Solution to Crackme v 1.3 by Greedy_Fly

author: baderj (www.johannesbader.ch)

Introduction

The crackme **Crackme v 1.3** by *Greedy_Fly* is written in Assembler and runs on Windows. It has been published October 1st 2014 and is rated **3 - Getting harder**. You can find the crackme here. The description of the crackme is::

```
Hi, All!!!

I guess difficulty of this crackme - 3?

Solution: Valid Serial and solution.txt

//Don't post your solution(Serial) on the board!

Have Fun!

Greedy_Fly
```

Serial Length and Format

Locating the Key Validation Routine

To find the key validation routine, I searched for `GetDlgItem` calls. All of the call are made by `DialogFunc`. The routine handles all `DialogFunc` calls, including the one triggered when pressing the `Register` button. To get to the key validation routine the message can't be `110h` (which is `WM_INITDIALOG`):

```
.text:004011BE 0                cmp     [ebp+uMsg], WM_INITDIALOG
.text:004011C5 0                jnz     short loc_40121A
```

We also skip the routine if the message is `WM_PAINT` (constant `0xF`):

```
.text:0040121A 0                cmp     [ebp+uMsg], WM_PAINT
.text:0040121E 0                jnz     loc_4012AF
```

If, however, the message is `WM_COMMAND` (constant `0x111`) with `wParam` set to `LB_SETTABSTOPS` (`0x192`) we reach the key validation algorithm::

```
.text:004012AF 0                cmp     [ebp+uMsg], WM_COMMAND
.text:004012B6 0                jnz     loc_4014BE
.text:004012BC 0                cmp     [ebp+wParam], LB_SETTABSTOPS
```

```
.text:004012C3 0                jnz     loc_4014A5
.text:004012C9 0                push    68h             ; nIDDlgItem
.text:004012CB 0                push    [ebp+hDlg]      ; hDlg
.text:004012CE 0                call    GetDlgItem
```

Serial Length

The first steps of key validation are to retrieve the serial (I renamed the variables accordingly). The code fetches at most 32 bytes, so the serial can't be longer than that:

```
.text:004012CE 0                call    GetDlgItem
.text:004012D3 0                mov     dword_403074, eax
.text:004012D8 0                push    0CFh
.text:004012DD 0                lea     eax, serial
.text:004012E3 0                push    eax             ; lParam
.text:004012E4 0              push    20h           ; wParam = nr of chars max
.text:004012E6 0                push    WM_GETTEXT      ; Msg
.text:004012E8 0                push    dword_403074    ; hWnd
.text:004012EE 0                call    SendMessageA
```

The number of characters of the serial - returned in `eax` by `SendMessageA` - is stored in `nr_of_chars`.

```
.text:004012F3 0                mov     edx, 19FCh
.text:004012F8 0                mov     ebx, 1895h
.text:004012FD 0                mov     dword ptr nr_of_chars, eax
```

Next we enter a sequence of floating point operations::

```
.text:00401302 0                finit
.text:00401305 0                fild    con_2_dword_403000
.text:0040130B 1                fild    dword ptr nr_of_chars
.text:00401311 2                fyl2x
```

The first `fild` loads the constant 2 from `dword_403000`, the second `fild` loads the number of character in the serial - let that be n. Finally `fyld2x` calculates `ST1*log2(ST0)`, i.e.,:

$$ST_1 = 2 \cdot \log_2(n)$$

The next codes lines are::

```
.text:00401313 1                fld     st
.text:00401315 2                frndint
```

The instruction `frndint` rounds `ST` according to the rounding mode set by `RC`, in our case it is still set to the default which is *round to nearest (even)*:

$$ST = \text{round}(2 \cdot \log_2(n))$$

Next:

```
.text:00401317 2                fxch    st(1)
.text:00401319 2                fsub    st, st(1)
```

These instruction calculate the difference between the unrounded and rounded value:

$$ST = 2 \cdot \log_2(n) - \text{round}(2 \cdot \log_2(n))$$

The next instruction is `f2xml`:

```
.text:0040131B 2                f2xm1
```

It calculates `2^ST - 1`:

$$ST = 2^{2 \cdot \log_2(n) - \text{round}(2 \cdot \log_2(n))} - 1$$

The next instructions add 1 to the result::

```
.text:0040131D 2                fld1
.text:0040131F 3                faddp   st(1), st
```

which leads to

$$ST = 2^{2 \cdot \log_2(n) - \text{round}(2 \cdot \log_2(n))}$$

The last math operation is:

```
.text:00401321 2                fscale
```

The instruction `fscale` calculates `ST0*2^(roundtowardszero(ST1))`. In `ST1` we still have `round(2*log_2(n))`, so we end up with:

$$\begin{aligned} ST &= \left(2^{2 \cdot \log_2(n) - \text{round}(2 \cdot \log_2(n))}\right) \cdot 2^{\text{round}(2 \log_2(n))} \\ &= \frac{2^{2 \cdot \log_2(n)}}{2^{\text{round}(2 \cdot \log_2(n))}} \cdot 2^{\text{round}(2 \log_2(n))} \\ &= 2^{2 \cdot \log_2(n)} \\ &= n^2 \end{aligned}$$

The result `n^2` is stored in eax and tested with the lines that follow::

```
.text:00401323 2                fistp   l_squared
.text:00401329 1                mov     eax, l_squared
.text:0040132E 1                mov     edi, eax
.text:00401330 1                mov     esi, edx
.text:00401332 1                not     esi
.text:00401334 1                and     eax, esi
.text:00401336 1                not     edi
.text:00401338 1                and     edi, edx
.text:0040133A 1                add     edi, eax
.text:0040133C 1                sub     edi, ebx
.text:0040133E 1                jz      short loc_401346
.text:00401340 1                push    offset loc_401487
.text:00401345 1                retn
```

Those instructions boil down to:

$$
\begin{aligned}
a &:= 0x19FC \\
b &:= 0x1895 \\
b &\stackrel{!}{=} \neg a \wedge n^2 + a \wedge \neg n^2 \\
\Rightarrow b &= a \oplus n^2 \\
\Rightarrow n^2 &= a \oplus b \\
\Rightarrow n &= \sqrt{a \oplus b} = \sqrt{361} = 19
\end{aligned}
$$

Conclusion: The sequence of FPU instructions, followed by some logical operators, boil down to the simple test if the serial has 19 characters.

Serial Format

If the serial has 19 characters, we get to these lines:

```
.text:00401346 1                pop     ecx
.text:00401347 1                sub     dl, cl
.text:00401349 1                sub     ecx, ecx
.text:0040134B 1                cmp     byte ptr serial_14, dl
.text:00401351 1                jnz     loc_401487
.text:00401357 1                cmp     byte ptr serial_9, dl
.text:0040135D 1                jnz     loc_401487
.text:00401363 1                cmp     byte ptr serial_4, dl
.text:00401369 1                jnz     loc_401487
```

Since `edx` is still set to the constant `0x19FC`, and `ecx` is also a constant set in "`004012D8 0 push 0CFh`", the result of `sub dl, cl` is always `dl = 2D`. The lines therefore check if the fourth, ninth and fourteenth character of the serial are 2D, which is the ASCII code for `"-"`. So we know the serial has the format::

`XXXX-XXXX-XXXX-XXXX`

Where the `X` can be anything.

First Group of Four Characters

After the format of the serial is checked, the four parts are concatenated on the stack, and passed to `parse_serial`:

```
.text:0040136F 1 lea     edi, empty
.text:00401375 1 push    dword ptr serial_15
.text:0040137B 1 push    dword ptr serial_10
.text:00401381 1 push    dword ptr serial_5
.text:00401387 1 push    dword ptr serial
.text:0040138D 1 push    esp
.text:0040138E 1 pop     ecx
.text:0040138F 1 push    offset empty
.text:00401394 1 push    ecx
.text:00401395 1 call    parse_serial
```

So if the entered serial is `1234-5678-90AB-CDEF` we get `1234567890ABCDEF`. The routine `parse_serial` has the following disassembly::

```
.text:00401522  parse_serial proc near             ; CODE XREF: DialogFunc+1DDp
.text:00401522
.text:00401522   serial_concat= dword ptr  8
.text:00401522   res= dword ptr  0Ch
.text:00401522
.text:00401522   push    ebp
.text:00401523   mov     ebp, esp
.text:00401525   push    esi
.text:00401526   push    edi
.text:00401527   push    ebx
.text:00401528   mov     esi, [ebp+serial_concat]
.text:0040152B   mov     edi, [ebp+res]
.text:0040152E   jmp     short loc_401538
.text:00401530  ; ---------------------------------------------------------------------------
.text:00401530
.text:00401530  loc_401530:                       ; CODE XREF: parse_serial+41j
.text:00401530   and     ebx, 0Fh
.text:00401533   add     eax, ebx
.text:00401535   mov     [edi], al
.text:00401537   inc     edi
.text:00401538
.text:00401538  loc_401538:                        ; CODE XREF: parse_serial+Cj
.text:00401538   movzx   edx, byte ptr [esi]
.text:0040153B   cmp     edx, 40h
```

```
.text:0040153E   sbb     ebx, ebx
.text:00401540   sub     edx, 37h
.text:00401543   and     ebx, 7
.text:00401546   inc     esi
.text:00401547   add     ebx, edx
.text:00401549   js      short loc_401565
.text:0040154B   mov     eax, ebx
.text:0040154D   shl     eax, 4
.text:00401550   mov     [edi], al
.text:00401552   movzx   edx, byte ptr [esi]
.text:00401555   cmp     edx, 40h
.text:00401558   sbb     ebx, ebx
.text:0040155A   sub     edx, 37h
.text:0040155D   and     ebx, 7
.text:00401560   inc     esi
.text:00401561   add     ebx, edx
.text:00401563   jns     short loc_401530
.text:00401565
.text:00401565  loc_401565:                      ; CODE XREF: parse_serial+27j
.text:00401565   pop     ebx
.text:00401566   pop     edi
.text:00401567   pop     esi
.text:00401568   leave
.text:00401569   retn    8
.text:00401569   parse_serial endp
.text:00401569
.text:0040156C  ; ---------------------------------------------------------------------------
.text:0040156C   push    ebp
.text:0040156D   mov     ebp, esp
.text:0040156F   push    edi
.text:00401570   push    esi
.text:00401571   push    ebx
.text:00401572   mov     ebx, [ebp+0Ch]
.text:00401575   mov     edi, [ebp+10h]
.text:00401578   test    ebx, ebx
.text:0040157A   mov     esi, [ebp+8]
.text:0040157D   jz      short loc_4015B5
.text:0040157F
.text:0040157F  loc_40157F:                       ; CODE XREF: .text:004015B3j
.text:0040157F   movzx   eax, byte ptr [esi]
.text:00401582   mov     ecx, eax
.text:00401584   add     edi, 2
.text:00401587   shr     ecx, 4
.text:0040158A   and     eax, 0Fh
.text:0040158D   and     ecx, 0Fh
.text:00401590   cmp     eax, 0Ah
.text:00401593   sbb     edx, edx
.text:00401595   adc     eax, 0
```

```
.text:00401598   lea     eax, [eax+edx*8+37h]
.text:0040159C   cmp     ecx, 0Ah
.text:0040159F   sbb     edx, edx
.text:004015A1   adc     ecx, 0
.text:004015A4   shl     eax, 8
.text:004015A7   lea     ecx, [ecx+edx*8+37h]
.text:004015AB   or      eax, ecx
.text:004015AD   inc     esi
.text:004015AE   mov     [edi-2], ax
.text:004015B2   dec     ebx
.text:004015B3   jnz     short loc_40157F
.text:004015B5
.text:004015B5  loc_4015B5:                          ; CODE XREF: .text:0040157Dj
.text:004015B5   mov     eax, edi
.text:004015B7   mov     byte ptr [edi], 0
.text:004015BA   sub     eax, [ebp+10h]
.text:004015BD   pop     ebx
.text:004015BE   pop     esi
.text:004015BF   pop     edi
.text:004015C0   leave
.text:004015C1   retn    0Ch
.text:004015C4  ; ---------------------------------------------------------------------------
.text:004015C4   add     edx, 1
.text:004015C7   shl     edx, 3
.text:004015CA   mov     esi, 80h
.text:004015CF   lea     edi, [edx+esi]
.text:004015D2   imul    edx, edi, 1E6h
.text:004015D8   and     esi, 0
.text:004015DB   xchg    esi, edx
.text:004015DD   xor     edx, edx
.text:004015DF   retn
```

The code is quite long, but all it does is convert the serial (given as ASCII codes) to hex. So if the serial starts with `01AD-030B` we get the sequence of bytes: `0x01, 0xAD, 0x03, 0x0B`. You can still enter non hex characters like `G`, the code will convert those as well as long as the ASCII code is above 48. I will ignore this fact in the following and assume the entered serial consists of hex characters.

After parsing the serial, the code checks the first two bytes, i.e., the first four characters:

```
.text:004013AE 1 push    small word ptr [edi]
.text:004013B1 1 pop     ecx
.text:004013B2 1 xchg    ch, cl
.text:004013B4 1 mul     ecx
.text:004013B6 1 imul    eax, ecx
.text:004013B9 1 imul    ecx, 3E80h
.text:004013BF 1 sub     eax, ecx
.text:004013C1 1 cmp     eax, 0FF0BDC00h
.text:004013C6 1 jnz     loc_
```

These lines boils down to the equation:

$$4 \cdot v^2 - 16000 \cdot v - 4278967296 \equiv 0 \mod 2^{32}$$

where v is the integer value of the first four serial characters (for instance for `CAFE-` we get `v=51966`). Solving the quadratic equation gives a couple of solutions: `v_1 = 34768` and `v_2 = -30768`; `v_1 = 67536` and `v_2 = -63536`; and probably more. So which number should we take? I think you can't possibly know at this point. I first went with 34768 which lead to an impossible to solve equation later on. Next I tried 67536, which worked. The number 67536 has hex representation `0x107d0`. We can only represent the lowest 2 bytes with the first four serial characters, but since the rest will overflow anyway that's all we need. So we got our first four characters of the serial::

```
07D0-????-????-????
```

Second Group of Four Characters

Next there is a call to `sub_4014E4` that takes the constant 2 as one parameter, and the serial as the other parameter:

```
.text:004013CC 1                xchg    esi, edi
.text:004013CE 1                push    esi
.text:004013CF 1                mov     edi, con_2_dword_403000
.text:004013D5 1                call    sub_4014E4
```

The Routine is:

```
.text:004014E4  ; =============== S U B R O U T I N E =======================================
.text:004014E4
.text:004014E4
.text:004014E4  ; __int16 __usercall sub_4014E4<ax>(int con_2<edi>, int hexx<esi>)
.text:004014E4  sub_4014E4    proc near           ; CODE XREF: DialogFunc+21Dp
.text:004014E4                  cld
.text:004014E5                  xor     ecx, ecx
.text:004014E7                  dec     ecx
.text:004014E8                  mov     edx, ecx
.text:004014EA
.text:004014EA  loc_4014EA:                       ; CODE XREF: sub_4014E4+2Fj
.text:004014EA                  xor     eax, eax
.text:004014EC                  xor     ebx, ebx
.text:004014EE                  lodsb
.text:004014EF                  xor     al, cl
.text:004014F1                  mov     cl, ch
.text:004014F3                  mov     ch, dl
.text:004014F5                  mov     dl, dh
.text:004014F7                  mov     dh, 8
```

```
.text:004014F9
.text:004014F9  loc_4014F9:                             ; CODE XREF: sub_4014E4+28j
.text:004014F9                  shr     bx, 1
.text:004014FC                  rcr     ax, 1
.text:004014FF                  jnb     short loc_40150A
.text:00401501                  xor     ax, 8320h
.text:00401505                  xor     bx, 0EDB8h
.text:0040150A
.text:0040150A  loc_40150A:                             ; CODE XREF: sub_4014E4+1Bj
.text:0040150A                  dec     dh
.text:0040150C                  jnz     short loc_4014F9
.text:0040150E                  xor     ecx, eax
.text:00401510                  xor     edx, ebx
.text:00401512                  dec     edi
.text:00401513                  jnz     short loc_4014EA
.text:00401515                  not     edx
.text:00401517                  not     ecx
.text:00401519                  mov     eax, edx
.text:0040151B                  rol     eax, 10h
.text:0040151E                  mov     ax, cx
.text:00401521                  retn
.text:00401521   sub_4014E4     endp
```

The code calculates four bytes based on the first two bytes of the serial (which we know are `0x07D0`). See the `helper_scripts` for a Python script that calculates the value, or simply use a debugger. For our first four characters `07D0` of the serial we get the value `88 4B 56 EC`.

Right after the call come these lines::

```
.text:004013DA 1                bswap   eax
.text:004013DC 1                pop     esi
.text:004013DD 1                movzx   ecx, word ptr [esi+2]
.text:004013E1 1                xchg    cl, ch
.text:004013E3 1                xor     ecx, 4E62h
.text:004013E9 1                cmp     ax, cx
.text:004013EC 1                jnz     loc_401487
```

They switch around the bytes in `88 4B 56 EC` to get the two bytes `4B 88`. These bytes are then `XOR`ed with `0x4E62` which gives `0x05EA`. These two bytes need to match the next group in our serial, so we end up knowing:

```
07D0-05EA-????-????
```

Third Group of Four Characters

The next lines are::

```
.text:004013F2 1                mov     ecx, 114Fh
.text:004013F7 1                shr     eax, 10h
.text:004013FA 1                xchg    ah, al
.text:004013FC 1                movzx   ebx, word ptr [esi+4]
.text:00401400 1                sub     ebx, ecx
.text:00401402 1                cdq
.text:00401403 1                div     ebx
.text:00401405 1                sub     eax, 4
.text:00401408 1                or      eax, eax
.text:0040140A 1                jnz     short loc_401416
.text:0040140C 1                sub     edx, 400h
.text:00401412 1                jnz     short loc_401416
```

After `shr` and `xchg`, the value of `eax` is `0x56EC = 22252`, which are the third and fourth byte generated by the `sub_4014E4` call. Next, the lines calculate:

$$eax = \left\lfloor \frac{22252}{s - 4431} \right\rfloor$$
$$edx = 0x56EC \mod s - 4431$$

where s is the value of the third group of the serial. For a valid serial the code requires `eax = 4`, and `edx = 1024`, so our serial group needs to have the following value:

$$s = \frac{22252 - 1024}{4} + 4431 = 9738 = 0x260A$$

so the serial is:

```
07D0-05EA-260A-????
```

Fourth Group of Four Characters

Finally, the last check based on the four last characters of the serial. The code first loads the two bytes of the fourth group into edx, and the two bytes of the second group into eax (the byte order is switched). Next follows a sequence of XOR and XCHG operations, at some point also including the two bytes of the third serial group:

```
.text:0040141C 1                movzx   edx, word ptr [esi+6]
.text:00401420 1                movzx   eax, word ptr [esi+2]
.text:00401424 1                xor     al, dl
.text:00401426 1                xchg    ah, al
.text:00401428 1                xor     al, dl
.text:0040142A 1                xor     al, dh
.text:0040142C 1                xchg    ah, al
.text:0040142E 1                xor     al, dh
.text:00401430 1                shl     eax, 10h
```

```
.text:00401433 1                mov     ax, [esi+4]
.text:00401437 1                xor     al, dl
.text:00401439 1                xchg    ah, al
.text:0040143B 1                xor     al, dl
.text:0040143D 1                xor     al, dh
.text:0040143F 1                xchg    ah, al
.text:00401441 1                xor     al, dh
.text:00401443 1                bswap   eax
.text:00401445 1                cmp     eax, 3F1330DFh
```

The result is compared to `0x3F1330DFh`, which gives these four conditions:

- EA XOR dl XOR dh = DF
- 05 XOR dl XOR dh = 30
- 26 XOR dl XOR dh = 13
- 0A XOR dl XOR dh = 3F

where `dh` is the byte given by the first two characters of the last serial group, and `dl` is the byte given by the last two characters of the last serial group. All four conditions boil down to:

$$dl \text{ XOR } dh = 35$$

So we have 256 choices for the last serial group, e.g., '0035'. Valid serials therefore are:

```
07D0-05EA-0A26-0035
07D0-05EA-0A26-0134
07D0-05EA-0A26-0237
07D0-05EA-0A26-0336
07D0-05EA-0A26-0431
07D0-05EA-0A26-0530
07D0-05EA-0A26-0633
07D0-05EA-0A26-0732
...
```

This keygen produces all 256 valid serials:

```
serial = "07D0-05EA-0A26-"
for i in range(256):
    print('{}{:02x}{:02x}'.format(serial, i, i ^ 0x35))
```