Malware

du groupe: Clément Bailly-Grandvaux et Lucas Gouin

Michael Enescu - Nicolas Beaudouin

1. Main

```
.text:004014F1 8B EC                   mov     ebp, esp
.text:004014F3 83 EC 08                sub     esp, 8
.text:004014F6 83 7D 08 02             cmp     [ebp+argc], 2
.text:004014FA 75 63                   jnz     short loc_40155F
.text:004014FC 68 D8 20 40 00          push    offset Format   ; "Le nombre :\n\n"
.text:00401501 FF 15 A0 20 40 00       call    ds:printf
.text:00401507 83 C4 04                add     esp, 4
.text:0040150A 8B 45 0C                mov     eax, [ebp+argv]
.text:0040150D 8B 48 04                mov     ecx, [eax+4]
.text:00401510 51                      push    ecx
.text:00401511 E8 7A FB FF FF          call    sub_401090
.text:00401516 83 C4 04                add     esp, 4
.text:00401519 89 45 F8                mov     [ebp-8], eax
.text:0040151C 89 55 FC                mov     [ebp-4], edx
.text:0040151F 81 7D F8 C0 7D BB+      cmp     dword ptr [ebp-8], 40BB7DC0h
.text:00401526 75 20                   jnz     short loc_401548
.text:00401528 81 7D FC 68 4B 27+      cmp     dword ptr [ebp-4], 6F274B68h
.text:0040152F 75 17                   jnz     short loc_401548
.text:00401531 8B 55 0C                mov     edx, [ebp+argv]
.text:00401534 8B 42 04                mov     eax, [edx+4]
.text:00401537 50                      push    eax
.text:00401538 68 E8 20 40 00          push    offset aSBravoCEstLeBo ; "%s\n\nBravo, c'est le bon nombre.\n"
.text:0040153D FF 15 A0 20 40 00       call    ds:printf
.text:00401543 83 C4 08                add     esp, 8
.text:00401546 EB 15                   jmp     short loc_40155D
.text:00401548                ; ---------------------------------------------------------------------------
.text:00401548
.text:00401548                loc_401548:                             ; CODE XREF: _main+36↑j
.text:00401548                                                        ; _main+3F↑j
.text:00401548 8B 4D 0C                mov     ecx, [ebp+argv]
.text:0040154B 8B 51 04                mov     edx, [ecx+4]
.text:0040154E 52                      push    edx
.text:0040154F 68 0C 21 40 00          push    offset aSNon_CeNEstPas ; "%s\n\nNon. Ce n'est pas possible avec c"...
.text:00401554 FF 15 A0 20 40 00       call    ds:printf
.text:0040155A 83 C4 08                add     esp, 8
.text:0040155D
.text:0040155D                loc_40155D:                             ; CODE XREF: _main+56↑j
.text:0040155D EB 0E                   jmp     short loc_40156D
.text:0040155F                ; ---------------------------------------------------------------------------
```

1. Main

```
undefined4 __cdecl Main(uint32_t arg_8h, int32_t arg_ch)
{
    int64_t iVar1;
    uint32_t var_8h;
    uint32_t var_4h;

    if (arg_8h == 2) {
        (*MSVCR100.dll_printf)("Le nombre :\n\n");
        iVar1 = Fonction_principale(*(int32_t *)(arg_ch + 4));
        if (iVar1 == 0x6f274b6840bb7dc0) {
            (*MSVCR100.dll_printf)("%s\n\nBravo, c\'est le bon nombre.\n", *(undefined4 *)(arg_ch + 4));
        } else {
            (*MSVCR100.dll_printf)("%s\n\nNon. Ce n\'est pas possible avec ce nombre.\n", *(undefined4 *)(arg_ch + 4));
        }
    } else {
        (*MSVCR100.dll_printf)("Nombre d\'argument non valable, il en faut exactement 1.\n");
    }
    (*MSVCR100.dll_getchar)();
    return 0;
}
```

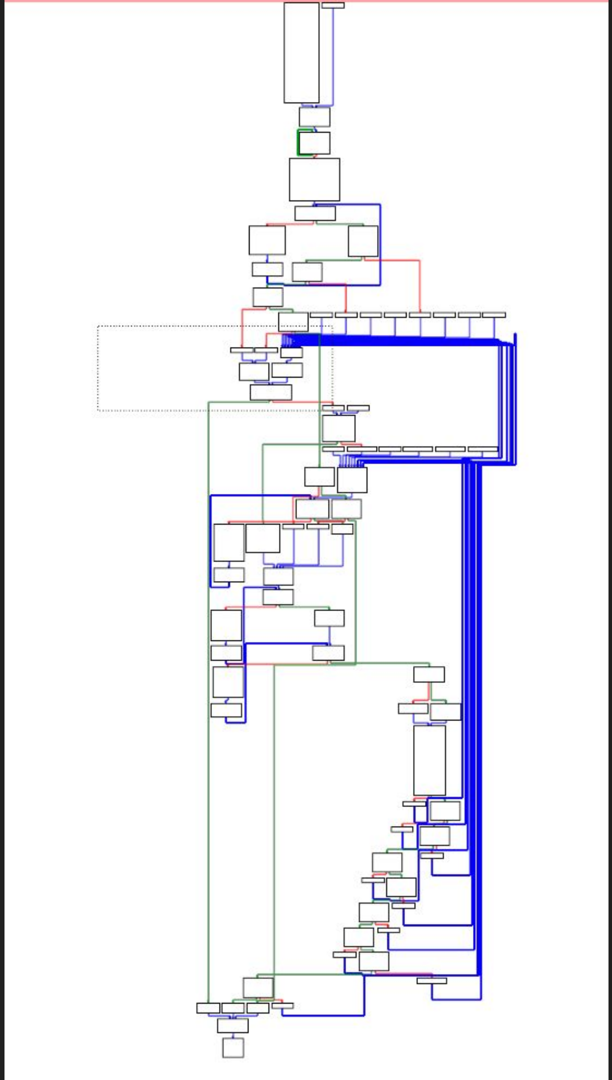
Informations non codées

2. Fonction principale

- Beaucoup de jumps vers diverses endroits

- Possibilités de diviser ces jumps en 13 labels que l’on nommera label_1, label_2….
  (On a récupérer les noms des labels donnés par l’outil cutter)

- On peut supposer que cela provient d’un certains nombre de conditions



2. Fonction principale

- On peut partir de la fin: du return de la fonction afin de retrouver l’ordre des passages possible

- On a donc trouvé deux ordre particuliers qui sont intéressants, dont 1 qui semble être correct.

  5 -> 13 -> 4 -> 7 -> 9 -> 1 -> 10 -> 11 -> 0 -> 6

  ou

  5 -> 13 -> 1 -> 10 -> 11 -> 0 -> 6

- Le reste des ordres des labels sont certainement utilisé pour l’anti-debug

```
    int32_t var_34h;
    int32_t var_30h;
    int32_t var_2ch;
    int32_t var_28h;
    int32_t var_24h;
    int32_t var_20h;
    int32_t var_1ch;
    int32_t var_14h;
    int32_t var_10h;
    int32_t var_ch;
    void * var_8h;
    intptr_t size;
    antiDebug_1 ();
    AntiDebug_2 ();
    *(0x403380) = fp_stack[0];
    fp_stack--;
    var_10h = 1;
    var_ch = 0;
    goto label_5;
label_0:
    eax = var_10h;
    edx = var_ch;
    goto label_6;
label_4:
    var_10h = 0;
    var_ch = 0;
    var_14h = 0;
    goto label_7;
label_2:
    eax = var_14h;
    eax++;
    var_14h = eax;
label_7:
    ecx = eax;
    if (ecx >= size) {
        goto label_8;
    }
    goto label_9;
label_1:
    var_20h = 1;
    var_1ch = 0;
    var_44h = 0;
    while (1) {
        edx = var_44h;
```

2. Fonction principale

- Avec Ghidra on peut obtenir une vision différente de la fonction mais il reste des goto

```c
    antiDebug_1();
    Var4 = (unkfloat10)AntiDebug_2();
    *(double *)0x403380 = (double)Var4;
    iVar2 = 1;
    var_5ch = arg_8h;
    do {
        cVar1 = *(char *)var_5ch;
        var_5ch = var_5ch + 1;
    } while (cVar1 != '\0');
    var_5ch = var_5ch - (arg_8h + 1);
    puVar3 = (undefined *)(*MSVCR100.dll_calloc)(1, var_5ch);
    *puVar3 = *(undefined *)arg_8h;
    for (var_50h = 1; var_50h < var_5ch; var_50h = var_50h + 1) {
        puVar3[var_50h] = puVar3[var_50h + -1] ^ *(uint8_t *)(arg_8h + var_50h);
    }
    Var4 = (unkfloat10)AntiDebug_2();
    *(double *)0x403378 = (double)(Var4 - (unkfloat10)*(double *)0x403380);
    Var4 = (unkfloat10)AntiDebug_2();
    *(double *)0x403380 = (double)Var4;
    var_68h = var_5ch;
    if ((uint16_t)((uint16_t)(0.0001 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.0001) << 0xe) ==
        0) goto code_r0x004010e0;
    iVar2 = 1;
    if ((uint16_t)((uint16_t)(0.001 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.001) << 0xe) == 0)
    goto code_r0x004010e0;
    if (((uint16_t)((uint16_t)(0.01 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.01) << 0xe) == 0)
       || ((uint16_t)((uint16_t)(0.05 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.05) << 0xe) == 0
          )) {
        iVar2 = 0;
        var_14h = 0;
        goto code_r0x004010e9;
    }
    iVar2 = 1;
    if (((uint16_t)((uint16_t)(0.1 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.1) << 0xe) != 0) &&
       (iVar2 = 1,
       (uint16_t)((uint16_t)(1.0 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 1.0) << 0xe) != 0)) {
        return 1;
    }
code_r0x004010ff:
    do {
        uVar5 = 1;
        var_44h = 0;
```

3. Fonctions d’anti-debug

- Deux fonctions:
  - 0x00401000
  - 0x00401050
- qui utilisent les fonctions:
  - QueryPerformanceCounter
  - QueryPerformanceFrequency

```
.text:00401003 sub     esp, 8
.text:00401006 lea     eax, [ebp+Frequency]
.text:00401009 push    eax
.text:0040100A call    ds:QueryPerformanceFrequency
.text:00401010 test    eax, eax
.text:00401012 jnz     short loc_40101C
.text:00401014 push    0FFFFFFFFh
.text:00401016 call    ds:exit
.text:0040101C ; ---------------------------------------
.text:0040101C
.text:0040101C loc_40101C:
.text:0040101C fild    qword ptr [ebp+Frequency]
.text:0040101F fdiv    ds:dbl_402178
.text:00401025 fstp    dbl_403390
.text:0040102B lea     ecx, [ebp+Frequency]
.text:0040102E push    ecx
.text:0040102F call    ds:QueryPerformanceCounter
```

- Appelés à plusieurs endroits, dont:
  - Au début de la fonction principale
  - à d’autres endroits à l’intérieur de la fonction, comme dans le label 11

3. Fonctions d’anti-debug

- Ce qu’on pense qui se passe :

QueryPerformanceCounter: Mesure le temps d'exécution d’une opération. Si l’on est dans un debugger, ce temps sera plus élevé que dans une exécution normale.

La première fonction stocke la valeur dans une case mémoire et la deuxième fonction utilise la valeur de la première.

3. Fonctions d’anti-debug

- Fonction 1:

```
void queryFreqCount1(void)
{
    int32_t iVar1;
    LARGE_INTEGER *lpPerformanceCount;
    int32_t var_4h;

    // [00] -r-x section size 4096 named .text
    iVar1 = (*KERNEL32.dll_QueryPerformanceFrequency)(&lpPerformanceCount);
    if (iVar1 == 0) {
        (*MSVCR100.dll_exit)(0xffffffff);
    }
    *(double *)0x403390 = (double)CONCAT44(var_4h, lpPerformanceCount) / 1000.0;
    (*KERNEL32.dll_QueryPerformanceCounter)(&lpPerformanceCount);
    *(LARGE_INTEGER **)0x403398 = lpPerformanceCount;
    *(int32_t *)0x40339c = var_4h;
    return;
}
```

3. Fonctions d’anti-debug

- Fonction 2:

```
unkfloat10 queryCount2(void)
{
    int32_t var_10h;
    int32_t var_ch;
    LARGE_INTEGER *lpPerformanceCount;
    int32_t var_4h;

    (*KERNEL32.dll_QueryPerformanceCounter)(&lpPerformanceCount);
    return (unkfloat10)
           CONCAT44((var_4h - *(int32_t *)0x40339c) - (uint32_t)(lpPerformanceCount < *(LARGE_INTEGER **)0x403398),
                    (int32_t)lpPerformanceCount - (int32_t)*(LARGE_INTEGER **)0x403398) /
           (unkfloat10)*(double *)0x403390;
}
```

3. Fonctions d’anti-debug

Idée de l'anti debug:

```
bool IsDebugged(DWORD64 qwNativeElapsed)
{
    LARGE_INTEGER liStart, liEnd;
    QueryPerformanceCounter(&liStart);
    // ... some work
    QueryPerformanceCounter(&liEnd);
    return (liEnd.QuadPart - liStart.QuadPart) > qwNativeElapsed;
}
```

```
Var4 = (unkfloat10)queryCount2();
*(double *)0x403380 = (double)Var4;
```

du code …

```
Var4 = (unkfloat10)queryCount2();
*(double *)0x403378 = (double)(Var4 - (unkfloat10)*(double *)0x403380);
```

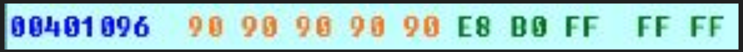
```
if (((uint16_t)((uint16_t)(0.01 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.01) << 0xe) == 0)
   || ((uint16_t)((uint16_t)(0.05 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.05) << 0xe) == 0
      )) {
```

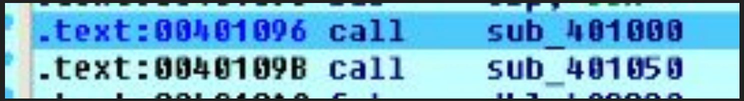
3. Fonctions d’anti-debug

- Outrepasser cet anti-debug:

Dans l'hexadécimal, dans IDA, on remplace les appels de fonctions par des NOP (x90) et on peut donc éviter qu’il y ait l’appel des fonctions.

On peut aussi écraser les tests et forcer le jump correspondant au bon résultat.





4. Précision sur le fonctionnement

- La clée est stockée en 2 parties:

```
mov     [ebp+val_1], 1
mov     [ebp+val_0], 0
```

- Calcul de la taille de la clé:

```
.text:004013BE mov     edx, [ebp+copyAddrKey]
.text:004013C1 mov     al, [edx]
.text:004013C3 mov     [ebp+proggrAddrKey], al
.text:004013C6 add     [ebp+copyAddrKey], 1
.text:004013CA cmp     [ebp+proggrAddrKey], 0
.text:004013CE jnz     short loc_4013BE
.text:004013D0 mov     ecx, [ebp+copyAddrKey]
```

- Calloc pour stocker valeure de la clé après le premier chiffrement:

```
.text:004013E5 call    ds:calloc
```

4. Précision sur le fonctionnement

```
.text:00401404 mov     ecx, [ebp+var_50]
.text:00401407 add     ecx, 1
.text:0040140A mov     [ebp+var_50], ecx
.text:0040140D
.text:0040140D loc_40140D:
.text:0040140D mov     edx, [ebp+var_50]
.text:00401410 cmp     edx, [ebp+SizeOfElements]
.text:00401413 jge     short loc_401434
.text:00401415 mov     eax, [ebp+var_8]
.text:00401418 add     eax, [ebp+var_50]
.text:0040141B movsx   ecx, byte ptr [eax-1]
.text:0040141F mov     edx, [ebp+addrKey]
.text:00401422 add     edx, [ebp+var_50]
.text:00401425 movsx   eax, byte ptr [edx]
.text:00401428 xor     ecx, eax
.text:0040142A mov     edx, [ebp+var_8]
.text:0040142D add     edx, [ebp+var_50]
.text:00401430 mov     [edx], cl
.text:00401432 jmp     short loc_401404
.text:00401434 ; ---------------------------------------
```

```
debug017:0000000000342A89 db  30h ; 0
debug017:0000000000342A8A db  78h ; x
debug017:0000000000342A8B db  31h ; 1
debug017:0000000000342A8C db  32h ; 2
debug017:0000000000342A8D db  33h ; 3
debug017:0000000000342A8E db  34h ; 4
```

```
debug017:0000000000342AA8 db  30h ; 0
debug017:0000000000342AA9 db  48h ; H
debug017:0000000000342AAA db  79h ; y
debug017:0000000000342AAB db  4Bh ; K
debug017:0000000000342AAC db  78h ; x
debug017:0000000000342AAD db  4Ch ; L
```

```
for (counter = 1; counter < copyAddrKey; counter = counter + 1) {
    puVar3[counter] = puVar3[counter + -1] ^ *(uint8_t *)(addrKey + counter);
}
```

```
for i in range(1,len(ss)):
    ss[i] = hex(int(ss[i-1], 16) ^ int(s[i], 16))[2:]
```

4. Précision sur le fonctionnement

```
Var4 = (unkfloat10)queryCount2();
*(double *)0x403378 = (double)(Var4 - (unkfloat10)*(double *)0x403380);
// re
//
Var4 = (unkfloat10)queryCount2();
*(double *)0x403380 = (double)Var4;
sizeKey = copyAddrKey;
if ((uint16_t)((uint16_t)(0.0001 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.0001) << 0xe) ==
    0) goto code_r0x004010e0;
iVar2 = 1;
if ((uint16_t)((uint16_t)(0.001 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.001) << 0xe) == 0)
goto code_r0x004010e0;
if (((uint16_t)((uint16_t)(0.01 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.01) << 0xe) == 0)
   || ((uint16_t)((uint16_t)(0.05 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.05) << 0xe) == 0
      )) {
    iVar2 = 0;
    index14 = 0;
    goto code_r0x004010e9;
}
iVar2 = 1;
if (((uint16_t)((uint16_t)(0.1 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.1) << 0xe) != 0) &&
   (iVar2 = 1,
   (uint16_t)((uint16_t)(1.0 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 1.0) << 0xe) != 0)) {
    return 1;
}
```

```
text:00401468 loc_401468:
text:00401468 fld     ds:dbl_402190
text:0040146E fcomp   dbl_403378
text:00401474 fnstsw  ax
text:00401476 test    ah, 41h
text:00401479 jnz     short loc_401480
text:0040147B jmp     loc_4013A5
```


```
text:00401468 loc_401468:
text:00401468 fld     ds:dbl_402190
text:0040146E fcomp   dbl_403378
text:00401474 fnstsw  ax
text:00401476 test    ah, 41h
text:00401479 jnz     short loc_401480
text:0040147B jmp     loc_4013A5
```


```
90 90
```

```
text:0040148E test    ah, 41h
text:00401491 nop
text:00401492 nop
text:00401493 jmp     loc_4010C9
```

4. Précision sur le fonctionnement

```c
if ((int32_t)(char)puVar3[index14] % 3 == 0) {
    var_30h = (int32_t)(char)puVar3[index14] ^ 0xff;
    goto code_r0x004010ff;
}
```

```asm
text:004012F2 mov     ecx, [ebp+var_8]
text:004012F5 add     ecx, [ebp+var_14]
text:004012F8 movsx   eax, byte ptr [ecx]
text:004012FB cdq
text:004012FC mov     ecx, 3
text:00401301 idiv    ecx
text:00401303 test    edx, edx
text:00401305 jz      short loc_40130D
text:00401307 jmp     short loc_40132A
```

```c
} else if ((uint16_t)
           ((uint16_t)(5e-05 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 5e-05) << 0xe) != 0
          ) {
    if ((uint16_t)
        ((uint16_t)(0.0005 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.0005) << 0xe) != 0)
    {
        if ((uint16_t)
            ((uint16_t)(0.000503 < *(double *)0x403378) << 8 |
            (uint16_t)(*(double *)0x403378 == 0.000503) << 0xe) == 0) goto code_r0x0040132a;
        if (((uint16_t)
             ((uint16_t)(0.0005 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.0005) << 0xe)
             != 0) &&
           ((uint16_t)
            ((uint16_t)(0.05 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.05) << 0xe) != 0)
           ) {
            if ((uint16_t)
                ((uint16_t)(0.5 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 0.5) << 0xe) ==
                0) goto code_r0x0040132a;
            if ((uint16_t)
                ((uint16_t)(1.0 < *(double *)0x403378) << 8 | (uint16_t)(*(double *)0x403378 == 1.0) << 0xe) !=
                0) {
                return iVar2;
            }
        }
    }
    goto code_r0x004010e0;
}
```

4. Précision sur le fonctionnement

```
code_r0x004010ff:
    do {
        uVar5 = 1;
        counterLab1 = 0;
        while( true ) {
            ancienMult = (int32_t)((uint64_t)uVar5 >> 0x20);
            multVal = (int32_t)uVar5;
            if (index14 <= counterLab1) break;
            uVar5 = flirt.allmul(sizeKey, ancienMult, var_30h, var_30h >> 0x1f, multVal);
            counterLab1 = counterLab1 + 1;
        }
        iVar6 = 1;
        var_48h = 1;
        while( true ) {
            var_34h = (int32_t)((uint64_t)iVar6 >> 0x20);
            var_38h = (int32_t)iVar6;
            if (index14 < var_48h) break;
            iVar6 = flirt.allmul(sizeKey, var_34h, var_48h, var_48h >> 0x1f, var_38h);
            var_48h = var_48h + 1;
        }
        if (iVar6 == 0) {
            iVar6 = 1;
        }
        sizeKey = multVal;
        iVar6 = flirt.aulldiv(uVar5, iVar6);
        iVar2 = iVar6 + iVar2;
```

```python
copyKey = "1000000";
s= list(copyKey);
ss = list(copyKey)

for i in range(1,len(ss)):
    ss[i] = hex(int(ss[i-1], 16) ^ int(s[i], 16))[2:]

res= 0;
for i in range(0,len(ss)):
    if int(ss[i], 16)%3 == 0:
        x = hex(int(ss[i],16) ^ int("ff", 16));
        miniRes1=1;
        miniRes2=1;
        compteur1=0;
        for j in range(0,i):
            miniRes1 = miniRes1*int(x, 16);
        for j in range(0,i-1):
            miniRes2 = miniRes2*int(x, 16);

        res += miniRes1/miniRes2;

if hex(res)[2:] == "6F274B6840BB7DC0":
            print(hex(res)[2:])
            print("\nest le bon nombre")
else:
    print("\nFAIL on a ça:")
    print(hex(res)[2:])
    print("\net on non ça:")
    print("6F274B6840BB7DC0")
```

4. Précision sur le fonctionnement

- Utilisation de allmul et aulldiv

```
.text:00401DE0                 ; =============== S U B R O U T I N E =======================================
.text:00401DE0
.text:00401DE0                 ; Attributes: library function
.text:00401DE0
.text:00401DE0                 __allmul proc near                    ; CODE XREF: sub_401090+A5↑p
.text:00401DE0                                                       ; sub_401090+E8↑p
.text:00401DE0 8B 44 24 08     mov     eax, [esp+8]
.text:00401DE4 8B 4C 24 10     mov     ecx, [esp+10h]
.text:00401DE8 0B C8           or      ecx, eax
.text:00401DEA 8B 4C 24 0C     mov     ecx, [esp+0Ch]
.text:00401DEE 75 09           jnz     short hard
.text:00401DF0 8B 44 24 04     mov     eax, [esp+4]
.text:00401DF4 F7 E1           mul     ecx
.text:00401DF6 C2 10 00        retn    10h
.text:00401DF9                 ; ---------------------------------------------------------------------------
```

```
;   AB
; x CD
; ----
;   DB
;  DA0
;  CB0
; CA00
; ----
; RRRR
```

```
R[0:31] = DB[0:31]
R[32:63] = DB[32:63] + DA[0:31] + CB[0:31]
```

```
uint64_t flirt.allmul(uint32_t param_1, uint32_t param_2, uint32_t param_3, uint32_t param_4)
{
    if ((param_4 | param_2) == 0) {
        return (uint64_t)param_1 * (uint64_t)param_3;
    }
    return (uint64_t)param_1 * (uint64_t)param_3 & 0xffffffff |
           (uint64_t)((int32_t)((uint64_t)param_1 * (uint64_t)param_3 >> 0x20) + param_2 * param_3 + param_1 * param_4)
           << 0x20;
}
```

5. Hash de la clé

0x6f274b6840bb7dc0,

```
text:00401519 mov     [ebp+var_8], eax
text:0040151C mov     [ebp+var_4], edx
text:0040151F cmp     [ebp+var_8], 40BB7DC0h
text:00401526 jnz     short loc_401548
text:00401528 cmp     [ebp+var_4], 6F274B68h
text:0040152F jnz     short loc_401548
```

Merci de votre attention

Nous laissons la main à l’autre groupe