By Horgh

Malware Unpacking 1

Version 1.0
28/10/2011

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
   ANNEXES
4. Payload du malware
   a) Etude de la DLL *embedded*
   b) Etude du payload de l'exécutable unpacké
5. Conclusion et remerciements
6. Références et documentation

1. Propos liminaires

a) Généralités

La cible du jour est un trojan de type password stealer, visant les jeux en ligne et principalement connu sous les noms de *Win32.OnlineGameHack* ou *Win32.Magania*, que vous pourrez trouver dans l'archive chiffrée ci-jointe. 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*.

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. (Cette partie n'a pas été finie, cf. Note introductive)

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.

NOTE : J'ai commencé à rédiger ce papier à la fin de l'année 2010, et j'ai arrêté de continuer sa rédaction (épisodique certes) en juillet, étant donné que l'article du malware corner de MISC n°56 est bien plus avancé que le mien (Je n'ai ni l'expérience, ni les moyens de l'auteur). Je publie néanmoins ce papier car je pense qu'au vu des retours préliminaires que j'en ai eu, il pourrait intéresser quelques personnes. Vous trouverez donc une partie seulement de l'analyse du malware (car je ne vais pas la terminer, c'est inutile de réinventer la roue), certaines choses différant avec l'article de MISC, mais dans l'ensemble l'action effectuée est la même.

Bonne lecture.

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.

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 / Process explorer / Process Monitor / 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 une piste : il est fort probable qu'il soit packé. 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 [2] du cheval de Troie, avec LordPE (Fig. 1).

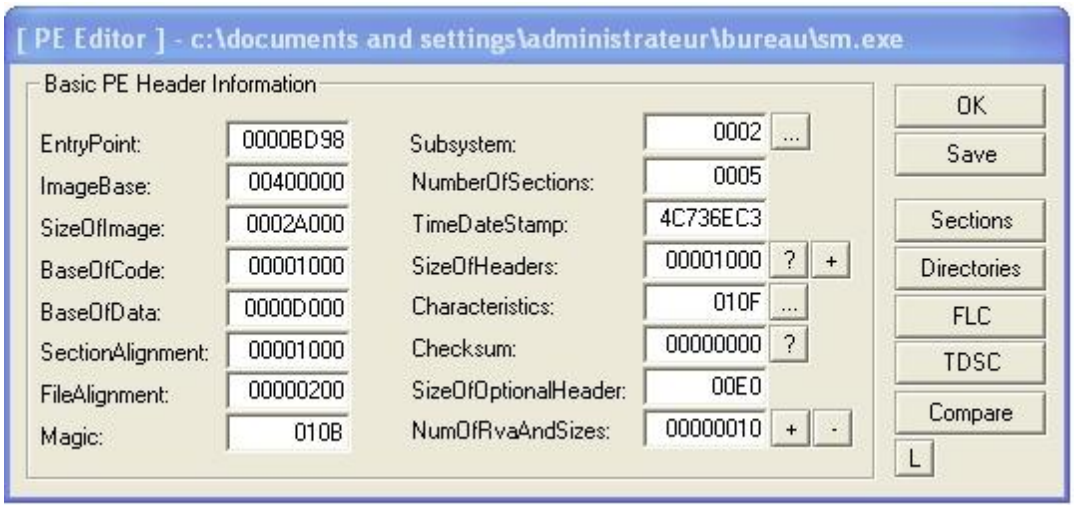

Figure 1 : PE Header du fichier

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* [3] est activé, on remarquera la présence d'une section *.textbss*. (Une section *.rdata* peut être également présente). [4]

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 devrons chercher au cours de l'unpacking.

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.textbss	00001000	00009700	00000000	00000000	E00000A0
.text	0000B000	00002000	00000400	00000D8D	60000020
.data	0000D000	0000E000	00001200	0000D800	C0000040
.idata	0001B000	0000B000	0000EA00	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 anormal, car il doit se situer dans la section .textbss pour ce genre d'exe. On remarque également que cette section est vide (ROffset et RSize sont à zéro), ce qui laisse supposer que l'exe est bel et bien packé. Examinons maintenant les caractéristiques de ces sections.

.textbss
Executable as code / readable / writeable / contains executable code / contains unitialized 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 / writeable / 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 1er 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 ce paramètre ne soit pas fiable dans ce cas présent, ce genre de malwares étant très fréquemment repacké ce qui laisse une durée de vie très courte à chaque sample (celle-ci ayant été capturée en novembre 2010).

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 [5], et cette api n'a soit peu de sens à être utilisée seule (QueryServiceConfigA par exemple), 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 ;)

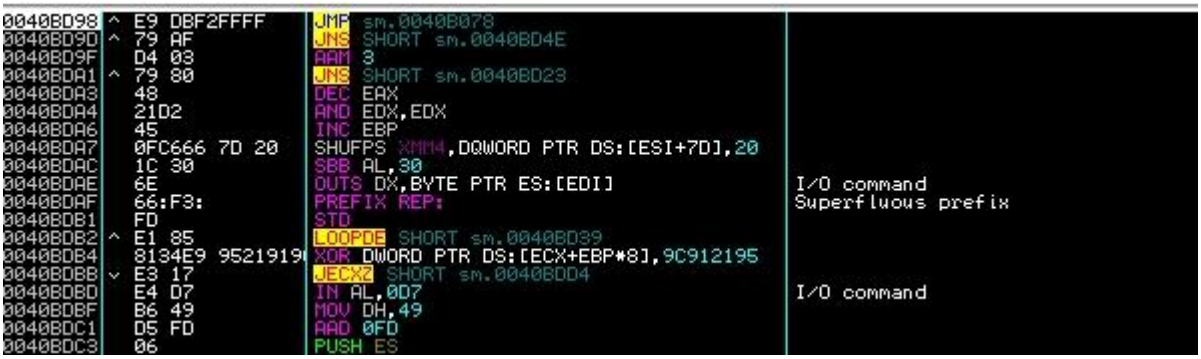

Figure 3 : EP du trojan

Nous retrouvons donc notre entrypoint fortement obfusqué, qui commence directement par un saut dans une zone mémoire située plus en arrière. 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"*).

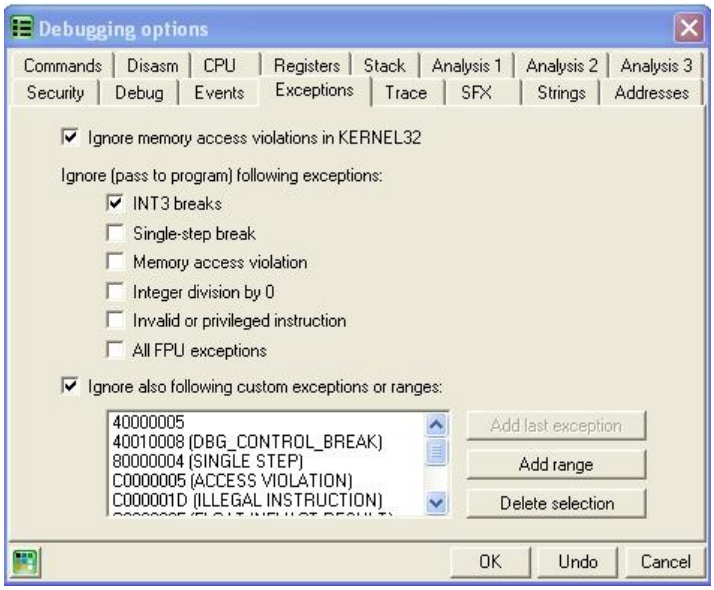

Figure 4 : Réglages des options de debugging

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.

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'exécutable.

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)

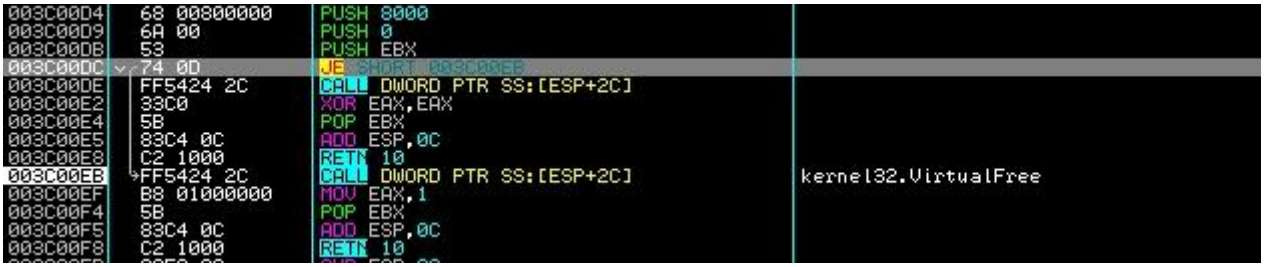

Figure 7 : Passage sur VirtualFree

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.

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*.

Figure 9 : Arrivée sur l'OEP analysé

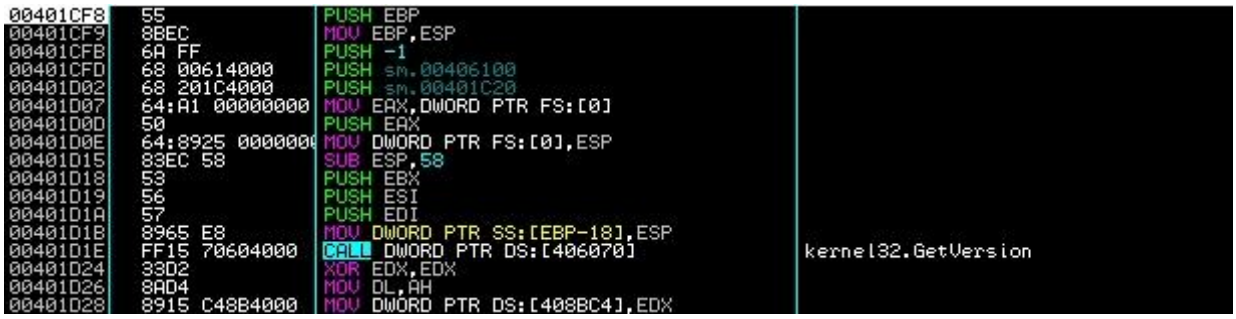

Figure 10 : OEP de notre trojan, typique de MSVC++

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, ou 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-ci 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                  ; rééquilibrage de la pile ?
0041465E    DEC ECX                  ; décrémente la taille restante à 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 533D9C60     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    8801            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 [6] 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             ; correspond au <sm.badboy>
0040F54A    POP DWORD PTR FS:[EAX]  ; restauration de l'ancien SEH
0040F54E    POP EBX
```

Le trojan va donc installer un SEH (*Structured Exception Handler*) [7] 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 [xxxxxxxxx]* dans Olly (exception numéro 0x0C0000005h)). 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" 9. Le fait d'effectuer des vérifications sur ces caractéristiques ci 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 0x4E52454Bh, 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 0x32334C45h soir "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]
; le champ "e_lfanew", à l'offset 0x3Ch (50d), 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 (kernel32.7C80262C)
0040F225    ADD ESI,EBP       ; export table RVA + adresse base kernel32
0040F227    PUSH ESI          ; pousse l'adresse
0040F228    MOV ESI,DWORD PTR DS:[ESI+20] ; DS:[7C80264C]=0000353C
0040F22B    ADD ESI,EBP       ; ESI pointe sur un ptr_array (7C80353C)
0040F22D    XOR ECX,ECX       ; ECX mis à 0
0040F22F    DEC ECX           ; junk ?

bad_api :
0040F230    INC ECX           ; junk ?
0040F231    LODS DWORD PTR DS:[ESI] ; charge dans EAX le dword dans ESI
0040F232    ADD EAX,EBP       ; ptr vers le nom de la première api contenue
dans l'Export Table (EAX 7C804BA5 ASCII "ActivateActCtx")
0040F234    XOR EBX,EBX                   ; ebx est mis a 0

loop_api :
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

api_terminated :
0040F245    CMP EBX,DWORD PTR DS:[EDI] ; cmp le résultat à 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 consiste 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 recherchées. Si les deux concordent, c'est que nous avons la bonne api, il 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 de l'une des quatre apis recherchées.

```
0040F249    POP ESI                       ; kernel32.7C80262C
0040F24A    MOV EBX,DWORD PTR DS:[ESI+24] ; DS:[7C802650]= 04424h
0040F24D    ADD EBX,EBP                   ; 7C800000 + 4424 = 0x7C804424
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
```

Le LOOPD en 0040F215 nous fait effectuer quatre fois la routine, afin de récupérer quatre apis : *kernel32.VirtualAlloc, kernel32.VirtualFree, kernel32.LoadLibraryA, kernel32.GetProcAddress*).

- Copie en mémoire de la dll contenue dans l'exécutable

Après avoir passé la récupération des apis, on prend le JMP en 0040F19B et on arrive directement avant l'appel de VirtualAlloc donc voici une analyse détaillée des paramètres ci-dessous :

```
0040F1B0    PUSH 40     ; PAGE EXECUTE READWRITE 0x40. Enables execute,
read-only, or read/write access to the committed region of pages.
0040F1B2    PUSH 1000   ; 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  ; The size of the region, in bytes.
0040F1BC    PUSH 0      ; If this parameter is NULL, the system determines
where to allocate the region.
0040F1BE    CALL EBX    ; kernel32.VirtualAlloc
```

Dans EAX on repère l'adresse où la mémoire a été allouée, 0x00910000h chez moi. Suivons cette valeur dans le dump afin de voir quel contenu y sera copié plus tard. Je précise que toutes les zones mémoire allouées depuis le début et jusqu'à la fin de ce tutoriel ont des adresses susceptibles de varier chez vous, ne vous étonnez donc pas de la différence. Dans ce cas présent, le choix de l'adresse d'allocation étant laissé à l'ordinateur, il est peu probable que vous ayez la même :)

```
0040F1CE    MOV EBX,500299D5
0040F1D3    SUB EBX,50029651        ; EBX = 384h
0040F1D9    ADD EAX,EBX             ; on obtient une nouvelle adresse
0040F1DB    PUSH EAX                ; EAX=0040F551 (sm.0040F551)
0040F1DC    CALL <sm.copy_dll>
```

Jetons un coup d'œil dans ce CALL. Nous remarquons quatre REP MOVS [xxx] qui sont des instructions utilisées pour recopier des zones mémoire d'une adresse à une autre. Prenons un exemple ci-dessous, avec le premier de ceux-ci :

```
0040F2DB    REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
DS:[ESI]=[0040F551]=00905A4D  ; Octets à copier (nous remarquons qu'il
; s'agit des quatre premiers octets d'un fichier PE (signature MZ)
ES:[EDI]=[00910000]=00000000  ; Zone qui reçoit, pour l'instant vide
```

Ces instructions vont donc copier dans la zone mémoire que nous venons d'allouer un fichier au format PE qui était contenu dans l'exécutable. Voici un aperçu du dump après que la copie soit finie (l'adresse d'allocation a changée, le screen ayant été fait plus tard) (Fig. 12) :

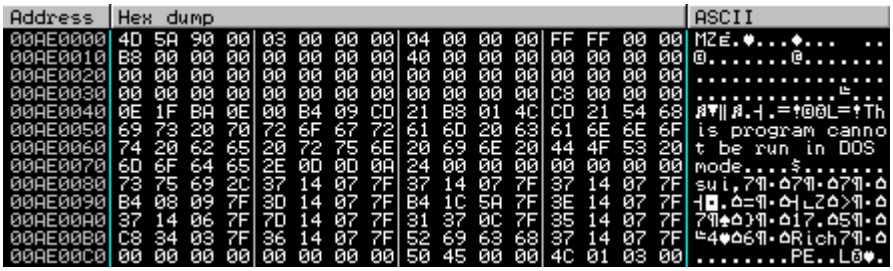

Figure 12 : Copie de la dll en mémoire

- Arrivée à l'OEP

Une fois sortis de la routine précédente, nous nous retrouvons face à un call qui appelle la dll que nous venons de voir recopiée en mémoire. Pour des raisons de convenance, j'analyserai son action dans la partie n°4, sa place me paraissant plus appropriée dans l'analyse des payloads du malware que dans la section traitant de l'unpacking de ce binaire malicieux.

```
0040F404    CALL EDX    ; appelle la dll
```

On retourne ensuite dans le code de l'exe. On passe du stuff déjà vu (IAT, layers, etc.). Au fil du tracing nous arrivons à ceci :

```
0040D73F    PUSH EAX
0040D740    PUSH 4              ; PAGE_WRITECOPY
0040D742    PUSH 1000           ; taille de la région concernée
0040D747    PUSH EDX            ; 00400000h, adresse de la région concernée
0040D748    CALL DWORD PTR SS:[EBP+16]       ; kernel32.VirtualProtect
```

En faisant Alt+M juste avant dans Olly, nous nous apercevons que la colonne "Access" en face de notre processus est à R (READONLY). Après l'exécution de VirtualProtect, cette colonne prend la valeur "RW CopyOnWr". Il semblerait donc que le paramètre *fdwNewProtect* contenait la valeur PAGE_WRITECOPY, ce qui d'après la documentation Microsoft donne un accès en copie & écriture à la région désirée.

Nous rencontrons au cours de la suite deux VirtualAlloc et quelques layers, avant d'atteindre les appels à VirtualFree proches de l'OEP du programme. Je conclurais ici cette partie III), la suite ne présentant qu'un intérêt limité à l'étude. Passons maintenant à l'étude du comportement de l'exe et de sa dll.

ANNEXES

4) Payload du malware

a) Etude de la DLL *embedded*

On rentre dans une routine UPX v. 2.03 (Fig. 14), reconnaissable au premier coup d'œil (Fig. 13) :

```
00C92940   807C24 08 01    CMP BYTE PTR SS:[ESP+8],1
00C92945 v 0F85 B9010000   JNZ 00C92B04
00C9294B   60              PUSHAD
00C9294C   BE 00F0C800     MOV ESI,0C8F000
00C92951   8DBE 0020FDFF   LEA EDI,DWORD PTR DS:[ESI+FFFD2000]
00C92957   57              PUSH EDI
00C92958   83CD FF         OR EBP,FFFFFFFF
00C9295B v EB 0D           JMP SHORT 00C9296A
00C9295D   90              NOP
00C9295E   90              NOP
```

Figure 13 : Routine UPX

L'unpacking est tout ce qu'il y a de plus classique, je ne reviendrai pas dessus une énième fois. Passons donc ceci et intéressons nous au corps de la dll.

```
000003C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000003D0  00 00 00 00 00 00 00 00  00 00 00 32 2E 30 33 00  ...........2.03.
000003E0  55 50 58 21 0D 09 02 08  DE 82 A2 1E FC 4F FE 1B  UPX!....Þ‚¢.üOþ.
000003F0  08 07 03 00 37 39 00 00  00 A0 00 00 26 04 00 0E  ....79... ..&...
```

Figure 14 : Version d'UPX

- Récupération de la langue du système

Nous passons donc l'EP et nous intéressons à l'action effectuée par le malware juste après. Celui-ci va appeler l'api GetACP, qui sert à récupérer une valeur correspondant à la langue active sur le système comme illustré dans le code ci-dessous :

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

On sait donc que le malware va tenter de déterminer la langue du système. On remarque à la ligne suivante qu'une comparaison est faite avec la valeur de retour de l'api, qui se trouve dans EAX. Nous obtenons ceci :

```
0x348h = 936      Chinese (PRC, Singapore)
0x4E4h = 1252     Windows 3.1 Latin 1 (US, Western Europe)
```

La valeur qui est retournée est 0x4E4h, qui correspond à un Windows situé dans l'Europe de l'ouest (France ici) / Etats-Unis. On lui compare la valeur 0x348h, qui correspond à un système d'exploitation ayant le chinois simplifié comme langage. Il n'y a que deux régions où ce langage est utilisé : La République Populaire de Chine (PRC) et Singapour. En effet le code ANSI pour Hong Kong et Taïwan diffère. Nous avons donc une idée assez précise de la localisation de l'auteur de notre malware (quoiqu'en dise l'article dans MISC, pour moi les preuves de l'origine de cette menace sont suffisamment claires pour pouvoir en tirer des conclusions sans trop de risques de commettre une erreur…). Si le résultat de la comparaison est égal donc, on prend le JE pour aller directement à la fin de la routine, la dll n'effectue donc aucune action particulière.
On rentre ensuite dans le CALL juste après :

```
0091355A    PUSH EAX          ; pHandle = 0012FF3C
0091355B    PUSH 20019        ; Access = KEY_READ
00913560    PUSH EDI          ; Reserved = 0
00913561    PUSH 918A1C     ; Subkey = "SYSTEM\CurrentControlSet\Control\Nls\Language"
00913566    PUSH 80000002     ; hKey = HKEY_LOCAL_MACHINE
00913572    CALL DWORD PTR DS:[914018]    ; advapi32.RegOpenKeyExA
```

Le malware s'ouvre un handle sur le chemin de la clé qui l'intéresse, et tente ensuite de lire une clé située dans HKEY_LOCAL_MACHINE, qui correspond à la langue d'installation de Windows.

```
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
```

Il regarde ensuite la valeur de la langue d'installation
0804 = "zh-cn;Chinese (China)"
040C = "fr;French (France)"

```
009135AC    PUSH 918A04       ; ASCII "0804"
009135B1    PUSH EAX          ; valeur InstallLanguage, 040C
009135B2    CALL 00913BC0     ; compare les deux
```

A partir d'ici c'est une analyse partielle et incomplète :

```
009137A3    MOV EBX,918A78; ASCII "MN_XADLEBDGTWUIAIHDIIASDOOAOIDCDDD0"
009137B1    PUSH EBX          ; nom de l'event
009137B2    PUSH ESI          ; push 0
009137B3    PUSH 1F0003       ; EVENT_ALL_ACCESS (0x1F0003)
009137B8    CALL EBP         ; kernel32.OpenEventA

009137C9    PUSH EBX ; EventName = "MN_XADLEBDGTWUIAIHDIIASDOOAOIDCDDD0"
009137CA    PUSH 1            ; InitiallySignaled = TRUE
009137CC    PUSH ESI          ; ManualReset = FALSE
009137CD    PUSH ESI          ; pSecurity = NULL
009137CE    CALL DWORD PTR DS:[91408C]     ; kernel32.CreateEventA
```

Je suppose que c'est un test pour savoir si le malware est déjà en exécution : suivant le résultat de OpenEvent, on sait si le om existe déjà ou pas, si c'est le cas on passe à la suite, sinon on créé l'event (un peu comme pour les mutex en fait). Après je sais pas pourquoi il fait ce test sur deux events différents.

```
009137E5    PUSH EAX    ; pThreadId = 0012FF54
009137E6    PUSH ESI    ; CreationFlags = 0
009137E7    PUSH ESI    ; pThreadParm = NULL
009137E8    PUSH 912B01 ; ThreadFunction = 00912B01 // thread anti av
009137ED    PUSH ESI    ; StackSize = 0
009137EE    PUSH ESI    ; pSecurity = NULL
009137EF    CALL DWORD PTR DS:[914088]    ; kernel32.CreateThread
```

Tout est dans le commentaire ☺
Je n'ai pas vraiment réussi à cerner l'intérêt de l'autre thread.

Thread anti av :

```
00C62B3E    MOV DWORD PTR SS:[EBP-4],0C66540  ; ASCII "LIVESRV.EXE(COMMON)"
```

⇨ Nom du module à rechercher

```
00C613C3    CALL 00C6386C        ; JMP to kernel32.CreateToolhelp32Snapshot
```

⇨ Capture tout les processus en cours

```
00C613DC    CALL 00C63866          ; JMP to kernel32.Process32First
```

⇨ Prendre le premier

```
00C613E9    CALL 00C61012   ; test si les noms des processus sont les mêmes
00C613EF    TEST EAX,EAX
00C613F2    JNZ SHORT 00C61407      ; si eax = 0 continue (pas trouvé)

00C613FC    CALL 00C63860          ; JMP to kernel32.Process32Next
```

⇨ Si ce n'était pas le bon prend le processus suivant dans le snapshot

Quand l'énumération est finie (et si le processus est trouvé) : on fait un OpenProcess dessus. Je ne suis pas allé plus loin, la liste des noms recherché se situe à la fin de cette section.

Dans une autre proc il va rechercher les modules d'un processus via PSAPI.EnumProcessModules ; puis va récupérer leur nom via PSAPI.GetModuleFileNameExA. On a ensuite une recherche des extensions ".vcd", "*.dll", "*.exe" via kernel32.FindFirstFileA & kernel32.FindNextFileA. Je n'ai

pas regardé pourquoi faire ni ce qu'il recherchait précisément. Une autre procédure va relever des informations sur le disque C:\ via un call à kernel32.GetVolumeInformationA. On a ensuite ceci dans le code :

```
00C61214     PUSH 0C68730                    ; ASCII "FAT32"
00C61219     PUSH EAX                        ;  ASCII "NTFS"
00C6121A     CALL 00C61012
```

Ceci sera ensuite suivi d'appels à kernel32.CreateFileA, kernel32.DeviceIoControl et kernel32.GetFileSize. Je n'ai pas cherché à voir à quoi ils correspondaient.

Les informations ici sont très parcellaires, car elles sont issues du peu d'analyse statique que j'avais fait sur la dll ; beaucoup de ces procédures n'ayant pas été exécutées au cours du debugging du programme. A noter aussi que le serveur vers lequel les requêtes sont lancées ne répondait plus.

Liste des processus recherché via les différentes fonctions :

Stack address = 00EDFC2A, (ASCII "minisniffer.exe;smartsniff.exe;packetcapture.exe;peepnet.exe;capturenet.exe;wireshark.exe;aps.exe;sockmon5.exe;gametroyhorsedetect.exe;filemon.exe;regmon.exe;")

```
00EDFC1C  63 61 70 74 75 72 65 3B 73 6E 69 66 66 3B D7 A5  capture;sniff;×¥
00EDFC2C  B0 FC 3B 73 79 73 69 6E 74 65 72 6E 61 6C 73 3B  °ü;sysinternals;
00EDFC3C  00 72 74 73 6E 69 66 66 2E 65 78 65 00 70 61 63  .rtsniff.exe.pac
00EDFC4C  6B 65 74 63 61 70 74 75 72 65 2E 65 78 65 00 70  ketcapture.exe.p
00EDFC5C  65 65 70 6E 65 74 2E 65 78 65 00 63 61 70 74 75  eepnet.exe.captu
00EDFC6C  72 65 6E 65 74 2E 65 78 65 00 77 69 72 65 73 68  renet.exe.wiresh
00EDFC7C  61 72 6B 2E 65 78 65 00 61 70 73 2E 65 78 65 00  ark.exe.aps.exe.
00EDFC8C  73 6F 63 6B 6D 6F 6E 35 2E 65 78 65 00 67 61 6D  sockmon5.exe.gam
00EDFC9C  65 74 72 6F 79 68 6F 72 73 65 64 65 74 65 63 74  etroyhorsedetect
00EDFCAC  2E 65 78 65 00 66 69 6C 65 6D 6F 6E 2E 65 78 65  .exe.filemon.exe
00EDFCBC  00 72 65 67 6D 6F 6E 2E 65 78 65                 .regmon.exe
```

00DDFC54 00C66540 ASCII "LIVESRV.EXE(COMMON)"
- ⇨ Processus BitDefender security Update Service

00DDFC54 00C66560 ASCII "BDAGENT.EXE"
- ⇨ Processus BitDefender Agent

00DDFD94 00C66840 ASCII "VCRMON.EXE"
- ⇨ Virus Monitor de Virus Chaser

00DDFC54 00C66B40 ASCII "CCSVCHST.EXE"
- ⇨ Symantec Service Framework

00DDFC54 00C66B60 ASCII "ALUSCHEDULERSVC.EXE"
- ⇨ Recherche de mises à jour pour les logiciels de symantec

00DDFC54 00C66E40 ASCII "ASHDISP.EXE"
- ⇨ Processus appartenant à Avast

00DDFC54 00C67140 ASCII "EKRN.EXE"
- ⇨ Service de l'antivirus ESET

00DDFC54 00C67440 ASCII "AVP.EXE"
- ⇨ Module de Kaspersky

00DDFC54 00C67740 ASCII "AYAGENT.AYE"
⇨ Processus appartenant à l'AV Alyac ?
00DDFC54 00C67A40 ASCII "UFSEAGNT.EXE"
⇨ TrendMicro Server Agent
00DDFC54 00C67D40 ASCII "AVGNT.EXE"
⇨ Processus appartenant à Avira Internet Security
00DDFC54 00C68040 ASCII "VSTSKMGR.EXE"
⇨ MacAfee Virus Scan Task Manager
00DDFC54 00C68340 ASCII "AVGRSX.EXE"
⇨ Processus d'AVG Internet Security

Contient plusieurs url :

ESI 01110000 ASCII "http://www.sohucct.com/1/t.rar"
Peut-être une sample de win32.RaMag.a

EBX 00407035 ASCII "http://kmcmapo.net/link/img/s.exe;"
L'url vers laquelle le main exe lance une requête via le internet explorer lancé en processus caché.

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

date : 2010-11-11 08:32:21
closed : 2010-11-12 19:50:02

5) Conclusion et remerciements

Le reversing de ce packer custom était au final très simple, car il n'implémente que des techniques classiques et n'utilise pas de tricks particuliers comme des anti-vm, anti-dump ou anti-debuggers, qui sont monnaie courante pour les fakes av par exemple. Les deux seules « protections » que l'on a pu remarquer étaient le recours aux exceptions et l'utilisation de junk code. On remarque aussi que ce malware n'est packé qu'une fois avec le packer custom ; bien souvent ceux-ci sont composés de deux couches de packing : une custom pour la protection et une UPX pour la compression.

Voilà, s'en est finalement fini du paper ici, je vous laisse la suite en annexe (sait on jamais que ça intéresse des gens). Peut être que je ferais une prochaine fois un papier sur mystic compressor, ou bien sur l'unpacking d'un fake AV ; je n'en sais rien.

Je tiens à remercier :

Shub-Nigurrath / ARTeam, pour le template de mise en page
Les gens que je côtoie au quotidien sur irc et que j'apprécie :)
Un merci particulier aux relecteurs de l'article :)

6) Références

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

[5] : - Kernel32.dll → GetModuleHandleA
- User32.dll → DefFrameProcW
- Advapi32.dll → QueryServiceConfigA
- Shell32.dll → SHGetDiskFreeSpaceA
- Gdi32.dll → GetBkColor

[6] : http://fat.vxer.org/data/win32tut.txt
[7] : http://www.woodmann.com/crackz/Tutorials/Seh.htm
[8] : https://secure.wikimedia.org/wikipedia/fr/wiki/Endianness

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/

Articles de MISC intéressants (techniques semblables) :

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
MISC n°56 : Win32/PSW.OnlineGames.OUM : c'est pas du jeu !, pages 11 à 15

7) Historique

• Version 1.0 : first public release