

VIROLOGIE INFORMATIQUE

INSA Centre Val De Loire - DINH THANH HAI &
NGUYEN ANH NGUYEN

ANALYSER LE PROGRAMME DE TYPE "ECHO" DE GROUPE: ANDRO DADIC & FLAVIEN EHRET

0. Tables adresses hex des fonctions dans le code

Fonction	Adresse
IsDebuggerPresent	7c82f6ef
CheckRemoteDebuggerPresent	7c85b690
sctrcpy	78aa2700
SetConsoleCursorPosition	7c875ab2
fopen	78b03f54
printf	78b056b4
VirtualProtect	7c801ad4
GetCurrentProcess	7c80de95
strcat	78aa2710

1. Echo ou Pas

Après qu'on a étudié dans la structure par l'IDA, on constate que ce programme est totalement un type de "echo".

L'adresse de la fonction qui imprime est **0x401B01**.

```

.text:00401AF1 ; _main+331fj
.text:00401AF1 sbb     eax, eax
.text:00401AF3 sbb     eax, 0FFFFFFFh
.text:00401AF6 loc_401AF6:
.text:00401AF6 test    eax, eax ; CODE XREF: _main+33F1j
.text:00401AF8 jz      short loc_401B06
.text:00401AFA lea     eax, [ebp+var_110]
.text:00401B00 push    eax
.text:00401B01 call    esi
.text:00401B03 add     esp, 4
.text:00401B06 loc_401B06:
.text:00401B06 ; CODE XREF: _main+125fj
.text:00401B06 ; _main+17F1j ...
.text:00401B06 jmp     short loc_401B06
.text:00401B06 _main endp
.text:00401B08 ; [0000000F BYTES: COLLAPSED FUNCTION _security_check_cookie(x). PRESS KEYPAD CTRL-'" TO EXPAND]
.text:00401B17 ; [0000004B BYTES: COLLAPSED FUNCTION _pre_cpp_init. PRESS KEYPAD CTRL-'" TO EXPAND]
.text:00401B62 ; [00000189 BYTES: COLLAPSED FUNCTION tmainCRTStartup. PRESS KEYPAD CTRL-'" TO EXPAND]

```

Figure 1. - call esi : on appelle la fonction "printf"

2. Chiffrement

Dans la structure, la fonction qui chiffre la texte est **sub_4011D0**. Cette fonction reçoit 2 paramètres: 1ère est le ciphertext, 2ème est la clé.

```
String sub_4011D0(char ciphertext, char cle){}
```

```

.text:00401A0C mov     [eax], 01
.text:00401A0E inc     eax
.text:00401A0F inc     ecx
.text:00401A10 cmp     ecx, 8
.text:00401A13 jb      short loc_401A07
.text:00401A15 lea     ecx, [ebp+var_110]
.text:00401A1B push    offset a332?720uq ; ">< 3' 32<?:7= ,2ou'+&"
.text:00401A20 push    ecx
.text:00401A21 lea     ecx, [ebp+var_210]
.text:00401A27 call    sub_4011D0
.text:00401A2C lea     edx, [ebp+var_110]
.text:00401A32 push    offset a332?720uq ; ">< 3' 32<?:7= ,2ou'+&"
.text:00401A37 push    edx
.text:00401A38 lea     ecx, [ebp+var_210]
.text:00401A3E call    sub_4011D0
.text:00401A43 add     esp, 10h
.text:00401A46 mov     ecx, 0
.text:00401A4B xor     ecx, ecx
.text:00401A4D xor     eax, eax
.text:00401A4F lea     eax, a332?720uq ; ">< 3' 32<?:7= ,2ou'+&"

```

La clé de chiffrement initialement (clé0) est **kiufrufgijnbhuyg**.

En plus, il utilise le **XOR 55h** pour 8 caractères pour déchiffrer et chiffrer un text (le but est de cacher la clé de chiffrement initialement).

```

.text:00401A4F      lea     eax, a3327720uq ; ">< 3' 32<?;7= ,2ou' +ç"
.text:00401A55
.text:00401A55 loc_401A55: ; CODE XREF: _main+2B1↓j
.text:00401A55      mov     bl, [eax]
.text:00401A57      xor     bl, 55h
.text:00401A5A      mov     [eax], bl
.text:00401A5C      inc     eax
.text:00401A5D      inc     ecx
.text:00401A5E      cmp     ecx, 8
.text:00401A61      jnb     short loc_401A55
.text:00401A63      mov     ecx, 0
.text:00401A68      xor     eax, eax
.text:00401A6A      lea     eax, a3327720uq+8
.text:00401A70 loc_401A70: ; CODE XREF: _main+2CC↓j
.text:00401A70      mov     bl, [eax]

```

XOR b1 && 55h

L'étape de chiffrement est comme ça:

1. Au début, il a utilisé la clé 0 pour chiffrer un texte qui est "ument" (var_210) par la fonction **sub_4011D0**. On reçoit un result **text_1**. Ce texte est très important.

```

.text:00401A07 loc_401A07: ; CODE XREF: _main+203↓j
.text:00401A07      mov     bl, [eax]
.text:00401A09      xor     bl, 55h
.text:00401A0C      mov     [eax], bl
.text:00401A0E      inc     eax
.text:00401A0F      inc     ecx
.text:00401A10      cmp     ecx, 8
.text:00401A13      jnb     short loc_401A07
.text:00401A15      lea     ecx, [ebp+var_110]
.text:00401A18      push    offset a3327720uq ; ">< 3' 32<?;7= ,2ou' +ç"
.text:00401A20      push    ecx
.text:00401A21      lea     ecx, [ebp+var_210]
.text:00401A27      call    sub_4011D0
.text:00401A2C      lea     edx, [ebp+var_110]
.text:00401A32      push    offset a3327720uq ; ">< 3' 32<?;7= ,2ou' +ç"
.text:00401A37      push    edx
.text:00401A38      lea     ecx, [ebp+var_210]
.text:00401A3E      call    sub_4011D0
.text:00401A43      add     esp, 10h
.text:00401A46      mov     ecx, 0
.text:00401A4B      xor     ecx, ecx
.text:00401A4D      xor     eax, eax
.text:00401A4F      lea     eax, a3327720uq ; ">< 3' 32<?;7= ,2ou' +ç"

```

2. De 0x401A2C à 0x401AB1, c'est des pièges. Ses résultats n'ont pas utilisé. Donc on m'en fou.

```

.text:00401A2C      lea     edx, [ebp+var_110]
.text:00401A32      push    offset a3327720uq ; ">< 3' 32<?;7= ,2ou' +ç"
.text:00401A37      push    edx
.text:00401A38      lea     ecx, [ebp+var_210]
.text:00401A3E      call    sub_4011D0
.text:00401A43      add     esp, 10h
.text:00401A46      mov     ecx, 0
.text:00401A4B      xor     ecx, ecx
.text:00401A4D      xor     eax, eax
.text:00401A4F      lea     eax, a3327720uq ; ">< 3' 32<?;7= ,2ou' +ç"

```

```

.text:00401A55
.text:00401A55 loc_401A55:                                ; CODE XREF: _main+2B1□j
.text:00401A55      mov     bl, [eax]
.text:00401A57      xor     bl, 55h
.text:00401A5A      mov     [eax], bl
.text:00401A5C      inc     eax
.text:00401A5D      inc     ecx
.text:00401A5E      cmp     ecx, 8
.text:00401A61      jnb     short loc_401A55
.text:00401A63      mov     ecx, 0
.text:00401A68      xor     eax, eax
.text:00401A6A      lea     eax, a332?72ouq+8
.text:00401A70
.text:00401A70 loc_401A70:                                ; CODE XREF: _main+2CC□j
.text:00401A70      mov     bl, [eax]
.text:00401A72      xor     bl, 55h
.text:00401A75      mov     [eax], bl
.text:00401A77      inc     eax
.text:00401A78      inc     ecx
.text:00401A79      cmp     ecx, 8
.text:00401A7C      jnb     short loc_401A70
.text:00401A7E      lea     eax, [ebp+var_110]
.text:00401A84      push    (offset a332?72ouq+8)
.text:00401A89      push    eax
.text:00401A8A      lea     ecx, [ebp+var_210]
.text:00401A90      call    sub_4011D0
.text:00401A95      add     esp, 8
.text:00401A98      mov     ecx, 0
.text:00401A9D      xor     eax, eax
.text:00401A9F      lea     eax, a332?72ouq+8
.text:00401AA5
.text:00401AA5 loc_401AA5:                                ; CODE XREF: _main+301□j
.text:00401AA5      mov     bl, [eax]
.text:00401AA7      xor     bl, 55h
.text:00401AAA      mov     [eax], bl
.text:00401AAC      inc     eax
.text:00401AAD      inc     ecx
.text:00401AAE      cmp     ecx, 8
.text:00401AB1      jnb     short loc_401AA5

```

3. De 0x401A5 à 0x401ACE, il est très important pour décider que ce programme est echo ou non.

D'abord, dans le rectangle rouge, c'est la fonction **strncpy** (il copie 7 caractères de var_210 à var_C, d'ici, var_210 est **text_1 + XOR 55h**). Et après, dans la rectangle noir, il utilise le var_C pour comparer avec un text **aS_2Zgj**. Il est possible que si var_c == aS_2Zgj, il n'imprime pas.

```

.text:00401AA5      mov     bl, [eax]
.text:00401AA7      xor     bl, 55h
.text:00401AAA      mov     [eax], bl
.text:00401AAC      inc     eax
.text:00401AAD      inc     ecx
.text:00401AAE      cmp     ecx, 8
.text:00401AB1      jnb     short loc_401AA5
.text:00401AB3      push    7
.text:00401AB5      lea     ecx, [ebp+var_210]
.text:00401ABB      push    ecx
.text:00401ABC      lea     edx, [ebp+var_C]
.text:00401ABF      push    edx
.text:00401AC0      call    [ebp+var_220]
.text:00401AC6      add     esp, 0Ch
.text:00401AC9      mov     ecx, offset aS_2Zgj ; "æ_2:âj"
.text:00401ACE      lea     eax, [ebp+var_C]
.text:00401AD1      loc_401AD1: ; CODE XREF: _main+33B↓j
.text:00401AD1      mov     dl, [eax]
.text:00401AD1      cmp     dl, [ecx]

```

4. Mais après qu'avoir étudié la structure de 0x401AD1 à 0x401AEF, on voit que dans tous les cas, lorsque ce programme n'est pas echo, il faut sauter à l'étape loc_401AE5. Mais pour sauter, il faut que dl a la value 0, mais avec une ligne: `** mov dl, [eax+1] **`, cela prouve qu'il ne saute jamais à l'étape loc_401AE5.

```

.text:00401AD1      loc_401AD1: ; CODE XREF: _main+33B↓j
.text:00401AD1      mov     dl, [eax]
.text:00401AD3      cmp     dl, [ecx]
.text:00401AD5      jnz     short loc_401AF1
.text:00401AD7      test    dl, dl
.text:00401AD9      jz      short loc_401AED
.text:00401ADB      mov     dl, [eax+1]
.text:00401ADE      cmp     dl, [ecx+1]
.text:00401AE1      jnz     short loc_401AF1
.text:00401AE3      add     eax, 2
.text:00401AE6      add     ecx, 2
.text:00401AE9      test    dl, dl
.text:00401AEB      jnz     short loc_401AD1
.text:00401AED      loc_401AED: ; CODE XREF: _main+329↑j
.text:00401AED      xor     eax, eax
.text:00401AEF      jmp     short loc_401AF6
; Toujours donner le eax = 0
.text:00401AF1      loc_401AF1: ; CODE XREF: _main+325↑j
; _main+331↑j
.text:00401AF1      sbb     eax, eax
.text:00401AF3      sbb     eax, 0FFFFFFFh
; Toujours donner le eax = 1
.text:00401AF6      loc_401AF6: ; CODE XREF: _main+33F↑j
.text:00401AF6      test    eax, eax
.text:00401AF8      jz      short loc_401B06
.text:00401AFA      lea     eax, [ebp+var_110]
.text:00401B00      push    eax
.text:00401B01      call    esi
; printf
.text:00401B03      add     esp, 4

```

3. Anti-Debug

Dans le code, il a utilisé 2 types de Anti-Debug: **IsDebuggerPresent** et **CheckRemoteDebuggerPresent**.

- L'adresse de la fonction **IsDebuggerPresent** quand il est appelé est 0x4018D5.

```

.text:004018CD
.text:004018CD loc_4018CD:                ; CODE XREF: _main+103↑j
.text:004018CD call    [ebp+var_214]
.text:004018D3 test    eax, eax
.text:004018D5 jnz     loc_401B06
.text:004018D8 mov     esi, [ebp+var_218]
.text:004018E1 sub     ebx, (offset loc_4050CF+1)
.text:004018E7 mov     edx, ebx
.text:004018E9 lea     esp, [esp+0]
.text:004018F0
.text:004018F0 loc_4018F0:                ; CODE XREF: _main+166↓j

```

Quand l'IDA lance là-bas, l'adresse hex de pointeur [ebp+var_214] est **0x7c82f6ef**. C'est exactement la fonction Anti-Debug **IsDebuggerPresent**. Et après ce ligne, on voit que:

```
jnz loc_401B06
```

loc_401B06 est utilisé contre Debug car c'est une boucle infinie.

```

.text:00401B00
.text:00401B06 loc_401B06:                ; CODE XREF: _main+125↑j
.text:00401B06                                     ; _main+17F↑j ...
.text:00401B06 jmp     short loc_401B06
.text:00401B06 _main      endp

```

Sur la structure de cette fonction, je prédis que IsDebuggerPresent est écrit comme ça:

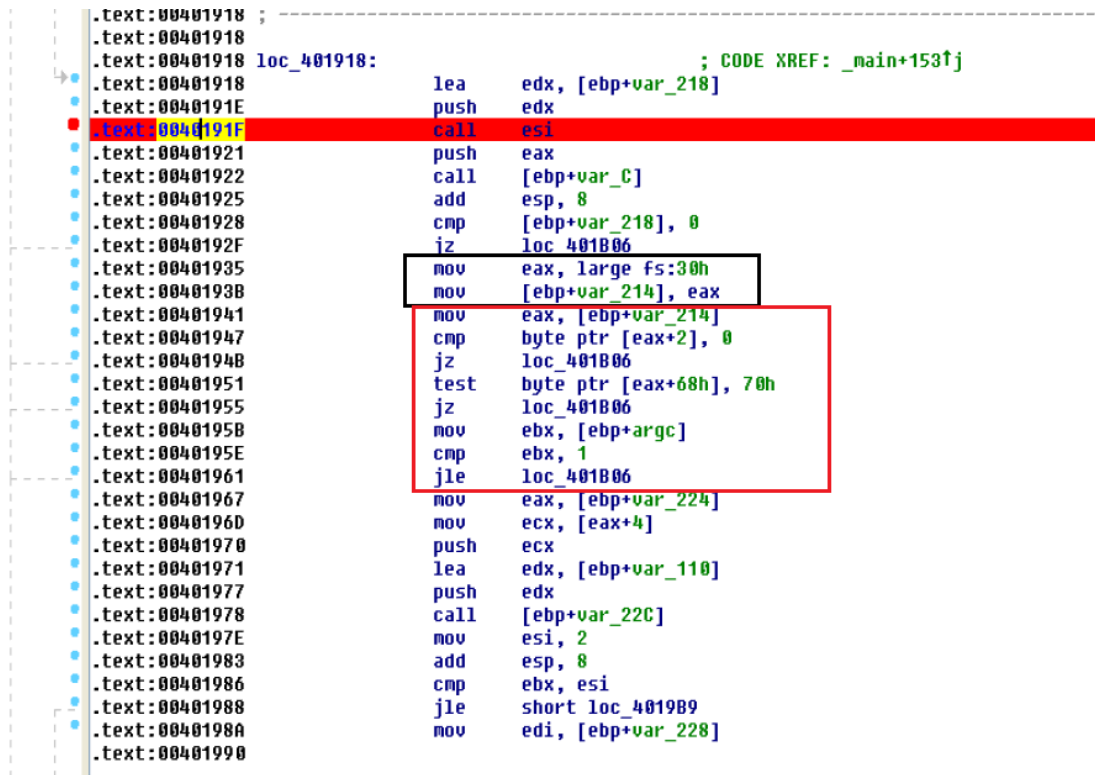
```

if(IsDebuggerPresent()){
    while(1);
} else{
    \\
}

```

Pour passer ce anti-debug, on change seulement de *jnz* à *jz* et dans le code, on cherche le *loc_401B06* et change le saut inverse (mais pas du tout, par exemple, à 0x40161, il ne faut pas changer parce que à d'ici, il vérifie que si l'argv est inférieur que 0, il ne fonctionne pas).

- L'adresse de la fonction **CheckRemoteDebuggerPresent** quand il est appelé est 0x40191F.



D'ici, quand l'IDA lance, l'adresss à registre *esi* est **7c85b690**. C'est exactement la fonction Anti-Debug **CheckRemoteDebuggerPresent**. Des lignes suivantes, on voit que:

- Rectangle noir: Quelques points à noter: **fs:30**, **mov eax**. Exactement dans le code, il a une ligne :

```

ppeb = IsDebuggerPresent; // ppeb est la fonction IsDebuggerPresent
__asm{
    mov eax, fs:[0x30]; //ProcessEnvironmentBlock(documentation de
Microsoft)
    mov ppeb, eax
}

```

Le but de rectangle noir est d'appelé la fonctionn Anti-Debug IsDebuggerPresent.

- Rectangle rouge: Quelques points à noter: **[eax+2]**, **[eax+68]**, **70h**. Exactement il a utilisé ppeb[2] et ppeb[104] & 0x70.

En général, la technique Anti-Debug est comme le TD, pas nouveau.

4. Auto-modification

Dans la structure, il a utilisé la fonction **VirutalProtect** avec la taille **dwSize = 10** (*push 0Ah*) et **flNewProtect = page_execute_readwrite** (*push 40h*) . L'adresse de cette fonction est **0x4017DF** comme l'image ci-dessus.

```

.text:004017C9      lea     ecx, [ebp+f10ldProtect]
.text:004017CF      push    ecx                ; lpf10ldProtect
.text:004017D0      push    40h                ; flNewProtect
.text:004017D2      push    0Ah                ; dwSize
.text:004017D4      push    offset _main       ; lpAddress
.text:004017D9      mov     [ebp+var_224], eax
.text:004017DF      call    ds:VirtualProtect
.text:004017E5      mov     esi, ds:fopen

```

En général, on prédit que dans la code, il y a une ligne :

```

char * pointeur = (char*) _tmain;
VirtualProtect(pointeur, 10, PAGE_EXECUTE_READWRITE, lpf10ldProtect);

```

Le code n'a pas été caché. Il est appelé directement au début de code.

5. Obfuscation

Des fonctions caché sont: **strcpy**, **strcat**, **IsDebuggerPresent**, **CheckRemoteDebuggerPresent**, **strncpy**, **printf**

1. Il utilise la fonction **fopen** pour cacher 3 fonctions strcpy, strcat et strncpy.

```

strcat = fopen - 61844
strcpy = fopen - 61854

```

```

.text:004017C8      push    edi
.text:004017C9      lea     ecx, [ebp+f10ldProtect]
.text:004017CF      push    ecx                ; lpf10ldProtect
.text:004017D0      push    40h                ; flNewProtect
.text:004017D2      push    0Ah                ; dwSize
.text:004017D4      push    offset _main       ; lpAddress
.text:004017D9      mov     [ebp+var_224], eax
.text:004017DF      call    ds:VirtualProtect
.text:004017E5      mov     esi, ds:fopen
.text:004017EB      mov     ebx, ds:SetConsoleCursorPosition
.text:004017F1      lea     eax, [esi-61854h]
.text:004017F7      mov     [ebp+var_22C], eax
.text:004017FD      lea     eax, [esi-61844h]
.text:00401803      mov     ecx, ebx
.text:00401805      mov     [ebp+var_214], ebx
.text:0040180B      mov     [ebp+var_C], ebx
.text:0040180E      mov     [ebp+var_218], ebx
.text:00401814      mov     [ebp+var_228], eax
.text:0040181A      mov     [ebp+var_220], esi
.text:00401820      sub     ecx, offset unk_4050B0
.text:00401826      jmp     short loc_401830
.text:00401826 ; -----
.text:00401828      align 10h

```



```

.text:004017FD      mov     [esi-61844h]
.text:00401803      mov     ecx, ebx
.text:00401805      mov     [ebp+var_214], ebx
.text:0040180B      mov     [ebp+var_C], ebx
.text:0040180E      mov     [ebp+var_218], ebx
.text:00401814      mov     [ebp+var_220], eax
.text:0040181A      mov     [ebp+var_220], esi
.text:00401820      sub     ecx, offset unk_4050B0
.text:00401826      jmp     short loc_401830
.text:00401826      ; -----
.text:00401828      align 10h      |
.text:00401830

```

2. Il utilise la fonction **SetConsoleCursorPosition** pour cacher 2 fonctions Anti-debug `IsDebuggerPresent` et `CheckRemoteDebuggerPresent`. Autre que `fopen`, il utilise une autre fonction pour changer l'adresse de **SetConsoleCursorPosition** à **IsDebuggerPresent** et **CheckRemoteDebuggerPresent**

```

.text:004018A0      mov     edx, esi
.text:004018A2      ; CODE XREF: _main+118↓j
.text:004018A2      loc_4018A2:      mov     ecx, 10h
.text:004018A2      mov     ecx, 10h
.text:004018A7      mov     eax, offset loc_4050E0
.text:004018AC      lea     esp, [esp+0]
.text:004018B0      ; CODE XREF: _main+112↓j
.text:004018B0      loc_4018B0:      cmp     ecx, 4
.text:004018B0      cmp     ecx, 4
.text:004018B3      jb     short loc_4018CD
.text:004018B5      mov     esi, [edx+eax]
.text:004018B8      cmp     esi, [eax]
.text:004018BA      jnz     short loc_4018C4
.text:004018BC      sub     ecx, 4
.text:004018BF      add     eax, 4
.text:004018C2      jmp     short loc_4018B0
.text:004018C4      ; -----
.text:004018C4      loc_4018C4:      ; CODE XREF: _main+10A↑j
.text:004018C4      dec     [ebp+var_220]
.text:004018C4      dec     [ebp+var_220]
.text:004018C9      dec     edx
.text:004018CB      jmp     short loc_4018A2
.text:004018CD      ; -----
.text:004018CD      loc_4018CD:      call    [ebp+var_214]
.text:004018CD      call    [ebp+var_214]
.text:004018D3      test    eax, eax
.text:004018D3      test    eax, eax
.text:004018D5      jz      loc_4018A0

```

Changer l'adresse par l'opérateur soustraction

Les valeurs de soustractions sont **107554 bits** (pour le **CheckRemoteDebuggerPresent**) et **287683 bits** (pour le **IsDebuggerPresent**)

3. Il utilise la fonction **scanf** pour cacher la fonction `printf`.

```
.text:004019B9
.text:004019B9 loc_4019B9:                                ; CODE XREF: _main+1D8↑j
.text:004019B9      mov     eax, ds:scanf
.text:004019BE      mov     esi, eax
.text:004019C0      sub     eax, offset unk_405024
.text:004019C5      mov     edx, eax
.text:004019C7      jmp     short loc_4019D0
.text:004019C7      -----
.text:004019C9      align 10h
.text:004019D0
.text:004019D0 loc_4019D0:                                ; CODE XREF: _main+217↑j
.text:004019D0      ; _main+246↓j
.text:004019D0      mov     ecx, 0Ch
.text:004019D5      mov     eax, offset unk_405024
.text:004019DA      lea     ebx, [ebx+0]
.text:004019E0
.text:004019E0 loc_4019E0:                                ; CODE XREF: _main+242↓j
.text:004019E0      cmp     ecx, 4
.text:004019E3      jb     short loc_4019F8
```