

Notre Attaque

BOUCHER, DA SILVA, SMAGGHE

December 2023

1 Introduction

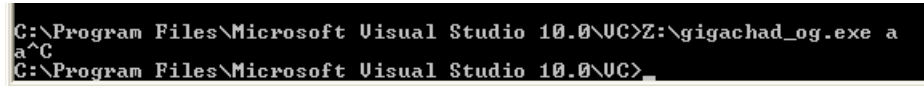
Ce document décrit l'analyse/attaque du malware que nous avons reçu.

2 Vue Globale

Lors de la première réception du malware, on effectue instinctivement plusieurs tests pour vérifier ce qu'effectue le malware en fonction des entrées et des consignes de conception.

2.1 Sanity check 1

Le premier test effectué consiste à vérifier le comportement du malware lorsqu'une entrée valide lui est donnée. Une entrée valide est une clé qui suit les consignes données lors de la conception, c'est donc une clé au format hexadécimal qui a une longueur maximale de 32 caractères.



```
C:\Program Files\Microsoft Visual Studio 10.0\VC>Z:\gigachad_og.exe a  
a^C  
C:\Program Files\Microsoft Visual Studio 10.0\VC>
```

Figure 1: Echo lors d'une entrée correcte

2.2 Sanity check 2

Le deuxième test effectué est, à l'inverse du premier, de fournir une entrée non valide au malware. Ainsi, il s'agit d'envoyer une entrée ne respectant pas les conditions du malware. Cela peut être une chaîne de caractères soit plus longue que 32 caractères, soit ne respectant pas le format hexadécimal. Une autre option est de ne pas respecter le nombre d'arguments à envoyer, c'est-à-dire de ne pas donner de chaînes de caractères ou d'envoyer trop d'arguments.

Ci-dessus, on observe que lors de l'envoi d'une entrée non valide, le système s'éteint, mais la machine virtuelle n'est pas cassée, ce qui est à noter.



Figure 2: Résultat d'une entrée ne respectant pas les conditions

En observant ce que nous donne IDA, on trouve dans le désassembleur un appel à la fonction `ExitWindowsEx` de la bibliothèque "winuser.h".

```

.text:00405212 85 C0          test     eax, eax
.text:00405214 75 B1          jnz     short loc_4051C7
.text:00405216 68 01 00 02 80 push     80020003h          ; dwReason
.text:00405218 6A 05          push     5                  ; uFlags
.text:0040521D FF 15 90 A1 35 01 call     ds:ExitWindowsEx
.text:0040521D
.text:00405223 8B 4D FC          mov     ecx, [ebp-4]
.text:00405226 F7 D8          neg     eax
.text:00405228 1B C0          sub     eax, eax

```

Figure 3: ExitWindows dans IDA

On remarque que la fonction est appelée avec deux arguments, `uFlags` et `dwReason`. L'argument `dwReason` spécifie la raison pour laquelle le système est arrêté ou redémarré. De son côté, `uFlags` indique le type d'arrêt que l'on souhaite effectuer. Ici, l'argument utilisé est 5, ce qui indique un arrêt du système qui, de plus, est forcé.

Nous avons ainsi recherché si une solution simple pouvait nous permettre de passer outre ce problème de mauvaise entrée. Nous avons remarqué que l'endroit où est appelée cette fonction possède un retour `retn (C3)`. Ainsi, nous avons modifié la première ligne d'appel de cette fonction que nous avons remplacée par un `retn (C3)`, ce qui nous permet maintenant d'effectuer des exécutions du programme avec des entrées ne respectant pas les conditions puisque la fonction ne va jamais rien faire.

```

.text:00405195 ; -----
.text:00405196 CC CC CC CC CC CC CC CC align 10h
.text:004051A0
.text:004051A0 chiant: ; CODE XREF: _main:loc_405E181p
.text:004051A0 ; _main:loc_4061101p
.text:004051A0 ; _main+E2B1p
.text:004051A0 ; _main+E4F1p
.text:004051A0 ; _main:loc_4061781p
.text:004051A0 ; _main+EAA1p
.text:004051A0 ; _main:loc_1353A161p
.text:004051A0 ; DATA XREF: _main+2601o
.text:004051A0 C3 retn |
.text:004051A1 ; -----
.text:004051A1 88 EC mov ebp, esp
.text:004051A3 83 EC 18 sub esp, 18h
.text:004051A6 A1 18 C0 35 01 mov eax, ___security_cookie
.text:004051A8 33 C5 xor eax, ebp
.text:004051AD 89 45 FC mov [ebp-4], eax
.text:004051B0 8D 45 E8 lea eax, [ebp-18h]
.text:004051B3 50 push eax ; TokenHandle
.text:004051B4 6A 28 push 28h ; '(' ; DesiredAccess
.text:004051B6 FF 15 38 A0 35 01 call ds:GetCurrentProcess
.text:004051B6
.text:004051BC 50 push eax ; ProcessHandle
.text:004051BD FF 15 04 A0 35 01 call ds:OpenProcessToken
.text:004051BD

```

Figure 4: Patch pour bypass l'exit Windows

3 Énigmes à résoudre

Nous avons enquêté sur la présence de méthodes anti-debug, de l'auto-modification, du chiffrement et de l'obfuscation syntaxique et évidemment de la présence d'une éventuelle clé. Nous n'avons pas trouvé de trace de chiffrement ni de trace d'obfuscation syntaxique.

3.1 Méthodes Anti-Debug

Dans cette section, nous présentons les méthodes anti-debug utilisées et trouvées, avec l'adresse correspondante à l'appel de ces méthodes.

Adresse(s)	Nom	Que fait la méthode
00405DCE	IsDebuggerPresent	Vérifie si le processus appelant est débogué par un débogueur en mode utilisateur. Si le processus actuel s'exécute dans le contexte d'un débogueur, la valeur de retour est différente de zéro. Si le processus actuel n'est pas en cours d'exécution dans le contexte d'un débogueur, la valeur de retour est zéro.
0040536C et 00405DC2	clock	Calcule le temps horloge utilisé par le processus appelant. Si le temps est supérieur au temps d'exécution normal, alors le malware suspecte une exécution en mode débogage
004060C4	CheckRemoteDebuggerPresent	Vérifie si le processus spécifié est en cours de débogage, en récupérant la valeur pointée par le pointeur pbDebuggerPresent. Cette valeur est à TRUE si le processus spécifié est en cours de débogage, et FALSE dans le cas contraire.

3.2 Auto-Modification

Nous avons détecté de l'auto-modification à l'adresse 004053B5 avec un appel à la fonction VirtualProtect qui modifie la protection sur une région de pages validées dans l'espace d'adressage virtuel du processus d'appel. VirtualProtect a été appelé avec un *size* = 6.

3.3 Obfuscation Syntaxique / Clé ?

Environ 13 000 clés générées de cette manière.

```
.text:00427A73 8B 55 0C          mov     edx, [ebp+argv]
.text:00427A76 8B 42 04          mov     eax, [edx+4]
.text:00427A79 0F BE 48 1B      movsx   ecx, byte ptr [eax+18h]
.text:00427A7D 83 F9 32          cmp     ecx, 32h ; '2'
.text:00427A80 0F 85 5B 02 00 00 jnz     loc_427CE1
```

Figure 5: Pattern des clés sur IDA

Nous avons réalisé un script Python qui récupère les caractères qui suivent ce pattern. Grâce à cela, on peut ensuite toutes les essayer une par une. Grâce à l'assembleur, on peut déduire un code C correspondant.

```
void main() {
    if (input[0] == '9' && input[1] == 'F' && input[2] == '6' ) {
        printf("%c", '9');
        printf("%c", 'F');
        printf("%c", '6');
    }
}
```

Figure 6: Modèle de génération de clés

Sauf sur une clé, la bonne :

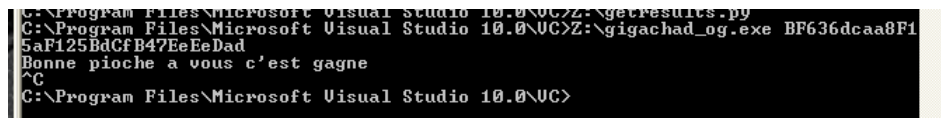
```
void main() {
    if (input[0] == 'A' && input[1] == 'B' && input[2] == 'C' ) {
        printf("%c", 'B');
        printf("%c", 'o');
        printf("%c", 'n');
        printf("%c", 'n');
        printf("%c", 'e');
        printf("%c", ' ');
        printf("%c", 'p');
        ...
    }
}
```

Figure 7: Modèle de génération de la bonne clé

La clé est celle-ci : {BF636dcaa8F15aF125BdCfB47EeEeDad}.

4 Type du Malware

Nous avons trouvé la bonne clé parmi les 13 000 clés, nous affirmons donc que le type du malware est A avec la clé {BF636dcaa8F15aF125BdCfB47EeEeDad}.



```
C:\Program Files\Microsoft Visual Studio 10.0\VC>Z:\getResults.py  
C:\Program Files\Microsoft Visual Studio 10.0\VC>Z:\gigachad_og.exe BF636dcaa8F1  
5aF125BdCfB47EeEeDad  
Bonne pioche a vous c'est gagne  
^C  
C:\Program Files\Microsoft Visual Studio 10.0\VC>
```

Figure 8: Réponse lors de l'utilisation de la bonne clé