COMPRENDRE TELOCK 0.51

par BeatriX

1 . INTRODUCTION

Cet article (plus que tutorial) a pour objectif de vous expliquer le fonctionnement du packer tElock 0.51. Il s'adresse avant tout aux crackers d'un niveau «débutant» qui connaissent déjà les bases élémentaires du cracking, à savoir :

1 . Utiliser un désassembleur/debugger
2 . Utiliser un éditeur héxa.
3 . Avoir quelques notions simples sur le format PE

Je précise qu'il ne s'agit pas ici de vous apprendre une technique particulière comme le unpacking (même si j'aborde le sujet plus bas) mais plutôt d'étudier les techniques de anti-unpacking (modification du header, redirection de l'IAT), de cryptage (layers, CRC32), d' anti-debuggage (SEH, checksum, protection silencieuse) en analysant le code assembleur d'un programme packé avec tElock 0.51.

tElock 0.51 est un packer qui a été réalisé en 2000 par tE! Alias The Egoist, cracker de la Team [TMG]. « tElock fût conçu à l'origine par les membres de TMG uniquement dans le but de compresser et protéger leurs keygens contre les voleurs [...] j'ai décidé quelques temps après de le rendre public[...] ». Un unpacker générique pour les versions 0.41, 0.42, 0.51 appelé teunlock v1.0 a vu le jour en septembre 2000 réalisé par r!sc et Cyber Daemon. Il était accompagné d'une note intitulée « Game Over » ce qui eu l'effet escompté sur l'auteur du packer ! Cette version fût le début d'une petite guerre entre packers tElock et unpackers et toutes les versions de tElock visent à mettre en échec des unpackers génériques.

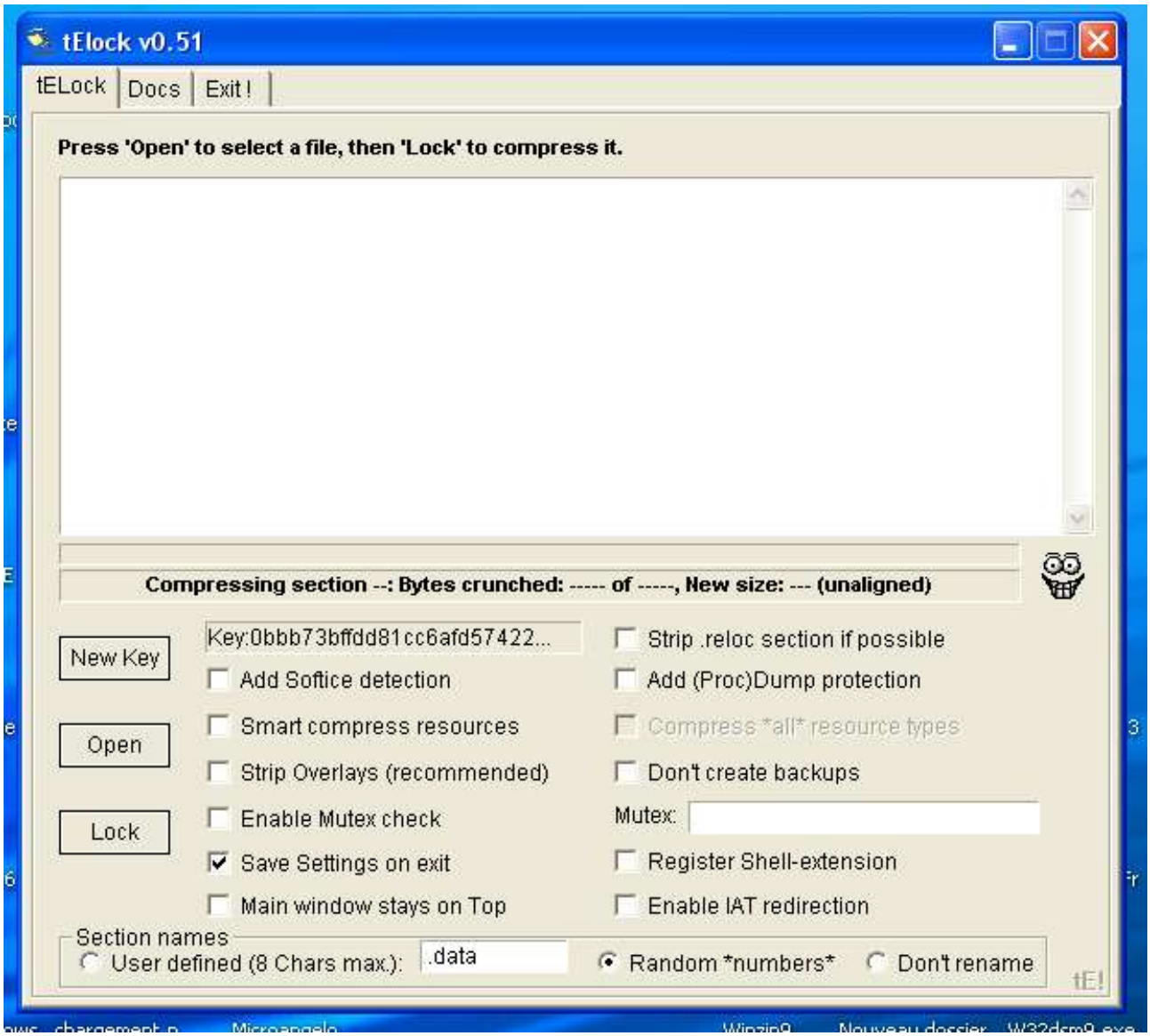
J'espère que vous aurez tout le plaisir que j'ai pu avoir à étudier et décortiquer ce packer.

Comme l'a si bien dit Daemon le 10 septembre 2000,

« debugging telocked files was quite funny @ first :) »

2 . PACKER UN EXE AVEC TELOCK 0.51

Lorsque vous lancez tElock 0.51, vous voyez la fenêtre suivante :



Un certain nombre d'options vous est proposé en plus de la protection par défaut. Nous allons donc étudier le programme calc.exe (calculatrice de Windows) packé par tElock 0.51. Je ne passerai pas en revue toutes les options proposées, seules seront étudiées :

1) Add SoftIce Detection
2) Enable Mutex Check.
3) Add (Proc)Dump protection.
4) Enable IAT redirection.

Tout le travail a été fait sous Windows XP Pro.

Pour le debuggage, j'ai utilisé un debugger ring 3 : OLLYDEBUGGER 1.09b.

Voilà une comparaison des différentes sections de calc.exe avant le packing/cryptage et après :

Avant le packing de l'exe :

Raw Offset	Raw Size	*Nom*
0	400	*Header*
400	12800	*.text*
12C00	A00	*.data*
13600	8C00	*.rscr*

Après le packing de l'exe :

Raw Offset	Raw Size	*Nom*
0	400	*Header (modifié)*
400	7C00	*3735848 (renommé, crypté et compressé)*
8000	400	*5958302 (renommé, crypté et compressé)*
8400	8C00	*.rscr (inchangé)*
11000	1800	*.data (ajout du loader de tElock)*

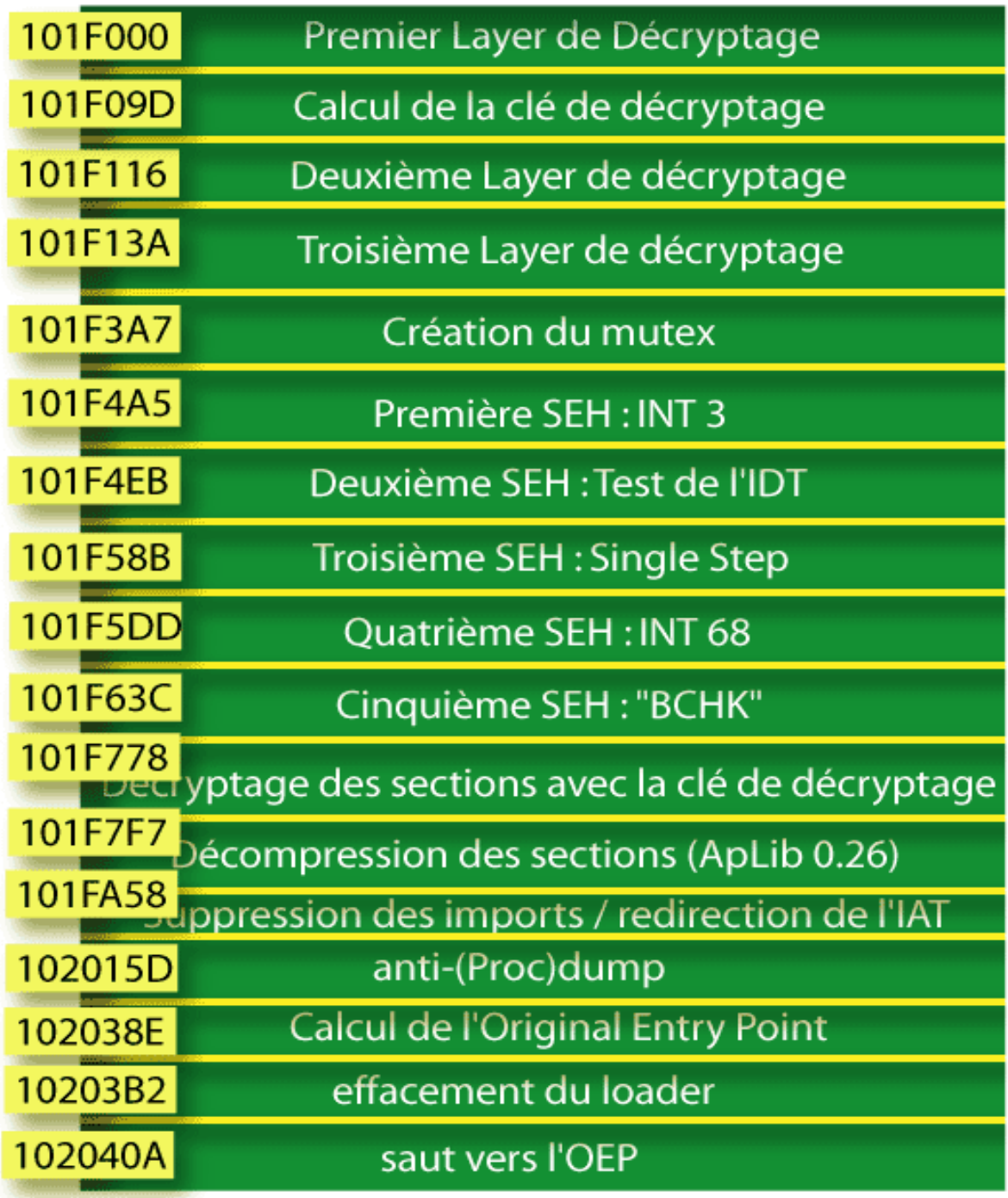
1) Le header est modifié
2) La section .text qui contient le code est packée, cryptée et renommée.
3) La section .data qui contient les imports est packée, cryptée et renommée.
4) La section .rscr subi le même sort si l'option est cochée.
5) Une section nommée .data est ajoutée à la fin et contient le loader tElock qui est chargé de décrypter puis unpacker les 2/3 sections précédentes.

Nous allons analyser ce loader qui est lui même protégé par cryptage et anti-debuggers.

3 . LE LOADER DE TELOCK 0.51

a . Schéma du loader

Pour commencer, voici un schéma de la section .data qui est chargée de décrypter et unpacker les sections de l'exécutable. Je vous proposerai ensuite d'analyser chaque partie de façon thématique. Je ne suivrai donc pas le loader dans son déroulement chronologique.


101F000
Premier Layer de Décryptage
101F09D
Calcul de la clé de décryptage
101F116
Deuxième Layer de décryptage
101F13A
Troisième Layer de décryptage
101F3A7
Création du mutex
101F4A5
Première SEH : INT 3
101F4EB
Deuxième SEH : Test de l'IDT
101F58B
Troisième SEH : Single Step
101F5DD
Quatrième SEH : INT 68
101F63C
Cinquième SEH : "BCHK"
101F778
yptage des sections avec la clé de décryptage
101F7F7
écompression des sections (ApLib 0.26)
101FA58
uppression des imports / redirection de l'IAT
102015D
anti-(Proc)dump
102038E
Calcul de l'Original Entry Point
10203B2
effacement du loader
102040A
saut vers l'OEP

b . Techniques d'anti-debuggage/désassemblage

1 . Le CCA (Code Changeant d'Apparence ou overlapping)

Le CCA appelé par les puristes *overlapping* est une technique visant à dérouter les désassembleurs (je dis désassembleur et non débuggers). Elle exploite une « faiblesse » des désassembleurs qui lisent le code de façon linéaire. De plus, pour un programmeur, le CCA est très facile à mettre en place...Je vous propose l'exemple suivant qui illustre la mise en place d'un tel dispositif :

Voici votre code de départ vu par votre désassembleur :

```
401000    6A 00          PUSH 0
401002    68 00304000    PUSH 403000
401007    68 17304000    PUSH 403017
40100C    6A 00          PUSH 0
40100E    E8 13000000    CALL <JMP.&user32.MessageBoxA>
```

Une classique MessageBoxA....

Première étape de la modification :
On ajoute un jump et un nop....

```
401000    EB 01          JMP 401003
401002    90             NOP
401003    6A 00          PUSH 0
401005    68 00304000    PUSH 403000
40100A    68 17304000    PUSH 403017
40100F    6A 00          PUSH 0
401011    E8 13000000    CALL <JMP.&user32.MessageBoxA>
```

Deuxième étape de la modification :

On change le code 90h par EBh...

```
401000    EB 01          JMP 401003
401002    EB             ???
401003    6A 00          PUSH 0
401005    68 00304000    PUSH 403000
40100A    68 17304000    PUSH 403017
40100F    6A 00          PUSH 0
401011    E8 13000000    CALL <JMP.&user32.MessageBoxA>
```

Votre code à l'arrivée, comme votre désassembleur vous le présente :

```
401000      EB 01           JMP 401003
401002      EB 6A           JMP 40106E
401004      00 6800         ADD BYTE PTR DS:[EAX],CH
401007      3040 00         XOR BYTE PTR DS:[EAX],AL
40100A      68 17304000     PUSH 403017
40100F      6A 00           PUSH 0
401011      E8 13000000     CALL <JMP.&user32.MessageBoxA>
```

Voilà !...un simple jump suivi d'un nop qu'on modifie et le tour est joué ! Votre désassembleur perd les pédales...Lorsque vous débuggez ligne par ligne, vous voyez apparaître sur la ligne courante le code de départ, vous avez donc l'impression que le code se modifie, qu'il change d'apparence ! L'objectif de cette technique est évident : éviter le dead listing !

Si vous tentez de debugger tElock, vous serez confronté à cette technique qui est employée toutes les 3 lignes de code environ...Dans la suite de ce tutorial, je ferai abstraction de cette technique donc toutes les portions de code que vous verrez seront celles d'origine, sans le CCA.

2 . Les layers de décryptage.

Un layer est une boucle chargée de décrypter une portion de code du loader lui-même. Le but est bien sûr de ralentir la progression du cracker. Comme vous pouvez le voir sur le schéma du loader, ils sont au nombre de 3.

Premier layer de décryptage :

Voici le code commenté :

```
101F013     POP ESI                 ESI = 101F013
            ADD ESI, 5E             ESI = 101F071 (début du décryptage)
            MOV EDI, ESI            EDI = 101F071
            PUSH 179                Taille de la portion de code à décrypter
            POP ECX                 ECX = 179
101F01F     LODS BYTE PTR DS:[ESI]  Stocke l'octet situé à l'adresse ESI dans EAX
            DEC AL                  Décryptage
            XOR AL, CL              Décryptage
            STOS BYTE PTR ES:[EDI]  Replace l'octet modifié en mémoire
            DEC ECX
            JG 101F01F              Boucle jusqu'à ECX = 0

101F071     PUSH 5                  Fin du premier layer.
```

Voici une interprétation du décryptage :

Il commence le décryptage en 101F071.

1) On « xor » l'octet situé en 101F071 avec 179.
2) On « xor » l'octet situé en 101F072 avec 178.
3) On « xor » l'octet situé en 101F073 avec 177.
...etc

Remarque : Ce premier layer sert à décrypter la boucle de calcul de la clé de décryptage qui se termine en 101F0EA.

Deuxième layer de décryptage :

Voici le code commenté :

101F116	LEA ESI, [EDI + 0112h]	*ESI = 101F112*
	LEA EDI, [EDI + 01632h]	*EDI = 1020632 (début du décryptage)*
	MOV ECX, 039h	*Taille du décryptage*
101F12A	LODSB	*EAX récupère le contenu situé en ESI*
	XOR [EDI], AL	*Xor le contenu situé en EDI avec AL*
	DEC ECX	
	JG 101F12A	*Boucle jusqu'à ECX = 0*
101F13A		*Fin du second layer.*

Remarque : Ce layer sert à décrypter la chaîne de caractères suivante :

Error! Loader failed Debugger detected CRC check error

Troisième layer de décryptage :

Voici le code commenté :

101F13A	LEA ESI, [EBP + 018Eh]	*ESI = 101F206 (début du décryptage)*
	MOV ECX, 01208h	*Taille de décryptage*
	MOV EDX, 014h	*Compteur*
	SUB WORD [EBP + 16Eh], 2DD	*Leurre...inutile pour le décryptage*
	MOV EDI, ESI	
	MOV BL, 073h	
101F15F	LODSB	*EAX récupère le contenu situé en ESI*
	XOR AL, [EBP+EDX+179]	
	ADD WORD [EBP + 15Ah], 5F03	*Leurre !*

```
            DEC EDX
            JG 101F183                  Condition : Si EDX = 0 alors EDX = 14
            MOV EDX, 14                 Suite...
101F183     SUB AL, 025h
            XOR AL, CL
            ROR WORD [EBP+153h], CL     Leurre
            MOV BYTE [EDI], AL
            ROR AL, CL
            INC BYTE [EBP+131h]         Leurre
            XOR WORD [EBP + 122h], DX   Leurre
            XOR AL, BL
            ADD WORD [EBP + 146h], AX   Leurre
            ROL WORD [EBP + 10Fh], CL   Leurre
            ADD BL, BYTE [EDI]
            STOSB BYTE [EDI]            Replace l'octet modifié en mémoire
            SUB WORD [EBP + Cfh], BX    Leurre
            ADD BYTE [EBP + E4h], DL    Leurre
            DEC ECX
101F1EA     JG 101F15F                  Boucle jusqu'à ECX = 0

101F1F0                                 Fin du troisème layer.
```

Remarque :

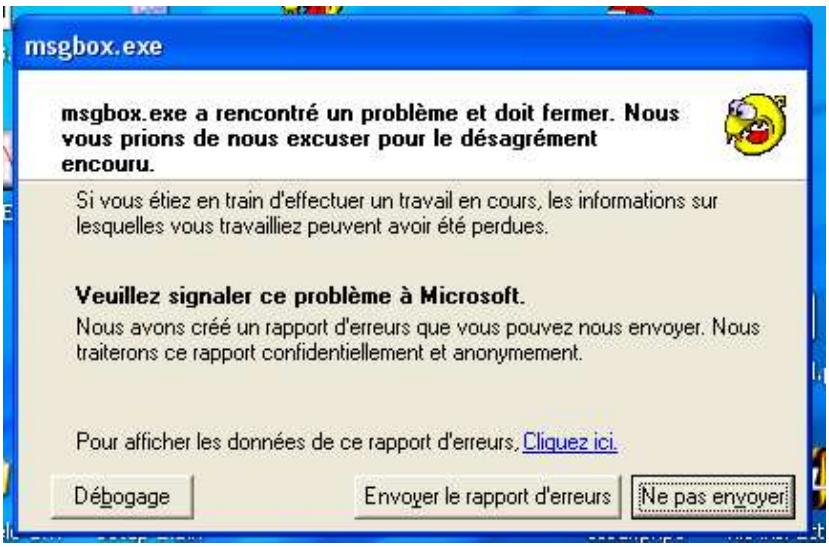
Ce layer permet de décrypter tout le reste du loader de 101F206 à 10240E. Vous remarquez aussi des lignes inutiles (leurre !) qui servent à modifier le layer lui-même. C'est ce qu'on appelle du code polymorphe qui vise à dérouter le cracker.

3 . LES SEH

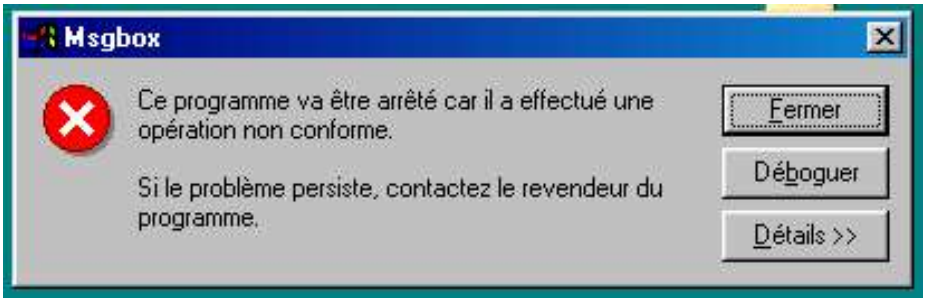
Avant de se lancer dans l'étude de ces anti-debuggers, je vous propose une rapide explication de ce qu'est une SEH et comment ça marche pour l'anti-debuggage.

Les SEH au service de l'anti-debuggage.

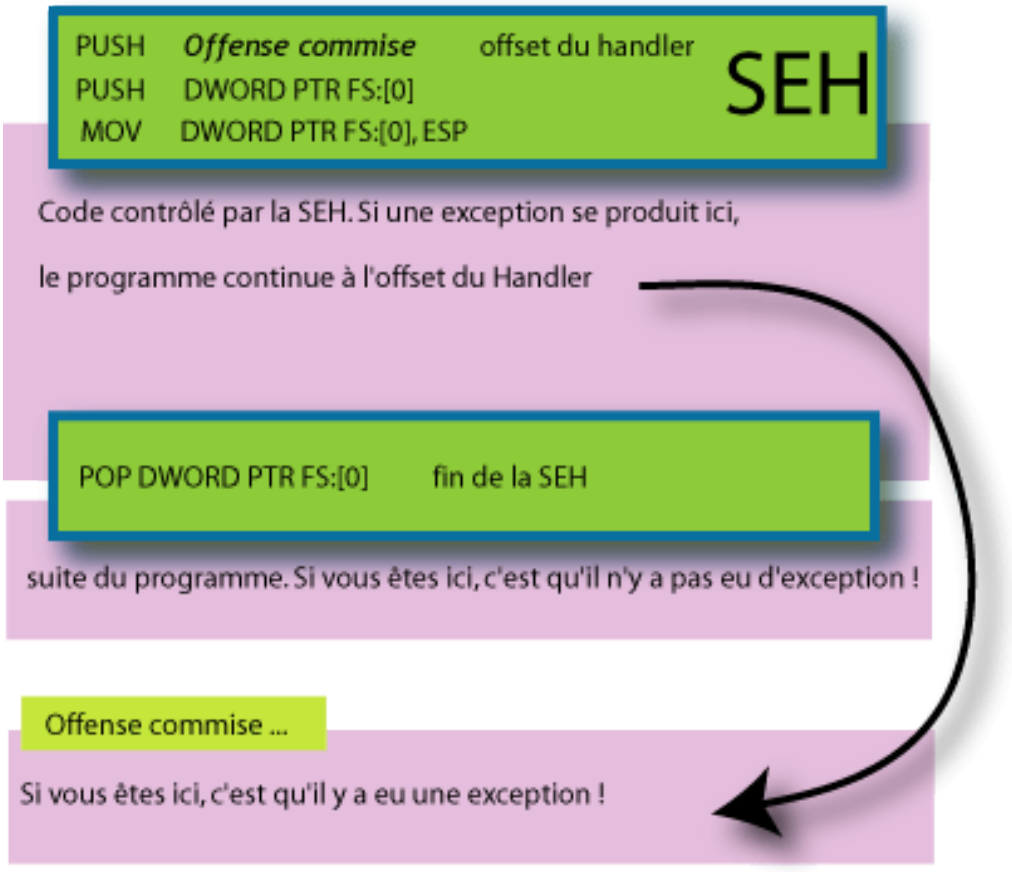
Les SEH (Structured Exception Handler) sont là pour gérer les exceptions (offenses commises au système Windows). Par exemple, si votre programme fait une opération non conforme (tentative d'accès à une zone de la RAM interdite, division par zéro,...), Windows vous affiche en général ceci :



ou ceci si vous êtes sous Win98 :



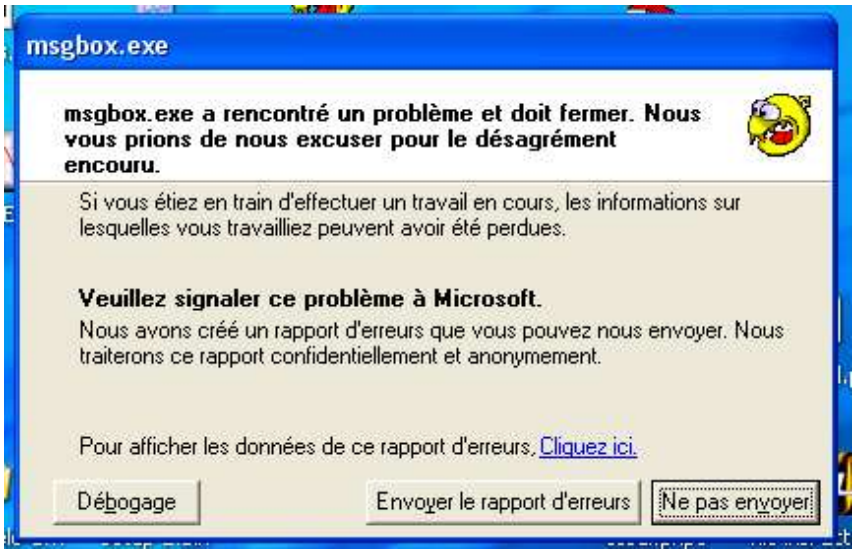
Votre programme peut faire ce travail à la place de Windows et réagir comme bon vous semble. Pour cela, il suffit que vous installiez une SEH qui va surveiller l'exécution de votre code et qui va vous envoyer vers l'OFFSET du HANDLER (un endroit précis dans votre programme) si une exception est commise.



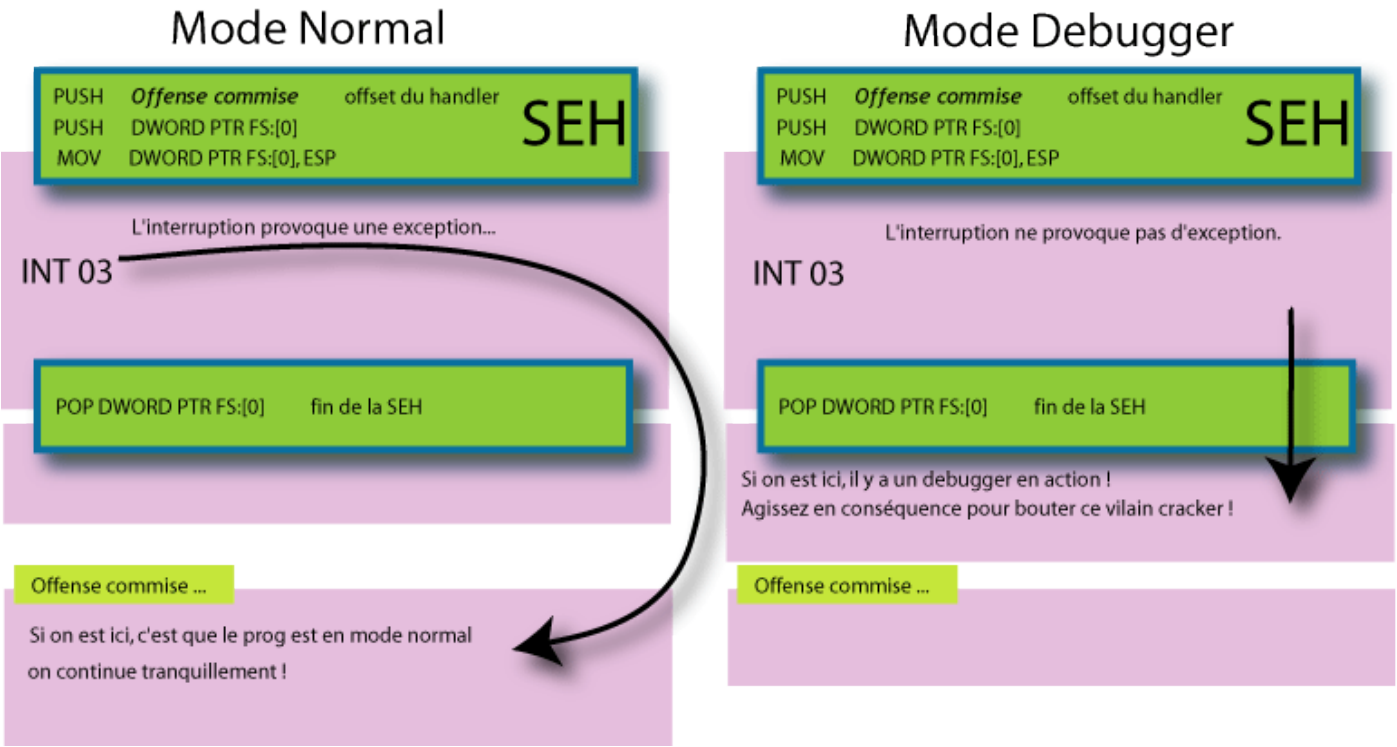
Mais on peut utiliser ces SEH d'une toute autre façon dans un but bien différent ! Elles peuvent servir à détecter la présence de debuggers !! Voici l'astuce...

Certaines interruptions logicielles comme INT 1, INT 3, INT 68... sont utilisées pour le debuggage. Par exemple, quand vous posez un BPX (F2) pour que votre programme breake en cours d'exécution, le debugger modifie le code de votre programme à l'endroit du BP et inscrit l'octet CCh qui correspond à l'instruction INT 3...l'interruption INT 3 est en fait un simple breake point ! Dès que votre programme tombe sur ce CCh , il donne la main au système, le vecteur de INT 3 envoie via l'IDT vers une routine qui gère le BP. Ceci étant dit, que se passerait-il si vous posiez dans votre code un INT 3, c'est-à-dire un CCh ?

Si vous lancez votre programme en mode normal (sans passer par un debugger), il plante lamentablement car l'INT 3 n'est pas gérée ! L' opcode CCh génère une exception ! Vous avez droit à la fenêtre suivante :



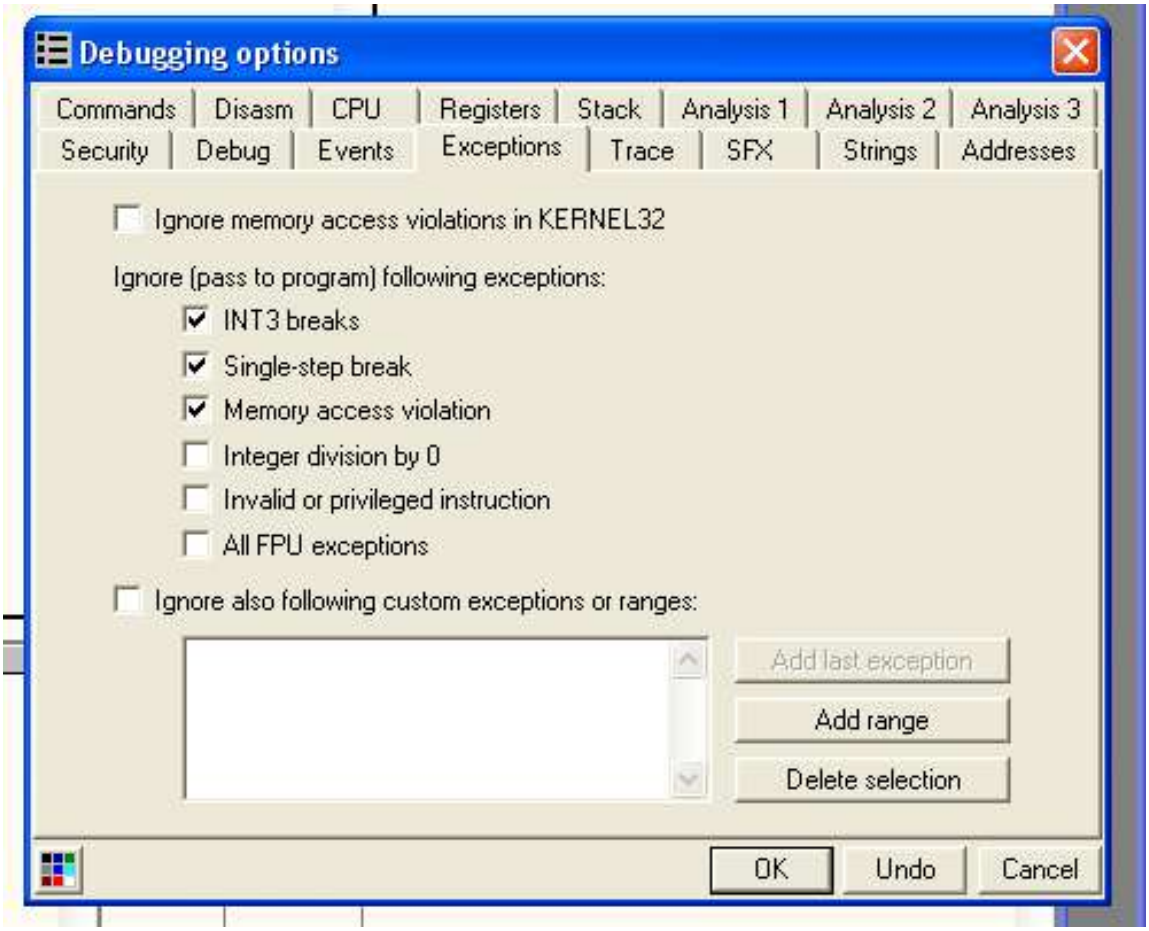
Si maintenant, en plus du INT 3 que vous avez inscrit dans votre code, vous installez une SEH, que va-t-il se passer ? Elle va gérer votre INT 3 ! En mode normal, l'offense sera forcément commise et le programme va continuer à partir de l'offset du handler. En mode debugger, l'offense ne sera pas commise, le programme va continuer son bonhomme de chemin.



Lors du debuggage, Comment franchir une SEH avec Olly ?

Olly est capable de se « camoufler » face à ce genre de piège. Pour effectuer ce camouflage, il suffit d'aller dans :

↳OPTIONS ↳ DEBUGGING OPTIONS ↳ EXCEPTIONS



Pour tElock, seules les 3 cochées ci-dessus suffisent.
Pour franchir la SEH, si vous tracez ligne par ligne avec F8, avant que l'exception soit commise, posez un BP sur l'offset du handler, et lancez l'exécution de cette SEH avec F9...vous vous retrouvez alors sur votre BP et vous pouvez continuer à tracer tranquillement.

Première SEH : Simulation d'un Software Break Point (BPX) par appel de INT 3.

L'offset du handler est 101F4B3

```
101F4A5    PUSH DWORD PTR FS:[EAX]       EAX = 0
101F4AB    MOV DWORD PTR FS:[EAX], ESP
101F4D1    INT 03                        Génère une exception
```

Deuxième SEH : Test de l'IDT.

L'IDT (Interrupt Descriptor Table) contient les adresses des routines des interruptions logicielles. Lorsque le programme rencontre un INT *n*, il regarde le vecteur n de l'interruption et se branche sur l'IDT pour lire l'adresse de la routine à exécuter. Certains debuggers modifient les adresses de l'IDT pour pouvoir exécuter leurs propres routines. (exemple : INT 3 et INT 68 pour Soft Ice). Ces modifications peuvent être repérées en testant l'IDT. Il faut pour cela localiser l'IDT grâce à l'IDTR en utilisant l'instruction SIDT qui stocke le contenu de l'IDTR dans une zone mémoire. Voici un schéma issu des docs Intel qui montre bien la relation entre IDT et IDTR :

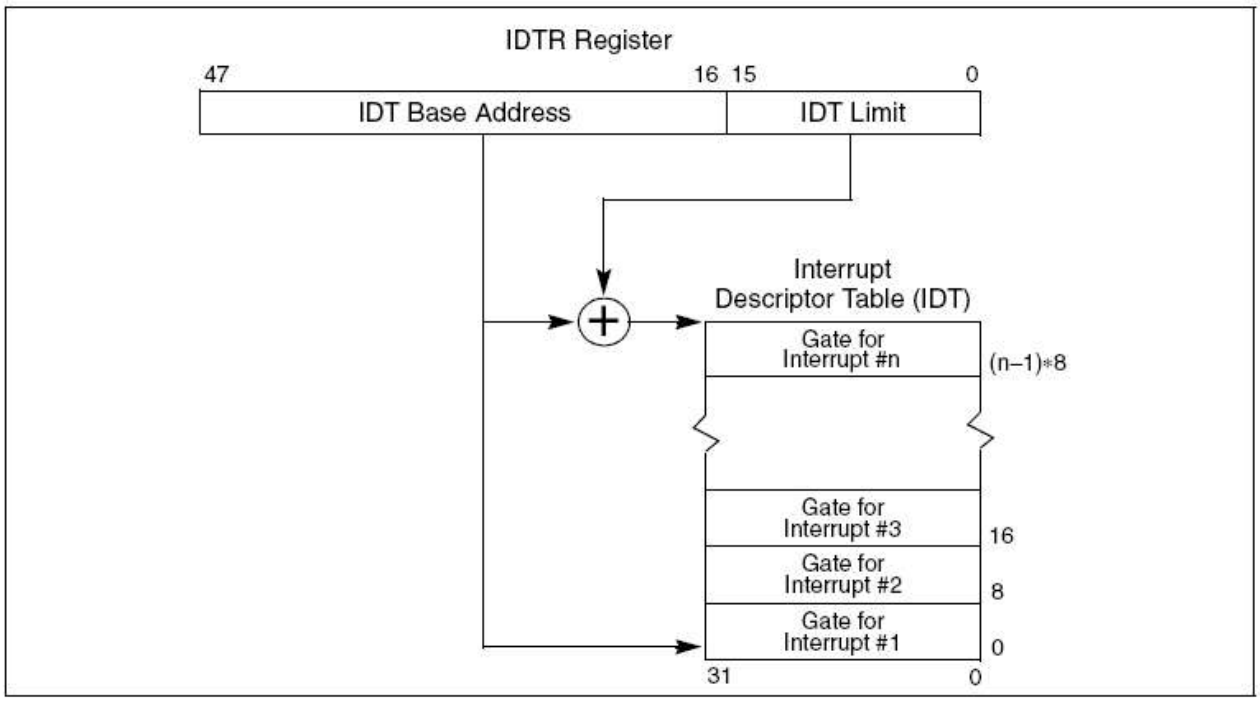


Figure 5-1. Relationship of the IDTR and IDT

Voici le code :

```
101F4EB  MOV DWORD PTR FS:[0], ESP
         SIDT FWORD PTR SS:[EBP+15A6]        Stocke le contenu de l'IDTR en 102061E
         MOV EAX, DWORD PTR SS:[EBP+15A8]    EAX = 8003F400 (adresse de l'IDT)
         ADD EAX, 4E                         EAX = 8003F44E
         MOV BX, WORD PTR DS:[EAX]           Génère une exception sous XP
         SHL EBX, 10
         MOV BX, WORD PTR DS:[EAX + 2]       Scanne l'IDT à la recherche
         MOV EAX, 0C00                       De deux nombres...
         CMP DWORD PTR DS:[EBX], 48455245
         JE 101F6A4
         CMP DWORD PTR DS:[EBX], 53474F52
         JE 101F557
```

```
        INC EBX
        DEC EAX
        JG 101F54E
```

Sous XP, l'accès à l'IDT n'est plus possible et ce test ne concerne finalement que Soft Ice.

Troisième SEH : Single Step (F8)

L'offset du handler est 101F596

Lorsqu'un debugger fait du single step (tracer pas à pas avec F8), il arme le drapeau TF (Trap Flag). On peut forcer cet armement pour simuler un Single Step.

```
101F5B4   PUSHFD                    Envoie le registre EFLAG sur la pile
          OR [ESP],100              TF = 1
          POPFD                     Restaure EFLAG
```

Quatrième SEH : INT 68 (pour détecter Soft Ice)

L'offset du hanlder est 101F5E8

```
101F5DD   MOV DWORD PTR FS:[0], ESP
          MOV AX, 4300              Paramètre transmis à l'interruption
          INT 68                    Génére une exception
```

Avec Olly, il y a une simple exception, avec Soft Ice, l'interruption renvoie une valeur que l'on peut tester.

Cinquième SEH : Option Add SI Detection (BCHK)

Cette SEH fait partie des options que propose tElock. Elle ne servira pas si l'option n'a pas été cochée avant la compression de l'exe.

```
101F63C   MOV DWORD PTR FS:[0], ESP
101F664   TEST BYTE PTR SS[EBP+1397], 0FF   Si [102040F] = 0, alors option cochée
101F66E   JE 101F68E                        Saut si option non cochée
101F673   MOV EBP, 4343484B                 Paramètre transmis à l'interruption
                                            - BCHK-
          MOV AX, 4                         Code de retour
          INT 03
          CMP AX, 4
          JNE WINICE PRESENT
101F68E   POP DWORD PTR FS:[0]
```

4 . L'anti Software Break Point (BPX) (Protection silencieuse)

Le loader est muni d'un système de défense efficace puisqu'il détecte les Breaks Points (BPX) posés avec F2. Ce qui est interéssant ici c'est qu'il ne cherche pas les BP en scannant le loader pour trouver d'éventuels CCh. Si vous posez un BPX, le loader se plante bêtement en générant une exception...

Ce loader est en fait équipé d'un checksum. Il va se scanner entièrement et utiliser tout son code pour générer une clé sur 4 octets. Si le code du loader a été modifié avant le calcul de cette clé (par exemple en posant un BPX), le programme va se planter. Comment se plante-t-il ?

Cette clé est en fait utilisée pour décrypter le programme ! Si la clé est mauvaise, le programme sera mal décrypté et va générer une exception ! C'est ce qu'on appelle une protection silencieuse. Autant dire que pour un cracker débutant, il est assez difficile de localiser un tel dispositif !
Le code de ce checksum est vu plus loin dans la rubrique « décryptage des sections de l'EXE ».

Pour finir, pour tracer ce loader, utilisez des Hardware Break Point (BPM) qui ne sont pas détectés. Ce n'est pas toujours le cas !!

c . Le décryptage/décompression des sections du PE

Cette partie concerne le décryptage puis la décompression de l'exe pour le remettre dans son état d'origine. Ce travail se fait donc en deux passes :

1 . Décryptage des 2 sections grâce à la clé de décryptage.
2 . Décompression des 2 sections en utilisant l'alogorithme d'ApLib v 0.26, utilisé pour aPACK de Joergen Ibsen.

Poue effectuer le décryptage, il faut une clé de décryptage qui est calculée après le premier layer de décryptage.

Calcul de la clé (CRC32): Ce calcul s'effectue en utilisant les valeurs des octets compris entre les offsets 101F000 (début du loader) et 102040E (fin du loader). Ceci signifie que le code ne doit en aucun cas être altéré au risque de fausser le calcul de la clé (attention aux BPX !) C'est une sorte de CheckSum.

Voici le code commenté du calcul de la clé :

ESI = 101F000 (début du loader)

```
101F09D   POP EBX
          MOV EBX, 140E                 Longueur du loader
          XOR ECX, ECX                  Compteur de boucle
          LEA EDX, [ECX – 1h]           EDX = FFFFFFFFh et contiendra la clé
101F0B1   XOR EAX, EAX
          LODSB                         EAX récupère le contenu situé en ESI
          XOR AL, DL
101F0BD   SHR EAX,01h
          JNB 101F0CA                   Saut si carry flag = 0
          XOR EAX, CDB792E1h            Valeur fixe choisie par tE!
          INC ECX
          AND CL, 07h                   Permet de boucler 8 fois
          JNZ 101F0BD
          SHR EDX, 08h
          XOR EDX, EAX
          DEC EBX
101F0EA   JG 101F0B1                    Boucle tant que EBX > 0
101F0FB   MOV DWORD [EBP + 150A], EBX   Stocke la clé calculée en 1020582
```

La première passe : décryptage à l'aide de la clé de décryptage stockée en 1020582.

```
101F778   MOV ESI, EDI                  ESI = 1001000 (première section)
          MOV EBX, clé calculée
          XOR EAX, EAX
101F77E   LODSB                         EAX récupère le contenu situé en ESI
          SUB AL, 043h
          SUB AL, 1h
          XOR AL, CL                    ECX = 7C00 (longueur de la section)
          MOV BYTE PTR [EDI], AL        Stockage en 101F000
          ROR AL, CL
          XOR AL, BL
          ADD BL, BYTE PTR [EDI]
          ADC BL, CL
          TEST CL, 01h                  ECX est impair ?
          JNZ 101F7CC                   Si oui, saute, sinon, continue
          SHR EBX, 01h
          TEST EBX, 4h
          JNZ 101F7CC
          ROL EBX, CL
          LEA EBX, [EBX*8 + EBX]
101F7CC   STOSB                         Replace l'octet modifié en mémoire
          DEC ECX
          JG 101F77E                    Saut si ECX > 0
```

La deuxième passe : Décompression des deux sections.

Cette fois ci, le code est nettement plus long et plus complexe. Il s'agit en faite de l'alogorithme de décompression proposé par Joergen Ibsen, auteur du packer aPACK. Il propose librement ces algorithmes dans l'ApLib, bibliothèque de procédures utilisées pour aPACK. Il s'agit ici de l'ApLib version 0.26 selon Joergen Ibsen.

Je vous livre le code sans commentaires comme il est présenté dans l'ApLib :

```
literal:
    movsb
nexttag:
    call   getbit
    jnb    literal
    xor    ecx, ecx
    call   getbit
    jnb    codepair
    xor    eax, eax
    call   getbit
    jnb    shortmatch
    mov    bl, 2
    inc    ecx
    mov    al, 10h
getmorebits:
    call   getbit
    adc    al, al
    jnb    getmorebits
    jnz    domatch
    stosb
    jmp    short nexttag
codepair:
    call   getgamma_no_ecx
    dec    ecx
    loop     normalcodepair
    call   getgamma
    jmp    short domatch_lastpos

shortmatch:
```

```
    lodsb
    shr    eax, 1
    jz     donedepacking
    adc    ecx, ecx
    jmp    short domatch_with_2inc

normalcodepair:
    xchg   eax, ecx
    dec    eax
    shl    eax, 8
    lodsb
    call   getgamma
    cmp    eax, 7D00h
    jnb    domatch_with_2inc
    cmp    ah, 5
    jnb    domatch_with_inc
    cmp    eax, 7fh
    ja     domatch_new_lastpos

domatch_with_2inc:
    inc    ecx

domatch_with_inc:
    inc    ecx

domatch_new_lastpos:
    xchg   eax, ebp
domatch_lastpos:
    mov    eax, ebp

domatch:
    push   esi
    mov    esi, edi
    sub    esi, eax
    rep    movsb
    pop    esi
    jmp    short nexttag

getbit:
    add    dl, dl
```

```
    jnz     stillbitsleft
    mov     dl, [esi]
    inc     esi
    adc     dl, dl
stillbitsleft:
    ret

getgamma:
    xor     ecx, ecx
getgamma_no_ecx:
    inc     ecx
getgammaloop:
    call    getbit
    adc     ecx, ecx
    call    getbit
    jb      getgammaloop
    ret

donedepacking:
    sub     edi, [esp + 40]
    mov     [esp + 28], edi     ; return unpacked length in eax

    popad
    ret
```

Là, vous n'avez que l'algorithme. Il y a néanmoins un petit travail avant. D'abord, cet algo est répété deux fois (il y a 2 sections) :

```
101F7DC   POP EDI
          POP ESI                                 ESI = 2 (nbre de sections)
          MOV ECX, DWORD PTR [EDI+40E4E4]         ECX = 80007C00 pour la 1ère section
          MOV EAX, DWORD PTR [EDI+40E4E0]         EAX = 1000 (RVA de la première section)
          TEST ECX, 80000000                      Test du bit de poids fort
          JE 101F835                              Saut si section déjà décompressée
          AND ECX, 7FFFFFFF                       Mise à zéro du bit de poids fort
          ADD EAX, DWORD PTR [EBP+40E4C0]         EAX = 1001000
          PUSH EAX                                Offset de la section
          PUSH ECX                                Taille de la section
          CALL 101F84A                            Appel du décompresseur aPACK
          CMP EAX, -1                             Erreur de décompression ?
          JE 101FC39
```

```
101F835   ADD EDI, 8
          DEC ESI
          JG 101F73B
```

Dans le call précédent, juste avant la décompression, le contenu des sections est copié vers des zones mémoires virtuelles comme suit :

1 . D'abord, avec la fonction VirtuelAlloc, le loader réserve une nouvelle zone mémoire vide.

2 . Puis, il copie intégralement la première section de l'exe dans cette zone mémoire.

3 . Ensuite, il décompresse la zone mémoire allouée dans la première section.

4 . Enfin, avec la fonction VirtualFree, il libère la zone mémoire allouée.

5 . Il recommence avec la deuxième section.

d . Techniques anti-unpacking/anti-dump

1 . le mutex

Apparemment, ce fût une grande première dans les techniques anti-unpacking. La technique est astucieuse : Le loader crée un mutex auquel on a donné un nom au moment du packing.(par exemple : Mon_Mutex_A_MOI). Quand le loader a fini son travail, il passe la main au programme.

Là, le programme doit tester la présence du mutex pour que cette protection serve à quelquechose. Si le mutex n'est pas présent, ou s'il n'a pas le nom Mon_Mutex_A_MOI, c'est que le programme s'est fait unpacker à la main ou par un unpacker générique. Vous pouvez imaginer les conséquences : ExitProcess dans le meilleur des cas, mauvaise blague du programmeur dans le pire !! Dans les deux cas, il faudra virer le test du mutex dans l'exe par la suite ! Un conseil : Faites gaffe aux Keygen de TMG compressés par tElock ! ;-)

Voici ce que préconise tE! Pour tester le mutex :

Example code for your check(s):

```
ASM (TASM):
-----------
.DATA
    mymutex db "YourMutexStringHere",0        ; use same string in tElock!
.CODE
```

```
call    CreateMutexA, 0, 0, offset mymutex
call    GetLastError
cmp     eax, ERROR_ALREADY_EXISTS
jnz     MyEvilRoutine
```

DELPHI:

```
CreateMutex(nil, False, 'YourMutexStringHere');
if (GetlastError() <> ERROR_ALREADY_EXISTS) then close;
```

C:

```
CreateMutex(NULL, FALSE, "YourMutexStringHere");
if (GetLastError() != ERROR_ALREADY_EXISTS) {
  YourEvilRoutinesHere;
  ...
}
```

Protection utilisable par n'importe quel programmeur !!

2 . effacement du loader

Il ne s'agit pas à proprement parler d'un anti-unpacking. Il s'agit ici de faire disparaître le contenu du loader une fois l'exe décompressé. On ne peut donc pas au moment du dump récupérer le loader décrypté...

```
        LEA EDI, DWORD [EBP + 40E372]    ESI = 1020420 (début de l'effacement)
        XOR EAX, EAX
        MOV ECX, 24B                     Longueur de l'effacement
        REP STOS BYTE ES:[EDI]           1ère effacement
        LEA EDI, DOWRD [EBP + 40CF58]    EDI = 101F006 (début de l'effacement)
        MOV ECX, 13F5                    Longueur de l'effacement
        REP STOS BYTE ES:[EDI]           2ème effacement
        STOS WORD ES:[EDI]               ...
        POPAD

102040A JMP WORD [ESP-24]                Saut vers l'OEP en 4012475
```

3 . modification du header

En fait ce code modifie simplement l'ImageSize du programme en mémoire. Ainsi les

programmes de dump sont plantés car soit ils lisent trop de mémoire soit pas assez. Dans ce cas précis, on fait croire que l'Imagesize = 0 !

Voici le principe expliqué par ***Karneth*** :

« D'abord on récupère l'adresse du PEB (Process Environnement Block):

```
MOV EAX, DWORD FS:[30].....................EAX = 7FFDF000
```

Dont voici le début de la structure:

```
typedef struct _PEB {
/*000*/ BOOLEAN  InheritedAddressSpace;
/*001*/ BOOLEAN   ReadImageFileExecOptions;
/*002*/ BOOLEAN   BeingDebugged;
/*003*/ BOOL          SpareBool; // alloc size
/*004*/ HANDLE     Mutant;
/*008*/ PVOID        SectionBaseAddress;
/*00C*/ PPROCESS_MODULE_INFO ProcessModuleInfo;
/*010*/ PPROCESS_PARAMETERS  ProcessParameters;
/*...*/
} PEB, *PPEB;
```

Ensuite on récupère l'adresse du ProcessModuleInfo:

```
MOV EAX, DWORD PTR DS:[EAX + C]...........EAX = 191EA0
```

Dont voici la structure:

```
typedef struct _PROCESS_MODULE_INFO {
/*000*/ DWORD    Size;
/*004*/ DWORD    Initalized;
/*008*/ HANDLE    SsHandle;
/*00C*/ LIST_ENTRY ModuleListLoadOrder;
/*014*/ LIST_ENTRY ModuleListMemoryOrder;
/*018*/ LIST_ENTRY ModuleListInitOrder;
/*020*/ } PROCESS_MODULE_INFO, *PPROCESS_MODULE_INFO;
```

Puis enfin l'adresse du ModuleListLoadOrder:

```
MOV EAX, DWORD PTR DS:[EAX + C]...........EAX = 191EE0
```

Dont voici la structure:

```
typedef struct _MODULE_ITEM {
/*000*/ LIST_ENTRY   ModuleListLoadOrder;
/*008*/ LIST_ENTRY    ModuleListMemoryOrder;
/*010*/ LIST_ENTRY    ModuleListInitOrder;
/*018*/ DWORD          ImageBase;
/*01C*/ DWORD          EntryPoint;
/*020*/ DWORD          ImageSize;
/*024*/ UNICODE_STRING PathFileName;
/*02C*/ UNICODE_STRING FileName;
/*034*/ ULONG          ModuleFlags;
/*038*/ WORD           LoadCount;
/*03A*/ WORD           Fill;
/*03C*/ DWORD          dw3c;
/*040*/ DWORD          dw40;
/*044*/ DWORD          TimeDateStamp;
/*048*/ } MODULE_ITEM, *PMODULE_ITEM;
```

A l'adresse +20h on trouve la valeur de ImageSize qu'il suffit de remplacer par ce qu'on veut. »

Voici donc le petit code qui permet ce miracle :

```
102014E   ADD AH, 4B
          JNZ 10202DA                        Saut si option Anti-dump non cochée.
102015D   MOV EAX, DWORD FS:[30]             EAX = 7FFDF000
          TEST EAX, EAX
          JS 102018B
          MOV EAX, DWORD PTR DS:[EAX + C]    EAX = 191EA0
          MOV EAX, DWORD PTR DS:[EAX + C]    EAX = 191EE0
          MOV DWORD PTR DS:[EAX + 20], 0     Remplace 21000 par 0.

10202DA   POP DWORD PTR FS:[0]               Fin de la SEH qui protégeait le code.
```

« Pour contrer les anti-procdump, on met un bmp fs:30 et on lance jusqu'à ce qu'il breake puis on trace un peu pour tomber sur **js xxxxxx** » ***Christal***

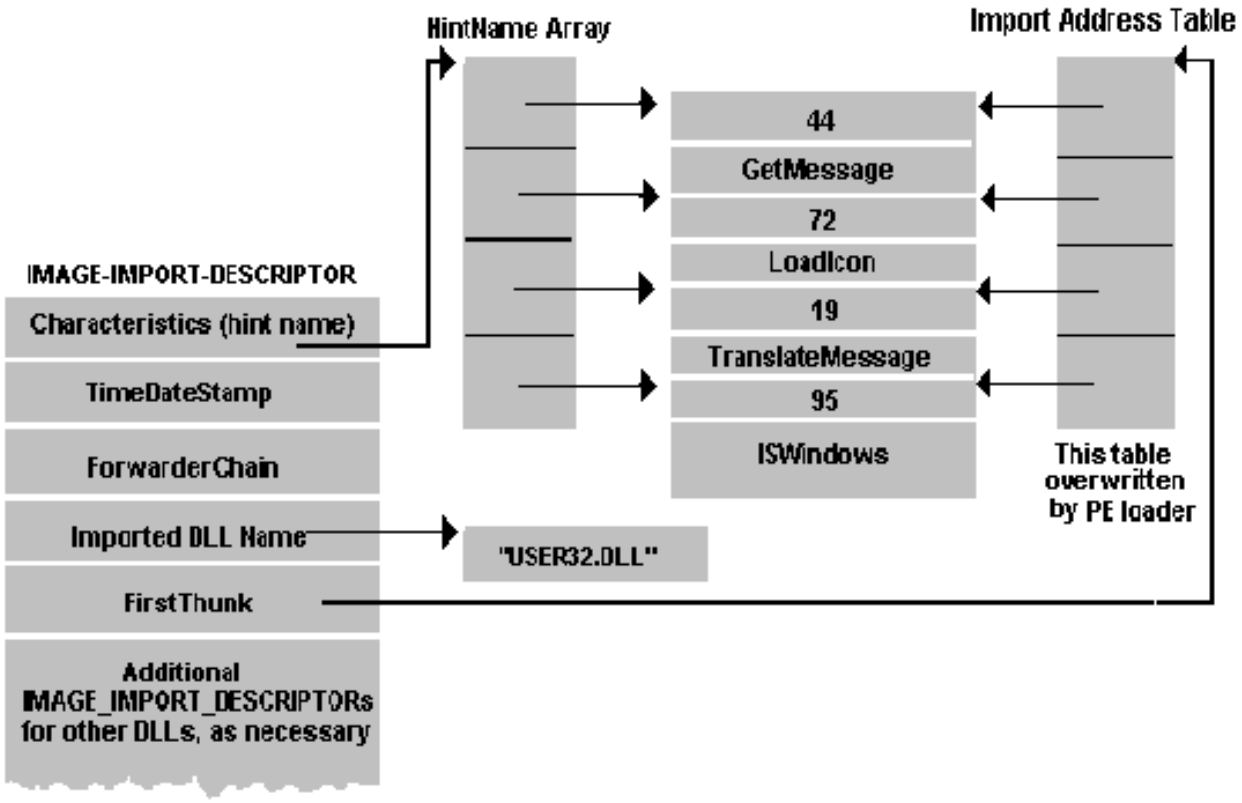
4 . redirection de l'IAT

Heu, c'est quoi l'IAT m'sieur ?

L'IAT (Imports Adress Table) contient les adresses de toutes les APIs utilisées par l'exe. Ce tableau est rempli à chaque lancement de l'exe grâce aux noms des APIs qui se

trouvent dans la section des imports. En gros, au démarrage, le loader de windows liste les apis de l'exe, et pour chacune d'elle il va chercher l'adresse de sa routine pour la stocker dans l'IAT. (Si on dumpe le programme quand il est chargé en mémoire, on copie intégralement l'IAT avec les adresses des APIs.)

Voici un schéma issu de « Peering inside the PE » de Matt PIETREK (mars 1994) qui illustre bien le rôle de l'IAT :

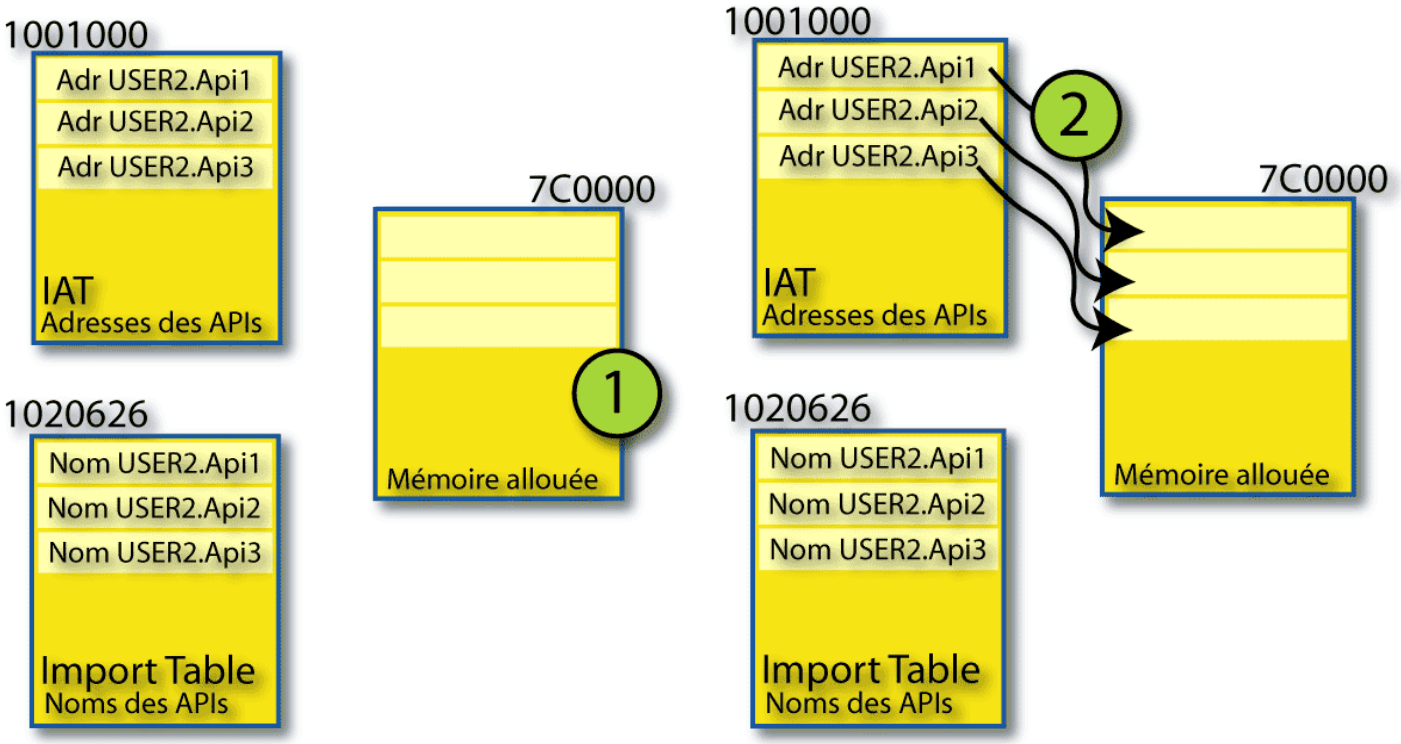


Le rôle de ImportReconstructor est de justement récupérer les adresses stockées dans l'IAT pour déterminer les APIs utilisées. La redirection de l'IAT permet de mettre partiellement cette technique en échec. L'idée est simple : L'IAT ne contient plus les adresses des APIs mais pointe vers des zones mémoires qui contiennent les bonnes adresses. Du coup, ImportReconstructor ne parvient pas à résoudre les imports directement ! Dans le cas présent, quasiment toutes les APIs sont redirigées. Nous verrons plus bas comment résoudre ce problème de redirection !

A cela s'ajoute un effacement pur et simple de tous les Imports au fur et à mesure que le loader de tElock récupère les adresses. On ne peut donc pas non plus utiliser ces noms pour reconstruire l'Import Table.

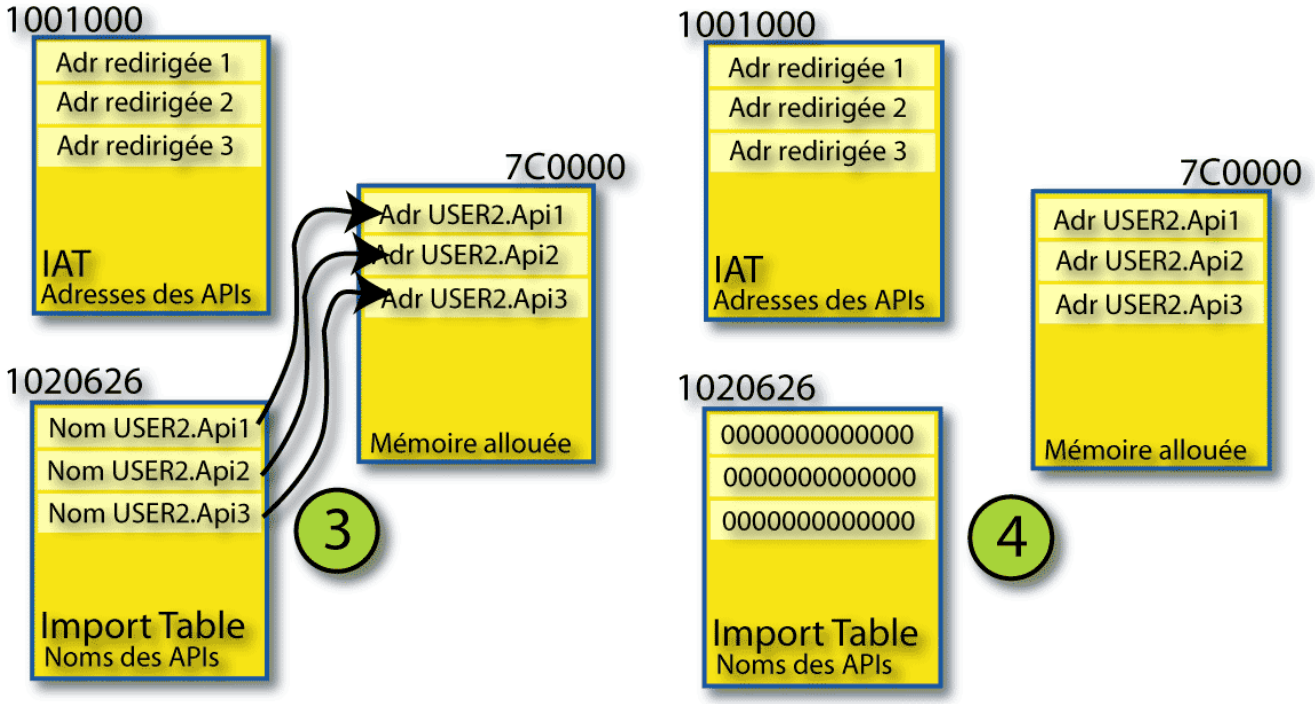
Voici illustré le procédé pour rediriger l'IAT.

On imagine que notre programme n'utilise que 3 apis appelées api1, api2 et api3 issues de USER32.DLL. La première étape consiste à allouer une zone mémoire grâce à la commande VirtualAlloc.



La deuxième étape consiste à faire pointer les adresses des apis vers la nouvelle zone mémoire allouée.

La troisième étape consiste à récupérer les adresses des Apis à partir de l'import table grâce à la commande GetProcAdress et de les coller dans la zone allouée.



La quatrième étape consiste à effacer les noms des Apis dans l'import !

Pour chaque Dll, ces 4 étapes seront réalisées dans des zones mémoires différentes. Je ne vous donne pas le code qui est en réalité très simple mais assez rigolo à tracer.

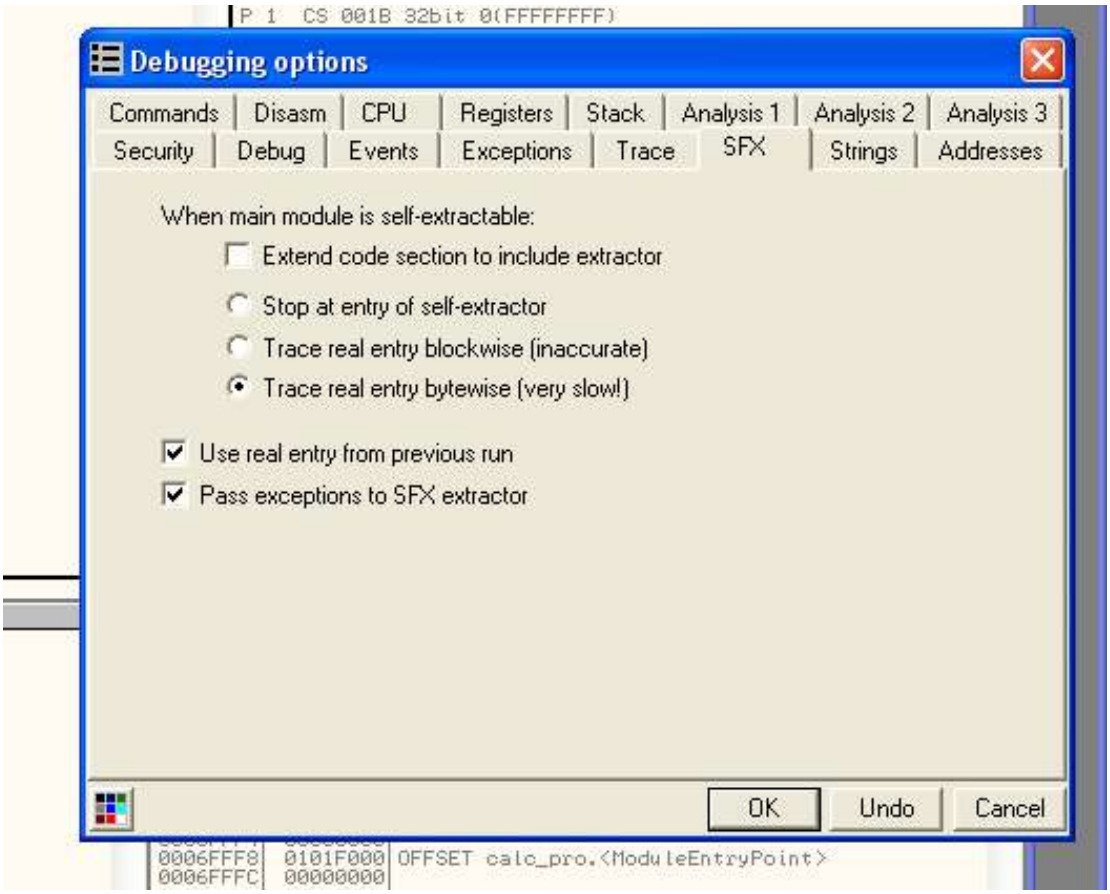
4 . MANUAL UNPACKING

Je rappelle les étapes du manual unpacking :

1) Chercher l'OEP (Original Entry Point)
2) Dumper le programme en se plaçant sur le saut qui nous envoie à l'OEP.
3) Changer l'EP dans le header.
4) Restaurer les imports.

Bien sûr, ça, c'est la théorie ! Dans la pratique, chaque étape peut être problématique.

1) Pour trouver l'OEP, on peut décortiquer le loader ligne par ligne (comme je l'ai fait ici) mais on peut essayer de s'économiser un peu en cherchant l'OEP par la technique la plus rapide possible. Ici, le module SFX (Self Extractor) de OllyDebugger me semble être une bonne méthode.



2) Pour dumper le programme, on a vu qu'il existait des protections : modif du header et surtout le mutex !!). Bon, une fois ces deux soucis écartés, on dumpe avec ProcDump. LordPE ne fonctionne pas bien car il ne nous permet pas de fixer les imports !

3) Si l'option « redirection de l'IAT » n'a pas été cochée lors du packing de l'exe, toutes les adresses de l'IAT sont valides et avec ImportReconstructor, il est aisé de les fixer dans le fichier dumpé.
Si l'option a été cochée, ImportReconstructor ne résout pas les imports.

L'IAT a été modifiée pour qu'elle ne contienne plus les adresses des APIs d'origine. Elle pointe en fait vers une autre « IAT » . Cette nouvelle IAT contient les bonnes adresses.

Pour restaurer les imports avec ImportReconstructor, il suffit d'utiliser le **traceur de niveau 3 (Trap Flag)** sur chaque apis erronée. Attention, certaines apis peuvent mal réagir avec un Trace level 3, allez y doucement ! Pour la calculatrice windows, il y a quand même 133 apis redirigées sur les 136 utilisées !!

5 . REMERCIEMENTS/SOURCES

Voici mes sources :

1) te_unpack.asm de r!sc (dur si vous ne lisez pas l'asm !) (site Unpacking gods)
2) un-te.asm de Cyber Daemon (idem)
3) Les SEH de Christal (dossier6.zip de groupe de travail) (site de Christal)
4) Peering inside the PE de Matt PIETREK (site microsoft)
5) Documentation Intel sur l'IA-32. (site Intel)

MERCI à DAEMON et R!SC qui nous ont laissé le code commenté de leur unpacker de tElock 0.51

MERCI à Gbillou pour m'avoir mis sur la piste de aPACK pour la partie unpacker.

MERCI à Karneth pour l'explication de l'anti-dump.

MERCI à Joergen Ibsen pour m'avoir fourni la version de l'ApLib utilisée.

MERCI à Kaine qui m'a énormément aidé à comprendre le fonctionnement des anti-debuggers.

MERCI à tous ceux qui contribuent au développement de la connaissance en matière de cracking.

Août 2004 - BeatriX