Mutation d'un virus

Abstract
Ce document présente les méthodes de détection des *malwares*. Il montre également les faiblesses des technologies utilisées en s'orientant sur des codes polymorphiques.

La dernière version de ce document est accessible sur le Web à l'adresse :
http://esl.epitech.net/~arnaud/mutation-d'un-virus

Pour toutes questions et commentaires concernant ce document, merci de contacter:
maillard.arnaud@gmail.com

Table des matières

Introduction

Depuis l'apparition des premiers virus, le marché des produits de sécurité s'est considérablement développé. Pour les créateurs de codes malicieux - autant que pour ceux essayant de les circonvenir - il a fallu -inventer de nouvelles méthodes d'infection et de détection.

Les premiers codes auto-reproducteurs étaient bien des prouesses, mais des prouesses où les architectures matérielles et logicielles ne restreignaient aucun accès, que ce soit sur les fichiers, la mémoire, ou même l'écran ou le disque dur. De plus, rien ne venait contrôler leur prolifération au sein d'une machine, et le concept d'anti-malwares n'existait pour ainsi dire pas.

Aujourd'hui, cette situation a considérablement changé. Processeurs et systèmes d'exploitations restreignent les accès, le parc d'utilisateur s'est étoffé, et les codes malicieux doivent tenir une course permanente avec les outils visant à leur élimination.

On s'intéresse dans ce document aux termes de cette course, présentant d'abord comment on peut affirmer de la présence d'un virus sur une machine, puis comment s'évader de ces mécanismes.

Détection d’un virus

Il existe nombres de méthodes pour détecter les virus - ou au moins avertir l’utilisateur d’un fichier suspect - mais toutes peuvent se classer dans au moins l’une de ces trois familles:

- détection par signature
- détection par heuristiques
- détection par émulation

Détection par signature

Une signature est une suite d’octets caractéristiques, une séquence identifiable servant à identifier la présence d’un code malicieux à l’intérieur d’un exécutable. Il s’agit d’un *pattern* dégageant des propriétés, celui-ci pouvant être construit autour d’un modèle plus ou moins complexe, pouvant tirer profit des expressions régulières.

Un antivirus par signature procède donc à une analyse minutieuse, comparant deux suites d'octets. La première constitue le fichier soumis à vérification, la seconde déclarée comme constituante d'un code malicieux. Ce type de logiciel est spécifique, n’étant capable de reconnaître une menace que si elle est connue et présente dans ses bases de donnée.

Un exemple de signature: les fichiers EICAR

Le code EICAR est un code compris dans les bases de l’ensemble des produits de sécurité du marché. Chaque fois que l’un d’eux le rencontre, il le signale par convention comme un virus, même si ce code ne présente aucune menace.

Historiquement, cette signature a été intégrée pour que les utilisateurs puissent vérifier que le logiciel qu’il venait d’acquérir fonctionnait correctement, qu’il réagissait à la présence de menaces présentes sur la machine. Divers organismes indépendants ont essayé de faire la même chose, sans succès toutefois.

Le code EICAR se présente comme suit :

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

En copiant ce texte dans un éditeur de texte puis modifiant l’extension du fichier, on obtient un fichier executable au format COM.


```
$ hexdump eicar.com
0000000 58 35 4f 21 50 25 40 41 50 5b 34 5c 50 5a 58 35
0000010 34 28 50 5e 29 37 43 43 29 37 7d 24 45 49 43 41
0000020 52 2d 53 54 41 4e 44 41 52 44 2d 41 4e 54 49 56
0000030 49 52 55 53 2d 54 45 53 54 2d 46 49 4c 45 21 24
0000040 48 2b 48 2a
0000044
```

Détection par heuristiques

Une heuristique peut être considéré comme un schéma, un fait vérifiable concernant:

- la structure d’un fichier exécutable
- le code contenue à l’intérieur de ce fichier

La détection par heuristique se base sur l'idée que la plupart des codes malicieux présentent des caractéristiques communes et génériques, isolables comme le sont les signatures, mais n'en possédant pas la rigidité.

Heuristiques sur le format

La plupart des virus vont effectuer des opérations mettant à jour différents champs dans la structure PE (*Portable Executable*) du fichier infecté. Chaque heuristique va alors pouvoir inspecter ces modifications et les rapporter à l'utilisateur.

- point d'entrée sur la dernière section

Il est très inhabituel que le champ *EntryPoint* du PE pointe ailleurs que sur la première section.

- point d'entrée avant la première section

Ce cas est plus rare que le premier mais est utilisé quelquefois, constituant un comportement suspicieux manifeste.

- SizeOfCode

Le champ *SizeOfCode* comptabilise les octets contenus dans les sections marqués *execute*. La plupart des virus ne le mettent pas à jour, le loader Windows ne le vérifiant pas.

- caractéristiques des sections

En général, les compilateurs et assembleurs forment la première section comme étant la seule exécutable. Il est donc suspect de trouver la dernière section du fichier comme présentant la même caractéristique.

Dans le même ordre d'idée, il est suspect de trouver une section marquée des en même temps des drapeaux *execute* et *writable*.

- nom et caractéristiques des sections

On rapporte ici le nom d'une section avec les caractéristiques qui lui ont été attribué. Il est par exemple illogique que la section des ressources, usuellement nommée « .rsrc », possède le drapeau *execute*.

Heuristiques sur le corps du programme

Ici, les heuristiques vont s'intéresser au code exécutable d'une application, y recherchant des comportements caractéristiques, ou au moins suspicieux.

- redirection du flux d'exécution

Les virus ne modifiant pas l'EntryPoint du fichier infecté peuvent substituer aux premières instructions originales une opération de modification du flux (JMP / CALL).

Il est suspect que les instructions d'une section exécutable appellent des instructions d'une autre section exécutable.

Il est également suspect de trouver une séquence d'instruction de type PUSH / RET.

- recherche et ouverture de fichiers PE

Pour infecter d'autres programmes, le virus va devoir utiliser les APIs Windows de recherche et d'ouverture de fichier (FindFirstFile, FindNextFile, CreateFile, etc.).

- chaînes de caractères

En plus de l'utilisation d'API, le virus peut embarquer des chaînes de caractères suspectes, par exemple « *.exe » ou les signatures DOS et PE des fichiers trouvés par le couple FindFirstFile / FindNextFile.

- appel d’APIs via une adresse mémoire

Certains virus appellent les APIs Windows en utilisant une adresse mémoire, comportement très improbable pour un programme sain.

- récupération de limageBase de KERNEL32

Pour utiliser les APIs Windows, un virus - s’il n’adresse pas directement l’espace de KERNEL32.DLL - doit retrouver où se situe la DLL en mémoire.

- delta offset

La technique du delta offset est utilisée dans nombre de virus pour obtenir un code « position independent ».

Détection par émulation

Une méthode simple pour contrecarrer les méthodes mises en oeuvre par les précédentes techniques de détection est le chiffrement du corps du virus, ajoutant à l’infection un mécanisme de chiffrement et d'exécution du code, cette-fois sous une forme compréhensible par le processeur.

Un produit de sécurité analysant un programme chiffré pourra bien tester les heuristiques sur la structure PE, mais seront faussées:

- la recherche de signature
- les heuristiques sur le corps du programme

Le flux analysé n’étant pas le code réellement exécuté, le produit a de grandes chances de signaler un fichier sain alors qu’il est infecté.

Le principe de l’émulation est de laisser s'exécuter le programme dans un environnement cloisonné puis de commencer l’analyse à partir d’un certain point, faisant entrer en jeu les principes évoqués précédemment.

Mutation d’un virus

Le chiffrement du virus sert à contrer son identification. Le but ici est d’empêcher une méthode générique de désassemblage, donc l’identification d’une signature.

Dans les exemples suivants, correspondant à une possible première génération d’un virus, la section de code n’a pas la propriété *write*. On utilisera donc le programme ci-dessous.

```
/**
 ** pewrite.c
 ** changes the write attribute of the code section in a PE file
 **
 **/

#include <windows.h>
#include <stdio.h>

int     main(int argc, char **argv)
{
  PIMAGE_SECTION_HEADER pImageSectionHeader;
  PIMAGE_DOS_HEADER     pImageDosHeader;
  PIMAGE_NT_HEADERS     pImageNtHeaders;
  PIMAGE_FILE_HEADER    pImageFileHeader;
  HANDLE                hFile;
  HANDLE                hFileMappingObject;
  LPVOID                lpFileMappedView;
  DWORD                 dwCount;

  if (argc < 2)
    return (-1);

  hFile = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ,
                     NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
  if (hFile == INVALID_HANDLE_VALUE)
    return (-1);

  hFileMappingObject = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0,
                                         NULL);
  if (hFileMappingObject == NULL)
    return (-1);

  lpFileMappedView = MapViewOfFile(hFileMappingObject, FILE_MAP_ALL_ACCESS,
0,
                                   0, 0);
  if (lpFileMappedView == NULL)
    return (-1);
```

```
  pImageDosHeader = (PIMAGE_DOS_HEADER) lpFileMappedView;
  if (pImageDosHeader->e_magic != IMAGE_DOS_SIGNATURE)
    return (-1);

  pImageNtHeaders = (PIMAGE_NT_HEADERS) ((DWORD) lpFileMappedView +
                                         pImageDosHeader->e_lfanew);
  if (pImageNtHeaders->Signature != IMAGE_NT_SIGNATURE)
    return (-1);

  pImageSectionHeader = (PIMAGE_SECTION_HEADER) ((DWORD) pImageNtHeaders +
                                                 sizeof (IMAGE_NT_HEADERS));

  for (dwCount = 0;
       dwCount != pImageNtHeaders->FileHeader.NumberOfSections;
       dwCount++) {
    if (pImageNtHeaders->OptionalHeader.BaseOfCode ==
        pImageSectionHeader->VirtualAddress)
      pImageSectionHeader->Characteristics |= IMAGE_SCN_MEM_WRITE;
    pImageSectionHeader = (PIMAGE_SECTION_HEADER)
      ((DWORD) pImageSectionHeader + sizeof (IMAGE_SECTION_HEADER));
  }

  return (0);
}
```

Chiffrement et déchiffrement

Les programmes suivants illustre le chiffrement d'un programme. On applique simplement une opération de type XOR.

Manipuler les opcodes

```
; m1.asm

.586
.model flat, stdcall
option casemap :none

.code

start:

      mov    eax,1
      mov    ebx,2
      mov    ecx,3
      mov    edx,4

      ret
```

```
end    start

0:000> u $exentry
image00400000+0x1000:
00401000 b801000000      mov     eax,1
00401005 bb02000000      mov     ebx,2
0040100a b903000000      mov     ecx,3
0040100f ba04000000      mov     edx,4
00401014 c3              ret
```

```
; m2.asm
; code XOR 0x06

.586
.model flat, stdcall
option casemap :none

.code

start:

       mov    esi,code_begin
       mov    edi,code_begin
       mov    ecx,code_end - code_begin

l:
       lodsb
       xor    eax,06h
       stosb
       loop   l

       jmp    code_end

code_begin:
       mov    eax,1
       mov    ebx,2
       mov    ecx,3
       mov    edx,4
code_end:

       ret

end    start

0:000> u $exentry
image00400000+0x1000:
00401000 be18104000      mov     esi,offset image00400000+0x1018 (00401018)
00401005 bf18104000      mov     edi,offset image00400000+0x1018 (00401018)
0040100a b914000000      mov     ecx,14h
0040100f ac              lods    byte ptr [esi]
00401010 83f006          xor     eax,6
00401013 aa              stos    byte ptr es:[edi]
00401014 e2f9            loop    image00400000+0x100f (0040100f)
00401016 eb14            jmp     image00400000+0x102c (0040102c)
00401018 b801000000      mov     eax,1
0040101d bb02000000      mov     ebx,2
```

```
00401022 b903000000      mov     ecx,3
00401027 ba04000000      mov     edx,4
0040102c c3              ret

0:000> g

0:000> u $exentry
image00400000+0x1000:
00401000 be18104000      mov     esi,offset image00400000+0x1018 (00401018)
00401005 bf18104000      mov     edi,offset image00400000+0x1018 (00401018)
0040100a b914000000      mov     ecx,14h
0040100f ac              lods    byte ptr [esi]
00401010 83f006          xor     eax,6
00401013 aa              stos    byte ptr es:[edi]
00401014 e2f9            loop    image00400000+0x100f (0040100f)
00401016 eb14            jmp     image00400000+0x102c (0040102c)
00401018 be07060606      mov     esi,6060607h
0040101d bd04060606      mov     ebp,6060604h
00401022 bf05060606      mov     edi,6060605h
00401027 bc02060606      mov     esp,6060602h
0040102c c3              ret
```

```
; m3.asm
; (code XOR 0x06) XOR 0x06

.586
.model flat, stdcall
option casemap :none

.code

start:

      mov   esi,code_begin
      mov   edi,code_begin
      mov   ecx,code_end - code_begin

l:
      lodsb
      xor   eax,06h
      stosb
      loop  l

code_begin:
      db    0beh
      dd    06060607h
      db    0bdh
      dd    06060604h
      db    0bfh
      dd    06060605h
      db    0bch
      dd    06060602h
code_end:

      ret
```

```
end    start

0:000> u $exentry
image00400000+0x1000:
00401000 be16104000      mov     esi,offset image00400000+0x1016 (00401016)
00401005 bf16104000      mov     edi,offset image00400000+0x1016 (00401016)
0040100a b914000000      mov     ecx,14h
0040100f ac              lods    byte ptr [esi]
00401010 83f006          xor     eax,6
00401013 aa              stos    byte ptr es:[edi]
00401014 e2f9            loop    image00400000+0x100f (0040100f)
00401016 be07060606      mov     esi,6060607h
0040101b bd04060606      mov     ebp,6060604h
00401020 bf05060606      mov     edi,6060605h
00401025 bc02060606      mov     esp,6060602h
0040102a c3              ret

0:000> bp 0040102a

0:000> g

0:000> r eax,ebx,ecx,edx
eax=00000001 ebx=00000002 ecx=00000003 edx=00000004
```

Polymorphisme

Le but du polymorphisme est de brouiller l'ensemble des signatures potentielles, extractibles à partir d'un virus. La naissance de cette technique est étroitement liée à la situation initiale des programmes antivirus, soit des bases de signatures. Le polymorphisme tend à éliminer toutes les parties fixes du corps d'un virus.

En effet, même avec une technique de chiffrement, quelques octets restent fixes. Le décrypteur ne change pas. Alors plutôt que de s'attaquer au corps du virus proprement dit, les chercheurs des laboratoires antivirus ont simplement identifié les décrypteurs. La signature passait ici de constituante d'un code malicieux à … constituante du mécanisme de déchiffrement du code.

Le polymorphisme signifie donc la génération de multiples décrypteurs. Le principe est simple : ne jamais produire de décrypteurs identiques, avec pour contrainte de garder un schéma opératif intact.

Idée générale

Puisque le polymorphisme est essentiellement une génération de décrypteurs, il faut bien penser le fonctionnement interne de celui-ci. Usuellement on aura un code similaire à celui-ci:

```
      mov   ecx,(taille du code à chiffrer/déchiffrer)
      lea   edi,(pointeur vers le code à chiffrer/déchiffrer)
      mov   eax,(clé de chiffrement/déchiffrement)

Boucle :
      xor   dword ptr [edi],eax
      add   edi,4
      loop  Boucle
```

Ce code est facilement marquable. On peut aisément en extraire une signature. Quels sont les éléments qui s'offrent à nous pour rendre ce code moins identifiable ? On peut :

- modifier les registres utilisés
- modifier l'ordre des instructions

- utiliser des instructions « équivalentes »
- insérer des instructions ne servant à rien (*junk code*)

Un moteur de polymorphisme va donc se décomposer essentiellement en deux parties:

- Un générateur de nombres pseudo aléatoires
- Un i générateur de décrypteur

Concepts

La génération de nombres

Le générateur de nombres pseudo aléatoires (GNPA) est utile puisqu'il va nous permettre d'opérer des changements sur le code produit.

Par exemple, si on veut utiliser un registre au hasard, ou produire une valeur immédiate aléatoire (*imm32*), ou encore générer des *opcodes*.

Un générateur pauvre en entropie pourrait se baser sur l'API **GetSystemTime**.

```
; m4.asm
; using GetSystemTime

.586
.model flat, stdcall
option casemap :none

  include \masm32\include\windows.inc
  include \masm32\include\user32.inc
  include \masm32\include\kernel32.inc

  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib

.data

  _SYSTEMTIME       STRUC
  Year        dw    ?
  Month             dw    ?
  DayOfWeek   dw    ?
  Day         dw    ?
  Hour        dw    ?
  Minute            dw    ?
  Second            dw    ?
  Milliseconds      dw    ?
  _SYSTEMTIME       ENDS

  STime       _SYSTEMTIME  <>

.code

start:

  call random

  push 0
  call ExitProcess

random proc
```

```
  push offset STime
  call GetSystemTime
  xor  eax,eax
  mov  ax,[STime.Milliseconds]
  ret
random endp

end    start
```

A la sortie de ce programme, le registre AX contient un nombre généré aléatoirement.

La structure SYSTEMTIME ne comportant que des WORD, il serait judicieux d'améliorer la sortie de la fonction de génération en utilisant des DWORD (en décalant les bits par exemple).

Il est très important de se concentrer sur cette partie du système polymorphique, puisque plus le générateur est bon, plus il va nous permettre d'encoder des opérations diverses, et donc d'améliorer la génération du code.

La génération de code

D'abord, améliorons le GNPA en lui fournissant une fonction par laquelle on peut spécifier une valeur maximale.

```
; m5.asm
; using GetSystemTime with RNDMAX

.586
.model flat, stdcall
option casemap :none

       include \masm32\include\windows.inc
       include \masm32\include\user32.inc
       include \masm32\include\kernel32.inc

       includelib \masm32\lib\user32.lib
       includelib \masm32\lib\kernel32.lib

.data

       _SYSTEMTIME  STRUC
       Year         dw     ?
       Month        dw     ?
       DayOfWeek    dw     ?
       Day          dw     ?
       Hour         dw     ?
       Minute       dw     ?
       Second       dw     ?
       Milliseconds dw     ?
       _SYSTEMTIME  ENDS

       STime  _SYSTEMTIME  <>
       RNDMAX equ    33

.code

start:

       mov    eax,RNDMAX
       call   rand
```

```
        push   0
        call   ExitProcess

rand    proc

        push   ecx
        push   edx
        mov    ecx,eax
        call   random
        xor    edx,edx
        div    ecx
        mov    eax,edx
        pop    edx
        pop    ecx
        ret

rand    endp

random  proc

        pusha

        push   offset STime
        call   GetSystemTime

        popa

        xor    eax,eax
        mov    ax,[STime.Milliseconds]

        ret

random  endp

end     start
```

Dans cet exemple, la valeur retournée dans EAX ne dépassera pas RNDMAX. Ceci est très utile si on veut codifier les registres.

```
        _EAX   equ    0
        _ECX   equ    1
        _EDX   equ    2
        _EBX   equ    3
```

De cette manière, nous pouvons désormais gérer les instructions qui font intervenir un registre (MOV REG,IMM32 par exemple).

```
; m6.asm
; generates MOV reg32, imm32

.586
.model flat, stdcall
option casemap :none

        include \masm32\include\windows.inc
        include \masm32\include\user32.inc
        include \masm32\include\kernel32.inc

        includelib \masm32\lib\user32.lib
```

```
        includelib \masm32\lib\kernel32.lib

.data

        _SYSTEMTIME  STRUC
        Year         dw     ?
        Month        dw     ?
        DayOfWeek    dw     ?
        Day          dw     ?
        Hour         dw     ?
        Minute       dw     ?
        Second       dw     ?
        Milliseconds dw     ?
        _SYSTEMTIME  ENDS

        STime  _SYSTEMTIME  <>

        RNDMAX equ    4

        _EAX   equ    0
        _ECX   equ    1
        _EDX   equ    2
        _EBX   equ    3

.code

start:
        jmp    realstart

codebuffer:
        db     0            ; mov reg32
        dd     0            ; imm32
        db     0            ; ret

realstart:
        mov    edi,offset codebuffer

        mov    eax,RNDMAX
        call   rand

        or     al,0B8h
        stosb

        call   random
        stosd

        mov    al,0c3h             ; ret opcode
        stosb

        push   offset _end
        jmp    codebuffer

_end:
        push   0
        call   ExitProcess

rand   proc
```

```
        push   ecx
        push   edx
        mov    ecx,eax
        call   random
        xor    edx,edx
        div    ecx
        mov    eax,edx
        pop    edx
        pop    ecx
        ret

rand    endp

random proc

        pusha
        push   offset STime
        call   GetSystemTime
        popa
        xor    eax,eax
        mov    ax,[STime.Milliseconds]
        shl    eax,16
        mov    ax,[STime.Second]
        ret

random endp

end     start
```

```
0:000> u $exentry
image00400000+0x1000:
00401000 eb06            jmp     image00400000+0x1008 (00401008)
00401002 0000            add     byte ptr [eax],al
00401004 0000            add     byte ptr [eax],al
00401006 0000            add     byte ptr [eax],al
00401008 bf02104000      mov     edi,offset image00400000+0x1002 (00401002)
0040100d b804000000      mov     eax,4
00401012 e81a000000      call    image00400000+0x1031 (00401031)
00401017 0cb8            or      al,0B8h
...

0:000> bp 00401028

0:000> g
image00400000+0x1028:
00401028 ebd8            jmp     image00400000+0x1002 (00401002)

0:000> u 00401002
image00400000+0x1002:
00401002 ba2a001400      mov     edx,14002Ah
00401007 c3              ret

0:000> .reload

0:000> g
image00400000+0x1028:
00401028 ebd8            jmp     image00400000+0x1002 (00401002)
```

```
0:000> u 00401002
image00400000+0x1002:
00401002 bb0b008302      mov     ebx,283000Bh
00401007 c3              ret
```

Le même principe peut être appliqué aux instructions du décrypteur.

Conclusion & ressources

La génération de codes polymorphiques est une bonne manière de s'initier à la manière dont sont conduits les flux d'instruction. Ils permettent également de se rapprocher du traitement effectué par le processeur en termes d'*opcodes*.

Ressources

Intel 64 and IA-32 Architectures Software Developer's Manuals:

Volume 2A: Instruction Set Reference, A-M
http://www.intel.com/Assets/PDF/manual/253666.pdf

Volume 2B: Instruction Set Reference, N-Z
http://www.intel.com/Assets/PDF/manual/253667.pdf

The Molecular Virology of Lexotan32: Metamorphism Illustrated
https://www.openrce.org/articles/full_view/29

The Viral Darwinism of W32.Evol
https://www.openrce.org/articles/full_view/27

MASM32 (Microsoft assembler)
http://www.masm32.com

OllyDbg
http://www.ollydbg.de

Debugging tools for Windows
http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx