Techniques de Analyse

Ce qu'en dit la loi...

Le *reverse engineering*, ou décompilation, est autorisé par la loi selon certains critères. Le plus connu est bien sûr la décompilation à des fins d'interopérabilité ou, en d'autres termes, la décompilation justifiée par le besoin de faire fonctionner un programme dans un environnement donné et avec d'autres programmes.

Le lecteur soucieux de respecter la loi trouvera toutes les informations utiles et avérées dans la Directive 91/250/CEE du 14 mai 1991, concernant la protection juridique des programmes d'ordinateur. Cette directive a fait l'objet d'une parution au Journal Officiel n° L 122 du 17/05/1991 p. 0042 - 0046.

Egalement disponible sur le Web à l'adresse : **http://europa.eu.int/smartapi/cgi/sga_doc?smartapi!celexapi!prod!ELEXnumdoc&lg=FR&numdoc=31991L0250&model=guichett** ou en faisant une recherche par mot clef ("décompilation") sur **http://europa.eu.int/eur-lex/fr/search/search_lif.html**

Dans le cadre de l'espionnage sous toutes ses formes, le reverse engineering s'avère être une technique très efficace d'analyse de fichiers, dont le fonctionnement est gardé secret par l'éditeur. Certains programmes espions (en premier les fameux spywares) pourraient bien être un jour protégés contre les analyses par désassemblage et l'utilisation de débogueurs. C'est pourquoi, dans cet article, nous présentons quelques techniques anti-reverse engineering, et comment les contourner.

RÉCUPÉRATION D'INFORMATIONS

L'exécutable que nous allons analyser [7] est un challenge lancé par un groupe de reverse engineers. Ce genre de challenges, souvent appelés "Crackmes", sont lancés par des membres, afin de tester les connaissances d'autres personnes, et de pouvoir les recruter par la suite. J'ai choisi cet exécutable pour sa quantité de "pièges" et de techniques d'anti-debugging.

Pour effectuer cette analyse, je me suis servi d'IDA Pro [1], de Compuware Soft ICE [2], d'Icedump [3] (plugin pour SoftICE), de LordPE [4] et d'un simple éditeur hexadécimal. Une bonne connaissance de l'assembleur x86 et de l'architecture des processeurs Intel est recommandée pour la compréhension de cet article. Je vous invite à lire les documentations citées en références.

Pour récupérer des informations pertinentes, les caractéristiques des sections et l'entry point, nous employons un éditeur de fichiers PE [4] (voir [0] pour plus d'informations).

```
Nb Sections: 6
Entry Point RVA: 6000
Image Base: 400000

CODE   Vsize:1000  RVA:1000  Psize:1000  offset:600   Flags:60000020
DATA   Vsize:1000  RVA:2000  Psize:600   offset:1600  Flags:C0000040
.idata Vsize:1000  RVA:3000  Psize:200   offset:1C00  Flags:C0000040
.reloc Vsize:1000  RVA:4000  Psize:200   offset:1E00  Flags:50000040
.rsrc  Vsize:1000  RVA:5000  Psize:A00   offset:2000  Flags:50000040
Yado   Vsize:2000  RVA:6000  Psize:2000  offset:3000  Flags:E0000080
```

L'*entry point* (début du programme) correspond à l'adresse relative (*RVA* ou *Relative Virtual Address*) de la dernière section du programme (Yado). Aucun compilateur ne place l'entrée d'un programme dans la dernière section, ce qui signifie que le programme a été modifié soit par un *PE crypteur*, soit manuellement. Les caractéristiques E0000080 signifient `section writeable`, `readable executable`. Généralement cela indique que la première section est chiffrée et que le programme commence par en déchiffrer le code avant de l'exécuter (pour plus d'informations, voir [0]).

Reverse Engineering 2

d'un exécutable "verrouillé"

DÉBUT DE L'ANALYSE

Nous lançons l'application et nous obtenons comme message "*Can Not Start Application*". Cela provient probablement d'une détection du débogueur, actuellement en mémoire. A l'aide d'Icedump, voyons si nous pouvons lancer l'application. Icedump est un plugin pour SoftICE pour en dissimuler la présence et ajouter des commandes (tracer, dumper, etc.). Je relance l'application et j'obtiens : *"Your Trial is Expired"* (période d'essai expirée). Examinons le contenu du fichier texte fourni avec le programme.

```
"Welcome !
This crackme is a trial simulator.
You must patch the exe to make the original Fake Trial application run
again.
You will finish this crackme only when you have done the patched exe
that runs
the trial program again.Patching Krypton2 only in memory is not a valid
solution
of the crackme, only the patched exe it is. This is what you must do.
There are no limitations.
You can use :  disassembler , debugger , dumper , editor ecc.. ecc...
Naturally Krypton2 uses some kind of protections that you must defeat.
Good Work reverser :)"
```

Le message d'erreur est une MessageBox classique. Je place donc un point d'arrêt sur l'API MessageBoxA afin de comprendre ce qui se passe (sous SoftICE `bpx MessageBoxA`). Je relance le programme et l'ordinateur bloque. Je suis contraint de redémarrer. Il est évident que le programme contient une détection des points d'arrêt (BPX). Je décide de m'appuyer sur les BPM afin de contourner ce probleme.

La commande BPM, *Breakpoint on memory access*, aussi connue sous le nom de `watchpoint` utilise les Debug Registers du processeur pour le débogage. Contrairement à la commande BPX, elle ne modifie pas le programme débogué, ce qui permet de passer outre les checksums et détections de points d'arrêts. (Détection qui consiste à rechercher l'octet 0xCC dans le code, opcode de l'interruption 3). Je reviendrai sur ces points par la suite.

Sous SoftICE, "bpm MessageBoxA x". Je relance l'application et l'ordinateur se bloque une fois de plus forçant à rebooter. Changeons de méthode.

Placer un point d'arrêt à l'*entry point* du programme protégé permet de tracer pas à pas ce qui se passe. En effet, la détection des points d'arrêts ne pouvant se faire au début du programme (puisque aucune instruction n'a encore été exécutée), aucun problème de détection ne se pose.

LordPE permet de placer une interruption 3h (point d'arrêt CPU atteint) sur la première instruction exécutée par le programme grâce à la commande "`Break'n Enter`". Sous SoftICE, tapons "`bpint 3`" pour s'arrêter sur l'interruption 3h placée à l'entry point. Je presse OK quand LordPE me demande si je suis prêt, je confirme et SoftICE réapparaît :

```
00405FFE  ADD     [EAX],AL
00406000  INT     3               <=== On break ici
00406001  CMP     BL,BH
00406003  IMUL    ECX,[ESI+58],F67459DF
0040600A  JMP     0040600D
```

LordPE nous indique que l'ancien octet est "EB".

```
ÉÍ[LordPE]ÍÍÍÍÍÍÍÍÍÍÍÍÍ»
º Type "eb eip EB" !   º
ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼
```

L'ancien octet est l'octet remplacé par LordPE lors de l'insertion de l'interruption 3 dans le programme.

L'OBFUSCATION

Sous SoftICE nous tapons "`eb eip EB`" pour rétablir le code d'origine. Le code est étrange, je présume que nous sommes en présence d'un code "polymorphe", une obfuscation utilisée pour empêcher les analyses par désassemblage, et rendre le débogage plus long. Je commence à tracer avec F10 (step over) et ma machine se bloque sur le premier `call` rencontré. Redémarrons ! Voilà à quoi ressemble le code d'obfuscation :

```
00406000  JMP     0040603C <== Un jmp pour passer le "code poubelle".
00406002  FILD    QWORD PTR [ECX+4E]
00406005  POP     EAX
00406006  FISTP   WORD PTR [ECX+74]
00406009  IMUL    BL
0040600B  ADD     EDI,EBX
0040600D  JNZ     00406000
0040600F  FISTP   WORD PTR [ECX-64]
00406012  ADD     ECX,-19
```

```
00406015  POPFD
00406016  JMP       ECX
00406018  JMP       00406027
0040601A  PUSH      ECX
0040601B  CALL      00406010
00406020  JMP       0040600D
00406022  JMP       0040605E
00406024  FILD      QWORD PTR [ECX+4E]
00406027  POP       EAX
00406028  FISTP     WORD PTR [ECX+74]
0040602B  IMUL      BL
0040602D  ADD       EDI,EBX
0040602F  JNZ       00406022
00406031  FISTP     WORD PTR [ECX-64]
00406034  ADD       ECX,-19
00406037  POPFD
00406038  JMP       ECX
0040603A  JMP       00406049

 ==> 0040603C  PUSH      ECX                    on arrive ici.
0040603D  CALL      00406032
```

En traçant, nous arrivons au `PUSH ECX`, puis ensuite le `call` (appel). Si nous passons le `call` en *step over* (F10), la machine se bloque. On n'a pas le choix, il faut rentrer dans la fonction pour comprendre. On utilise la fonction *step into* (touche F8). L'adresse que le `call` appelle ne correspond pas avec les adresses des instructions avant le `call` (`406031` et `406034`). Pourquoi ?

SoftICE a désassemblé les instructions une premiere fois, mais le programme appelle des adresses comprises à l'intérieur d'instructions déjà existantes, ce qui a pour effet d'assembler de nouvelles instructions pendant que nous tracons le programme. Voici le nouveau code assemblé :

```
0040602C  JMP       0040602F  <=== il n'y avait pas de 40602C tout à
l'heure. (40602B et 40602D)
0040602E  FBSTP     TBYTE PTR [EBP-0F]
00406031  FISTP     WORD PTR [ECX-64]
```

Il n'y a pas de `40602F` pour le moment, seulement des instructions commencant en `40602E` et `406031`. Pourtant le programme appelle une instruction comprise entre ces adresses, formant ceci :

```
00406022  JMP       0040605E
00406024  FILD      QWORD PTR [ECX+4E]
00406027  POP       EAX
00406028  FISTP     WORD PTR [ECX+74]
0040602B  IMUL      BL
0040602D  ADD       EDI,EBX
0040602F  JNZ       00406022  <== *on arrive ici.*
```

> **Note**
>
> Il ne faut pas se fier aux adresses puisque le code du programme s'assemble sous nos yeux, en appellant des adresses en milieu d'instructions. Celles-ci changent au fur et à mesure que l'on avance dans le code.

Remarquons la présence de l'adresse `40602Fh` ! Elle n'existait pas jusqu'à ce que le code se modifie. Pour l'instant, il nous est difficile de différencier les instructions factices des instructions importantes.

Revenons maintenant sur ce qui se passe dans le `call` qui suit le `push ecx` pour vous expliquer l'obfuscation.

Nous arrivons sur un PUSHF qui, une fois exécuté, change de forme. Ensuite ceci :

```
00406012  ADD       ECX,-19
00406015  POPFD
00406016  JMP       ECX
```

La valeur dans `ECX` est modifiée et sert de pointeur vers le code à exécuter (`JMP ECX`). C'est ce mécanisme qui permet de calculer l'adresse de la prochaine instruction à exécuter. Le code est rempli de cette routine et nous ne savons pas où nous allons atterrir. Le code n'est pas lisible du tout, chaque instruction est mélangée avec le code parasite. A partir de maintenant, je ne donnerai que le code important pour la compréhension de la protection, mais il faut savoir que le code est quant à lui toujours "illisible" et il faut le tracer avec une très grande attention pour déterminer quelles sont les instructions pertinentes. Le désassemblage est possible mais le code n'est pas très pratique à lire. N'étant pas vraiment linéaire, nous ne connaissons pas exactement l'ordre des instructions.

Cette obfuscation comprend en plus des sauts de plusieurs blocs d'instructions non valides jusqu'à ce que le code se modifie et les rende exécutables. Commençons à tracer et notons les instructions importantes.

HOOK D'INTERRUPTIONS

```
00406066  SIDT      FWORD PTR [004073D3] ; Accès a l'IDT
004060D3  MOV       EBX,[004073D5]       ; ebx = IDT base addresse
004060FB  ADD       EBX,08               ; ebx = selector pour l'int.
00406120  CLI                            ; clear interrupts.
00406169  SHL       EDX,10
0040618E  MOV       DX,[EBX]
004061B3  MOV       [004073CB],EDX
004061FD  MOV       EAX,00406BBA         ; nouvel int 1 handler.
00406224  MOV       [EBX],AX             ; on crash ici si on trace
00406249  SHR       EAX,10
```

Nous sommes en présence d'un détournement d'interruption. Il est important de noter que ce bout de code fonctionne seulement sous Windows 9x (95, 98 et ME). Ces versions de Windows autorisent l'accés en écriture à l'IDT (*Interrupt Descriptor Table,* voir ci-après) en ring 3 (user level), ce qui nous permettra de passer en ring 0 (kernel level) sans avoir besoin d'écrire de pilote !

Quel est l'intérêt de détourner une interruption ? Principalement exécuter du code en ring 0, et par conséquent lire tous les registres du processeur, même ceux réservés tels que les *Debug Registers* par exemple. Nous verrons plus tard à quoi cela peut servir dans la lutte contre le reverse engineering. Le code exécuté en ring 0 ne pourra pas être débogué par un débogueur ring 3 tel que `Ollydbg`,

le nouveau débogueur d'IDA ... qui ne peut déboguer que des applications fonctionnant en ring 3. Pour finir avec l'utilité d'un détournement d'interruption, cela permet aussi d'embrouiller la personne qui tente d'analyser le fichier si elle n'est pas familière avec ces techniques. Continuons donc.

Les 3 lignes importantes ici sont :

```
00406066  SIDT      FWORD PTR [004073D3]    ; Accès à l'IDT
```

Le programme accède à l'IDT. L'IDT est un tableau de descripteurs. Chaque descripteur fait 8 octets et représente une interruption. L'IDT est la table des interruptions pour le mode protégé (en mode réel, il s'agit de l'IVT (*Interrupt Vector Table*); pour plus d'informations sur l'IDT voir **[5]**).

```
0187:004060FB  ADD     EBX,08     ; ebx = sélecteur pour l'interruption.
```

Cette ligne ajoute 8 octets à l'adresse de base de l'IDT. Etant donné que chaque descripteur d'interruption fait 8 octets, le programme travaille sur l'interruption 1. (1 * 8 = 8).

```
0187:004061FD  MOV       EAX,00406BBA     ; nouvel int 1 handler.
```

Le but d'un détournement est d'appeler une routine de notre choix quand l'interruption détournée est exécutée. L'adresse qui est mise dans EAX (406BBA) est le nouveau *interrupt handler*. En clair, à partir de maintenant, à chaque fois qu'on rencontrera une interruption 1, le code situé à l'adresse 406BBA sera exécuté.

Le programme génère une erreur si nous essayons de tracer l'instruction suivante :

```
0187:00406224  MOV       [EBX],AX
```

On ne peut donc pas continuer à tracer l'exécution du programme. Étant donné qu'une interruption vient d'être détournée, il est fort probable que celle-ci soit appelée par la suite pour exécuter du code en ring 0. Nous allons donc émuler cette interruption 1, c'est-à-dire nous rendre à l'adresse de son handler. EAX contient l'adresse de l'*interrupt handler* (le gestionnaire associé à l'interruption), il nous suffit donc de nous y rendre.

Attention : N'oublions pas que le programme détecte les interruptions 3 (BPX) et d'autres protections. Nous serions donc tentés d'utiliser la commande G sous SoftICE. (G Adresse = permet de se rendre à l'adresse passée en paramètre, soit dans notre cas G eax). Cependant, avec cette commande, la machine se bloque, nous obligeant à redémarrer une fois de plus.

Comment faire alors ? Le processeur contient une série de registres très intéressants, et dans notre cas, celui qui nous intéresse le plus est le registre EIP (*Extended Intruction Pointer*) : il contient l'adresse de la prochaine instruction à exécuter.

Exemple:

```
00401000    push ebp          eip contient 401000
00401001    mov ebp,esp       eip contient 401001
00401003    xor eax,eax       eip contient 401003
```

La commande `R` (`R destination source`) sous SoftICE place une valeur dans un registre par exemple. Dans notre cas, il suffit de faire : `R EIP EAX`. Cela a pour effet de copier le contenu de EAX dans EIP. La prochaine instruction à exécuter est maintenant celle de l'interrupt handler. Afin d'avoir un rapide coup d'œil sur cette routine, nous utilisons la commande `U` (`U adresse l longueur`, la commande s'écrit aussi avec une adresse ou un registre en paramètre) et `U EIP` nous donne :

```
00406BBA  EB3C              JMP      00406BF8      <== ici.
                                    ---- code "poubelle" ----
00406BF8  51      PUSH      ECX
00406BF9  E8F0FFFFFF        CALL     00406BEE
```

Toujours notre obfuscateur, donc je passe ce code qui ne sert qu'à nous embrouiller. On trace un peu et nous arrivons sur une boucle infinie (`jmp eip`).

```
00406BDC  EBFE              JMP      00406BDC
```

Quel est l'intérêt d'une boucle infinie ? Revenons un peu dans le contexte du programme. Nous étions en présence d'un détournement d'interruption (interruption 1 pour être précis, utilisée par les débogueurs). Lors d'un détournement d'interruption, le nouvel *interrupt handler* est exécuté en ring 0, c'est-à-dire au plus haut niveau de privilèges système. Que se passe-t-il si on exécute une boucle infinie dans ce mode ? La machine boucle sur elle-même et ne répond plus ! En ring 3, il suffit simplement de "tuer" le processus et le problème est résolu. En revanche, en ring 0, on ne peut rien faire à part redémarrer. Vous vous souvenez des blocages précédents ? En voilà la raison !

Pour résumer

Si une interruption 1 est rencontrée (débogueur), la machine est bloquée. Étant donné que nous sommes sous SoftICE, la boucle ne nous dérange pas. Nous sommes en plus en train d'émuler l'interruption handler, nous ne sommes pas en ring 0. Nous pourrions remplacer la boucle infinie par des instructions NOP (*No Operation*), mais je préfère ne pas modifier le code de la protection. Nous allons la passer sans toucher au code. Le registre EIP s'avère très utile une fois de plus. Tapons `R eip eip+2` et nous voilà sur la prochaine instruction après la boucle. (`JMP EIP` est codé sur 2 octets, d'où le `eip+2` pour passer à l'instruction d'après). On continue à tracer avec F8 (et pas F10 si on ne veut pas redémarrer, voir ci-après pour l'explication).

Après quelques instructions on retrouve une boucle infinie :

```
00406C66  EBFE              JMP      00406C66
```

Tapons `R eip eip+2` pour contourner la boucle. On reprends le tracing avec F8 et peu de temps après nous tombons sur un passage très intéressant :

```
Contrôle des DEBUG REGISTERS:

00406D78  0F21C0            MOV      EAX,DR0
00406D9D  85C0              TEST     EAX,EAX
00406DC1  0F850F050000      JNZ      004072D6   (NO JUMP)
```

Ce code qui semble banal est en fait particulièrement intéressant. N'oublions pas que nous sommes normalement en ring 0 et donc que nous pouvons accéder à tous les registres du système, même ceux qui ne sont normalement pas accessibles en user level, comme les *Debug Registers*.

DR0 est un *Debug Register*. Il en existe 8, de DR0 à DR7. Ils n'ont pas tous la même fonction. Les 4 premiers sont utilisés par SoftICE (entre autres) quand nous appelons la commande BPM pour placer un point d'arrêt "matériel" sur une adresse mémoire (aussi connu sous le nom de *watchpoint* et à ne pas confondre avec les BPX qui eux insèrent une interruption 3h).

Ceux qui sont déjà familiers avec SoftICE se sont peut-être demandé pourquoi ils ne pouvaient utiliser que 4 BPM à la fois. Tout simplement car nous sommes limités par l'architecture du processeur.

DR4 et DR5 sont réservés. Les deux derniers registres DR6 et DR7, quant à eux, servent à stocker des informations sur le type de watchpoint, c'est-à-dire les accès en lecture, écriture ou exécution (explication simplifiée).

Pour vérifier ceci, tapons la commande `CPU` sous SoftICE :

```
DR0   00000000  DR1   00000000
DR2   00000000  DR3   00000000
DR6   FFFF0FF0  DR7   00000400
```

Voir la documentation **[5]** pour plus d'informations sur les *Debug Registers*. Revenons maintenant à l'analyse du code :

```
00406D78  0F21C0            MOV       EAX,DR0
```

Elle déplace le contenu du registre DR0 dans le registre EAX. Les seules instructions autorisées sur les *Debug Registers* sont des `MOV` (ex : `mov eax, dr2`).

```
00406D9D  85C0              TEST      EAX,EAX
00406DC1  0F850F050000      JNZ       004072D6   (NO JUMP)
```

L'instruction `TEST` est en fait un "ET LOGIQUE" (`TEST EAX, EAX` signifie `EAX AND EAX`). Si le résultat de cette opération est 0, alors le *zero flag* est positionné à 1. `JNZ` est un saut conditionnel qui dépend du *zero flag*. Si celui-ci n'est pas positionné à 1, alors le saut est exécuté, direction l'adresse 4072D6h.

Dans notre exécutable, si le registre DR0 est à zéro, alors le résultat du `TEST` positionne le *zero flag* à 1 et le saut n'est pas pris. Au contraire, si le registre DR0 est différent de 0, alors le résultat du `TEST` positionne le *zero flag* à 0, le saut conditionnel est exécuté (`JNZ` signifie : *jmp if not zero*, donc si le résultat du `TEST` est différent de zéro, il saute. C'est une facon plus simple de se rappeller du fonctionnement de ce saut conditionnel).

Nous avons vu plus haut que les *Debug Registers* sont utilisés lors de la pose de BPM. Vous l'avez compris, si une personne utilise la commande BPM, les *Debug Registers* sont différents de zéro. Si c'est le cas, le programme nous branche vers la routine en 4072D6h.

```
U 4072D6

0187:004072D6  EBFE            JMP       004072D6
```

Nous avons encore un `jmp eip` en ring 0 qui bloque complètement la machine. Donc, pour résumer, si un BPM est utilisé (registre DR0 différent de 0), notre machine se bloque. Pour l'instant, nous n'avons utilisé aucun BPM, donc le registre est toujours à zéro. On continue de tracer avec F8 et on arrive ici :

```
00406DE9  0F21C8            MOV       EAX,DR1
00406E0E  85C0              TEST      EAX,EAX
00406E32  0F859E040000      JNZ       004072D6
```

Les *Debug Registers* de 0 à 3 sont testés un par un. Si un seul de ces registres est différent de zéro, alors le programme appelle une routine qui bouclera sur elle-même et bloquera le système. Toutes ces instructions sont noyées dans le code obfuscateur.

Pour permettre l'utilisation de BPM, il suffirait de remplacer les 2 octets des `JNZ` par des NOP ou alors, afin d'éviter de modifier le code (en cas de présence de contrôle d'intégrité de type CRC ou Checksum), nous pourrions effectuer la commande `R FL Z` sur chaque `JNZ` s'apprêtant à nous bloquer. La commande `R FL` signifie *Reverse Flag* (inversion d'un flag), et le `Z` indique que nous voulons inverser le *zero flag*.

Un peu plus loin, nous rencontrons la série d'instructions suivantes :

```
00406F5E  B800104000        MOV       EAX,00401000
:what eax
The value 401000 is (a) KRYPTON2!CODE
```

La commande `WHAT` est très utile. Elle nous indique que la valeur qui a été mise dans `EAX` représente la section `CODE` du programme que nous sommes en train de tracer.

```
00406FA7  B900204000        MOV       ECX,00402000
:what ecx
The value 402000 is (a) KRYPTON2!DATA
```

Cette fois-ci, l'adresse est celle de la section `DATA`.

```
00406FF0  BB00200101        MOV       EBX,01012000
:what ebx
The value (1012000) was not identified as any known type
```

En revanche cette valeur ne semble correspondre à rien de défini pour l'instant. Nous continuons notre exploration :

```
00407007D  3118             XOR       [EAX],EBX
```

Intéressant :

Un OU EXCLUSIF (`XOR`) entre [`EAX`] (pointeur) et `EBX`. Plus exactement entre les 4 octets pointés par `EAX`, c'est-à-dire à l'adresse 401000h, et `EBX` qui contient 1012000h. Le OU exclusif est généralement utilisé pour remettre à zéro un registre en assembleur (ex: `XOR EAX, EAX`, et après cette instruction `EAX = 0`). Autrement, le `XOR` sert à effectuer un chiffrement primitif par masquage constant. Ici les données chiffrées sont à l'adresse 401000 (soit la première section du programme qui est censée contenir du code exécutable en temps normal) et la clef de déchiffrement est dans `EBX` (valeur constante 1012000h). Utilisons F8 pour exécuter cette instruction et un message d'erreur apparaît. C'est une GPF (*General Page Fault* ou défaillance de page).

LE DÉCHIFFREMENT

Essayons de comprendre pourquoi nous avons eu cette erreur. L'instruction essaie d'écrire dans la section CODE pour déchiffrer le code qu'elle contient. Or, par défaut, les caractéristiques de la section CODE sont *readable* et *executable*, c'est-à-dire qu'il n'est pas possible d'y écrire. Bien sûr, le programme n'a normalement pas besoin d'avoir le droit d'écriture puisque ce déchiffrement est normalement effectué en ring 0.

Je relance l'application, et une fois sous SoftICE, j'utilise la commande MAP32 (pour afficher le détail des sections, leur adresse en mémoire et les droits d'accès).

```
:map32 krypton2
Owner     Obj Name  Obj#  Address        Size      Type
KRYPTON2  CODE      0001  0187:00401000  00001000  CODE  RO <= !!
KRYPTON2  DATA      0002  018F:00402000  00001000  IDATA RW
KRYPTON2  .idata    0003  018F:00403000  00001000  IDATA RW
KRYPTON2  .reloc    0004  018F:00404000  00001000  IDATA RO SHARED
KRYPTON2  .rsrc     0005  018F:00405000  00001000  IDATA RO SHARED
KRYPTON2  Yado      0006  018F:00406000  00002000  UDATA RW
```

La première section a pour droits d'accès RO (*Read Only*, lecture seule). Nous avions bien raison. Changeons les caractéristiques de la section code en RW. Pour cela, nous utilisons l'éditeur PE intégré à LordPE (note : nous avons vu dans le code que la section DATA est aussi utilisée ; il y a fort à parier qu'elle va être déchiffrée elle aussi. Cependant, il n'est pas nécessaire de modifier les propriétés de cette section car celle-ci est déjà en lecture - écriture (RW)).

Jetons un petit coup d'œil à la section .rsrc (ressources) et on s'aperçoit qu'elle est aussi chiffrée (aucune chaîne n'apparaît). Passons aussi cette section en écriture : clic droit sur l'exécutable, "Load in PE Editor (LordPE)". Nous cliquons sur "Sections" pour accéder au tableau des sections. Clic droit sur la première section et "List section header table".

```
1. item:
 Name:                 CODE
 VirtualSize:          0x00001000
 VirtualAddress:       0x00001000
 SizeOfRawData:        0x00001000
 PointerToRawData:     0x00000600
 PointerToRelocations: 0x00000000
 PointerToLinenumbers: 0x00000000
 NumberOfRelocations:  0x0000
 NumberOfLinenumbers:  0x0000
 Characteristics:      0x60000020
 (CODE, EXECUTE, READ)
```

Nous voyons clairement les caractéristiques de la section CODE : 60000020h correspond à EXECUTE - READ. On ferme la fenêtre, clic droit à nouveau et cette fois-ci on choisit l'option : "Edit section header". Cette fonction permet de modifier les caractéristiques d'une section et en particulier les flags de la section.

Nous voyons un champ "Flags" avec un bouton juste à côté. Il permet de calculer les droits des sections rapidement à l'aide de cases à cocher, et justement celle qui nous intéresse (*Writeable*) n'est pas encore cochée. Il suffit de la sélectionner et le programme nous indique la nouvelle valeur de la section : "E0000020". Faisons de même avec la section .rsrc. Nous pouvons fermer LordPE. Il ne faut pas oublier de presser "Save" pour enregistrer les changements. Relançons notre application pour nous retrouver au XOR, et reprendre l'analyse. Regardons rapidement ce qu'il y a en 401000h avant d'effectuer le XOR :

```
:d 401000
00401000 A4 3F 1C 04 00 8C 73 35-82 A0 04 09 0D 03 ED 9E
```

Nous passons le XOR, le premier chiffrement/déchiffrement a été fait.

```
0040707D  3118              XOR     [EAX],EBX
:d 401000
00401000 A4 1F 1D 05 00 8C 73 35-82 A0 04 09 0D 03 ED 9E
```

Les octets sont bien modifiés, le "d'écryptement" se passe très bien. Nous continuons notre petite exploration et nous arrivons sur ceci :

```
004070C3  8B10              MOV     EDX,[EAX]
```

Le résultat du XOR est placé dans EDX. L'ordre des octets est inversé, EDX contient donc 051D1FA4h. Ensuite arrive :

```
00407109  C1CA02            ROR     EDX,02
```

L'instruction ROR signifie *Rotate Right*.

```
00407150  8910              MOV     [EAX],EDX
```

Après la rotation, le résultat est placé à l'adresse pointée par EAX. Les 4 octets sont inversés par rapport à l'ordre qu'ils ont dans EDX.

```
00407196  3118              XOR     [EAX],EBX
```

Ensuite, un deuxième XOR vient modifier les 4 nouveaux octets pour donner les 4 octets déchiffrés à l'adresse 401000h.

```
004071DC  83C004            ADD     EAX,04
```

Nous venons de déchiffrer 4 octets et le programme incrémente le registre EAX de 4. Il pointe maintenant vers les nouveaux octets à déchiffrer. Le programme effectue une boucle pour cela.

```
00407223  3BC1              CMP     EAX,ECX
```

EAX contient 401004h alors qu'ECX contient 402000h.

Il s'agit d'une comparaison pour savoir si les octets entre l'adresse 401000h et l'adresse 402000h ont été déchiffrés. Le programme boucle jusqu'à ce que EAX = 402000h. Si nous traçons pas à pas, cela prend énormément de temps. Au début de l'analyse, on a vu que le programme detectait les BPX et bloquait la machine. Je présume qu'il détourne l'interruption 3 et la fait pointer vers un "jmp eip" comme pour l'interruption 1.

Nous n'avons pas encore rencontré de code détournant cette interruption, il est donc possible de déposer un point d'arrêt sans bloquer la machine. Pour éviter tout problème, j'ai préféré utiliser un BPX "intelligent" :

```
BPX EIP IF EAX==402000.
```

Cette commande mettra un BPX à l'adresse actuelle seulement quand EAX sera à 402000h. Cela permet de passer tout le déchiffrement en une fraction de seconde, et de récupérer la main juste à l'endroit où nous étions.

> **Remarque**
>
> Si l'auteur de la protection avait détourné l'interruption 3, il aurait fallu tracer tout le déchiffrement à la main, car mon BPX aurait bloqué la machine. (Enfin, il aurait été possible d'empêcher le hook de l'interruption comme nous allons le voir plus tard).

Après quelques instructions tracées, nous arrivons ici :

```
004072B3  CF      IRETD
```

IRET signifie *Interrupt Return*. Cela permet à un programme de revenir en ring 3 après avoir exécuté cette instruction. Nous ne pouvons pas la tracer puisque nous ne sommes pas en ring 0. Que faire ?

Nous avons vu que le programme déchiffrait du code. Il va certainement appeler ce code un peu après ce déchiffrement. Nous allons donc nous rendre manuellement en 401000h. Pourquoi 401000h ? Le programme, compte tenu de sa petite taille et des instructions rencontrées, a été écrit en assembleur. Les programmes en assembleur commencent "toujours" en 401000h (voir **[0]**). Tapons R EIP 401000 pour nous y rendre.

```
00401000  E92F060000    JMP       00401634
```

> **Important :** Avant toute chose, pour ne pas tout recommencer si nous avions à rebooter, il est préférable de "dumper"le processus. Deux solutions sont possibles :
>
> - insertion d'un JMP EIP à la place du JMP 401634. Il suffit de taper "A" -- "Enter", puis "jmp eip" -- "Enter". Il suffit de dumper le processus via LordPE par exemple.
> - Utiliser la commande PEDUMP d'Icedump : /PEDUMP 400000 1000 c:\dump1.exe
>
> Personnellement, je conseille la seconde solution. A partir de maintenant, si notre fichier crashe, il sera toujours possible de recommencer à partir du dump, et donc d'éviter de tout recommencer. De plus, nous pouvons déjà désassembler le fichier dump1.exe pour analyser un peu le programme.

Continuons. Le saut nous amène à deux parties importantes de la protection :

```
0167:00401634  E85E000000    CALL      00401697    ; decrypt les datas.
```

Comme nous le voyons ci-dessous, le code ressemble beaucoup à ce que nous avons déjà rencontré.

```
00401697  B800204000 MOV  EAX,00402000 ; EAX = adresse section data
0040169C  B91A040000 MOV  ECX,0000041A ; ECX = index de boucle.
004016A1  BB00104000 MOV  EBX,00401000 ; EBX = clef
004016A6  8B1B       MOV  EBX,[EBX]    ; EBX = dword pointé par EBX
004016A8  3118       XOR  [EAX],EBX    ; dword pointés par EAX xor EBX
004016AA  83E804     SUB  ECX,04       ; ECX = ECX-4 (on soustrait un dword)
004016AD  83C004     ADD  EAX,04       ; EAX = EAX+4 (on ajoute un dword)
004016B0  83F900     CMP  ECX,00       ; tant que
004016B3  7DF3       JGE  004016A8     ; ECX >= 0 on boucle
004016B5  C3         RET               ; retour
```

Pour l'instant, il n'y a toujours pas de détournement d'interruption 3. Nous pouvons sans crainte presser F10 sur le "CALL 401697" ou F12 si nous avions tracé à l'intérieur de la routine de déchiffrement. Avant déchiffrement :

```
016F:00402170            ......& ..& ..&
016F:00402180..& ..&..f` .@s
016F:00402190            .]c .@ik.Aa .Gos
016F:004021A0            .M□ .]cs.Fhg.[ne
016F:004021B0            .%On.@&b.[ro..rh
```

Après déchiffrement :

```
0167:00402170            y!..
0167:00402180            ..If you
0167:00402190            are looking this
0167:004021A0            by pressing the
0167:004021B0            ..Info button th
```

La section .data est maintenant déchiffrée, continuons avec la seconde partie :

```
00401639  E8B9000000    CALL      004016F7
```

Le CALL nous amène l'obfuscation habituelle : un JMP + boucle.

```
004016F7  EB41              JMP       0040173A
```

Nous examinons uniquement les instructions importantes pour la protection :

```
00401719  0F010D0A204000    SIDT      FWORD PTR [0040200A]
```

Accès à l'IDT. Le programme va sûrement détourner une interruption.

```
0040176A  83C318            ADD       EBX,18      ; 3 * 8
```

Il s'agit de l'interruption 3 (rappel : chaque descripteur fait 8 octets). Regardons le handler de cette interruption.

```
0040184A  B87F1D4000        MOV       EAX,00401D7F
```

Il suffit de taper U 401D7F

```
00401D7F  EBFE              JMP       00401D7F  ; jmp eip
```

Ce n'est pas surprenant.

L'auteur remplace le handler de interruption 3 par un `jmp eip`, ce qui bloque la machine quand un point d'arrêt est rencontré. C'est une technique d'anti-debugging assez simple à mettre en place, mais efficace. La seule solution est de redémarrer le système lorsqu'un point d'arrêt (int 3) est déclenché. Après examen du code qui suit le hook de l'interruption 3, nous ne pouvons donc pas sauter le `CALL 4016F7` pour empêcher le hook de l'interruption 3 puisque ce call calcule des éléments importants pour la suite du chargement du programme.

PETITE EXPLICATION SUR LE HOOK DE L'INTERRUPTION 3

En plus de bloquer les `BPX` que nous mettons dans le code, cette méthode nous empêche de tracer en `STEP OVER` (F10 sous SoftICE) :

```
401xxx      instruction
401xxx  call routine
401xxx      instruction
```

Lorsque nous pressons F10 pour passer le `call` sans rentrer dedans, SoftICE met un point d'arrêt sur l'adresse de retour, pour nous rendre la main juste après l'exécution du `call`. Si l'interruption 3 a été détournée, la machine est bloquée. Ce type d'anti-debugging est pénible puisque nous devons tracer tous les `call` entièrement, les boucles, les séries de tests sans fin ... Si nous tentons de presser sur F12 pour sortir du `call`, nous bloquons pour la raison expliquée ci-dessus. Il va donc falloir empêcher le hook de l'interruption 3, mais continuer l'exécution du `call` pour se rendre à la partie importante. Nous allons recommencer, mais cette fois-ci sans hooker l'interruption 3. Pour quitter sans risque de reboot, voici comment procéder sous SoftICE :

```
"A" -- Enter
"push 0" -- Enter
"call ExitProcess" -- Enter
```

Nous pouvons maintenant presser F5, ce qui quitte le programme proprement. Nous utilisons LordPE et son *break'n Enter* pour reprendre au début du fichier, mais cette fois-ci, nous travaillons sur `dump1.exe` précédemment sauvegardé. Quelques pressions sur F8 et nous revoilà dans le `call` qui détourne l'interruption 3. Le principe pour éviter le détournement est de sauter chaque instruction du code qui touche l'`IDT`.

```
00401719  0F010D0A204000      SIDT      FWORD PTR [0040200A]
```

Par exemple, ici, on tape `R EIP EIP+7` (l'instruction fait 7 octets).

```
0040176A  83C318              ADD       EBX,18
R EIP EIP+3 (3 octets), .....
```

Il suffit de passer toutes les instructions de cette manière pour éviter le détournement de l'int 3. Le code que je vous montre est au milieu de l'obfuscation, donc il n'est pas vraiment possible de savoir facilement où se termine le code "détournant", et de s'y rendre par un `R EIP adresse` . Une fois le détournement passé, on tombe sur les instructions importantes (sans obfuscation) :

```
004018E1  68EC204000          PUSH      004020EC  ;USER32.DLL
00401908  E87C050000          CALL      KERNEL32!GetModuleHandleA
0040192F  010504214000        ADD       [00402104],EAX
```

Récupération et sauvegarde de l'ImageBase de `USER32.DLL`. Avant :

```
D 402104: 0023:00402104 00 00 00 00 00 00 00 00 47 65 74 53 79 73 74 65
```

Après :

```
D 402104: 0023:00402104 00 00 F5 BF 00 00 00 00-47 65 74 53 79 73 74 65
```

```
00401957  68F7204000          PUSH      004020F7    ; KERNEL32.DLL
0040197E  E806050000          CALL      KERNEL32!GetModuleHandleA
004019A5  010508214000        ADD       [00402108],EAX
```

Récupération et sauvegarde l'ImageBase de `KERNEL32.DLL`. Avant :

```
0023:00402108 00 00 00 00 47 65 74 53-79 73 74 65 6D 54 69 6D
```

Après :

```
0023:00402108 00 00 F7 BF 47 65 74 53-79 73 74 65 6D 54 69 6D
```

SUITE DE L'ANALYSE - L'INTERRUPTION 5

Les mécanismes de détournement sont maintenant bien connus, et nous ne nous attarderons plus dessus.

```
004019CD  680C214000    PUSH  0040210C                ; GetSystemTime
004019F4  FF3508214000  PUSH  DWORD PTR [00402108] ; ImageBase Kernel32
00401A1C  E862040000    CALL  KERNEL32!GetProcAddress
```

Le programme sauvegarde l'ImageBase de `Kernel32` un peu plus tôt pour charger dynamiquement l'API `GetSystemTime`. `EAX` contient l'adresse de l'API après l'exécution de `GetProcAddress` et celle-ci est sauvegardée.

```
00401A43  010550214000    ADD       [00402150],EAX ; Sauvegarde
D 402150 ==> 00402150 0E 0F FA BF
```

```
00401A6B  681A214000    PUSH      0040211A   ; MessageBoxA
00401ABA  E8C4030000    CALL      KERNEL32!GetProcAddress
00401AE1  010554214000  ADD       [00402154],EAX  ; Sauvegarde
```

L'adresse de l'API `MessageBoxA` est sauvegardée, ainsi que `GetSystemTime`, `MessageBoxA`, `MessageBoxExA`, `ExitProcess`, et `GlobalAlloc`.

```
001B:00401CE3  C3           RET
```

Une fois terminée, nous sortons de cette routine pleine d'obfuscation, pour arriver ici :

```
0167:0040163E  B443         MOV       AH,43
```

Il est conseillé, une fois de plus, de dumper votre processus pour éviter de tout recommencer en cas de problème.

```
/PEDUMP 400000 163E c:\dump2.exe

0040163E  B443          MOV     AH,43
00401640  CD68          INT     68
00401642  663D86F3      CMP     AX,F386
00401646  0F84DF060000  JZ      00401D2B ; JZ softice_present
```

Il s'agit d'un code de détection du débogueur SoftICE (pour plus d'informations sur cette détection, lire **[0]**). Nous utilisons actuellement Icedump. Celui-ci nous rend indétectable vis-à-vis des méthodes de détection les plus courantes. Continuons :

```
0040164C  8B1D0C204000        MOV     EBX,[0040200C]
00401652  83C328              ADD     EBX,28      ; 5 * 8 = 40 = 28h.
interrupt 5 descriptor.
00401655  FA                  CLI
00401656  668B5306            MOV     DX,[EBX+06]
0040165A  C1E210              SHL     EDX,10
0040165D  668B13              MOV     DX,[EBX]
00401660  891502204000        MOV     [00402002],EDX
00401666  B8B6164000          MOV     EAX,004016B6
0040166B  668903              MOV     [EBX],AX
0040166E  C1E810              SHR     EAX,10
00401671  66894306            MOV     [EBX+06],AX
00401675  CD05                INT     05                  ;
Appel l'interruption une fois hookée.
00401677  8B1D0C204000        MOV     EBX,[0040200C]
0040167D  83C328              ADD     EBX,28
00401680  8B1502204000        MOV     EDX,[00402002]
00401686  668913              MOV     [EBX],DX
00401689  C1EA10              SHR     EDX,10
0040168C  66895306            MOV     [EBX+06],DX
00401690  E852070000          CALL    00401DE7
```

Nous sommes encore une fois en présence d'un hook d'interruption. Cette fois ci, le programme détourne l'interruption 5. Elle n'a aucune importance dans le débogage, l'auteur aurait pu prendre une autre interruption. Il se sert ici du détournement d'interruption pour exécuter du code en ring 0.

Nous allons nous rendre dans son handler :

```
R EIP 4016B6

004016B6  B834164000          MOV     EAX,00401634 ; EAX = 401634
004016BB  BB95164000          MOV     EBX,00401695 ; EBX = 401695
004016C0  33C9                XOR     ECX,ECX      ; ECX = 0
004016C2  0308                ADD     ECX,[EAX]    ; ECX = dwords
pointés par EAX.
004016C4  40                  INC     EAX          ; EAX = EAX+1
004016C5  3BC3                CMP     EAX,EBX      ; tant que
004016C7  7EF9                JLE     004016C2     ; EAX <= EBX on
boucle
```

Le handler effectue une sorte de checksum avec les opcodes entre les adresses 401634h et 401695h. A la fin de la boucle, ECX contient ce checksum, ensuite utilisé pour déchiffrer une partie de la section CODE :

```
004016C9  B8AF114000          MOV     EAX,004011AF ; EAX = 4011AF
004016CE  BB82040000          MOV     EBX,00000482 ; EBX = 482 --
index de boucle
004016D3  3108                XOR     [EAX],ECX    ; dword pointé par
EAX XOR ECX.(checksum)
004016D5  83C004              ADD     EAX,04       ; EAX = EAX+4
004016D8  83EB04              SUB     EBX,04       ; EBX = EBX-4
004016DB  83FB00              CMP     EBX,00       ; Tant que
004016DE  7DF3                JGE     004016D3     ; EBX >= 0 on
boucle.
004016E0  CF                  IRETD                ; Interrupt Return
(passage en ring 3)
```

Le déchiffrement s'effectue et le handler rend la main au programme. Nous sommes en ring 0 ici normalement, mais nous sommes en train d'émuler l'interruption en traçant le handler en ring 3 ; il ne faut donc pas exécuter le IRETD, sous peine de blocage violent. L'auteur de la protection utilise le checksum d'une partie du code pour déchiffrer une autre partie. Si nous avions modifié des instructions, le déchiffrement se serait bien effectué, mais le code résultant ne serait pas valide. Pour retourner après l'interruption 5, il suffit de taper : R EIP 00401690. Nous nous retrouvons sur le call juste après le "dé-hookage" de l'interruption 5.

```
00401690  E852070000          CALL    00401DE7 ; Checksum Check
```

Examinons ce qui se passe dans cette procédure :

```
00401DE7 60           PUSHAD              ; Sauvegarde les registres.
00401DE8 B800104000   MOV   EAX,00401000 ; EAX = 401000
00401DED BBA7114000   MOV   EBX,004011A7 ; EBX = 4011A7
00401DF2 33C9         XOR   ECX,ECX       ; ECX = 0
00401DF4 0308         ADD   ECX,[EAX]     ; ECX = ECX + dword pointé par
EAX.
00401DF6 40           INC   EAX           ; EAX = EAX + 1
00401DF7 3BC3         CMP   EAX,EBX       ;on boucle sur 401DF4 Tant que
00401DF9 7EF9         JLE   00401DF4      ; EAX <= EBX
00401DFB 81F9974B1C42 CMP   ECX,421C4B97  ; ECX = 421C4B97?
00401E01 0F8524FFFFFF JNZ   00401D2B      ; non! BAD checksum on va en
401D2B
00401E07 61           POPAD               ; On restaure les registres
00401E08  C3          RET
```

Il s'agit d'un simple checksum. Encore une fois, si nous avions modifié le programme, nous n'aurions pu continuer l'analyse de la protection (note : ici, il est possible de modifier le test du checksum puisque la valeur n'est pas utilisée pour déchiffrer quoi que ce soit). La valeur d'ECX n'est pas sauvegardée et les registres sont restaurés juste avant de sortir de la procédure. Continuons après le call.

```
00401695  EB4A          JMP     004016E1
```

Un saut vers le code suivant :

```
004016E1  68061D4000    PUSH    00401D06          ; Exception Handler
004016E6  6467FF360000  PUSH    DWORD PTR FS:[0000]
004016EC  646789260000  MOV     FS:[0000],ESP
004016F2  E90EF9FFFF    JMP     00401005          ; saut en 401005
```

Une pratique courante dans les techniques anti-debugging est d'utiliser les *except handlers* pour rediriger le code. C'est beaucoup moins visible qu'un simple `call`, ou saut. Vous connaissez sûrement l'utilisation des SEH (*Structured Exception Handler*) en langage de programmation haut niveau. En C++, par exemple, ils ressemblent à ceci :

```
__try{
      Code à protéger.
     };
__except{
          Le code ici sera exécuté seulement en cas d'erreur dans la
clause "try"
        };
```

Pour plus d'information regardez une documentation complète sur le sujet **[6]**.

Les SEH permettent d'accéder au contexte de l'application par exemple. Beaucoup de protections les utilisent pour effacer les BPM, ou de modifier les *debug registers* sans passer en ring 0. Revenons à l'analyse. L'adresse 401D06h sera appelée en cas d'erreur dans le code.

```
00401005  8B1D0C204000        MOV       EBX,[0040200C]
0040100B  83C328              ADD       EBX,28
....
0040101F  B8E1104000          MOV       EAX,004010E1
....
0040102E  EB63                JMP       00401093
00401030  CD05                INT       05
```

Encore un hook de l'interruption 5. Nous commençons à avoir l'habitude. Comme nous pouvons le voir, l'interruption n'est pas exécutée immédiatement, un saut inconditionnel nous détourne de l'exécution de l'interruption. Après quelques tests sans importance, l'int 5 est exécutée. L'int 5 regorge de tests différents qui influencent le déroulement du programme. Dans certaines conditions, le code en dessous de l'interruption est modifié par "FFFFh", ce qui aura pour résultat de provoquer une erreur lors de l'exécution. Le *except handler* prendra alors la main.

Le code est très embrouillé et pas vraiment simple à comprendre. Je préfère faire un dump de mon processus afin de ne pas perdre mon travail (`/PEDUMP 400000 101F c:\dump3.exe`). Je décide d'analyser le reste de la protection avec IDA. Nous fermons l'application de la même façon que tout à l'heure et nous désassemblons avec IDA notre `dump3.exe`. Après un rapide coup d'œil, nous trouvons ceci :

```
CODE:004011C4                 push    0               ; CODE XREF:
DialogFunc+155j
CODE:004011C6                 call    GetModuleHandleA
CODE:004011CB                 mov     ds:hInstance, eax
CODE:004011D0                 jmp     loc_401285
CODE:004011D5 ; ---------------------------------------------------
CODE:004011D5
CODE:004011D5 loc_4011D5:                             ; CODE XREF:
DialogFunc+B4j
CODE:004011D5                 jmp     loc_40146F
```

```
CODE:004011DA
CODE:004011DA                 push    0               ; dwInitParam
CODE:004011DC                 push    offset DialogFunc ; lpDialogFunc
CODE:004011E1                 push    0               ; hWndParent
CODE:004011E3                 push    67h             ; lpTemplateName
CODE:004011E5                 push    ds:hInstance    ; hInstance
CODE:004011EB                 call    DialogBoxParamA
```

L'API `DialogBoxParamA` utilise les ressources pour fonctionner correctement. Nous n'avons toujours pas déchiffré les ressources (adresse > 405000h). Apparemment tout le programme est déchiffré, excepté les ressources. Tapons `Ctrl+S` pour nous rendre à la section ressource.

```
.rsrc:00405000 _rsrc          segment para public 'DATA' use32
.rsrc:00405000                assume cs:_rsrc
.rsrc:00405000                ;org 405000h
```

Aucune référence à cette adresse ; nous faisons défiler cette section afin de voir si une référence existe, et nous trouvons ceci

```
.rsrc:00405430 unk_405430     db  97h ; ù           ; DATA XREF:
CODE:004012FDo
.rsrc:00405431                db  2Dh ; -
.rsrc:00405432                db 0BFh ; +
.rsrc:00405433                db 0E3h ; Ô
.rsrc:00405434                db  93h ; ô
```

Il ne nous reste plus qu'à nous rendre en 4012FDh pour trouver la routine de déchiffrement :

```
CODE:004012FD                 mov     eax, offset unk_405430
CODE:00401302                 mov     ebx, 3E0h
CODE:00401307
...
```

Nous avons trouvé la routine. Relançons l'application pour déchiffrer la section des ressources. Juste avant de retourner sous SoftICE, nous avons besoin de trouver le début de la routine. Grâce à IDA, je remonte légèrement dans le code, je modifie un peu le désassemblage qu'IDA a fait, car certaines instructions n'ont pas été désassemblées (utilisation de la touche "C" pour transformer des octets en code d'IDA).

```
CODE:0040128B                 add     ebx, 28h
CODE:0040128E                 cli
CODE:0040128F                 mov     dx, [ebx+6]
CODE:00401293                 shl     edx, 10h
CODE:00401296                 mov     dx, [ebx]
CODE:00401299                 mov     ds:dword_402002, edx
CODE:0040129F                 mov     eax, offset loc_4012CE
CODE:004012A4                 mov     [ebx], ax
CODE:004012A7                 shr     eax, 10h
CODE:004012AA                 mov     [ebx+6], ax
CODE:004012AE                 int     5
```

Une dernière redirection d'interruption. Nous allons donc nous rendre à l'adresse du handler sous SoftICE, et utiliser LordPE

pour relancer l'application comme nous avons maintenant l'habitude de faire : `R EIP 4012CE` :

```
004012CE  0F21C0     MOV   EAX,DR0      ; EAX = contenu de DR0
004012D1  85C0       TEST  EAX,EAX     ; si eax = 0 pas de BPM
004012D3  7540       JNZ   00401315    ; sinon erreur_crash
```

Les *debug registers* DR0 à DR3 sont vérifiés. Si leur contenu est différent de zéro, alors le programme en cours bloque.

```
004012EA  B834164000 MOV   EAX,00401634 ; EAX = 401634
004012EF  BBE71D4000 MOV   EBX,00401DE7 ; EBX = 401DE7
004012F4  33C9       XOR   ECX,ECX     ; ECX = 0
004012F6  0308       ADD   ECX,[EAX]   ; ECX = ECX + dword pointé par
EAX
004012F8  40         INC   EAX         ; EAX = EAX + 1
004012F9  3BC3       CMP   EAX,EBX     ; Tant que
004012FB  7EF9       JLE    004012F6   ; EAX <= EBX on boucle sur
4012F6
```

Le programme calcule la checksum entre l'adresse 401634h et 401DE7h. Il s'en sert pour déchiffrer les ressources :

```
004012FD  B830544000 MOV   EAX,00405430 ; EAX = 405430 Adresse dans la
section ressource.
00401302  BBE0030000 MOV   EBX,000003E0 ; EBX = 3E0 (taille à
décrypter)
00401307  3108       XOR   [EAX],ECX    ; dword pointé par EAX XOR
CheckSum (ECX)
00401309  83EB04     SUB   EBX,04       ; EBX = EBX - 4
0040130C  83C004     ADD   EAX,04       ; EAX = EAX + 40040130F  83FB00
CMP   EBX,00                            ; tant que·
00401312  7DF3       JGE   00401307     ; EBX >= à zero on boucle sur
401307
00401314  CF         IRETD              ; Retour en ring 3.
```

Comme nous le constatons, le déchiffrement s'effectue avec la checksum d'une partie du code comme clef. A partir de maintenant, notre application est complètement déchiffrée, nous pouvons effectuer le dump final du processus. Un coup d'œil rapide à la section ressource nous confirme le bon déchiffrement :

```
D 405430

0023:00405430 61 00 6B 00 65 00 20 00-74 00 72 00 69 00 61 00  a.k.e.
.t.r.i.a.
0023:00405440 6C 00 20 00 61 00 70 00-70 00 6C 00 69 00 63 00  l.
.a.p.p.l.i.c.
0023:00405450 61 00 74 00 69 00 6F 00-6E 00 73 00 20 00 3A 00
a.t.i.o.n.s. .:.
etc..
```

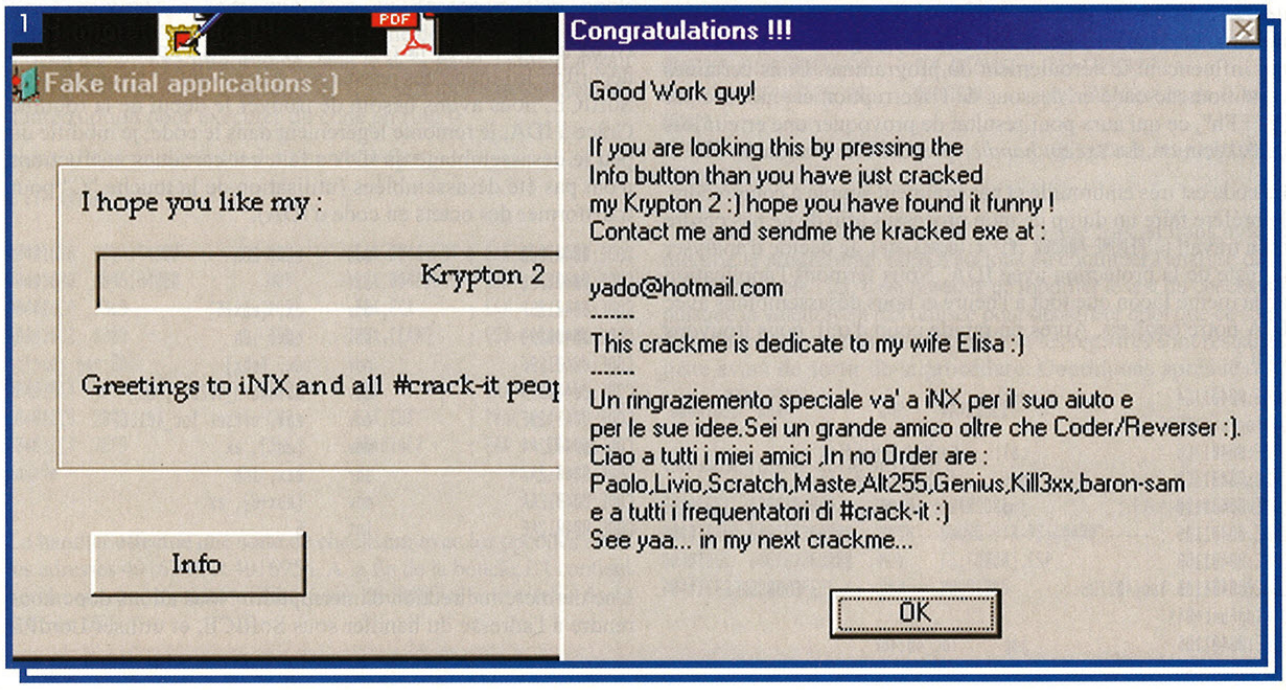
Tout a l'air normal. Étant donné que nous ne connaissons pas encore le véritable *entry point* à utiliser pour notre dump, j'utilise 1000 comme valeur (*entry point* original qui pointe vers la protection). Nous corrigerons ensuite, après analyse finale du fichier sous IDA.

```
/PEDUMP 400000 1000 c:\final.exe
```

Nous terminons le processus comme nous l'avons fait jusqu'à présent par insertion d'un *call exitprocess*.

PATCHAGE DU FICHIER

Relisons les instructions qui accompagnaient notre fichier protégé contre le reverse engineering. Le challenge est de patcher le fichier pour qu'il puisse se lancer sans limitation dans le temps. Les patches en mémoire ne sont pas autorisés. Nous ne pouvions analyser le code dès le départ grâce à IDA puisque celui-ci était chiffré. Nous



avons effectué tout le travail de déchiffrement, il ne nous reste plus qu'à modifier le programme pour qu'il se lance sans problème.

Petite remarque : dans la section des données déchiffrées, j'ai observé le message suivant :

```
DATA:00402164    db 'If you are looking this by pressing the',0Dh,0Ah
DATA:00402164    db 'Info button than you have just cracked',0Dh,0Ah
DATA:00402164    db 'my Krypton 2 :)
```

L'auteur fait référence à un bouton à presser. Comme je le disais plus tôt, l'API `DialogBoxParamA` utilise les ressources pour afficher une boite de dialogue.

Nous faisons débuter le code à cet emplacement.

```
CODE:004011C4    push    0               ; CODE XREF: DialogFunc+155j
CODE:004011C6    call    GetModuleHandleA
CODE:004011CB    mov     ds:hInstance, eax
CODE:004011D0    jmp     loc_401285
CODE:004011D5 ; ---------------------------------------------------
CODE:004011D5
CODE:004011D5 loc_4011D5:                ; CODE XREF: DialogFunc+B4j
CODE:004011D5    jmp     loc_40146F
CODE:004011DA
CODE:004011DA    push    0               ; dwInitParam
```

suite page 34 ➔ ➔

Quelques points importants

Avant de terminer cet article, revenons sur quelques points importants. Dans la procédure de hooking de l'interruption 3, le programme a calculé certaines adresses dynamiquement. Nous n'avons pas encore vu leur utilité. L'une d'elles servait à vérifier si la date d'utilisation du programme est expirée. Avec ma méthode, nous n'avons pas eu à nous préoccuper de cette vérification. En revanche, nous calculions l'adresse d'une API, tout particulièrement `MessageBoxA`. Celle-ci sert à afficher le message final lors de la pression sur le bouton. Pour retrouver ce code, il suffit de partir du code de la DialogBox qui gère la pression sur le bouton :

```
CODE:0040121C     cmp     [ebp+arg_4], 2
CODE:00401220     jz      short loc_401241
CODE:00401222     cmp     [ebp+arg_4], WM_COMMAND
CODE:00401229     jz      short bouton_presse
...
CODE:0040124F bouton_presse:                          ; CODE XREF: DialogFunc+14j
CODE:0040124F     cmp     [ebp+arg_8], 66h
CODE:00401253     jz      short Bouton_message
CODE:00401255     cmp     [ebp+arg_8], 67h
CODE:00401259     jz      short Exit

CODE:00401262 Bouton_message:                         ; CODE XREF: DialogFunc+3Ej
CODE:00401262     push    0
CODE:00401264     push    offset aCongratulation      ; "Congratulations !!!"
CODE:00401269     push    offset aGoodWorkGuyIfY      ; "Good Work guy!\r\n
CODE:0040126E     push    ds:hInstance
CODE:00401274     mov     eax, offset address_index   ; EAX = 0040214C
CODE:00401279     add     eax, 8; EAX = EAX + 8 = 00402154
CODE:0040127C     call    dword ptr [eax]             ;Appel l'adresse pointé par EAX.

DATA:00402154 AddrMessageBox  dd 0BFF5412Eh
```

Le `call [eax]` appelle l'adresse BFF5412Eh, qui correspond au MessageBoxA sur ma version de Windows (la valeur est "hardcodée" puisque nous avons fait un dump après le calcul de l'adresse). Voilà pourquoi il était important de tracer complètement la procédure qui détournait l'interruption 3. Celle-ci "calculait" aussi des adresses cruciales pour le fonctionnement du programme.

L'article est déjà bien long, c'est pourquoi je n'ai pas montré tout le code anti-reverse engineering. Il y avait dans cet exécutable des routines qui scannaient les adresses calculées précédemment, à la recherche de "0xCC" dans leur code (BPX). Étaient également présentes des routines qui détruisaient le code si elles détectaient la présence d'une personne qui déboguait la protection, soit en effaçant du code, soit en ré-écrivant certaines parties des données avec le mot "YADO" (pseudonyme de l'auteur de la protection).

```
CODE:004011DC    push    offset DialogFunc ; lpDialogFunc
CODE:004011E1    push    0                 ; hWndParent
CODE:004011E3    push    67h               ; lpTemplateName
CODE:004011E5    push    ds:hInstance      ; hInstance
CODE:004011EB    call    DialogBoxParamA
```

L'API `DialogBoxParamA` a besoin de l'instance du programme (calculé avec `GetModuleHandleA`) pour fonctionner. Nous voyons deux sauts qui détournent le bon fonctionnement du programme. Ils ne nous sont plus utiles désormais. Nous les remplaçons par des NOP.

Un clic sur le premier jmp en 4011D0h nous donne l'offset dans le fichier. IDA nous informe de celui-ci.

A l'aide d'un éditeur hexadécimal, il reste à changer les opcodes des `jmp` en `nop`.

```
CODE:004011D0 E9 B0 00 00 00 devient CODE:004011D0 90 90 90 90 90
CODE:004011D5 E9 95 02 00 00 devoent CODE:004011D5 90 90 90 90 90
```

Il nous faut aussi changer l'*entry point* du programme qui, pour l'instant, est à 1000h. Nous utilisons le RVA du code qui passe le premier paramètre à `GetModuleHandleA`.

```
"11C4" (RVA = Virtual Address - Imagebase = 4011C4 - 400000 =
11C4)
```

Pour changer l'*entry point* avec LordPE sur `final.exe`, on remplace "1000h" par "11C4h" dans le champ *Entry point* puis on sauvegarde les changements. Exécutons le programme. Une fenêtre s'affiche avec un bouton. Une pression sur celui-ci et nous observons un message nous expliquant que le challenge a été réussi (voir page 32).

CONCLUSION

L'objectif de cet article était de montrer diverses techniques pour lutter contre le reverse engineering : lutte contre le débogage, passage en ring 0 pour rendre inopérant tout débogueur fonctionnant en ring 3 (débogueur d'IDA, OllyDbg, ...), détection des watchpoints par simple lecture des debug registers, checksums utilisées comme clé de déchiffrement pour empêcher toute modification du code. Il s'agissait aussi de montrer comment les contourner compte tenu du thème du dossier de ce numéro.

Il est nécessaire de pouvoir analyser tout programme susceptible de pratiquer l'évasion de données (voir l'article de P. Chambet, E. Filiol et E. Detoisien à ce sujet dans ce numéro) , même si celui-ci est chiffré sur le disque et donc protégé contre le désassemblage, ou qu'il utilise des techniques d'anti-debug. L'utilisation groupée de nombreux outils nous a permis de comprendre le fonctionnement des protections, et de les contourner pour obtenir un exécutable complètement opérationnel, sans limite de temps.

Très peu de vers utilisent encore de vraies méthodes de protection contre l'analyse de leur code à l'heure actuelle (signalons toutefois le virus Whale dont le code est un modèle du genre). La plupart d'entre eux utilisent des packers d'exécutables peu complexes, fonctionnant sur le même principe que la protection présentée. Il suffit de dumper le processus quand la section code est décryptée. Il n'y a en général aucun code anti-debugging. Il est possible que dans le futur, des programmes espions, des vers, virus ou autres malwares soient protégés contre le reverse engineering, et j'espère que cet article vous aura donné un aperçu des techniques d'analyse de binaires protégés, et comment coupler l'utilisation d'un désassembleur et d'un débogueur, pour que vous puissiez vous aussi analyser des binaires protégés et détecter le futur malware caché sur votre disque dur.

Nicolas Brulez

Cartel Sécurité

brulez@cartel-securite.fr

http://www.cartel-securite.fr

The Armadillo Software Protection System

http://www.siliconrealms.com/armadillo.htm

RÉFÉRENCES

[0] N. Brulez - *Analyse d'un ver par désassemblage* - MISC Le journal de la sécurité informatique, no. 5, janvier 2003.

[1] IDA Pro © Datarescue - **http://www.datarescue.com/idabase/**

[2] SoftICE © Compuware - **http://www.compuware.com/products/driverstudio/softice/**

[3] IceDump - **http://ghiribizzo.virtualave.net/icedump/icedump.html**

[4] LordPE - **http://mitglied.lycos.de/yoda2k/LordPE/info.htm**

[5] Intel 386 Programmer's Reference - **http://www.online.ee/~andre/i80386/Chap9.html (9.4)**

[5bis] Debug Registers - **http://www.online.ee/~andre/i80386/Chap12.html**

[6] *Structured Exception Handling* par Jeremy Gordon - **http://spiff.tripnet.se/~iczelion/Exceptionhandling.html**

[7] L'exécutable Krypton2 - **http://www.miscmag.com/Download/Krypton2.zip**