Comprendre tElock 0.98 et private versions.

par BeatriX

1 . INTRODUCTION / AVERTISSEMENTS..... 3

2 . LE LOADER DE TELOCK 0.98.....5

a . Schéma du loader sans les protections..... 5
b . Schéma du loader avec les anti-dumps/anti-debuggers..... 6
c . Schéma complet du loader..... 7
d . Reconstruction de l'IAT du loader..... 8
e . Reconstruction de l'IAT de l'exe.....13
f . Techniques d'anti-debuggage/désassemblage..... 14

1 . Le CCA.....14
2 . Les Obfuscations.....14
3 . Les layers de décryptage.....15
4 . L'anti Software Break Point (BPX) (Protection silencieuse).....15
5 . L'anti Software Break Point sur APIs.....15
6 . L'anti Hardware Break Point (BPM) 16
a) Les Debugs Registers.....
b) Le CONTEXT.....
c) Passage de Ring 3 à Ring 0 par SEH.....
L'EXCEPTION_RECORD et le CONTEXT record.....
7 . CheckSum du header.....25

g **. Le décryptage/décompression des sections du PE..... 28**

0 . Calcul du CRC32, clé de décryptage..... 28
1 . Premier décryptage.....29
1 . Deuxième décryptage par clé..... 31
2 . Décompression suivant l'algorithme de ApLib v 0.26..... 31

h **. Techniques anti-unpacking/anti-dump.....31**

1 . le mutex..... 31
2 . effacement du loader.....31

3 . modification du header.....31
Modifier le nombre de sections avec VirtualProtectEx.....
Modifier l'ImageSize en accédant au PEB.....
4 . redirection de l'IAT/ Effacement des imports..... 32

3 . LE LOADER DES VERSIONS PRIVATE.....34

a . Schéma du loader simplifié.....34
b . Technique anti-debugging..... 34

l'API GetClassNameA..... 34
(patcher OllyDbg pour contourner l'obstacle)

4 . MANUAL UNPACKING..... 36

a . Trouver l'OEP.....36

Trouver l'OEP avec LordPE .....
Utiliser ShaOllyScript plugin v0.92 by ShaG.....
Utiliser le script de loveboom pour les versions 1.xx.....

b . Dumper l'exe..... 39
Comparer les dumps faits avec ProcDump – OllyDump – LordPE
(*Dans les entrailles de ProcDump 1.6f)*
c . Reconstruire les imports..... 41

5 . REMERCIEMENTS/SOURCES..... 42

1 . INTRODUCTION / AVERTISSEMENTS

Je vous présente la suite de mon étude sur le packer tElock. Il ne s'agit pas d'une répétition du travail sur tElock 0.51 et pour preuve, je considère comme acquis ici toutes les notions vues dans le précédent article. Ceci s'adresse donc à des crackers d'un niveau intermédiaire qui ont quelques bases en unpacking.

L'étude porte principalement sur tElock 0.98, dernière version publique de ce petit packer de PE. En étudiant certains Keygens récents de TMG (septembre 2003), on peut constater qu'ils sont protégés par une version private de tElock que l'on baptisera ici tElock 1.xx. Nous étudierons également les ajouts de ces versions 1.xx.

J'ai d'abord axé mon travail sur la reconstruction des imports (contrairement à la version 0.51 où je n'en ai pas du tout parlé). J'ai ensuite travaillé sur les nouvelles techniques d'anti-debuggage, à savoir les ANTI-BPM, Les OBFUSCATIONS, les détections de certains process via les noms de leurs classes. J'ai encore une fois étudié calc.exe que j'ai packé en modifiant les options suivant les effets voulus.

Avant de commencer, je vous propose un bref historique des différentes versions publiques de tElock :

???? :	Version tElock 0.41b	
???? :	Version tElock 0.41c	
20 septembre 2000 :	Version tElock 0.42	
05 octobre 2000 :	Version tElock 0.51	version étudiée précédemment
12 décembre 2000 :	Version tElock 0.60	
27 décembre 2000 :	Version tElock 0.70	
28 décembre 2000 :	Version tElock 0.71	
6 avril 2001 :	Version tElock 0.80	
11 juillet 2001 :	Version tElock 0.85f	
18 juillet 2001 :	Version tElock 0.90	
6 octobre 2001 :	Version tElock 0.92a	
16 octobre 2001 :	Version tElock 0.95	
18 octobre 2001 :	Version tElock 0.96	
26 octobre 2001 :	Version tElock 0.98	dernière version publique.

A cette petite liste s'ajoutent les versions privates qui comme leur nom l'indique ne sont pas accessibles au public. Il existe de source sûr les versions 0.81, 0.84, 0.99...introuvables sauf peut-être sur certains keygens de TMG !

Les changements d'une version à l'autre ne diffèrent que par la place des procédures, le nombre de layers, le type de SEH. On note quand même à partir de la

version 0.98 un anti-BPM. A cela s'ajoute, sur les dernières versions privates de tElock, un Anti-Olly, Anti-WinDasm, Anti-FileMon, Anti-DéDé.

Comment repérer tElock ?

Pour chaque version publique de tElock, il existe un unpacker. De plus, PEID détecte toutes les versions et les unpacke aisément. En gros, il n'y a quasiment aucun intérêt pratique à étudier ce packer sauf peut-être pour unpacker les private versions ! Le schéma de toutes les versions de tElock est toujours le même : chaque section de l'exe est cryptée, compressée et renommée à l'exception des ressources (sauf si l'option est cochée). Le loader est installé dans une section ajoutée à la fin de l'exe. On peut repèrer facilement tElock aux noms des sections. Elles sont nommées de 3 façons possibles :

Premier cas :

Nom de la section 1 : 1574859 (nombre aléatoire)
Nom de la section 2 : 8458791
Nom de la section 3 : .rscr (ressources)
Nom de la section 4 : .data (loader)

Deuxième cas :

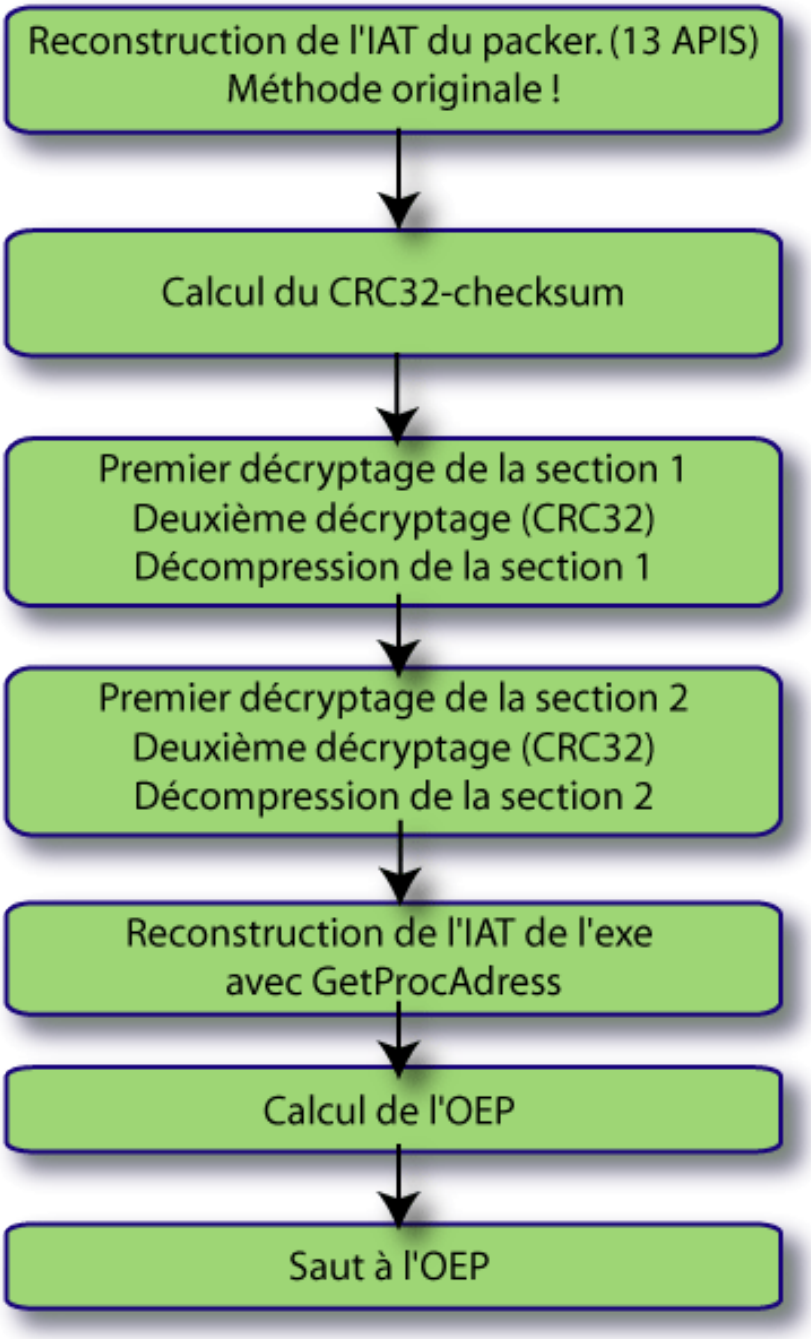
Nom de la section 1 : PEPACK!! (nom d'un packer choisi aléatoirement)
Nom de la section 2 : PEPACK!!
Nom de la section 3 : .rscr (ressources)
Nom de la section 4 : PEPACK!! (loader)

Autres cas : Les sections n'ont pas de noms ou les sections ont toutes le même nom. En réalité, tout ceci dépend de l'option choisie au moment de la compression.

2 . LE LOADER DE TELOCK 0.98

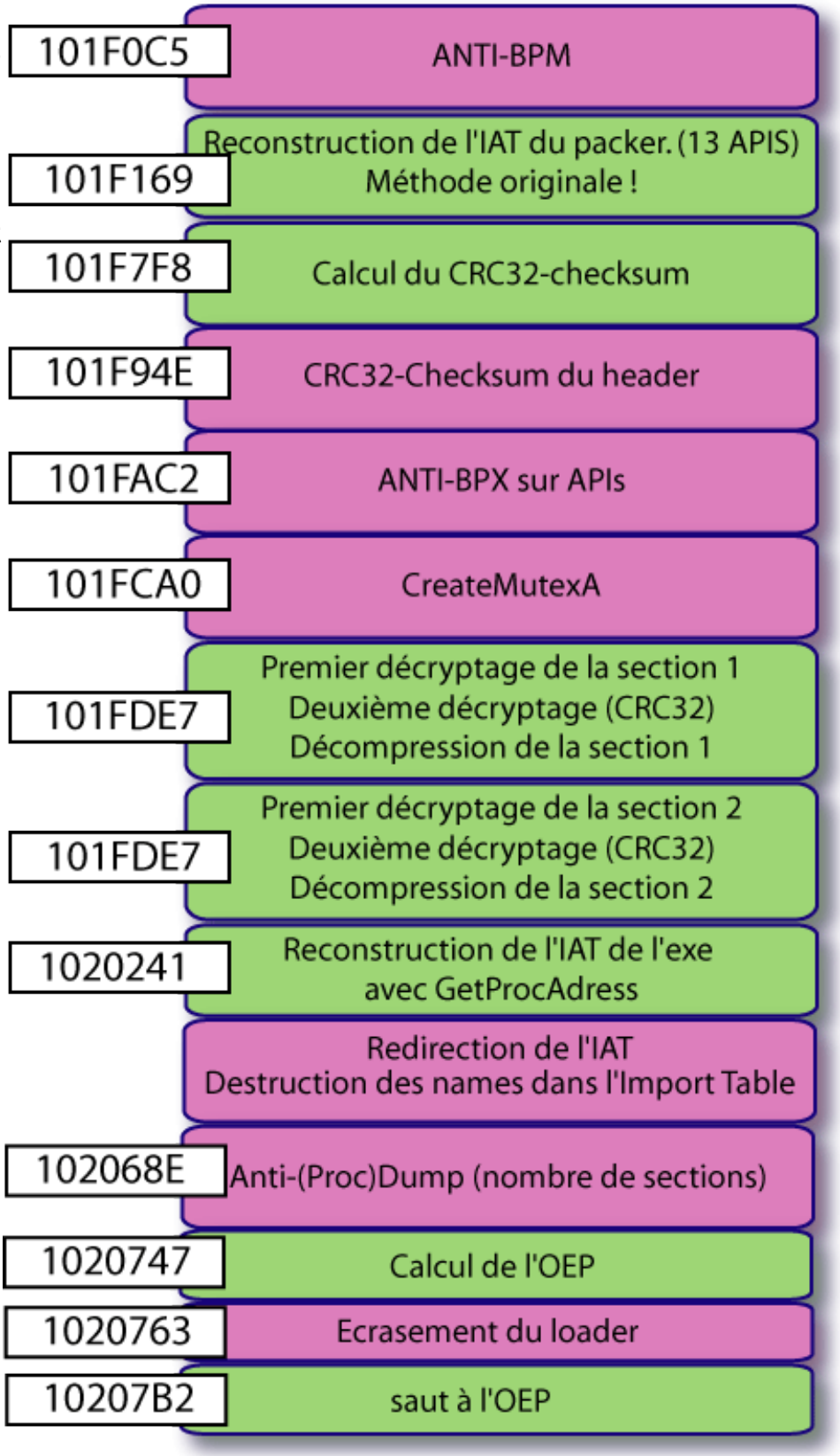
a . Schéma du loader sans les protections

Je vous présente ici l'ossature du loader indispensable pour unpacker et décrypter le PE. Comme je l'ai dit dans l'introduction, il n'y a pas beaucoup de changements par rapport aux versions précédentes. L'ordre des procédures change mais le code reste le même. A noter un petit ajout : les sections sont décryptées deux fois : une première fois avec un layer standard et une deuxième fois en utilisant le CRC32 calculé.



b . Schéma du loader avec les anti-dumps/anti-debuggers

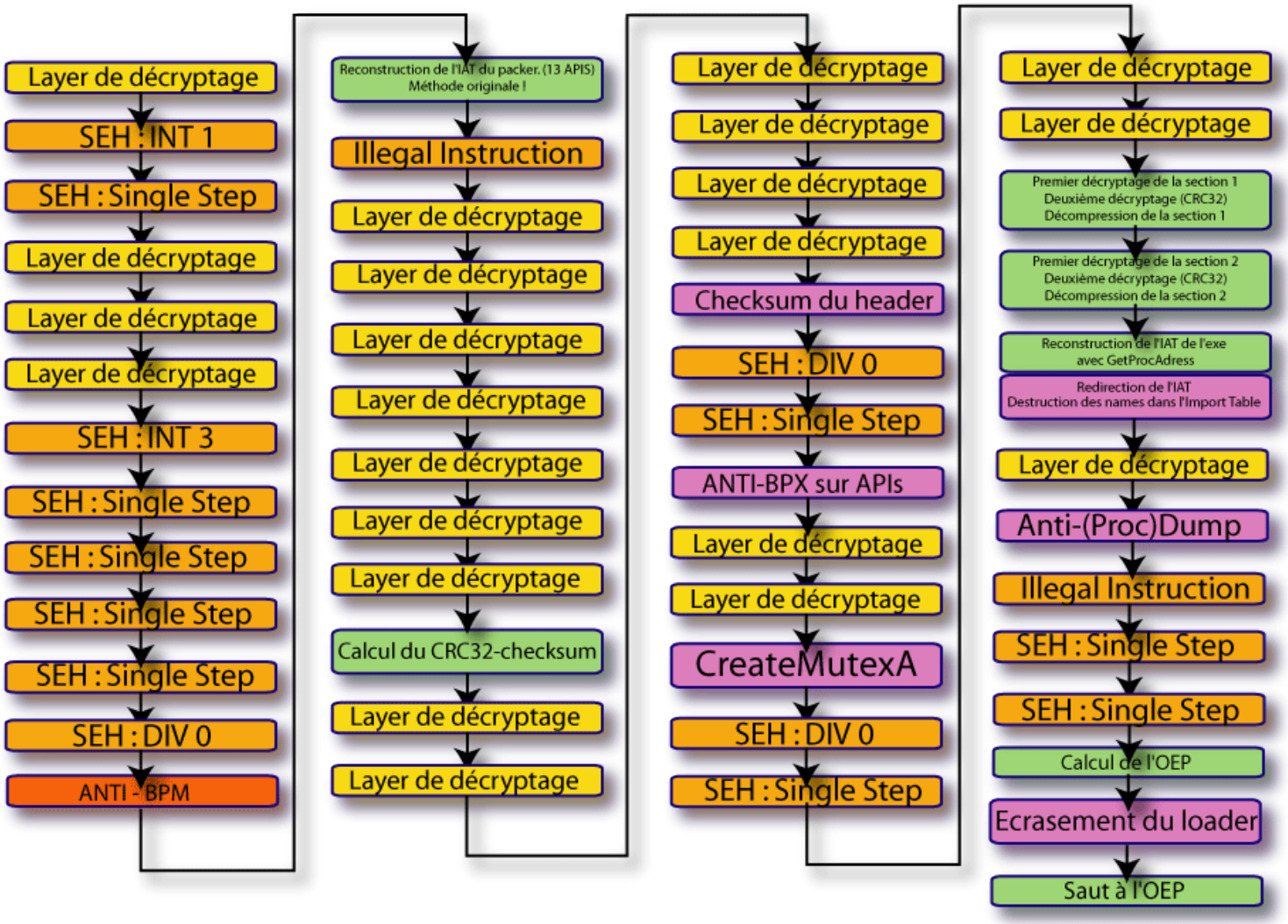
Ici, je vous propose le schéma précédent auquel j'ai ajouté en mauve les techniques anti-unpacking / anti-dump – anti-debugging qui existent en grande partie déjà dans les versions précédentes. Même remarque que précédemment, seul l'ordre des procédures change d'une version à l'autre.



c . Schéma complet du loader

Je vous présente ici le schéma complet du loader juste pour que vous puissiez vous rendre compte de la part occupée par les layers de décryptage et les SEH. Toutes les parties colorées en jaune / orangé sont inutiles au loader et ne servent qu'à ralentir le cracker. Vous constatez que le loader est carrément pollué par ce code ajouté. Il s'agit vraiment d'une nouveauté pour cette version. A cela s'ajoute le CCA et les OBFUSCATIONS qui ne facilitent pas le travail non plus ! Un ajout est à noter : En bas à gauche du schéma, vous voyez ANTI-BPM. Il s'agit cette fois d'empêcher les Hardware Break Points en plus des Software Break Points (BPX).

Evidemment, le schéma complet de ce loader tient sur une seule page. Certaines protections de PE plus robustes utilisent des centaines voire des milliers de layers, autant d'anti-debuggers et de SEH....impossible à tracer manuellement !



d . Reconstruction de l'IAT du loader

Je commence cette fois par vous expliquer comment le loader reconstruit les IATs. Car effectivement il a deux IATs à reconstruire, celle de l'exe packé et la sienne. Commençons par celle du loader.

Si, avant de désassembler l'exe, vous jetez un oeil à l'import table, vous voyez uniquement deux APIs :

USER32.MessageBoxA
KERNEL32.GetModuleHandleA

Le reste des APIs nécessaires est donc chargé dans le loader. En réalité, il manque à la liste précédente les 13 APIs suivantes :

Pour remplir l'IAT de l'exe
LoadLibraryA : Charge l'image Base de la dll voulue .
GetProcAdress : Charge l'adresse d'une fonction de l'API.

no comment...
ExitProcess : ferme le process en cours.

Pour effectuer la décompression
VirtualAlloc : alloue un segment de taille définie
VirtualFree : libère le segment alloué par VirtualAlloc

Pour empêcher le unpacking
CreateMutexA : Crée un mutex.

Pour mettre en échec ProcDump (modification du header)
GetCurrentProcessID : Renvoie l'ID du process en cours.
OpenProcess : Ouvre un Process.
VirtualProtectEx : Change les attributs des sections d'un process.

Pour empêcher la modification du header (caractéristiques des sections...)
GetModulefileNameA : Renvoie le chemin du fichier spécifié.
CreateFileA : Ouvre un fichier spécifique
ReadFile : Lit le contenu d'un fichier spécifique.
CloseHandle : Ferme un objet par son handle.

On dispose déjà de l'adresse d'entrée de KERNEL32.DLL. Pour récupérer les adresses de ces treize fonctions, le packer va lister toutes les fonctions de kernel32.dll et ne va retenir que les 13 précédentes. Pour identifier les fonctions voulues, le loader

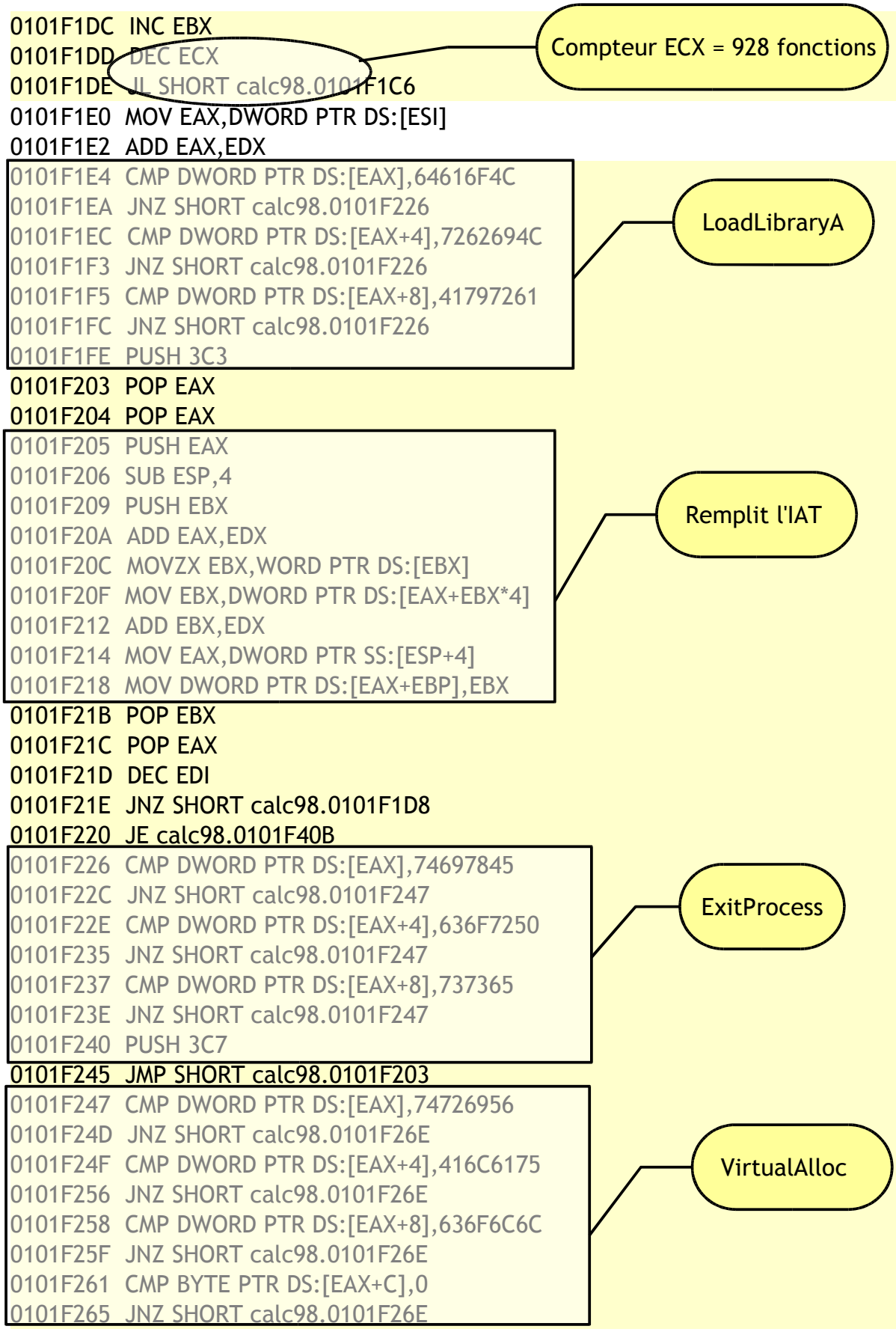
scanne chaque fonction à la recherche d'une sorte de signature.

Par exemple, pour ExitProcess, le loader va chercher l'API qui commence par le code héxadécimal suivant : 74697845636F7250737365. Il s'agit bien du code héxa du début de la fonction ExitProcess. Et il va faire cela pour les treize fonctions ci-dessus. A chaque fonction trouvée, il stocke l'adresse dans l'IAT, au bon endroit !

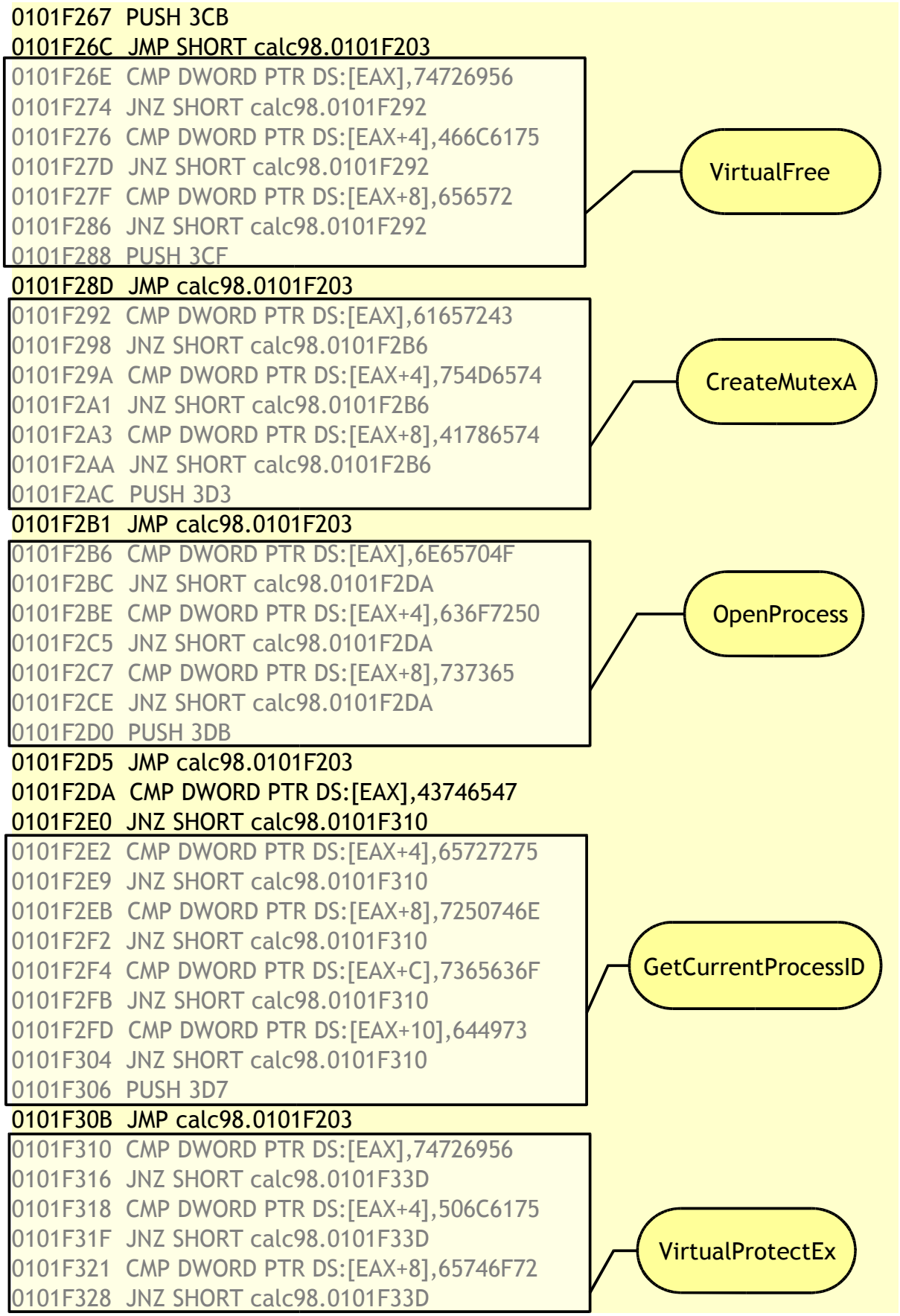
Voici le code commenté de cette petite manipulation :

```
0101F169  MOV EDX,DWORD PTR SS:[EBP+1BAF]      ; GetModuleHandleA
0101F16F  AND EDX,FFFF0000                     ; Récupère octets de poids fort
0101F175  MOV EAX,ESP
0101F177  XOR ESP,ESP
0101F179  MOV ESP,EAX
0101F17B  CMP WORD PTR DS:[EDX],5A4D           ; 2 premiers octets = « MZ » ?
0101F180  JE SHORT calc98.0101F18A             ; Recherche l'adresse du header de Kernel32.dll
0101F182  ADD EDX,FFFF0000
0101F188  JMP SHORT calc98.0101F17B
0101F18A  MOV EAX,DWORD PTR DS:[EDX+3C]        ; RVA du début du PE
0101F18D  CMP EAX,200
0101F192  JA SHORT calc98.0101F17B
0101F194  CMP DWORD PTR DS:[EAX+EDX],4550      ; 2 octets = « PE » ?
0101F19B  JNZ SHORT calc98.0101F17B
0101F19D  MOV EAX,DWORD PTR DS:[EAX+EDX+78]    ; RVA de l'export Table
0101F1A1  MOV EBX,DWORD PTR DS:[EAX+EDX+1C]
0101F1A5  PUSH EBX
0101F1A6  MOV EBX,DWORD PTR DS:[EAX+EDX+24]
0101F1AA  MOV ESI,DWORD PTR DS:[EAX+EDX+20]    ; Nombre de fonctions dans Kernel32 = 928 fonctions
0101F1AE  MOV ECX,DWORD PTR DS:[EAX+EDX+18]
0101F1B2  MOV EAX,DWORD PTR DS:[EAX+EDX+C]
0101F1B6  ADD EAX,EDX
0101F1B8  MOV EAX,DWORD PTR DS:[EAX]
0101F1BA  AND EAX,5F5F5F5F
0101F1BF  CMP EAX,4E52454B
0101F1C4  JE SHORT calc98.0101F1CD
0101F1C6  PUSH 0FEEB
0101F1CB  JMP ESP
0101F1CD  LEA ESI,DWORD PTR DS:[ESI+EDX-4]     ; Compteur EDI = 13 fonctions
0101F1D1  LEA EBX,DWORD PTR DS:[EBX+EDX-2]
0101F1D5  PUSH 0D
0101F1D7  POP EDI
0101F1D8  ADD ESI,4
0101F1DB  INC EBX
```

```
0101F1DC  INC EBX
0101F1DD  DEC ECX
0101F1DE  JL SHORT calc98.0101F1C6
0101F1E0  MOV EAX,DWORD PTR DS:[ESI]
0101F1E2  ADD EAX,EDX
0101F1E4  CMP DWORD PTR DS:[EAX],64616F4C
0101F1EA  JNZ SHORT calc98.0101F226
0101F1EC  CMP DWORD PTR DS:[EAX+4],7262694C
0101F1F3  JNZ SHORT calc98.0101F226
0101F1F5  CMP DWORD PTR DS:[EAX+8],41797261
0101F1FC  JNZ SHORT calc98.0101F226
0101F1FE  PUSH 3C3
0101F203  POP EAX
0101F204  POP EAX
0101F205  PUSH EAX
0101F206  SUB ESP,4
0101F209  PUSH EBX
0101F20A  ADD EAX,EDX
0101F20C  MOVZX EBX,WORD PTR DS:[EBX]
0101F20F  MOV EBX,DWORD PTR DS:[EAX+EBX*4]
0101F212  ADD EBX,EDX
0101F214  MOV EAX,DWORD PTR SS:[ESP+4]
0101F218  MOV DWORD PTR DS:[EAX+EBP],EBX
0101F21B  POP EBX
0101F21C  POP EAX
0101F21D  DEC EDI
0101F21E  JNZ SHORT calc98.0101F1D8
0101F220  JE calc98.0101F40B
0101F226  CMP DWORD PTR DS:[EAX],74697845
0101F22C  JNZ SHORT calc98.0101F247
0101F22E  CMP DWORD PTR DS:[EAX+4],636F7250
0101F235  JNZ SHORT calc98.0101F247
0101F237  CMP DWORD PTR DS:[EAX+8],737365
0101F23E  JNZ SHORT calc98.0101F247
0101F240  PUSH 3C7
0101F245  JMP SHORT calc98.0101F203
0101F247  CMP DWORD PTR DS:[EAX],74726956
0101F24D  JNZ SHORT calc98.0101F26E
0101F24F  CMP DWORD PTR DS:[EAX+4],416C6175
0101F256  JNZ SHORT calc98.0101F26E
0101F258  CMP DWORD PTR DS:[EAX+8],636F6C6C
0101F25F  JNZ SHORT calc98.0101F26E
0101F261  CMP BYTE PTR DS:[EAX+C],0
0101F265  JNZ SHORT calc98.0101F26E
```



```
0101F267  PUSH 3CB
0101F26C  JMP SHORT calc98.0101F203
0101F26E  CMP DWORD PTR DS:[EAX],74726956
0101F274  JNZ SHORT calc98.0101F292
0101F276  CMP DWORD PTR DS:[EAX+4],466C6175
0101F27D  JNZ SHORT calc98.0101F292
0101F27F  CMP DWORD PTR DS:[EAX+8],656572
0101F286  JNZ SHORT calc98.0101F292
0101F288  PUSH 3CF
0101F28D  JMP calc98.0101F203
0101F292  CMP DWORD PTR DS:[EAX],61657243
0101F298  JNZ SHORT calc98.0101F2B6
0101F29A  CMP DWORD PTR DS:[EAX+4],754D6574
0101F2A1  JNZ SHORT calc98.0101F2B6
0101F2A3  CMP DWORD PTR DS:[EAX+8],41786574
0101F2AA  JNZ SHORT calc98.0101F2B6
0101F2AC  PUSH 3D3
0101F2B1  JMP calc98.0101F203
0101F2B6  CMP DWORD PTR DS:[EAX],6E65704F
0101F2BC  JNZ SHORT calc98.0101F2DA
0101F2BE  CMP DWORD PTR DS:[EAX+4],636F7250
0101F2C5  JNZ SHORT calc98.0101F2DA
0101F2C7  CMP DWORD PTR DS:[EAX+8],737365
0101F2CE  JNZ SHORT calc98.0101F2DA
0101F2D0  PUSH 3DB
0101F2D5  JMP calc98.0101F203
0101F2DA  CMP DWORD PTR DS:[EAX],43746547
0101F2E0  JNZ SHORT calc98.0101F310
0101F2E2  CMP DWORD PTR DS:[EAX+4],65727275
0101F2E9  JNZ SHORT calc98.0101F310
0101F2EB  CMP DWORD PTR DS:[EAX+8],7250746E
0101F2F2  JNZ SHORT calc98.0101F310
0101F2F4  CMP DWORD PTR DS:[EAX+C],7365636F
0101F2FB  JNZ SHORT calc98.0101F310
0101F2FD  CMP DWORD PTR DS:[EAX+10],644973
0101F304  JNZ SHORT calc98.0101F310
0101F306  PUSH 3D7
0101F30B  JMP calc98.0101F203
0101F310  CMP DWORD PTR DS:[EAX],74726956
0101F316  JNZ SHORT calc98.0101F33D
0101F318  CMP DWORD PTR DS:[EAX+4],506C6175
0101F31F  JNZ SHORT calc98.0101F33D
0101F321  CMP DWORD PTR DS:[EAX+8],65746F72
0101F328  JNZ SHORT calc98.0101F33D
```



```
0101F32A  CMP DWORD PTR DS:[EAX+C],78457463
0101F331  JNZ SHORT calc98.0101F33D
0101F333  PUSH 3DF
0101F338  JMP calc98.0101F203
```

```
0101F33D  CMP DWORD PTR DS:[EAX],61657243
0101F343  JNZ SHORT calc98.0101F361
0101F345  CMP DWORD PTR DS:[EAX+4],69466574
0101F34C  JNZ SHORT calc98.0101F361
0101F34E  CMP DWORD PTR DS:[EAX+8],41656C
0101F355  JNZ SHORT calc98.0101F361
0101F357  PUSH 3EB
```

CreateFileA

```
0101F35C  JMP calc98.0101F203
```

```
0101F361  CMP DWORD PTR DS:[EAX],736F6C43
0101F367  JNZ SHORT calc98.0101F385
0101F369  CMP DWORD PTR DS:[EAX+4],6E614865
0101F370  JNZ SHORT calc98.0101F385
0101F372  CMP DWORD PTR DS:[EAX+8],656C64
0101F379  JNZ SHORT calc98.0101F385
0101F37B  PUSH 3E3
```

CloseHandle

```
0101F380  JMP calc98.0101F203
```

```
0101F385  CMP DWORD PTR DS:[EAX],64616552
0101F38B  JNZ SHORT calc98.0101F3A6
0101F38D  CMP DWORD PTR DS:[EAX+4],656C6946
0101F394  JNZ SHORT calc98.0101F3A6
0101F396  CMP BYTE PTR DS:[EAX+8],0
0101F39A  JNZ SHORT calc98.0101F3A6
0101F39C  PUSH 3E7
```

ReadFile

```
0101F3A1  JMP calc98.0101F203
```

```
0101F3A6  CMP DWORD PTR DS:[EAX],4D746547
0101F3AC  JNZ SHORT calc98.0101F3DB
0101F3AE  CMP DWORD PTR DS:[EAX+4],6C75646F
0101F3B5  JNZ SHORT calc98.0101F3DB
0101F3B7  CMP DWORD PTR DS:[EAX+8],6C694665
0101F3BE  JNZ SHORT calc98.0101F3DB
0101F3C0  CMP DWORD PTR DS:[EAX+C],6D614E65
0101F3C7  JNZ SHORT calc98.0101F3DB
0101F3C9  CMP WORD PTR DS:[EAX+10],4165
0101F3CF  JNZ SHORT calc98.0101F3DB
0101F3D1  PUSH 3EF
```

GetModuleFileNameA

```
0101F3D6  JMP calc98.0101F203
```

```
0101F3DB  CMP DWORD PTR DS:[EAX],50746547
0101F3E1  JNZ calc98.0101F1D8
0101F3E7  CMP DWORD PTR DS:[EAX+4],41636F72
0101F3EE  JNZ calc98.0101F1D8
```

GetProcAdress

```
0101F3F4  CMP DWORD PTR DS:[EAX+8],65726464
0101F3FB  JNZ calc98.0101F1D8
0101F401  PUSH 3BF
0101F406  JMP calc98.0101F203
```

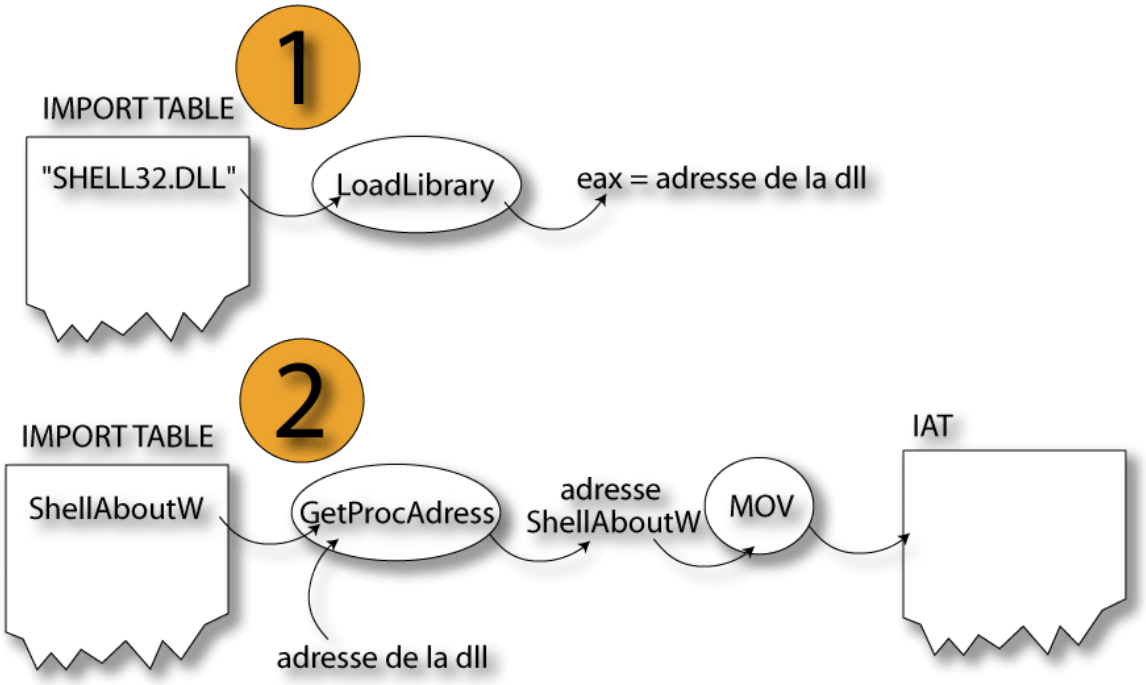
La procédure recherche d'abord l'adresse du header de Kernel32.dll en partant de l'adresse de l'API GetModuleHandleA. Puis, elle cherche l'entrée de l'export table ainsi que le nombre de fonctions dans kernel32.dll. Pour finir, elle récupère l'adresse de la première fonction et démarre sa recherche. Il y a deux boucles imbriquées, celle qui décompte le nombre de fonctions scannées dans kernel32.dll et celle qui décompte le nombre de fonctions nécessaires pour le packer.

Vous remarquez que les octets recherchés sont hardcodés dans le loader. En effet, le loader ne cherche pas à récupérer le début des APIs. Cela pose la question de l'OS qui supporte l'exe. tE! a-t-il vérifié que les APIs récupérées avaient toutes le même code quelque soit l'OS ?

Pour finir, tE! n'utilise pas GetProcAdress pour éviter les Break Point sur cette API. Ainsi, à moins de tracer le code du loader, il est difficile de localiser cette procédure.

e . Reconstruction de l'IAT de l'exe

Pour la reconstruction de l'IAT de l'exe, c'est nettement moins amusant. Il s'agit d'une technique standard qui utilise GetProcAdress et qui elle-même va utiliser l'import table pour remplir l'IAT. Nous avons donc un GetModuleHandleA suivi d'un LoadLibraryA puis des GetProcAdress. Je vous propose donc un petit schéma illustrant ce code on ne peut plus standard, qui se retrouve dans de nombreux packers/crypters et notamment dans les virus.



On récupère l'adresse de la DLL voulue puis celles des fonctions nécessaires que l'on copie dans l'IAT à l'aide d'un simple mov.

f . Techniques d'anti-debuggage/désassemblage

1 . Le CCA

Comme dans toutes les versions de tElock, le CCA est présent tout au long du loader. Cependant, selon moi, il ne présente pas une réelle difficulté en soi. Il suffit de s'y habituer et de tracer suffisamment lentement pour ne pas être embêté. Par contre, il empêche clairement une étude statique !

2 . Les Obfuscations

Je n'ai pas abordé cette technique dans la version 0.51 car elle n'était pas aussi présente que dans celle-ci. Le principe est très simple : il s'agit d'introduire du code inutile entre les lignes du code du loader. Un exemple :

```
INC EAX
DEC EAX
```

Parfaitement inutile, mais couplé avec d'autres codes de la même espèce, le loader devient assez rapidement difficile à lire. Le problème resterait mineur si tE! en était resté à là. En plus de ça, il a rendu son loader « non linéaire », à savoir que le code est criblé de call et de jump qui nous font sauter à peu près n'importe où. Un peu déroutant la première fois, je l'avoue ! On ajoute aussi des calls inutiles, des calls sans

retour...bref, l'idée est assez claire, il faut rendre le repas indigeste !

A priori, je n'ai pas vu de schéma de construction des obfuscations...il n'y a pas de bloc d'obfuscations qui se répète mais ceci est à confirmer !

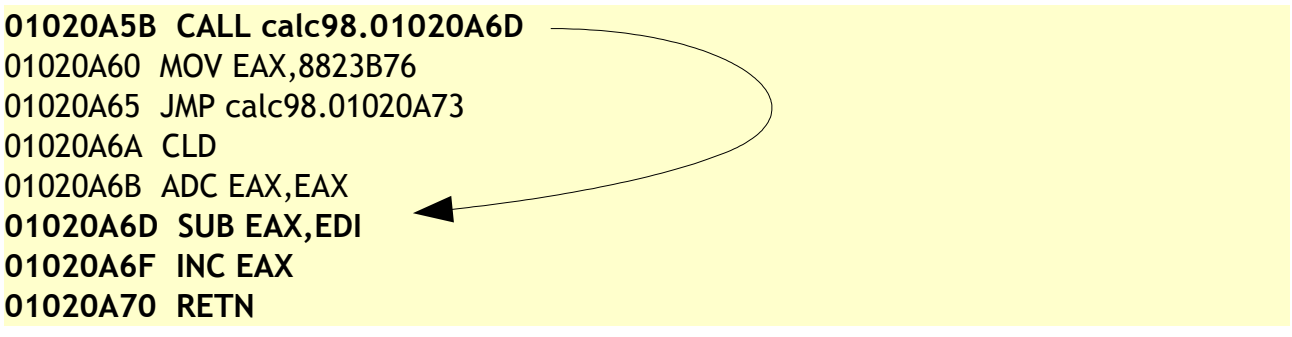
Premier exemple :(call sans retour – le pop dépile l'adresse de retour)

```
0101F113  CALL calc98.0101F119
0101F118  NOP
0101F119  POP EAX
0101F11A  INC BYTE PTR DS:[EAX]
```

Deuxième exemple : (idem mais là, on dépile de call de suite)

```
0101F009  CALL calc98.0101F00E
0101F00E  POP ESI
0101F00F  SUB ECX,ECX
0101F011  POP EAX
0101F012  JE SHORT calc98.0101F016
0101F014  INT 20
0101F016  MOV ECX,1951
0101F01B  MOV EAX,ECX
0101F01D  CLC
0101F01E  JNB SHORT calc98.0101F022
```

Troisième exemple : (call inutile)



3 . Les layers de décryptage

Nettement plus nombreux que dans la version 0.51, ils sont là pour décrypter de petites portions de code et pour ralentir le cracker. Selon mon décompte, dans cette version, ils sont au nombre de 22 contre 3 dans la version 0.51. Ceci étant dit, certains packers/crypters sont équipés de plusieurs centaines voire milliers de layers. Là, il ne s'agit ni plus ni moins que d'empêcher une étude statique du loader.

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

Il s'agit de la même protection que dans les versions précédentes. Le loader fait un CRC32-checksum basique qui empêche de modifier le code et également de poser des BPX. Ce CRC32 sert à décrypter les sections de l'exe. S'il est incorrect, une exception se produira au moment de la décompression. Il est donc difficile de repérer une protection de ce type.

Je signale néanmoins au passage que tE! n'utilise pas un algorithme de CRC32 efficace. Il applique le principe de base, à savoir, il réalise une division du code de l'exe par un nombre appelé Polynôme générateur. Dans l'art du CRC32, les Poly, comme on les appelle, ne peuvent pas être choisis au hasard. Suivant la valeur du Poly, la protection par CRC32-checksum sera plus ou moins efficace. Manifestement, tE! n'a pas estimé utile d'utiliser un poly « efficace ». Le CRC32 peut donc ne pas détecter certains changements ou certains BPX de par le choix de ce poly.

5 . L'anti Software Break Point sur APIs

Il s'agit ici d'empêcher les BPX sur les 15 APIs du loader. Le loader recherche le code CCh à l'entrée de chaque API d'une façon détournée. Il récupère pour chaque API le premier octet et lui ajoute 34h ???!!! Or, CCh + 34h = 100h...on obtient une retenue. Il suffit donc de vérifier si on obtient une retenue lors de l'addition et si c'est le cas, exitprocess . Voici la petite procédure :

```
0101FC3B  MOV ESI,DWORD PTR DS:[EDI]          ; (exemple : kernel32.OpenProcess)
0101FC3D  ADD EDI,4                           ; adresse de l'API suivante
0101FC40  LODS BYTE PTR DS:[ESI]              ; récupère le premier octet dans eax
0101FC41  ADD AL,34                           ; lui ajoute 34h
0101FC43  JE calc98.010202D6                  ; vérifie s'il y a une retenue (CF = 1)
0101FC49  LOOPD SHORT calc98.0101FC3B
```

6 . L'anti Hardware Break Point (BPM) (Debug registers)

Voilà une nouveauté dans les techniques anti-debugger de tElock. Si dans les versions précédentes, on pouvait abuser des Hardware Break Point (BPM), ici, ils sont à utiliser avec un peu plus de précaution puisque le loader tente de supprimer ceux que vous avez posé. Le dispositif a été mis en place sur le handler de plusieurs SEH. Etudions comment il est possible de modifier ces Hardware Break Points.

i) Les Debugs Registers

Tout d'abord, voici la liste des 7 Debug Registers (registres de debug) proposée par la Documentation INTEL. Les DR0, DR1, DR2 et DR3 sont utilisés pour stocker les adresses de 4 Break Points appelés Hardware Break Point ou Break Point Memory (BPM).

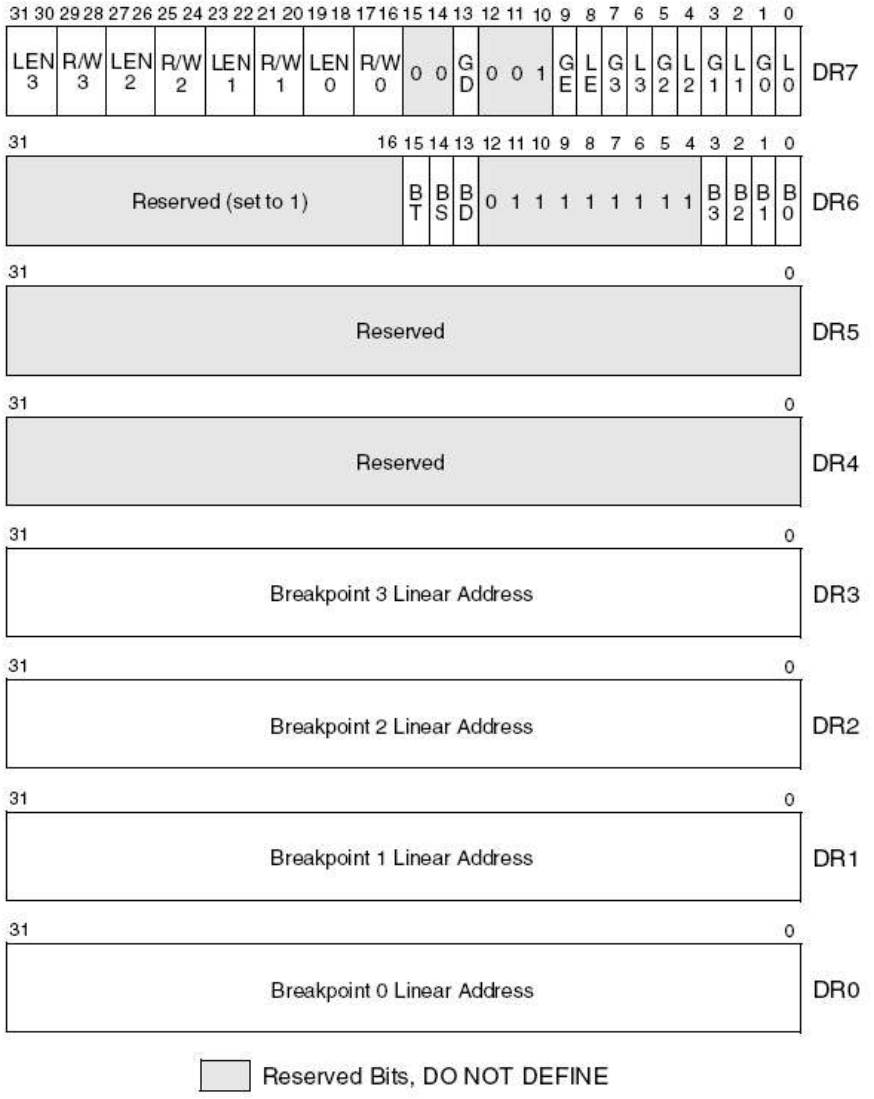


Figure 15-1. Debug Registers

ii) Le CONTEXT

Ces Debug Registers font partie d'une structure appelée CONTEXT. Ce CONTEXT contient en réalité tous les registres que vous connaissez :

(issu de la doc de J.GORDON)

```
+0 context flags
(used when calling
GetThreadContext)
DEBUG REGISTERS
+4 debug register #0 .....modifié par tElock
+8 debug register #1 .....modifié par tElock
+C debug register #2 .....modifié par tElock
+10 debug register #3 ..... modifié par tElock
+14 debug register #6 ..... modifié par tElock
+18 debug register #7 ..... modifié par tElock
FLOATING POINT / MMX registers
+1C ControlWord
+20 StatusWord
+24 TagWord
+28 ErrorOffset
+2C ErrorSelector
+30 DataOffset
+34 DataSelector
+38 FP registers x 8 (10 bytes each)
+88 Cr0NpxState
SEGMENT REGISTERS
+8C gs register
+90 fs register
```

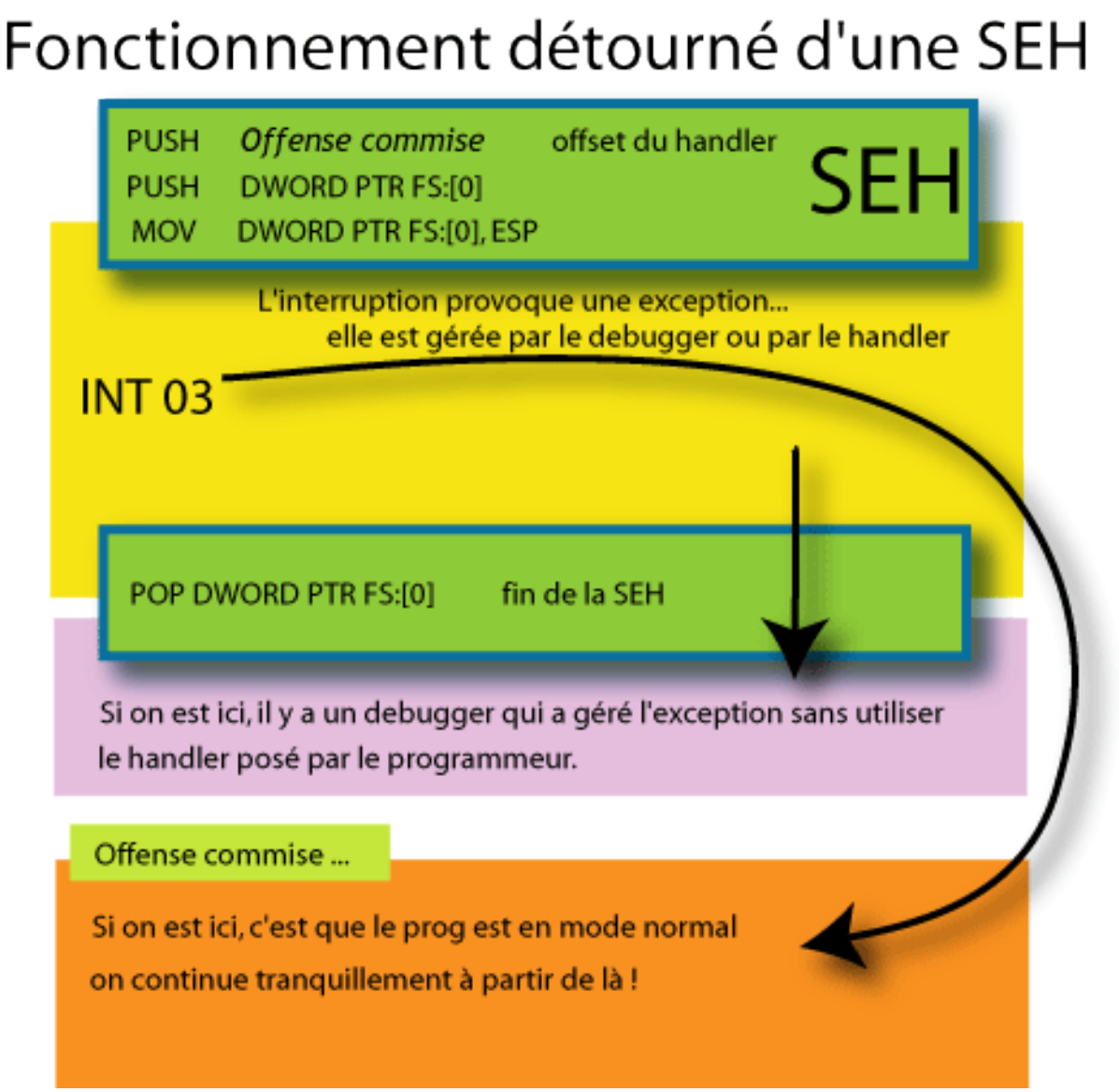
```
+94 es register
+98 ds register
ORDINARY REGISTERS
+9C edi register
+A0 esi register
+A4 ebx register
+A8 edx register
+AC ecx register
+B0 eax register
CONTROL REGISTERS
+B4 ebp register..... utilisé par tElock
+B8 eip register ..... modifié par tElock
+BC cs register
+C0 eflags register
+C4 esp register
+C8 ss register
```

Cependant, le CONTEXT n'est accessible qu'en Ring 0 or tElock parvient à modifier les DR sans passer en Ring 0 !

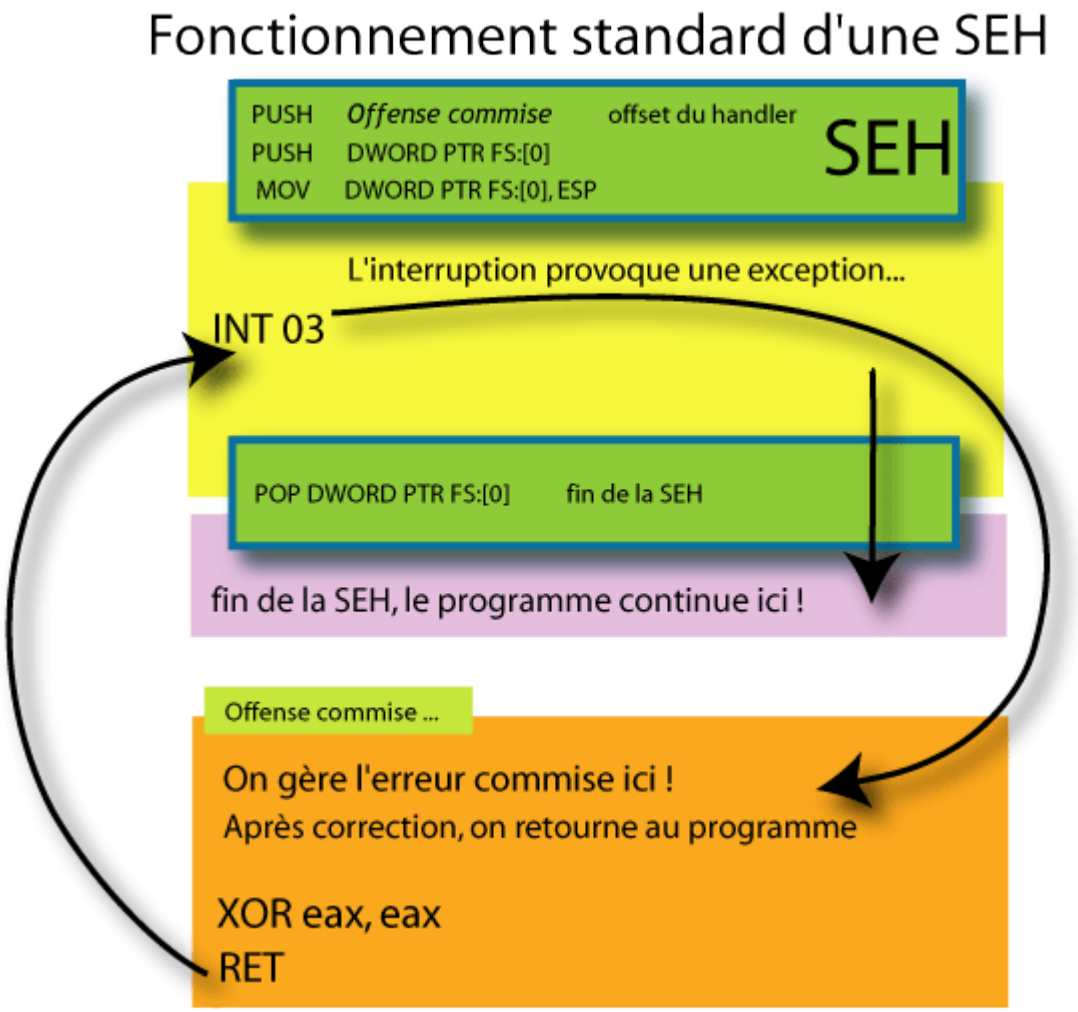
iii) Passage de Ring 3 à Ring 0 par SEH

Il existe un autre moyen d'accéder aux registres du CONTEXT en « restant » en Ring 3. Pour ça, il faut utiliser une SEH. Il se trouve que le handler d'une SEH peut accéder à une copie du CONTEXT. Depuis la fonction du handler, on peut modifier cette copie. Lorsque le handler rend la main au système, ce dernier recharge le CONTEXT (en Ring 0) avec les nouvelles valeurs. Et le tour est joué !

Voyons plus en détail ce qu'il se passe réellement. Voici tout d'abord schématisé l'utilisation des SEHs présentée dans le tutorial de tElock 0.51. Il s'agit en fait d'une utilisation détournée des SEH.


Fonctionnement détourné d'une SEH
PUSH Offense commise offset du handler
PUSH DWORD PTR FS:[0]
MOV DWORD PTR FS:[0], ESP
SEH
L'interruption provoque une exception...
elle est gérée par le debugger ou par le handler
INT 03
POP DWORD PTR FS:[0] fin de la SEH
Si on est ici, il y a un debugger qui a géré l'exception sans utiliser le handler posé par le programmeur.
Offense commise ...
Si on est ici, c'est que le prog est en mode normal
on continue tranquillement à partir de là !

Voici maintenant schématisée l'utilisation standard d'une SEH.



Lorsque le handler de la SEH a fini son travail, il rend la main au système qui retourne à l'eip spécifié dans le CONTEXT. (là où l'exception a eu lieu). Pour comprendre comment le handler peut « accéder » au CONTEXT, il faut détailler d'avantage ce qu'il se passe lorsqu'a lieu une exception. Voici un survol des manoeuvres du système :

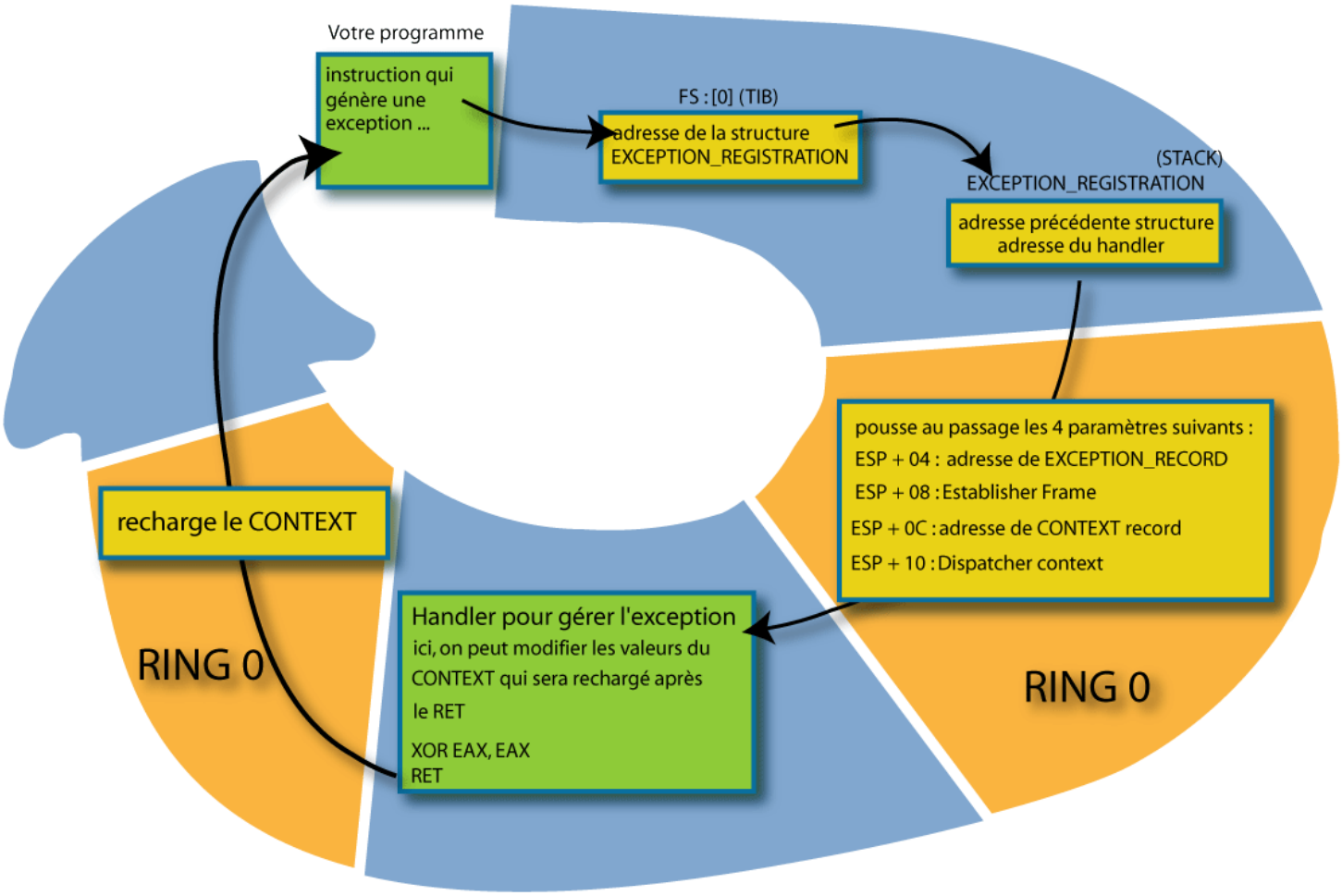
L'EXCEPTION_RECORD et le CONTEXT record

1) Lorsqu'une exception se produit, le système se branche sur le TIB (Thread Information Block) en FS:[0] pour récupérer l'offset d'une structure appelée EXCEPTION_REGISTRATION.
En fait, c'est l'utilisateur qui crée cette structure sur la pile au moment de la création de la SEH.

Push *offset du handler*	premier paramètre de la structure
Push FS:[0]	deuxième paramètre (offset de la structure précédente)

Mov FS:[0], ESP indique au TIB que la structure est à l'offset ESP.

2) Dans la structure EXCEPTION_REGISTRATION, le système récupère l'offset du handler de la SEH.



3) Puis, le système pousse 4 paramètres sur la pile dont l'offset du CONTEXT_record (la fameuse copie) et se branche sur la fonction du handler. (Matt Pietrek l'appelle callback function).

4) Du handler, on peut modifier le CONTEXT_record et notamment les DR.

5) Le handler rend la main au système qui recharge le CONTEXT avec les valeurs du CONTEXT_record.

Pour tous les détails sur le fonctionnement d'une SEH, je vous renvoie à trois documentations très complètes :

« Win32 Exception handling for assembler programmers. »
de Jeremy GORDON

«A Crash Course on the Depths of Win32 Structured Exception Handling»
de Matt PIETREK

« DOSSIER N°6 »
du groupe de travail

Revenons à tElock 0.98. Pour accéder au CONTEXT, tElock va générer trois sortes d'exceptions : Division by zero, Single Step, Break Point. Puis, le handler de ces SEH va se charger de faire le ménage dans les DR. Voici le code commenté de cette fonction :

Début de la fonction.....
D'abord, on récupère l'offset de la structure EXCEPTION_RECORD (stocké sur la pile) :
0101F0C5 MOV EAX,DWORD PTR SS:[ESP+4]

Puis, l'offset de la structure CONTEXT record (stocké sur la pile):
0101F0C9 MOV ECX,DWORD PTR SS:[ESP+C]

On augmente EIP de 1 (pour que le retour se fasse sur la ligne de code suivante)
0101F0CD INC DWORD PTR DS:[ECX+B8]

On récupère le code de l'exception situé dans l'EXCEPTION_RECORD :
0101F0D3 MOV EAX,DWORD PTR DS:[EAX]

On teste ce code : ici, on vérifie si l'exception est une division par zéro,
0101F0D5 CMP EAX,***C0000094***
0101F0DA JNZ SHORT calc98.0101F100

Si vous êtes ici, il y a eu une division by zero exception.....
On augmente EIP de 1 (pour que le retour se fasse sur la ligne de code suivante)
0101F0DC INC DWORD PTR DS:[ECX+B8]

Mise à zéro de DR0
0101F0E2 XOR EAX,EAX
0101F0E4 AND DWORD PTR DS:[ECX+4],EAX

Mise à zéro de DR1

```
0101F0E7  AND DWORD PTR DS:[ECX+8],EAX

Mise à zéro de DR2
0101F0EA  AND DWORD PTR DS:[ECX+C],EAX

Mise à zéro de DR3
0101F0ED  AND DWORD PTR DS:[ECX+10],EAX

Modification de DR6
0101F0F0  AND DWORD PTR DS:[ECX+14],FFFF0FF0

Modification de DR7
0101F0F7  AND DWORD PTR DS:[ECX+18],0DC00
0101F0FE  JMP SHORT calc98.0101F160
```

On teste le code de l'exception : ici, on vérifie si l'exception est un Single Step,

```
0101F100  CMP EAX,80000004
0101F105  JE SHORT calc98.0101F113
```

On teste le code de l'exception : ici, on vérifie si l'exception est un Break Point,

```
0101F107  CMP EAX,80000003
0101F10C  JE SHORT calc98.0101F120
```

L'exception rencontrée ne correspond à aucune des exceptions prévues. Mise à 1 de eax. Indique au système d'aller au prochain handler. (en fait, laisse Windows gérer l'exception)

```
0101F10E  PUSH 1
0101F110  POP EAX
0101F111  JMP SHORT calc98.0101F160
```

Si vous êtes ici, il y a eu une Single Step exception..... .

```
0101F113  CALL calc98.0101F119            ; Obfuscations
0101F118  NOP
0101F119  POP EAX
0101F11A  INC BYTE PTR DS:[EAX]
0101F11C  SUB EAX,EAX                     Mise à zéro de eax
0101F11E  JMP SHORT calc98.0101F160
```

Si vous êtes ici, il y a eu une break Point interruption.....

Modification de DR0

```
0101F120  MOV EAX,DWORD PTR DS:[ECX+B4]
0101F126  LEA EAX,DWORD PTR DS:[EAX+24]
0101F129  MOV DWORD PTR DS:[ECX+4],EAX
```

```
Modification de DR1
0101F12C  MOV EAX,DWORD PTR DS:[ECX+B4]
0101F132  LEA EAX,DWORD PTR DS:[EAX+1F]
0101F135  MOV DWORD PTR DS:[ECX+8],EAX

Modification de DR2
0101F138  MOV EAX,DWORD PTR DS:[ECX+B4]
0101F13E  LEA EAX,DWORD PTR DS:[EAX+1A]
0101F141  MOV DWORD PTR DS:[ECX+C],EAX

Modification de DR3
0101F144  MOV EAX,DWORD PTR DS:[ECX+B4]
0101F14A  LEA EAX,DWORD PTR DS:[EAX+11]
0101F14D  MOV DWORD PTR DS:[ECX+10],EAX

Mise à zéro de eax. Indique au système de recharger le CONTEXT et de continuer
l'exécution du programme.
0101F150  XOR EAX,EAX

Modification de DR6
0101F152  AND DWORD PTR DS:[ECX+14],FFFF0FF0

Modification de DR7
0101F159  MOV DWORD PTR DS:[ECX+18],155

0101F160  RETN
```

7 . CheckSum du Header.

Il arrive parfois que pour débugger un exe, on modifie certains paramètres du header. Par exemple, on peut changer ***Base Of Code*** pour debugger avec WinDasm. On peut aussi et surtout modifier les ***caractéristiques de certaines sections*** pour les rendre accessibles. tE! ne le sais que trop bien ! Il a donc jugé utile d'interdire ce genre de modification. Pour ça, tElock va faire une copie du header et va effectuer un CRC32-checksum dessus. Si le checksum s'est bien passé, il continue et utilise ce CRC32 pour décrypter une partie du loader...dans le cas contraire, c'est un exitprocess immédiat !

J'ai fait le test de modifier les caractéristiques de la section du loader...c'est sans appel. Il y a néanmoins un petit bug sous XP : manifestement, tE! avait prévu de marquer le coup avec une messageBox mais la fonction n'affiche rien ! Voyons maintenant le code qui fait ce travail :

Copie le header dans le loader.

Cherche son propre nom de fichier

```
0101F94E  CALL DWORD PTR SS:[EBP+3EF]       ; GetModuleFileNameA
0101F954  PUSH ESI
0101F955  XOR EAX,EAX
0101F957  LEA ECX,DWORD PTR DS:[EAX-1]
0101F95A  CLD
0101F95B  REPNE SCAS BYTE PTR ES:[EDI]
0101F95D  NOT ECX
0101F95F  LEA EDX,DWORD PTR DS:[ECX-1]
0101F962  STD
0101F963  DEC EDI
0101F964  MOV AL,5C
0101F966  REPNE SCAS BYTE PTR ES:[EDI]
0101F968  CLD
0101F969  INC EDI
0101F96A  TEST ECX,ECX
0101F96C  JE SHORT Copie_de.0101F970
0101F96E  INC ECX
0101F96F  INC EDI
0101F970  SUB EDX,ECX
0101F972  MOV ECX,EDX
0101F974  AND ECX,1F
0101F977  MOV ESI,EDI
0101F979  LEA EDI,DWORD PTR SS:[EBP+1D3D]
0101F97F  REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[>
0101F981  POP ESI
0101F982  POP EDI
```

Ouvre le fichier

```
0101F983  PUSH 0
0101F985  PUSH 80
0101F98A  PUSH 3
0101F98C  PUSH 0
0101F98E  PUSH 1
0101F990  PUSH 80000000
0101F995  PUSH EDI
0101F996  CALL DWORD PTR SS:[EBP+3EB]             ; CreatFileA
```

Copie du header dans le loader

```
0101F99C  PUSH EAX
0101F99D  PUSH 0
0101F99F  PUSH EDI
0101F9A0  PUSH EBX
0101F9A1  PUSH ESI
```

```
0101F9A2  PUSH EAX
0101F9A3  CALL DWORD PTR SS:[EBP+3E7]          ; ReadFile

Ferme le fichier

0101F9A9  CALL DWORD PTR SS:[EBP+3E3]          ; CloseHandle
```

effacement du checksum du header (qui change suivant le fichier)

```
0101F9AF  MOV EAX,DWORD PTR DS:[ESI+3C]
0101F9B2  XOR ECX,ECX
0101F9B4  ADD EAX,ESI
0101F9B6  AND DWORD PTR DS:[EAX+58],ECX
```

Calcul du CRC32-checksum sur la copie du header

```
0101F9B9  LEA EAX,DWORD PTR DS:[ECX-1]     EAX =FFFFFFFF
0101F9BC  MOV EDI,EDB88320                 Polynôme générateur du CRC32
0101F9C1  XOR EDX,EDX
0101F9C3  MOV DL,BYTE PTR DS:[ESI]
0101F9C5  XOR DL,AL
0101F9C7  SHR EDX,1
0101F9C9  JNB SHORT Copie_de.0101F9CD
0101F9CB  XOR EDX,EDI                      Soustraction !
0101F9CD  INC ECX
0101F9CE  AND CL,7
0101F9D1  JNZ SHORT Copie_de.0101F9C7
0101F9D3  SHR EAX,8
0101F9D6  XOR EAX,EDX
0101F9D8  INC ESI
0101F9D9  DEC EBX
0101F9DA  JG SHORT Copie_de.0101F9C1

0101F9DC  NOT EAX
0101F9DE  XOR DWORD PTR SS:[EBP+1B5F],EAX
[.......]
0101FC4B  MOV EAX,DWORD PTR SS:[EBP+40D280]
0101FC51  PUSH EAX
0101FC52  XOR EAX,3DA76A69
0101FC57  SUB EAX,56B82513
0101FC5C  POP EBX
0101FC5D  PUSH 1
0101FC5F  POP EAX
0101FC60  PUSH 8
0101FC62  POP ECX
```

Saut si le Checksum n'est pas bon

```
0101FC63  JNZ Copie_de.0102027B
```

```
0101FC69  JE SHORT Copie_de.0101FC84
```

```
0102027B  MOV EDX,DWORD PTR SS:[EBP+40D362]
01020281  ADD DWORD PTR SS:[EBP+40D32A],EDX
01020287  ADD DWORD PTR SS:[EBP+40D32E],EDX
0102028D  ADD DWORD PTR SS:[EBP+40D33E],EDX
01020293  ADD DWORD PTR SS:[EBP+40D342],EDX
01020299  ADD DWORD PTR SS:[EBP+40D346],EDX


Affiche une fenêtre précisant que le fichier a été altéré par un virus
(bug sous XP qui n'affiche pas la MessageBox)
0102029F  PUSH 30                                 Style de la fenêtre
010202A1  PUSH DWORD PTR SS:[EBP+40D32A]          Contenu de la fenêtre
010202A7  DEC EAX
010202A8  JNZ SHORT Copie_de.010202B2
010202AA  PUSH DWORD PTR SS:[EBP+40D346]          Titre de la fenêtre
010202B0  JMP SHORT Copie_de.010202CE
010202B2  INC EAX
010202B3  JNZ SHORT Copie_de.010202BD
010202B5  PUSH DWORD PTR SS:[EBP+40D32E]
010202BB  JMP SHORT Copie_de.010202CE
010202BD  INC EAX
010202BE  JNZ SHORT Copie_de.010202C8
010202C0  PUSH DWORD PTR SS:[EBP+40D33E]
010202C6  JMP SHORT Copie_de.010202CE
010202C8  PUSH DWORD PTR SS:[EBP+40D342]
010202CE  PUSH 0
010202D0  CALL DWORD PTR SS:[EBP+40D2D8]          ;MessageBoxA

Exit !
010202D6  MOV EAX,DWORD PTR SS:[EBP+40BAE8]
010202DC  MOV DWORD PTR SS:[ESP-4],EAX
010202E0  POPAD
010202E1  PUSH 0
010202E3  CALL DWORD PTR SS:[ESP-20]              ; ExitProcess
```

g . Le décryptage/décompression des sections du PE

a) . Calcul du CRC32, clé de décryptage

tE! Utilise le même principe pour décrypter les sections. Le CRC32 calculé sur les octets du loader sert dans l'algorithme de décryptage. Je ne redétaille pas ce procédé qui est le même pour toutes les versions. Seul le polynôme générateur change de valeur.

Valeur du Poly pour tElock 0.98 : **CDC795E1h**

b) . Premier décryptage

Dans cette version, tE! A ajouté un premier décryptage qui ne présente en soi aucune difficulté. J'ai cependant trouvé ce layer assez amusant puisqu'il applique pas moins de 46 opérations sur chaque octet pour le décryptage ! Voici le code :

```
0101FD09  LODS BYTE PTR DS:[ESI]....récupération de l'octet situé à l'adresse ESI
0101FD0A  ADD AL,DL
0101FD0C  LEA EBX,DWORD PTR DS:[EBX]
0101FD0E  TEST ECX,ECX
0101FD10  OR ESI,ESI
0101FD12  LEA EBX,DWORD PTR DS:[EBX]
0101FD14  ADD AL,0CF
0101FD16  ADD AL,0C5
0101FD18  ROR AL,CL
0101FD1A  ADD AL,2D
0101FD1C  OR CL,CL
0101FD1E  TEST EDX,EDX
0101FD20  MOV AL,AL
0101FD22  ADD AL,DL
0101FD24  ROR AL,CL
0101FD26  CLC
0101FD27  NOP
0101FD28  ADD AL,1
0101FD2A  LEA EDX,DWORD PTR DS:[EDX]
0101FD2C  ROR AL,CL
0101FD2E  NOT AL
0101FD30  NEG AL
0101FD32  INC AL
0101FD34  NOT AL
0101FD36  OR EAX,EAX
0101FD38  NEG AL
0101FD3A  ADD AL,CL
0101FD3C  LEA EBX,DWORD PTR DS:[EBX]
0101FD3E  TEST EDI,EDI
0101FD40  XOR AL,BL
0101FD42  OR ESI,ESI
0101FD44  XOR AL,19
0101FD46  NEG AL
0101FD48  LEA EBX,DWORD PTR DS:[EBX]
0101FD4A  NEG AL
0101FD4C  XOR AL,25
```

```
0101FD4E  OR CL,CL
0101FD50  XOR AL,9
0101FD52  XOR AL,9D
0101FD54  OR EAX,EAX
0101FD56  ADD AL,1
0101FD58  ADD AL,CL
0101FD5A  TEST EDI,EDI
0101FD5C  NEG AL
0101FD5E  INC AL
0101FD60  OR ESI,ESI
0101FD62  ROR AL,CL
0101FD64  NOT AL
0101FD66  NOT AL
0101FD68  ADD AL,1
0101FD6A  LEA EDX,DWORD PTR DS:[EDX]
0101FD6C  LEA EBX,DWORD PTR DS:[EBX]
0101FD6E  MOV EAX,EAX
0101FD70  ADD AL,DL
0101FD72  ROR AL,CL
0101FD74  ROR AL,CL
0101FD76  ADD AL,21
0101FD78  LEA EBX,DWORD PTR DS:[EBX]
0101FD7A  ADD AL,CL
0101FD7C  ADD AL,8B
0101FD7E  NEG AL
0101FD80  ADD AL,CL
0101FD82  XOR AL,BL
0101FD84  NOT AL
0101FD86  MOV BL,BL
0101FD88  NOP
0101FD89  NOP
0101FD8A  NOP
0101FD8B  NOP
0101FD8C  XOR AL,DL
0101FD8E  ROR AL,1
0101FD91  STOS BYTE PTR ES:[EDI]...stockage de  AL  à l'adresse EDI !
0101FD92  IMUL EDX,EDX,4BCDB0A5
0101FD98  STC
0101FD99  JB SHORT calc.0101FD9D
0101FD9B  INT 20
0101FD9D  ROL EDX,1
0101FD9F  IMUL EBX,EBX,6AEE1F70
0101FDA5  ADD EBX,EDX
0101FDA7  DEC ECX
```

0101FDA8 JG calc.0101FD09

c) . Deuxième décryptage

Je ne reviens pas dessus - voir tElock 0.51 - Il s'agit d'utiliser le CRC32 pour décrypter la même portion de code qui vient de l'être avec le layer précédent.

d) . Décompression - ApLib 0.26

Idem - tE! utilise toujours l'algorithme de décompression de l'ApLib 0.26b. voir tElock 0.51.

h . Techniques anti-unpacking/anti-dump

1 . le mutex

Même protection que celles des versions précédentes, si l'option est cochée, le loader crée un mutex dont la présence doit être testée par le logiciel protégé. En l'absence de mutex, le logiciel peut considérer qu'il a été unpacké et donc prévoir une procédure pour se protéger. J'ai regardé si les keygens de TMG étaient équipés d'un tel dispositif....et conclusion : l'équipe de TMG ne se sert pas de cette option !! (Pour plus de détails, voir le tutorial sur tElock 0.51). Petite note de l'auteur :

P.S.: Since v0.95 tElock ALWAYS generates a 8-char Mutex Object for each file. You can find the string used in the Listview control after packing process has been finished. The string is calculated individually for each file. Example: CMS:: xPkWZ8Ha

2 . effacement du loader

Toujours comme les versions précédentes, juste avant d'arriver au saut sur l'OEP, le loader s'efface en remplaçant ses données par des zéros. Le code est le même que pour la version 0.51.

3 . modification du header

Modifier le nombre de sections avec VirtualProtectEx

Ici, tElock change le nombre de sections spécifié dans le header.(C'est une protection que je n'ai pas vu dans la version 0.51 !) Cette section n'est normalement pas accessible en Ring 3 sauf si on modifie son accès à l'aide de la fonction VirtualProtectEx. Cette technique est efficace contre ProcDump (plantage simple !) mais inutile contre LordPE qui reconstruit le header tout seul. On récupère l'ID du

process puis on ouvre le process et on modifie les droits d'accès au header.

```
0102068E  CALL DWORD PTR SS:[EBP+40BAF8]          ; kernel32.GetCurrentProcessId
01020694  MOV EBX,EAX
01020696  PUSH EAX
01020697  PUSH 0
01020699  PUSH 1F0FFF
0102069E  CALL DWORD PTR SS:[EBP+40BAFC]          ; kernel32.OpenProcess
010206A4  INC EAX
010206A5  DEC EAX
010206A6  JE SHORT calc98.010206D3
010206A8  PUSH 0
010206AA  PUSH ESP
010206AB  PUSH 4
010206AD  PUSH 1000
010206B2  PUSH DWORD PTR SS:[EBP+40D362]
010206B8  PUSH EAX
010206B9  CALL DWORD PTR SS:[EBP+40BB00]          ; kernel32.VirtualProtectEx
010206BF  ADD ESP,4
010206C2  INC EAX
010206C3  DEC EAX
010206C4  JE SHORT calc98.010206D3
010206C6  MOV EDI,DWORD PTR SS:[EBP+40D362]     Récupère l'image base
010206CC  ADD EDI,DWORD PTR DS:[EDI+3C]          Récupère l'adresse PE
010206CF  OR WORD PTR DS:[EDI+6],SP              Modifie le nombres de sections.
```

Modifier l'ImageSize en accédant au PEB

Contrairement à la version 0.51, tElock 0.98 ne change pas l'ImageSize en accédant au PEB (ImageSize = 0). L'option n'est plus accessible ! Par contre, on peut la retrouver dans les versions 1.xx. J'ai rencontré cette option sur un release de TMG datant du 16 octobre 2003. Notez au passage que cette option ne sert à rien face à LordPE qui ne tiens absolument pas compte de cette supercherie lors du dump.

4 . redirection de l'IAT/ Effacement des imports

Idem, le loader redirige l'IAT de l'exe (si l'option est cochée). Cette redirection utilise le même principe que la version 0.51. On remarque une petite amélioration puisque les redirections se font au fur et à mesure que les imports sont reconstruits. Les deux codes ne sont pas disctincts, ce qui rend la redirection un peu plus difficile à contourner. A noter que toutes les APIs ne sont pas redirigées. Il teste le nom de l'API en cours qui doit faire partie des 3 suivantes : USER32.DLL – KERNEL32.DLL – SHELL32.DLL. Tout le reste, il n'y touche pas ! Il semblerait que certaines fonctions soient difficiles à

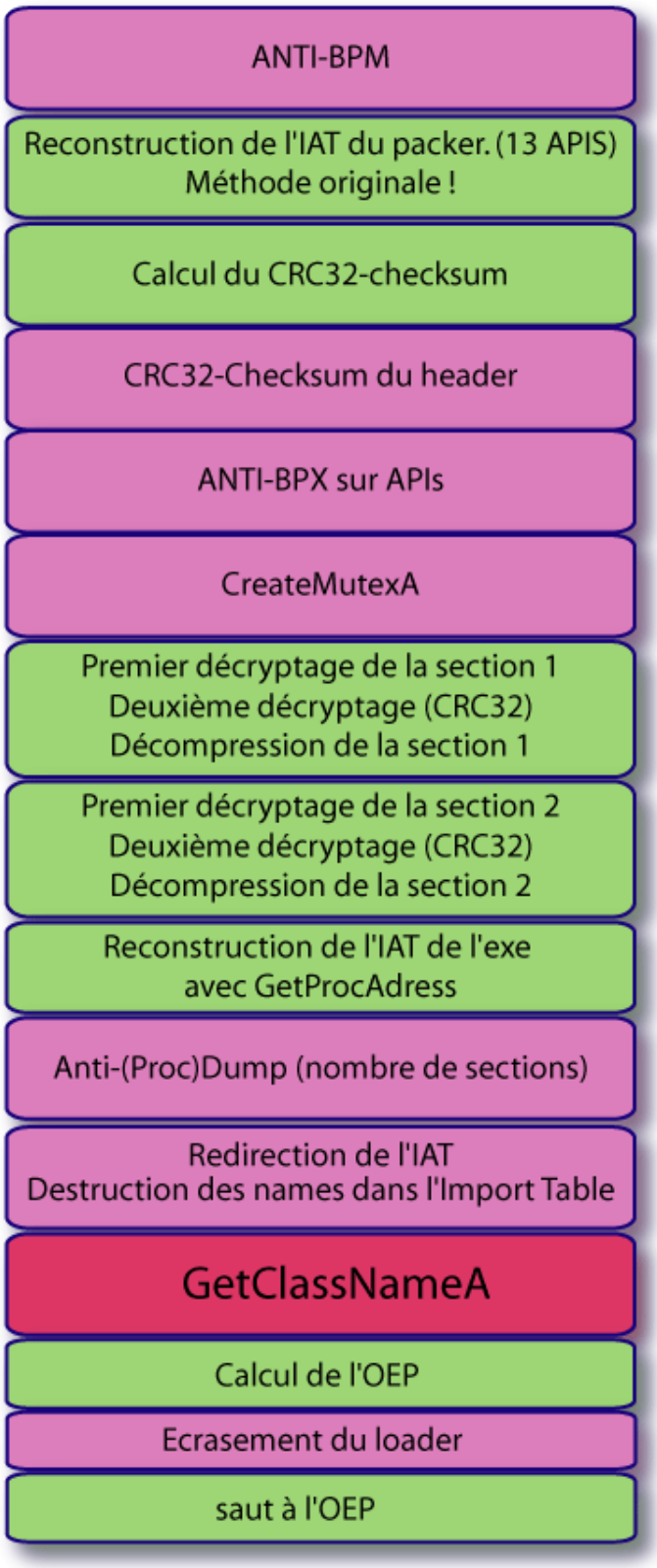
détourner (dixit +The Analyst) ce qui ne concernerait pas les 3 DLLs précédentes ainsi que GDI32.DLL.Ceci dit, tE! est bien conscient que c'est inutile face à Revirgin ou ImportReconstructor.

« ***[Enable IAT-Redirection]***
That's a useful feature to get rid of some Imports-rebuilding tools which are able to rebuild the importtable of a dumped PE-File using the fixed addresses in the IAT during runtime. Info: In the meanwhile there're a few (free) utilities available on the net which are able to rebuild the imports even if the IAT has been redirected 100 times or more. (Greetings to MackT and +tsehp) »

3 . LE LOADER DES VERSIONS PRIVATE 1.xx

a . Schéma du loader sans les protections

Pour les versions privates, celles qui protégent les keygens de TMG récents, qui sont baptisées versions 1.xx, il y a très peu de changement par rapport à la version 0.98. Les versions diffèrent par leurs signatures mais les protections sont les mêmes. Un petit ajout est néanmoins à noter dans les anti-debugging : tElock détecte désormais OllyDbg, WinDasm, DéDé, FileMon à l'aide de l'API GetClassNameA et les ferme de façon autoritaire s'il les détecte. Nous verrons le code de cette protection dans le prochain paragraphe.



b . Technique anti-debugging

l'API GetClassNameA

Voilà donc la nouveauté des versions 1.xx. Le principe est simple : le loader énumère toutes les classes des process en cours et va rechercher les classes des applications suivantes :

OLLYDEBUGGER (Classe = OLLYDBG)
WINDASM (Classe = OWL_Window)

FILEMON (Classe = FileMonC)
DéDé (Classe = TDeDeMainForm)

Dès qu'il en a trouvé une, il la ferme de façon autoritaire.

Voici le code :

```
0041B89E  CALL DWORD PTR DS:[EDI+C]          ;GetClassNameA
0041B8A1  MOV EAX,DWORD PTR DS:[EDI]

0041B8A3  CMP DWORD PTR DS:[EAX],594C4C4F    « OLLY »
0041B8A9  JE SHORT keygen.0041B8CC
0041B8AB  CMP DWORD PTR DS:[EAX],5F4C574F    « OWL_ »
0041B8B1  JE SHORT keygen.0041B8CC
0041B8B3  CMP DWORD PTR DS:[EAX],44654454    « TDeD »
0041B8B9  JE SHORT keygen.0041B8CC
0041B8BB  CMP DWORD PTR DS:[EAX],656C6946    « File »
0041B8C1  JNZ SHORT keygen.0041B8DF
0041B8C3  CMP DWORD PTR DS:[EAX+4],436E6F4D  « MonC »
0041B8CA  JNZ SHORT keygen.0041B8DF

0041B8CC  PUSH 0
0041B8CE  PUSH 0
0041B8D0  PUSH 10
0041B8D2  PUSH DWORD PTR SS:[EBP+8]
0041B8D5  CALL DWORD PTR DS:[EDI+8]          SendMessageA
0041B8D8  XOR EAX,EAX
0041B8DA  POP EDI
0041B8DB  LEAVE
0041B8DC  RETN 8

0041B8DF  PUSH 1
0041B8E1  POP EAX
0041B8E2  POP EDI
0041B8E3  LEAVE
0041B8E4  RETN 8
```

Pour contourner la difficulté, il suffit de poser un BP en 41B8A3 et de surveiller le contenu de EAX. Dès que [EAX] = « OLLYDBG », on le remplace (par exemple) par « ALLYDBG ».

Autre solution plus radicale (patcher OllyDbg) : Puisqu'il cherche une classe qui répond au nom de « OLLYDBG », on a juste à changer le nom de cette classe et le tour est joué.

Pour cela, il faut repérer dans OllyDbg.exe où se trouve le nom de la classe (regardez les paramètres de RegisterClass) et de le patcher avec un autre nom. Personnellement, j'ai utilisé LordPE pour faire ce travail. Pour la version OllyDbg 1.10, la string se trouve en 4B7218. Je l'ai remplacé par « ALLYDBG »...et c'est réglé !

4 . MANUAL UNPACKING

a . Trouver l'OEP

trouver l'OEP avec LordPE (valable pour toutes les versions)

Pour les versions publiques, le problème ne se pose pas puisqu'il existe des unpackers. En plus, OllyDbg, avec son module SFX permet de trouver l'OEP sans aucune difficulté.

Si vous n'avez pas modifié le nom de la classe de Olly, pour les versions privates 1.xx, la détection de la classe par l'API GetClassNameA rend le travail plus délicat. Il faut faire le travail à la main.

Nous allons utiliser une particularité des packers de tElock : lorsque le packer rend la main au programme (saut à l'OEP), la section du loader a été écrasée à grands coups de zéros...mais pas partout ! Aussi étrange que cela puisse paraître, tElock a copié au début du loader l'adresse de l'OEP sous forme d'un JMP OEP !! Nous vous y trompez pas, ceci n'est qu'un leurre puisque ce jump n'est jamais utilisé. Ceci dit, nous allons récupérer cette adresse sans plus tarder.

1) Lancer l'application packée par tElock.

2) Faire un « Dump Region » avec LordPE de la dernière section de l'exe (celle du loader) lorsque c'est possible. Si vous ne voyez qu'une grosse section contigüe, c'est normal, l'ImageSize est à zéro. Dans ce cas, dumpez quand même, il faudra juste chercher la dernière section dans cette copie.

3) Ouvrir le dump avec HexDecChar pour lire l'OEP :

Par exemple, si votre section commence en 101F000, vous devriez y voir :

101F000 **E9 7034FFFF** JMP 1012475

Evidemment, vous ne voyez que les octets E97034FFFF. E9 correspond au JMP et 7034FFFF signifie qu'il faut se déplacer de FFFF3470. Pour calculer l'OEP, on fait :

101F000h + 05h + FFFF3470h = 1012475

Utiliser ShaOllyScript plugin v0.92 by ShaG

ShaG nous propose OllyScript, un plugin qui permet de créer ses propres scripts pour automatiser certaines tâches. Il a introduit un script pour trouver l'OEP de tElock 0.98. Le principe est simple : s'aider des SEHs pour avancer dans le code. Ce sont des sortes de Break Points « naturels », autant s'en servir au maximum. Voici le code de ce script :

```
/*
      tElock 0.98 OEP finder v1.2
      ---------------------------
      Seems to work =)
      Please make sure no exceptions are passed to program
      i.e. uncheck all the boxes on the Exceptions tab
      in Debugging Options except the topmost one
*/

var cbase
gmi eip, CODEBASE
mov cbase, $RESULT
log cbase
var csize
gmi eip, CODESIZE
mov csize, $RESULT
log csize

var count
mov count, 13....ici, on initialise le nombre d'exceptions à franchir.
eob lbl1
eoe lbl1
run

lbl1:
cob
coe
msg count
cmp count, 0
je lbl2
esto
```

```
sub count, 1
jmp lbl1

lbl2:
esti
bprm cbase, csize
eob end
eoe end
run

end:
cmt eip, "OEP"
cob
coe
ret
```

ShaG dénombre 13h SEHs et malheureusement, sur mon calc.exe, ça ne fonctionne pas car je n'ai que Fh SEHs. Il faut donc pour que ça fonctionne ***modifier ce fameux 13 par F !***

Malgré son nom tElock098.txt, ce script fonctionne très bien pour les versions 1.xx si on a pris la précaution de changer le nom de la classe de Olly comme je l'ai précisé plus haut.

Utiliser le script de loveboom pour les versions 1.xx

Voici un autre script proposé par lovedoom qui nécessite également de changer le nom de la classe de Olly. Il précise en entête qu'il faut renommer OLLYDBG.EXE...je n'ai toujours pas compris pourquoi ! C'est tout simplement inutile. Par contre, il est plus confortable que le premier script car il ne nécessite aucune intervention de notre part. Le principe est néanmoins le même.

```
/*
//////////////////////////////////////////////////
tElock 0.9 - 1.0 (private) -> tE! OEP Finder v0.1
Author: loveboom
Email : bmd2chen@tom.com
OS : Win2kADV sp2,OllyDbg 1.1b,OllyScript v0.62
Date : 2004-3-29
Config: uncheck all the boxes on the Exceptions tab
in Debugging Options except the topmost one.Rename 'OLLYDBG.EXE'
Modify Classname 'OLLYDBG',I can email you a file patch(Only 1.1b) if you want.
Note : If you have one or more question, email me please,thank you!
//////////////////////////////////////////////////
```

```
*/

var count //Declare variant
var cbase
var csize

start:
gmi eip,CODEBASE
mov cbase,$RESULT
gmi eip,CODESIZE
mov csize,$RESULT
mov count,f.....ici, loveboom a initialisé avec la bonne valeur.
run

lbl1:
cmp count,0
je lbl2
sub count,1
esto
jmp lbl1

lbl2:
bprm cbase,csize
esto

lblend:
bpmc
cmt eip,"OEP found!please dumped it!"
msg "scrip't by loveboom[DFCG],Thank you for using my scrip't!"
ret
```

b . Dumper l'exe

Nous allons poser un BPM sur l'OEP pour ensuite dumper avec le plugin OllyDump. Pour poser ce BPM, nous devons faire face à trois cas de figure :

	ANTI-BPM	**GetClassNameA**
Versions tElock < 0.98	NON	NON
Versions tElock 0.98 et 0.99	OUI	NON
Versions tElock 1.xx	OUI	OUI

Premier cas : Aucun problème

Deuxième cas : Il faut franchir les anti-BPM pour pouvoir breaker sur l'OEP. Il faut s'aider des SEHs pour avancer dans le code.

Troisième cas :Il faut franchir le GetClassNameA. Posez un memory on access sur cette api pour localiser la procédure qui s'en sert. Evidemment, il n'y a pas de problème si vous avez changé le nom de la classe.

OllyDump dumpe l'exe et reconstruit le header sans difficulté.

Comparaison des dumps réalisés avec ProDump - LordPE - OllyDump

C'est une idée de eXXe qui m'a semblée très bonne. Je vous propose de jeter un oeil sur les headers des dumps et d'analyser les différences.

Déjà, ProcDump se plante lamentablement à cause de l'anti-(Proc)Dump ! On pourrait en rester là mais on va analyser ce que fait ProcDump et ce qui cause le plantage !

Dans les entrailles de ProcDump 1.6f

On désassemble ProcDump et on scrute son comportement. D'abord, il est clair que c'est le nombre de sections (qui est à FFA0 au lieu de 4) qui fait planter ProcDump. Il y a manifestement une exception qui se produit au moment de la création du dump. Il s'agit en fait d'un **Access Violation**. Pour créer le dump, ProcDump copie le contenu de l'exe à dumper en mémoire. Puis, il va chercher dans le header les noms des sections de cet exe. Pour effectuer cette recherche, il va récupérer le nom de la première section puis va sauter sur le nom de la deuxième..etc en utilisant comme compteur le nombre de sections...ça semble normal ! Or, si le nombre de sections est FFA0, il va faire plusieurs bonds en dehors de la section de l'exe et va surement atteindre une section protégée...du coup, on a droit à un Access Violation en bonne et dûe forme ! ProcDump n'est pas équipé d'une SEH pour se protéger de ce genre d'erreur et il se plante !

LordPE et OllyDump parviennent à faire un dump de l'exe en contournant l'anti-(Proc)Dump. Ils n'utilisent pas la donnée du nombre de sections pour lister toutes les sections du PE. Il faut néanmoins prendre une précaution avec LordPE : avant de faire un dump full, il faut ***décocher l'option « Realign file »***. Sinon, le fichier va être réduit et ImpRec sera incapable de rajouter une section (le header étant trop petit), donc sera incapable de fixer les imports. Moyennant cette petite modification, les headers sont identiques. Cependant, OllyDump a récupéré les imports et les a fixé tout seul dans une section supplémentaire. Il a donc fait le travail de ImpRec ! Sans aucune hésitation, OllyDump, vainqueur par chaos sur tous les plans !

c . Reconstruction de l'IAT

Si vous avez dumpé avec LordPE, comme pour les précédentes versions, l'IAT est au pire redirigé donc un simple **Trace Level 3** sous ImpRec suffit pour résoudre tous les imports.

Du point de vue du unpacker, tElock 0.98 n'a que peu d'intérêt puisque les anti-debuggers sont très facilement contournables (on n'a même pas besoin de les comprendre pour les éviter), et le dump se fait tout seul. Comme le précise tE! lui-même, rediriger les imports une fois ou 500 fois ne changera rien pour OllyDump , ImportReconstructor ou Revirgin. C'est donc une méthode à proscrire sans appel ! Il a néanmoins le mérite de ralentir les lamers qui désireraient ripper le code de l'exe ou juste modifier ses ressources pour le « personnaliser ».

Voilà ! Le second volet sur l'analyse de tElock s'achève. J'espère que ce travail vous a plu et que vous avez appris quelques bricoles (ne serait-ce que le schéma du loader de tElock). Personnellement, j'ai eu un plaisir immense à travailler sur ce petit packer d'exe.

6 . REMERCIEMENTS/SOURCES

Voici mes sources :

1) « Win32 Exception handling for assembler programmers. » de Jeremy GORDON

2) «A Crash Course on the Depths of Win32 Structured Exception Handling» de Matt PIETREK

3) « DOSSIER N°6 » du groupe de travail.(SEH)

4) « DOSSIER N°8 » du groupe de travail.(CRC32)

5) Documentation Intel sur l'IA-32.

Merci à tE! Pour avoir imaginé ce petit packer d'exe très intéressant.

Je remercie énormément tous ceux qui ont pris de leur temps pour me donner un lien, une doc, une explication sur un détail technique qui ont été des aides précieuses sans lesquelles, je le reconnais, je n'aurais pas pu écrire ceci. Merci donc à eXXe, Darus, Lautheking, elooo, +The Analyst, Kaine, Gbillou, Kharneth, Thierry The One, Neitsa.

Un grand merci également à Cyber Daemon et r!sc pour leur travail sur le unpacker générique des versions de tElock 0.41b, 0.42c et 0.51.

Merci à ShaG et loveboom pour leur travaux sur OllyScript.

Merci à J.GORDON, M.PIETREK et au Groupe de Travail pour leurs docs remarquables sur les SEHs.

Merci à Ross N. Williams pour son excellent travail sur le CRC32.

Enfin, un grand coucou et un grand merci à tous les membres actifs de forumcrack.

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

Jeudi 16 septembre 2004 - BeatriX