de_test` renvoie un valeur correcte (ce qui arrive rarement si `proc_de_test` consiste à contrôler la validité d'un mot de passe par exemple).

Vous vous dites certainement que ce type de protection est stupide et tellement simple à craquer, mais c'est pourtant ce à quoi se résument la plupart des protections : un test et un saut.

Un autre moyen de rentrer au cœur d'une protection est de rechercher dans le dead listing le texte inscrit dans la boîte à message affichée par `au_revoir`. En effet, ce texte est stocké dans la zone data du programme et son adresse est passée en paramètre de la fonction affichant la boîte à message. Cette technique est aussi efficace qu'un breakpoint sur la fonction. Donc, si vous utilisez votre propre fonction d'affichage des messages pour éviter les breakpoint, **n'écrivez pas en clair** votre message. Codez le de façon simple (le but est juste d'empêcher le repérage dans un fichier texte), décodez le grâce à une fonction et envoyez le à votre fonction d'affichage.

La science du cracker consiste à abuser largement des faiblesses de ce type. Dans les prochains chapitres je ne décrirai pas ce type de crack (étant entendu que ce sont les premiers à tenter).

La science du « protectionniste » consiste à éviter à tout prix un accès facile à sa protection, à bien cacher ses tests, à en effectuer plusieurs à des endroits très différents, ou encore mieux : ne pas utiliser de tests.

4 Les différents types de protection

4.1 CD check

Les protections de ce type sont très fréquentes dans les jeux. Elles empêchent simplement l'utilisateur de lancer le programme si le CD original n'est pas dans le lecteur CD. Dans beaucoup de cas toute l'information est copiée sur le disque dur à l'installation, faisant de la présence du CD une simple question de copyright et non une nécessité technique. Dans d'autres cas cependant une partie de l'information reste sur le CD rendant sa présence indispensable.

Dans le premier cas, la protection consiste à vérifier la présence (et éventuellement le contenu) d'un fichier clé sur le CD, voire même pour les protections les plus basiques, à contrôler uniquement la présence d'un CD. Les techniques de crackage consistent alors à placer le fichier clé sur le disque dur et à faire croire au programme que le disque dur est un lecteur CD, ou bien à utiliser des émulateurs de CD prêts à l'emploi (comme FakeCD ou Subst).

Les API utilisées pour ce genre de tests sont classiquement :

GetDriveType => détermine le type du disque passé en paramètre (renvoie 5 pour un lecteur CD)
GetLogicalDrives => détermine la présence de disques
GetLogicalDriveStrings => renvoie les noms des disques présents
FindFirstFile => recherche un fichier dans un répertoire
GetFileAttributes => récupère les attributs du fichier spécifié
GetFileSize => renvoie la taille en octets du fichier passé en paramètre
ReadFile => lit un certain nombre d'octets à une position spécifiée dans un fichier
GetLastError => renvoie un description détaillée de la dernière erreur

Des services DOS sont également disponibles:

Int 21

service 3D : open file

service 3F : read file

4.1.1 Vérification de la présence d'un CD

Essayer d'écrire sur le lecteur CD qu'on a détecté et activer la protection si l'écriture a été réussie (donc le lecteur n'était pas un lecteur CD) permet de piéger certains émulateurs CD qui ne désactivent pas les droits d'écriture sur le disque émulé.

Une technique classique pour déterminer quel est le lecteur CD et la suivante :

- placer "C:\\" dans une variable, par exemple xrom
- tester si xrom est un lecteur CD en utilisant GetDriveTypeA
- si le résultat est différent de 5, incrémenter la lettre dans xrom et retester
- sinon utiliser xrom comme lecteur CD

Ici le crack est simple:

- placer le nom de lecteur désiré à la place de "C:\\" (cette chaîne se trouve normalement dans la zone data du programme et est facile à modifier)
- remplacer l'appel à GetDriveTypeA (call GetDriveTypeA) par mov eax, 00000005 ou bien remplacer la comparaison avec 5 par une comparaison avec 3 (disque dur).

4.1.2 Vérification de la présence d'un fichier

```
push "%s\fichier_clé" ; passage de la chaîne "%s\fichier_clé" en paramètre à
                        complétion_du_chemin
call complétion_du_chemin ; remplacement de %s par le chemin du lecteur CD
push eax ; passage du chemin complet en paramètre à
        tests_sur_le_fichier_clé
call tests_sur_le_fichier_clé ; tests quelconques
```

Cette séquence est typique d'un `sprintf` où `%s` est remplacé par une chaîne de caractères. Ici le fichier clé est clairement identifié. On peut donc le copier sur disque dans le répertoire de l'exécutable et remplacer `%s` par `.` (\Rightarrow `push ".\fichier_clé"`) ce qui indique au programme de chercher `fichier_clé` dans le répertoire courant c'est-à-dire le répertoire de l'exécutable.

4.2 Mots de passe (*serial number*)

Ici la protection consiste à autoriser l'utilisation sans restriction du logiciel à l'utilisateur si celui rentre un bon mot de passe. Celui-ci devient alors un « utilisateur enregistré » (registered user). Généralement la procédure d'enregistrement consiste à entrer son nom puis à entrer un mot de passe calculé à partir du nom. Ce mot de passe s'achète au vendeur du logiciel. Lorsque le mot de passe est entré, il est comparé au mot calculé par le programme à partir du nom. Ou bien, seul un mot de passe est entré et son traitement active les procédures d'enregistrement de l'utilisateur.

Ici le travail du cracker consiste soit à retrouver en mémoire le mot de passe calculé, soit à étudier la procédure de calcul du mot de passe et à réaliser un programme qui utilisera cette procédure pour générer des mots de passe à partir de n'importe quel nom (on parle de générateur de clé). La deuxième option ne sera pas étudiée puisque les procédures de calcul peuvent être très différentes et qu'elles n'impliquent qu'une analyse mathématique.

En général les programmeurs utilisent des API standards pour effectuer la saisie des informations : une boîte de dialogue apparaît contenant un ou plusieurs champs de saisie de texte, un bouton OK et un bouton Annuler.

La récupération du texte se fait avec `GetDlgItemTextA` ou `GetWindowTextA` et de toute façon avec `Hmemcpy` qui est une primitive noyau pour écrire en mémoire. La récupération des caractères peut se faire un par un (privilégiez cette technique, c'est très énervant pour le cracker) ou en bloc après la validation de la saisie.

La technique de crackage consiste à repérer les informations saisies dans la mémoire et à suivre les traitements qui leur sont appliqués jusqu'au test final (pour les protections les plus simples). Pour cela il faut placer un breakpoint sur une des fonctions précédentes chercher dans la mémoire la chaîne qui vient d'être saisie, placer un breakpoint sur la lecture ou l'écriture dans cette zone de mémoire et se lancer dans l'analyse de toutes les sections de code manipulant la chaîne (ce qui peut être très long et fastidieux si la chaîne a été copiée en plusieurs endroits et est traitée par de nombreuses procédures, d'ailleurs ici rien n'empêche de créer des procédures bidon juste pour énerver).

Pour que cette protection soit efficace, une bonne procédure de cryptage est nécessaire et la validation du mot de passe ne doit pas se faire sur de simples comparaisons. Il vaut mieux utiliser les résultats de traitements sur les différentes chaînes pour activer ou non les bonnes procédures, placer certaines valeurs en mémoire et les réutiliser plus tard comme condition au bon déroulement du programme. D'une manière générale, il faut lier étroitement le résultat des calculs et les fonctions vitales du programme quitte à planter le programme si une valeur n'est pas correcte. De plus il faut différer au maximum les conséquences de la saisie d'un

mauvais mot de passe voire même ne pas prévenir l'utilisateur qui a entré un mauvais mot de passe.

Nous allons maintenant étudier une section de code classique (saisie du nom puis du mot de passe):

Ici, c'est un breakpoint sur GetDlgItemTextA qui nous fait rentrer dans le code.

Les paramètres de cette API sont :

- un pointeur sur la zone de saisie du texte
- un paramètre de contrôle (sans intérêt)
- l'adresse où devra être stockée la chaîne
- le nombre maximal de caractères à lire

Valeur de retour : le nombre de caractères de la chaîne ou 0 si l'opération échoue

Rappel : les paramètres sont empilés en sens inverse

```
push 0000000F      ; empilage du nombre maximal de caractères à lire
lea  eax, [ebp-2C] ; on charge eax avec l'adresse ebp-2C
push  eax          ; empilage de l'adresse de stockage de la chaîne
push 00000404      ; empilage du paramètre de contrôle
push [ebp+08]      ; empilage du pointeur sur la zone de saisie du texte
call [USER32!GetDlgItemTextA] ; un breakpoint sur GetDlgItemTextA nous amène ici
mov  edi, eax      ; eax contient la valeur de retour (longueur de la chaîne) qu'on stocke dans edi
```

A ce niveau, si on visualise le contenu de ebp-2C, on peut voir le nom que l'on a entré.

Il faut placer un breakpoint en lecture sur la zone mémoire ebp-2C → ebp-2C+edi pour surveiller les traitements effectués sur le nom.

```
push 0000000B
lea  ecx, [ebp-18]
push  ecx
push 0000042A
push [ebp+08]
call [USER32!GetDlgItemTextA]
mov  ebx, eax
```

La même procédure est utilisée pour lire le mot de passe

```
test edi, edi      ; le programme teste si le nombre de caractères dans le nom est égal à 0
jne  00402FBF      ; s'il n'est pas égal à 0, on continue à l'adresse 00402FBF

:00402FBF cmp  ebx, 0000000A ; comparaison du nombre de caractères du mot de passe avec 10
je   00402FDE      ; si égal, saut vers 00402FDE
```

ici on sait que le mot de passe doit faire 10 caractères.

```
:00402FDE xor  esi, esi ; on met esi à 0 (xor est souvent utilisé de cette façon et est équivalent à sub esi, esi)
```

```
:00402FE0 xor  eax, eax ; idem pour eax
```

```

:00402FE2 test edi, edi ; souvenez-vous que edi contient la longueur du nom (ce test est
superflu)
:00402FE4 jle 00402FF2
:00402FE6 movsx byte ptr ecx, [ebp - 2C + eax] ; ebp-2C contient l'adresse du
nom. On place donc dans ecx le ième caractère du nom. Le breakpoint sur l'adresse du nom
nous aurait amené ici.
:00402FEB add esi, ecx ; on ajoute ecx à esi
:00402FED inc eax ; on incrémente eax pour passer à la lettre suivante
:00402FEE cmp eax, edi ; on compare eax à la longueur du nom (i.e. est-on est au dernier
caractère du nom)
:00402FF0 jl 00402FE6 ; si eax est inférieur à edi on reboucle sur 00402FE6

```

Cette section de code additionne les caractères du nom et le résultat se trouve dans esi. Nous allons maintenant voir ce qui est fait avec cette valeur.

```

:00402FF2 push 0000000A ; empilage de paramètres pour la fonction suivante. Ici on
empile 10 (ce qui n'est probablement pas sans rapport avec la longueur attendue pour le mot
de passe)
:00402FF4 lea eax, [ebp-18] ; souvenez-vous que ebp-18 contient l'adresse du mot de
passe
:00402FF7 push 00000000 ; un autre paramètre
:00402FF9 push eax ; empilage de l'adresse du mot de passe
:00402FFA call 00403870 ; mais que fait donc cette fonction ?
:00402FFF add esp, 0000000C ; mise à jour de la pile (nettoyage des paramètres)
:00403002 cmp eax, esi ; on compare la valeur de retour de la fonction avec la
somme des caractères du nom !!!
:00403004 je 00403020 ; LE TEST

```

Si on visualise eax, on constate qu'il contient la valeur numérique du mot de passe entré (il fallait entrer des chiffres). La fonction 00403870 convertissait juste le nombre codé en ASCII en valeur numérique.

Ici le crack est évident : le programme a effectué un calcul sur le nom, puis un calcul sur le mot de passe et enfin il compare les deux valeurs. Il suffit donc de transformer le je 00403020 en jmp 00403020 et quel que soit le résultat de la comparaison, on se branchera sur la bonne section. Il ne faut pas oublier que le mot de passe doit contenir 10 caractères.

Dans ce cas-là il est même facile de faire un générateur de clé puisque pour n'importe quel nom, le mot de passe est la somme de ses caractères ASCII.

Cet exemple est tiré d'un programme existant. Dans ce cas-là l'algorithme de cryptage était très simple mais même dans le cas d'algorithmes plus compliqués la protection se termine souvent sur un test. Vous avez vu à quel point il est facile de cracker cette protection, donc évitez les tests de ce type c'est-à-dire :

```

if( nom_crypté == mot_de_passe )
    good_guy() ;
else
    bad_guy() ;

```

4.3 Time check

Ce type de protection est utilisé par les versions d'évaluation de logiciels qui autorisent l'utilisation du logiciel pendant un certain nombre de jours après l'installation (en général entre 15 et 90 jours) ou même pendant un certain laps de temps après le lancement de l'application (15 minutes par exemple). Cette protection est généralement couplée à un mot de passe qui permet de désactiver la protection.

Les API couramment utilisées sont :

- GetCurrentTime : retourne le temps écoulé depuis le démarrage de Windows
- GetFileTime : retourne les dates de création, de dernière modification et de dernière lecture d'un fichier
- GetLocalTime : retourne l'heure et et la date locales
- GetSystemTime : retourne l'heure et et la date système
- GetTickCount : cf. GetCurrentTime
- GetTimeZoneInformation : renvoie les informations nécessaires pour convertir l'heure universelle en heure locale
- SetTimer : initialise une horloge qui envoie un message Windows ou appelle une fonction TimerProc associée à l'horloge à intervalles réguliers
- KillTimer : désactive un timer
- TimerProc : fonction associée à une horloge

Services DOS :

Int 21 service 2A : renvoie la date système

Les protections qui désactivent le logiciel peu de temps après son lancement utilisent la fonction SetTimer.

Les programmes qui utilisent la date d'installation doivent stocker cette date. Ils le font généralement dans la base de registre ou dans un fichier qu'ils créent dans leur répertoire. Cette date est normalement cryptée. Ils peuvent également modifier un fichier lors de leur premier lancement et contrôler la date associée à ce fichier. A chaque lancement du programme, celui-ci récupère la date courante (grâce à GetLocalTime ou une fonction de même type) et la compare à la date d'installation.

Les points d'entrée dans cette protection se situent au niveau des fonctions manipulant des dates et des fonctions de lecture de fichiers ou de base de registre.

Pour comparer la date d'installation et la date courante, ne décryptez pas la date stockée pour la comparer avec la date renvoyée par une API. Cryptez plutôt la date courante et effectuez la comparaison.

4.4 Clé électronique

Les clés électroniques ou « bouchons » sont des composants que l'on branche généralement sur le port parallèle et qui offrent des services liés à la protection (décryptage de mots de passe, time check ...). De plus, la plupart du temps le programme se désactive si la clé n'est pas branchée. Ces clés ne font que fournir des services : elles sont l'équivalent de fonctions que vous auriez pu créer. A ce titre elles ne suffisent pas à garantir la sécurité de vos applications comme peuvent le prétendre les vendeurs de ces clés. L'implémentation de ces services dans votre programme est la véritable clé de la réussite comme pour toute autre protection. De plus la communication avec le port parallèle se fait sur le port 0x378 (quasiment tout le temps) ou 0x3BC ou 0x278, donc tout appel de service peut être repéré et propulse le cracker en plein coeur de votre protection.

5 Comment se protéger

Le protectionniste doit affronter le cracker sur son terrain. Il va donc sans dire qu'une bonne connaissance des méthodes de crackage et de l'assembleur sont nécessaires pour espérer vaincre les plus expérimentés. Dans ce chapitre je ne donnerai que des conseils et des astuces susceptibles de décourager les crackers. Je ne donnerai pas de protection toute faite, n'étant moi-même pas encore capable d'en réaliser une valable.

5.1 Principes de base

- Ne jamais coder en clair les chaînes de caractères utilisées dans les messages liés à la protection.
- Si vous décidez de diffuser une version d'évaluation dont certaines fonctions sont désactivées, ne les implémentez pas, tout simplement (utilisez la compilation conditionnelle).
- Jamais de test du style

```
if ( Vérifier_mot_de_passe() == OK )
    Continuer() ;
else
    Arrêter() ;
```
- La protection doit être étroitement liée aux parties vitales du programme
- Utilisez les sections faibles de votre protection susceptibles d'être patchées (comme les sauts conditionnels) comme clé dans un calcul lié à votre protection. Il est ainsi plus difficile de les changer.
- Effectuez des checksum sur les sections critiques et si le checksum est faux (i.e. on a patché votre exécutable), surtout n'émettez pas de message du style « checksum error » : plantez directement le PC (avec la séquence d'instructions F0 0F C7 C8 pour les Pentium par exemple).
Note : effectuez un checksum sur le code en mémoire et pas sur votre fichier exécutable. En effet, une technique de crackage pour déjouer les checksums est de réaliser un lanceur pour le programme avec des droits en écriture sur la mémoire du programme et ensuite à appliquer le patch directement sur le code en mémoire.

5.2 Vérification de clé

Une bonne procédure de vérification de clé ne doit pas permettre de déchiffrer son comportement afin de créer des générateurs de clé.

Si un RCL (Rotate through Carry to the Left : décale vers la gauche les bits de l'opérande en passant par le flag de retenue. Le bit le plus significatif passe dans CF et CF va dans le bit le moins significatif) ou un RCR (Rotate through Carry to the Right) est effectué sur une valeur cela devient difficile à cracker ; en effet, on ne peut pas inverser les effet d'un RCL sans connaître la valeur de CF avant l'opération. Si CF est le résultat d'une autre opération compliquée vous venez de jouer un coup gagnant.

Placez des saut conditionnels un peu partout pour activer ou non certaines parties de votre procédure. C'est très pénible à tracer et difficile à inverser.

Utilisez des portions de votre code (de préférence les sections critiques) comme « nombre magique » pour manipuler votre code. Par exemple utilisez une portion de code pour XORer une partie de votre la clé. Cela rend impossible la modification de ces portions de code.

5.3 Désactivation des débogueurs

Pour placer un breakpoint en exécution, un débogueur utilise l'interruption 3 (INT 3 ; opcode CC) à l'adresse spécifiée. Lorsque le programme exécute INT 3, il se branche sur le débogueur qui rétablit l'instruction d'origine.

Cette interruption peut être utilisée pour détecter la présence d'un débogueur.

Ex1 : on veut détecter un breakpoint sur une fonction importée GetDlgItemTextA qui sert à récupérer les saisies clavier (et par conséquent des mots de passe). Le principe consiste à détecter la présence d'un INT 3 au début de la fonction avant de la lancer.

```
    lea    esi,GetDlgItemTextA    ; on charge dans esi l'adresse de
GetDlgItemTextA
    call   CheckIce              ; on vérifie la présence de SoftIce
    cmp    eax, 1                ; on compare la valeur de retour à 1 (SoftIce actif)
    je    Crash_system          ; s'il est actif on crash le système
    call   esi                   ; sinon on appelle la fonction
    .
    .
CheckIce:
    push  esi                    ; sauvegarde de esi
    push  ds                     ; sauvegarde de ds
    push  cs                     ; empilage de cs
    pop   ds                     ; dépilage de cs dans ds
    mov   esi,[esi+2]           ; on récupère l'adresse de l'appel réel à la fonction (lors d'un
appel à une fonction importée, on se branche en fait sur une table de jump qui saute vers les
fonctions à l'adresse où elles ont réellement été chargées au lancement du programme). Esi
contenait l'adresse de l'instruction jmp xxxxxxxx (opcode :EB xxxxxxxx) donc esi+2
contient xxxxxxxx.
    mov   eax,[esi]             ; on place dans eax les premiers octets de la fonction
    and   eax, FF               ; on récupère uniquement le premier octet
    cmp   eax, CC               ; on teste si c'est une instruction INT 3
    mov   eax, 1                ; si oui on renvoie 1
    je    Ice_Fired             ; on quitte CheckIce
    xor   eax,eax               ; sinon on renvoie 0 et on quitte
Ice_Fired:
    pop   ds                    ; on restaure ds
    pop   esi                   ; on restaure esi
    ret
```

Ex2 : on recherche un breakpoint dans une fonction interne. Chaque octet est comparé avec CC. Il faut au préalable vérifier que la section testée ne contient pas d'octet qui vaut CC.

```
    mov   al, CC                ; on charge al avec CC
    mov   ecx, longueur_fonction ; on charge ecx avec la longueur de la section à
contrôler (ecx joue le rôle de compteur pour loop)
    Call  Check_CC              ; on vérifie la section
Routine_To_Check:
    .
    .
```

```

Check_CC:
    mov esi,[esp]; on stocke l'adresse de retour, c'est-à-dire l'adresse de
Routine_To_Check
    push ds      ; sauvegarde de ds
    push cs     ; empilage de cs
    pop ds      ; dépilage de cs dans ds
Do_More:
    cmp al,[esi+ecx] ; on part de la fin de la fonction et on compare chaque octet avec
CC
    je Crash_System ; si c'est égal on crash le système
    loop Do_More ; sinon boucle (décrémentation de ecx jusqu'à ce que ecx soit nul)
    pop ds      ; cette section est exécutée uniquement si on n'a pas trouvé de CC
    ret

Crash_System:
    pop cs      ; ceci crash le système immédiatement (dépilage de ds dans cs)
    ret

```

Une autre astuce consiste à désactiver le clavier avant d'entrer dans une section critique et à le réactiver ensuite. Ainsi lorsque le débogueur prendra la main grâce à un breakpoint bien placé, le cracker ne pourra effectuer aucune commande clavier et notamment quitter le débogueur. Il sera obligé de rebooter et de repérer où la désactivation se fait grâce à un dead listing.

```

Ex:  in al, 21 ; place le registre d'état du clavier (port 21) dans al
     or al, 02 ; masque l'interruption clavier
     out 21, al ; mise à jour du registre d'état

```

Autre astuce : changer les routines appelées par les interruptions utilisées par les débogueurs c'est-à-dire INT 3 et INT 1. Lorsqu'une interruption est appelée, le processeur regarde dans la table des interruptions à quelle adresse se situe la fonction de traitement de l'interruption. L'idée ici est de changer l'adresse de la fonction de traitement de manière à exécuter notre propre fonction ou à exécuter n'importe quelle partie de la mémoire (pour planter l'ordinateur).

Ex : l'entrée dans la table pour INT 3 se situe à l'adresse 0000:000C

```

    push ds      ; sauvegarde de ds
    xor bx, bx   ; bx=0
    mov ds, bx   ; ds=0 (ds pointe sur la table d'interruption)
    not [000C] ; on modifie l'adresse de la fonction de traitement de INT 3 (contenue
dans 0000:000C) en une adresse quelconque (pourquoi pas avec un not)
    pop ds      ; restaure ds
    call Section_critique
    push ds     ; sauvegarde de ds
    xor bx, bx  ; bx=0
    mov ds, bx  ; ds=0 (ds pointe sur la table d'interruption)
    not [000C] ; on rétablit l'adresse de la fonction de traitement de INT 3
    pop ds     ; restaure ds

```

5.4 Désactivation des désassembleurs

Les désassembleurs fournissent des capacités d'analyse de code en référencant tous les sauts et les appels de fonctions. Ainsi pour chaque section de code on sait qui l'a appelée. Les chaînes de caractères de la zone données et les fonctions importées sont également référencées dans le code. On peut donc savoir où la chaîne « Félicitations vous êtes maintenant un utilisateur enregistré » est utilisée et où est utilisée la fonction `GetDlgItemTextA`.

Pour contrer le crackage à partir du code désassemblé (dead listing), on peut utiliser des appels indirects pour les fonctions.

Ex : `call [Table_des_Call +8]` où `Table_des_Call` est une table de jump vers des fonctions

On peut aussi utiliser `call [Table_des_Call + eax*4]` pour appeler une fonction en fonction de la valeur de `eax`

Ceci évite une référence directe à l'adresse de la fonction et annule donc certaines des facilités que fournissent les désassembleurs pour suivre la succession des appels.

Cette table peut également figurer dans la zone des données et être cryptée ou bien être chargée dynamiquement depuis un fichier pour éviter toute possibilité de déterminer quelle fonction est appelée.

On peut aussi crypter du code dans la zone donnée, puis le décrypter et le charger dans la zone de code pendant le déroulement du programme.

Il existe aussi des séquences d'instruction qui peuvent dérouter les désassembleurs.

Ex :

- les boucles infinies : créez une section de code mort où vous placez l'instruction

```
0000 : EB FE jmp 0000
```

- le décalage de code

```
0000 : EB FF jmp 0001
```

```
0002 : XX YY
```

ici vous sautez sur `FF` qui doit être le début d'une instruction

le code désassemblé sera `XX YY` alors que le code exécuté sera `FF XX YY`

- pour effectuer un jump, utilisez plutôt :

```
push adr_saut
```

```
ret
```

Petit aperçu de code désassemblé :

```
:0020.5712 7403 je 5717
:0020.5714 E9A900 jmp 57C0
```

* Referenced by a (U)nconditional or (C)onditional Jump at Address:

```
|:0020.5712(C)
```

```
|
```

* Possible Reference to String Resource ID=16917: "Swap file directory '%s' doesn't exist. Do you want to creat"

```
|
:0020.5717 B81400 mov ax, 4251
:0020.571A 50 push ax
:0020.571B 8D46EA lea ax, [bp-16]
```

```

:0020.571E 50          push ax
:0020.571F B87601     mov ax, 0176
:0020.5722 50          push ax
:0020.5723 9AFF0000   call 0006.0402h
:0020.5728 83C406     add sp, 0006
:0020.572B 9AFF0000   call Ctl3dV2.CTL3DGETVER
:0020.5730 3D2002     cmp ax, 0220
:0020.5733 7222       jb 5757
:0020.5735 16         push ss
:0020.5736 8D46EA     lea ax, [bp-16]
:0020.5739 50          push ax
:0020.573A 9AFF0000   call KERNEL.GETMODULEHANDLE
:0020.573F 8946FE     mov [bp-02], ax
:0020.5742 0BC0       or ax, ax

```

Dans cet exemple on peut voir que l'on accède à l'adresse 0020.5717 par un saut effectué en 0020.5712, ensuite on voit que la chaîne « swap file ... » est placée dans ax et empilée comme paramètre à la fonction située à l'adresse 0006.0402, et enfin on voit deux appels aux fonctions externes CTL3DGETVER et GETMODULEHANDLE des bibliothèques Ctl3dV2 et KERNEL respectivement.

6 Conclusion

Les mises en œuvre des protections que j'ai détaillées sont bien sûr basiques. La plupart des protections industrielles utilisent des méthodes plus élaborées qui mettent en application les principes de protection que j'ai évoqués. Cependant l'idée reste la même.

La réalisation de bonnes protections nécessite beaucoup de travail et d'astuce. Elle nécessite également une bonne connaissance des techniques de crackage. Il serait stupide de croire que l'on peut se passer de cette connaissance. J'invite donc vivement ceux qui sont concernés par les protections à apprendre à cracker. De nombreux sites et tutoriels sont consacrés à ce sujet.

Pour ceux que cet apprentissage rebute (l'assembleur provoque des allergies chez certains), il existe des protections commerciales qui se greffent sur le programme fini ou qui offrent des bibliothèques de fonctions liées à la protection (comme Timelock ou RSAgent ou VBox). Cependant sachez qu'elles constituent des cibles de choix pour les crackers et qu'elles sont très rapidement déjouées (quoi que puissent dire les publicités, ne sous-estimez pas les capacités des crackers). En résumé, seul un cracker expérimenté peut devenir un bon protectionniste ...

Un dernier conseil pour les programmeurs de sharewares : si vous diffusez votre logiciel, c'est pour que les gens l'utilisent puis l'achètent s'ils en sont content, donc évitez les méthodes de protection agressives comme les nag screens qui énervent. Contentez-vous plutôt de désactiver certaines fonctions (c'est-à-dire diffusez une version light de votre produit) comme les sauvegardes : c'est increvable et si l'utilisateur prend du plaisir à utiliser votre logiciel, il sera plus enclin à payer pour avoir la version complète.

BIBLIOGRAPHIE

📖 Tutoriels de crackage : la magnifique collection de la Higher Cracking University sur le site <http://www.fravia.org>

Un must. De nombreux autres sujets y sont aussi traités.

📖 Tutorial d'assembleur : <http://www.programmersheaven.com>

Très pédagogique, mais en anglais.

📖 Aide-mémoire de l'assembleur :Eduard Pandeles, éd. Marabout 1996

ISBN 2-501-02607-1

Pas cher, précis mais manquent les instructions Pentium et les opcodes

📖 Opcodes Intel : fichier OpCode.pdf quelque part sur le web

📖 Liste des interruptions : intlst55.zip

📖 Les 1000 fonctions de programmation de Windows 95 TM : Bernard Frala, éd. Marabout 1997

ISBN 2-501-02797-3

Bien fait mais quelques fonctions manquent. Pour des les compléments, se référer aux docs en ligne.

📖 Documentation en ligne de l'API Windows que vous pouvez trouver avec tous les environnements de développement (Win32.hlp et Owl50.hlp pour Borland C++ 5.0).

📖 La doc de SoftIce3.2 : si30cr.pdf (Command Reference) et si30ug.pdf (User Guide)