Analyse du Scandal0us C0de de haiklr
Par Beginner – 02/07/2007

Introduction

Voilà un nouveau crackme, concocté par haiklr. Vu que je ne me suis jamais encore frotté à ce monsieur, je ne me suis pas encore aperçu du caractère sadique du personnage.

haikrl nous présente son crackme comme mélangeant maths, logique et crypto. Voyons de quoi il en retourne...

I – Généralités

Premier réflexe, PEiD. Comme on pouvais s'en douter, nous sommes gratifiés d'un magnifique « Nothing found * ». Le crackme doit être packé/protégé ou autre.

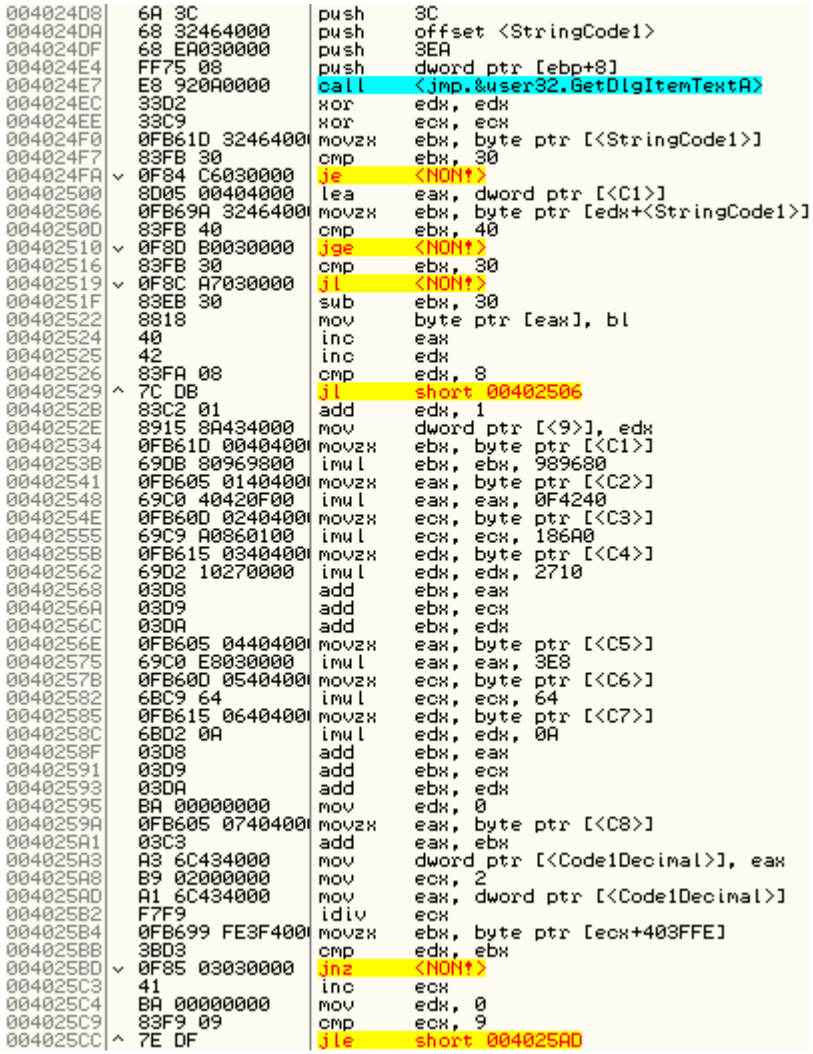
Après avoir chargé l'exécutable dans Olly, on peut observer une routine très propre, avec un petit antidebug à base de IsDebuggerPresent, qui va décrypter l'exécutable par un XOR 40, et qui va reconstruire la table des imports. Nous ne nous étendrons pas sur la partie unpacking de l'executable. Je suppose que vous savez le faire.

Une fois l'exécutable décrypté, on le repasse à la moulinette PEiD, et plus particulièrement son plugin « Krypto ANALyser ». Le plugin détecte alors deux signatures de hash MD5. Pourquoi deux ? C'est ce que nous verrons par la suite...

Ensuite, en posant un BP sur GetDlgItemTextA, on localise la routine de test, qui commence en 00402437. Un petit coup d'oeuil avec l'option graph de IDA nous montre que la routine est assez moche, avec des branchements dans tout les sens...
On peut d'ailleurs observer l'appel à deux API étonnantes : GetComputerNameA, GetLocalTime.
Nous verrons par la suite leur utilisation.

II – Premier code, partie I

Entrons dans le vif du sujet, avec le premier code à entrer. Voilà la routine.



```
004024D8   6A 3C          push    3C
004024DA   68 32464000    push    offset <StringCode1>
004024DF   68 EA030000    push    3EA
004024E4   FF75 08        push    dword ptr [ebp+8]
004024E7   E8 920A0000    call    <jmp.&user32.GetDlgItemTextA>
004024EC   33D2           xor     edx, edx
004024EE   33C9           xor     ecx, ecx
004024F0   0FB61D 3246400 movzx   ebx, byte ptr [<StringCode1>]
004024F7   83FB 30        cmp     ebx, 30
004024FA   0F84 C6030000  je      <NON!>
00402500   8D05 00404000  lea     eax, dword ptr [<C1>]
00402506   0FB69A 3246400 movzx   ebx, byte ptr [edx+<StringCode1>]
0040250D   83FB 40        cmp     ebx, 40
00402510   0F8D B0030000  jge     <NON!>
00402516   83FB 30        cmp     ebx, 30
00402519   0F8C A7030000  jl      <NON!>
0040251F   83EB 30        sub     ebx, 30
00402522   8818           mov     byte ptr [eax], bl
00402524   40             inc     eax
00402525   42             inc     edx
00402526   83FA 08        cmp     edx, 8
00402529   7C DB          jl      short 00402506
0040252B   83C2 01        add     edx, 1
0040252E   8915 8A434000  mov     dword ptr [<9>], edx
00402534   0FB61D 0040400 movzx   ebx, byte ptr [<C1>]
0040253B   69DB 80969800  imul    ebx, ebx, 989680
00402541   0FB605 0140400 movzx   eax, byte ptr [<C2>]
00402548   69C0 40420F00  imul    eax, eax, 0F4240
0040254E   0FB60D 0240400 movzx   ecx, byte ptr [<C3>]
00402555   69C9 A0860100  imul    ecx, ecx, 186A0
0040255B   0FB615 0340400 movzx   edx, byte ptr [<C4>]
00402562   69D2 10270000  imul    edx, edx, 2710
00402568   03D8           add     ebx, eax
0040256A   03D9           add     ebx, ecx
0040256C   03DA           add     ebx, edx
0040256E   0FB605 0440400 movzx   eax, byte ptr [<C5>]
00402575   69C0 E8030000  imul    eax, eax, 3E8
0040257B   0FB60D 0540400 movzx   ecx, byte ptr [<C6>]
00402582   6BC9 64        imul    ecx, ecx, 64
00402585   0FB615 0640400 movzx   edx, byte ptr [<C7>]
0040258C   6BD2 0A        imul    edx, edx, 0A
0040258F   03D8           add     ebx, eax
00402591   03D9           add     ebx, ecx
00402593   03DA           add     ebx, edx
00402595   BA 00000000    mov     edx, 0
0040259A   0FB605 0740400 movzx   eax, byte ptr [<C8>]
004025A1   03C3           add     eax, ebx
004025A3   A3 6C434000    mov     dword ptr [<Code1Decimal>], eax
004025A8   B9 02000000    mov     ecx, 2
004025AD   A1 6C434000    mov     eax, dword ptr [<Code1Decimal>]
004025B2   F7F9           idiv    ecx
004025B4   0FB699 FE3F400 movzx   ebx, byte ptr [ecx+403FFE]
004025BB   3BD3           cmp     edx, ebx
004025BD   0F85 03030000  jnz     <NON!>
004025C3   41             inc     ecx
004025C4   BA 00000000    mov     edx, 0
004025C9   83F9 09        cmp     ecx, 9
004025CC   7E DF          jle     short 004025AD
```

Que ce passe-t-il ? La grosse partie en 004024EC ... 004025A3, vérifie que le code est au bon format (chiffres, 1e caractère différent de 0) et ensuite convertit juste le code en décimal (0x989680 = 10000000, 0x0F4240 = 1000000, etc.)

La partie vraiment intéressante est celle située en 004025AD ... 004025CC. C'est 9 petites lignes de codes, mais particulièrement embêtantes...

Si on pose x, le premier code, avec : $x = \sum_{i=0}^{7} x_i 10^i \quad x_i \in \{0...9\}$ et $x_7 \neq 0$, alors x doit vérifier :

pour tout i entre 2 et 9 : $x \bmod i = x_{9-i}$

Soit, pour ceux qui ne n'ont pas forcément d'affinités avec les maths : le 1e caractère doit être le reste de la division par 2, le 2e le reste de la division par 3, ...

Il serait beaucoup trop long de tester toutes les possibilités, mais il est possible de la faire de manière intelligente : sachant que le premier caractère est un reste de division par 2 et doit être différent de 0, alors c'est forcément 1. Le dernier chiffre est alors impair.

De même, **le i[ème] caractère** x_{8-i} **est compris entre 0 et i**. Par conséquent on ne doit plus faire 10^8 tests, mais seulement $\frac{8!}{4} = 10080$, ce qui est facilement faisable.

On rechercha alors de manière exhaustive avec un code moche du genre :

```
a = 1
pour b de 0 à 2 ; pour c de 0 à 3
pour d de 0 à 4 ; pour e de 0 à 5
pour f de 0 à 6 ; pour g de 0 à 7
pour h de 0 à 8
x = a*10000000 + b*1000000 + c*100000 + d*10000 + e*1000 + f*100 + g*10 +h
si  (x mod 2)=a ET (x mod 3)=b ET (x mod 4)=c ET (x mod 5)=d ET (x mod 6)=e
ET (x mod 7)=f ET(x mod 8)=g ET(x mod 9)=h alors afficher(x)
fin pour ; fin pour
fin pour ; fin pour
fin pour ; fin pour
fin pour
```

Ceci nous amène après calcul à la première partie du code : **11121217**

Enfin, le programme vérifie en 004025DA que le 9[ème] caractère est « ? »

Le début du premier code est donc :

11121217?

III – Premier code, partie II

C'est maintenant que le côté sadique de haiklr apparaît dans toute sa splendeur...

Nous avons pu voir qu'avant le premier GetDlgItemTextA, l'exécutable passe par ces lignes :

```
004025E4   66:8B1D BA4340  mov     bx, word ptr [4043BA]
004025EB   66:A1 B8434000  mov     ax, word ptr [4043B8]
004025F1   66:8B0D BC4340  mov     cx, word ptr [4043BC]
004025F8   891D C4434000   mov     dword ptr [<DateDuJour>], ebx
004025FE   A3 D8434000     mov     dword ptr [<JourDeLaSemaine>], eax
00402603   890D CE434000   mov     dword ptr [<Heure>], ecx
```

Le crackme charge la date et l'heure du programme, nous allons voir pourquoi.

Tout content d'avoir trouvé la première partie du code, on tombe un peu perplexe sur cette section :

```
0040260E   33D2            xor     edx, edx
00402610   33C0            xor     eax, eax
00402612   A1 C4434000     mov     eax, dword ptr [<DateDuJour>]
00402617   B9 03000000     mov     ecx, 3
0040261C   F7F9            idiv    ecx
0040261E   85D2            test    edx, edx
00402620   0F84 F2000000   je      00402718
00402626   33D2            xor     edx, edx
00402628   A1 C4434000     mov     eax, dword ptr [<DateDuJour>]
0040262D   B9 02000000     mov     ecx, 2
00402632   F7F9            idiv    ecx
00402634   85D2            test    edx, edx
00402636   74 57           je      short 0040268F
```

Le comportement du crackme est différent selon la date à laquelle on le lance !

Il y a alors 3 cas :

III.a – les jours du mois multiples de 3

On arrive en 00402718 :

```
00402718   8D35 E2434000   lea     esi, dword ptr [<StringNomOrdi>]
0040271E   33D2            xor     edx, edx
00402720   33C9            xor     ecx, ecx
00402722   0FB60432        movzx   eax, byte ptr [edx+esi]
00402726   83F8 00         cmp     eax, 0
00402729 v 74 05           je      short 00402730
0040272B   03C8            add     ecx, eax
0040272D   42              inc     edx
0040272E ^ EB F2           jmp     short 00402722
00402730   A1 94434000     mov     eax, dword ptr [<MD5NomOrdi>]
00402735   8B1D D8434000   mov     ebx, dword ptr [<JourDeLaSemaine>]
0040273B   F7F9            idiv    ecx
0040273D   03D3            add     edx, ebx
0040273F   8915 80434000   mov     dword ptr [<SommeResteJour>], edx
00402745   8B15 8A434000   mov     edx, dword ptr [<9>]
0040274B   0FB69A 3246400( movzx   ebx, byte ptr [edx+<StringCode1>]
00402752   83FB 00         cmp     ebx, 0
00402755 v 74 15           je      short 0040276C
00402757   83FB 40         cmp     ebx, 40
0040275A v 0F8D 66010000   jge     <NON!>
00402760   83FB 30         cmp     ebx, 30
00402763 v 0F8C 5D010000   jl      <NON!>
00402769   42              inc     edx
0040276A ^ EB DF           jmp     short 0040274B
0040276C   E8 9FFCFFFF     call    <StringToDecimalECX>
00402771   A1 80434000     mov     eax, dword ptr [<SommeResteJour>]
00402776   3BC1            cmp     eax, ecx
00402778 v 0F85 48010000   jnz     <NON!>
```

Tout a en fait commencé au tout début du test :

```
0040245E   68 94434000     push    offset <MD5NomOrdi>
00402463   FF35 00424000   push    dword ptr [<LongueurNomOrdi>]
00402469   68 E2434000     push    offset <StringNomOrdi>
0040246E   E8 8DEBFFFF     call    <MD5>
```

Voilà le premier MD5. Il est calculé à partir du nom de l'ordinateur.

On comprend alors clairement la routine :

- on somme tous les caractères du nom de l'ordinateur (402718...40272E)
- on divise le premier dword du MD5 par le jour de la semaine (dimanche = 1,...)
- on somme le reste de la division et la somme des caractères du nom
- on vérifie que la partie entrée est la même que cette somme

Dans le cas ou le jour du mois est un multiple de 3, on a trouvé la 2e partie du code :

$$\sum_i NomOrdi[i] + \textbf{((1}^{\textbf{e}}\textbf{ dword MD5 de NomOrdi) MOD (jour de la semaine))}$$

III.b – les jours du mois pairs mais pas multiples de 3

On arrive en 00402718 :

```
0040268F   >  68 A4434000   push    offset <MD5ModifieNomOrdi>
00402694   .  FF35 0042400( push    dword ptr [<LongueurNomOrdi>]
0040269A   .  68 E2434000   push    offset <StringNomOrdi>
0040269F   .  E8 40F3FFFF   call    <MD5Modifie>
004026A4   .  A1 A4434000   mov     eax, dword ptr [<MD5ModifieNomOrdi>]
004026A9   .  8B1D A8434000 mov     ebx, dword ptr [4043A8]
004026AF   .  03C3          add     eax, ebx
004026B1   .  8B1D AC43400( mov     ebx, dword ptr [4043AC]
004026B7   .  8B0D B043400( mov     ecx, dword ptr [4043B0]
004026BD   .  03C3          add     eax, ebx
004026BF   .  03C1          add     eax, ecx
004026C1   .  A3 76434000   mov     dword ptr [<SommeMD5Modifie>], eax
004026C6   .  8B15 8A43400( mov     edx, dword ptr [<9>]
004026CC   >  0FB69A 32464( movzx   ebx, byte ptr [edx+<StringCode1>]
004026D3   .  83FB 00       cmp     ebx, 0
004026D6   .v 74 15         je      short 004026ED
004026D8   .  83FB 40       cmp     ebx, 40
004026DB   .v 0F8D E501000( jge     <NON!>
004026E1   .  83FB 30       cmp     ebx, 30
004026E4   .v 0F8C DC01000( jl      <NON!>
004026EA   .  42            inc     edx
004026EB   .^ EB DF         jmp     short 004026CC
004026ED   >  E8 1EFDFFFF   call    <StringToDecimalECX>
004026F2   .  A1 76434000   mov     eax, dword ptr [<SommeMD5Modifie>]
004026F7   .  3BC1          cmp     eax, ecx
004026F9   .v 0F85 C701000( jnz     <NON!>
```

Voilà le second MD5. Il est toujours calculé à partir du nom de l'ordinateur. Mais pourquoi avoir codé 2 fois l'algorithme MD5 alors que une seule fois suffirait ?

La réponse se trouve en allant examiner de plus près les routines que j'ai appelé <MD5> et <MD5Modifié> :

Dans <MD5> on a au début :

```
00401054  .  C706 0123456  mov     dword ptr [esi], 67452301
0040105A  .  C746 04 89AB  mov     dword ptr [esi+4], EFCDAB89
00401061  .  C746 08 FEDC  mov     dword ptr [esi+8], 98BADCFE
00401068  .  C746 0C 7654  mov     dword ptr [esi+C], 10325476
```

Alors que dans <MD5Modifié> on a :

```
00401A38  .  8B1D CE43400  mov     ebx, dword ptr [<Heure>]
00401A3E  .  891E          mov     dword ptr [esi], ebx
00401A40  .  C746 04 89AB  mov     dword ptr [esi+4], EFCDAB89
00401A47  .  C746 08 FEDC  mov     dword ptr [esi+8], 98BADCFE
00401A4E  .  C746 0C 7654  mov     dword ptr [esi+C], 10325476
```

Le hash est initialisé différemment, et en fonction de l'heure en plus !!!

Cette surprise passée, on peut alors effectuer sereinement l'analyse de la routine :

- on calcule le hash modifié du nom de l'ordinateur
- on somme les 4 dwords obtenus
- on vérifie que la partie entrée est la même que cette somme

Dans ce cas, on a encore trouvé la 2e partie du code :

c'est la somme des 4 dword du MD5 modifié du nom de l'ordinateur

III.c – les autres jours

Dans ce cas la routine est minuscule :

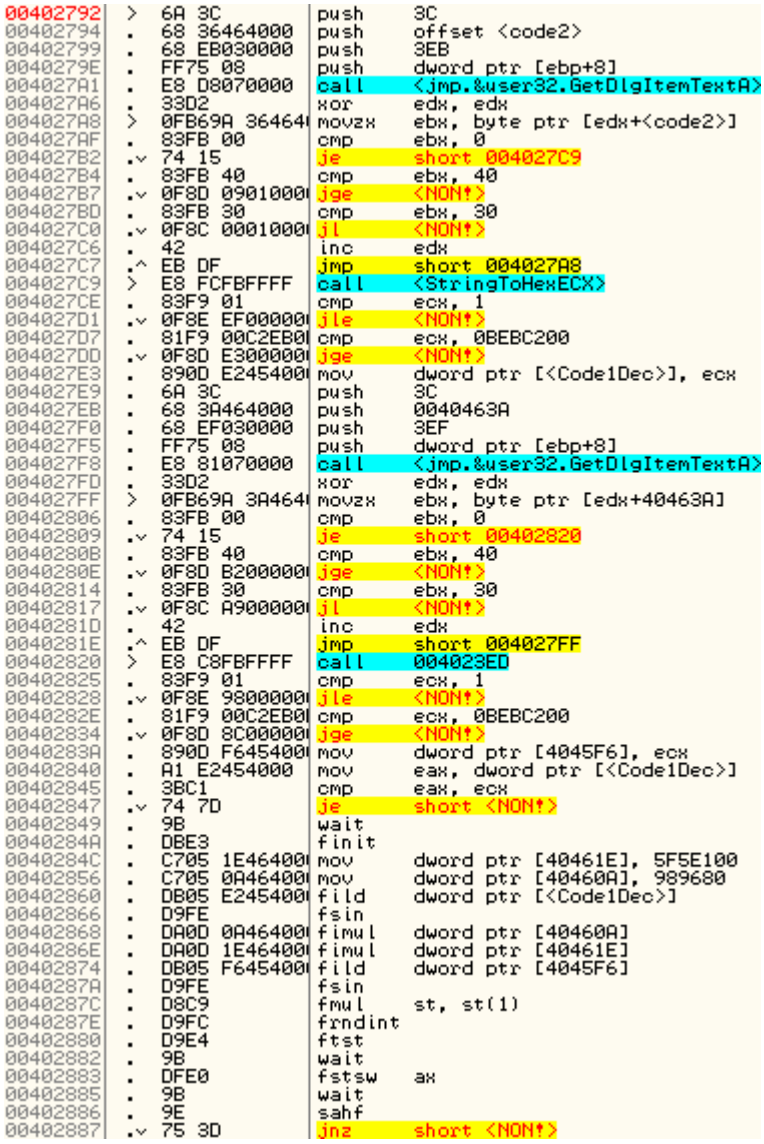
```
00402660  .  8B15 8A43400  mov     edx, dword ptr [<9>]
00402666  .  0FB69A 32464  movzx   ebx, byte ptr [edx+<StringCode1>]
0040266D  .  83FB 21       cmp     ebx, 21
00402670  .v 0F85 5002000  jnz     <NON!>
```

Elle se contente de vérifier si le 10e caractère est « ! »

IV – Deuxième et troisième codes

Maintenant que l'on a le premier code, il ne nous reste plus qu'à chercher la suite. Et là, le code est tout simple mais les maths commencent...

La routine est la suivante :



```
00402792  >  6A 3C          push    3C
00402794  .  68 36464000    push    offset <code2>
00402799  .  68 EB030000    push    3EB
0040279E  .  FF75 08        push    dword ptr [ebp+8]
004027A1  .  E8 D8070000    call    <jmp.&user32.GetDlgItemTextA>
004027A6  .  33D2           xor     edx, edx
004027A8  >  0FB69A 36464(  movzx   ebx, byte ptr [edx+<code2>]
004027AF  .  83FB 00        cmp     ebx, 0
004027B2  .v 74 15          je      short 004027C9
004027B4  .  83FB 40        cmp     ebx, 40
004027B7  .v 0F8D 0901000(  jge     <NON!>
004027BD  .  83FB 30        cmp     ebx, 30
004027C0  .v 0F8C 0001000(  jl      <NON!>
004027C6  .  42             inc     edx
004027C7  .^ EB DF          jmp     short 004027A8
004027C9  >  E8 FCFBFFFF    call    <StringToHexECX>
004027CE  .  83F9 01        cmp     ecx, 1
004027D1  .v 0F8E EF00000(  jle     <NON!>
004027D7  .  81F9 00C2EB0(  cmp     ecx, 0BEBC200
004027DD  .v 0F8D E300000(  jge     <NON!>
004027E3  .  890D E245400(  mov     dword ptr [<Code1Dec>], ecx
004027E9  .  6A 3C          push    3C
004027EB  .  68 3A464000    push    0040463A
004027F0  .  68 EF030000    push    3EF
004027F5  .  FF75 08        push    dword ptr [ebp+8]
004027F8  .  E8 81070000    call    <jmp.&user32.GetDlgItemTextA>
004027FD  .  33D2           xor     edx, edx
004027FF  >  0FB69A 3A464(  movzx   ebx, byte ptr [edx+40463A]
00402806  .  83FB 00        cmp     ebx, 0
00402809  .v 74 15          je      short 00402820
0040280B  .  83FB 40        cmp     ebx, 40
0040280E  .v 0F8D B200000(  jge     <NON!>
00402814  .  83FB 30        cmp     ebx, 30
00402817  .v 0F8C A900000(  jl      <NON!>
0040281D  .  42             inc     edx
0040281E  .^ EB DF          jmp     short 004027FF
00402820  >  E8 C8FBFFFF    call    004023ED
00402825  .  83F9 01        cmp     ecx, 1
00402828  .v 0F8E 9800000(  jle     <NON!>
0040282E  .  81F9 00C2EB0(  cmp     ecx, 0BEBC200
00402834  .v 0F8D 8C00000(  jge     <NON!>
0040283A  .  890D F645400(  mov     dword ptr [4045F6], ecx
00402840  .  A1 E2454000    mov     eax, dword ptr [<Code1Dec>]
00402845  .  3BC1           cmp     eax, ecx
00402847  .v 74 7D          je      short <NON!>
00402849  .  9B             wait
0040284A  .  DBE3           finit
0040284C  .  C705 1E46400(  mov     dword ptr [40461E], 5F5E100
00402856  .  C705 0A46400(  mov     dword ptr [40460A], 989680
00402860  .  DB05 E245400(  fild    dword ptr [<Code1Dec>]
00402866  .  D9FE           fsin
00402868  .  DA0D 0A46400(  fimul   dword ptr [40460A]
0040286E  .  DA0D 1E46400(  fimul   dword ptr [40461E]
00402874  .  DB05 F645400(  fild    dword ptr [4045F6]
0040287A  .  D9FE           fsin
0040287C  .  D8C9           fmul    st, st(1)
0040287E  .  D9FC           frndint
00402880  .  D9E4           ftst
00402882  .  9B             wait
00402883  .  DFE0           fstsw   ax
00402885  .  9B             wait
00402886  .  9E             sahf
00402887  .v 75 3D          jnz     short <NON!>
```

Son action est très simple :

- on récupère le 2e code ($code_1$), on vérifie qu'il est au bon format,
- on le convertit en décimal et on vérifie qu'il est supérieur à 0 et inférieur à 200,000,000
- on fait la même chose pour le 3e code ($code_2$)
- on calcule alors : $\sin(code_1)\cdot\sin(code_2)\cdot 10^{15}$
- on vérifie que l'entier le plus proche est zéro.

Voilà pour le code, place aux mathématiques.

Le but est de trouver deux entiers n_1, n_2 tels que $\left|\sin(n_1)\sin(n_2)\right| < 10^{-15}$

La théorie nous dis que la suite $(\sin(n))_{n\in\mathbb{N}}$ est dense dans $[-1,1]$.

Ou, pour reformuler : pour tout $\varepsilon > 0$ même très petit, il est possible de trouver un entier n tel que

$$\left|\sin(n)\right| < \varepsilon$$

La théorie, nous dit donc que ce qu'on cherche existe, il reste plus qu'à trouver ces entiers n_1, n_2 tels que :

$$|\sin(n_i)| < 10^{-8} \qquad \text{(comme ça on aura } |\sin(n_1)\sin(n_2)| < 10^{-16} < 10^{-15}\text{)}$$

On sait que $\sin(\pi) = 0$ et nous allons nous servir de ce résultat :
Soit a et b deux entiers tels que $\frac{a}{b}$ fraction irréductible soit une bonne approximation de π.

On a alors $\frac{a}{b} = \pi + \varepsilon$ avec ε très petit, d'où $a = b \cdot \pi + b \cdot \varepsilon$
dans ces conditions alors

$$\sin(a) = \sin(b \cdot \pi + b \cdot \varepsilon) = \sin(b \cdot \varepsilon)$$

Conclusion : **si** $\frac{a}{b}$ **est une bonne approximation de** π**, alors** $\sin(a)$ **est très proche de 0**

Il ne reste plus qu'à chercher de bonnes approximations de π avec a inférieur à 200,000,000.

On trouve par exemple (http://perso.orange.fr/jean-paul.davalan/arit/frcont/index.html):

$$\frac{5419351}{1725033}, \frac{80143857}{25510582} \text{ ou } \frac{165707065}{52746197}$$

Il nous suffit par exemple de prendre :

$$\begin{cases} code_1 = 5419351 \\ code_2 = 165707065 \end{cases}$$

Remerciements

Je tiens à remercier tout d'abord *haiklr* pour ce crackme qui a très joliment mis de vraies mathématiques à un niveau abordable dans ce crackme, mélangées à tout plein de bon trucs...

Merci beaucoup également à tous ceux de FC qui tiennent ce forum en vie, et grâce à qui on échange et apprend toujours plus. Je ne cite personne, sinon je vais en oublier.

Beginner