Solution to Crackme “DecryptMe #1” by “HMX0101”

Johannes Bader

2014-11-17

This crackme has been published April 20th 2006 with rating *“Difficulty: 1 - Very easy, for newbies”*. Despite the low level of difficulty, there are no accepted solutions yet. The correct solution has been posted in the comments by `AssemblyJammer69`:

> “Easy one, here is a working key:126, the rest of the keys are increments of 128 using 126 as the starting number.”

This solution is about how to get to this result. The crackme requires you to break an encryption routine, which is obfuscated with meaningless code.

Introduction

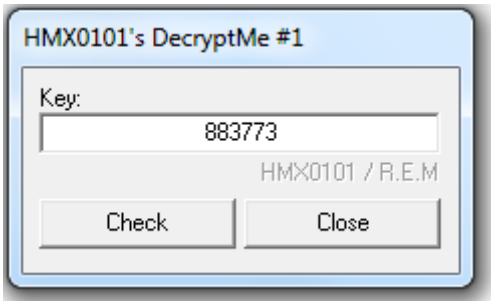

Figure 1: User Interface

The user interface of the crackme (see Figure 1) has an input box to enter the key. Only digits can be entered, i.e., only non-negative integers. After clicking `Check`, you are always presented with a message box titled `OK!`, see Figure 2. The text of the message box depends on the entered key, and is the ciphertext decrypted with the entered key $d_k(c)$.

The crackme does not reveal the correct plaintext and does not give feedback whether we entered the correct key or not. We therefore need to find the correct key *without* a known plaintext. However, we can work with the assumption that the message box text is in English and follows the corresponding letter frequency. But first, we need to reverse engineer the decryption routine.

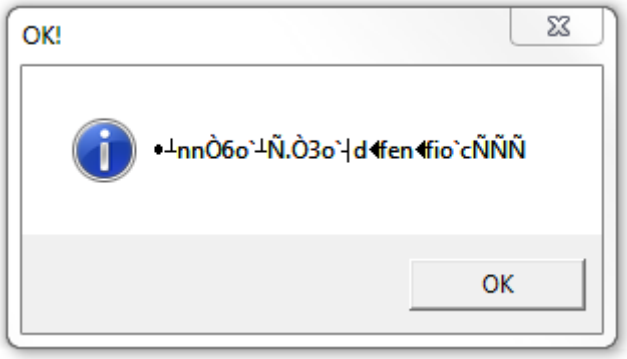

Figure 2: Messagebox after clicking `Check`

Code Overview

We know that the result of the decryption is probably shown in the `OK!` message box (Figure 2). The plaintext is therefore the `lpText` variable of a `MessageBox` call, see `MessageBox` on MSDN[1]. After loading the crackme in IDA Pro we find two references to `MessageBox`. The first one is related to a Runtime error and not the box in Figure 2:

```
CODE:00012E92 push    0               ; uType
CODE:00012E94 push    offset Caption  ; "Error"
CODE:00012E99 push    offset Text     ; "Runtime error     at 00000000"
CODE:00012E9E push    0               ; hWnd
CODE:00012EA0 call    MessageBoxA
```

The second place is more promising, these are the lines that lead up to the second message box:

```
CODE:0001400B push    eax             ; lpCaption OK!
CODE:0001400C lea     edx, [ebp+var_10]
CODE:0001400F mov     eax, ds:off_150D0
CODE:00014014 call    sub_13770
CODE:00014019 mov     eax, [ebp+var_10]
CODE:0001401C push    eax
CODE:0001401D mov     eax, [ebp+var_4]
CODE:00014020 call    sub_13C94
CODE:00014025 mov     edx, eax
CODE:00014027 lea     ecx, [ebp+var_C]
CODE:0001402A pop     eax
CODE:0001402B call    sub_13E1C
CODE:00014030 mov     eax, [ebp+var_C]
CODE:00014033 call    sub_1314C
CODE:00014038 push    eax             ; lpText
CODE:00014039 push    0               ; hWnd
CODE:0001403B call    MessageBoxA_0
```

At offset 1400B the caption "`OK!`" is pushed on the stack. In line 00014038 the text of the message box is pushed, i.e., `eax` holds the ciphertext at this point.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/ms645505%28v=vs.85%29.aspx

Using a debugger, we see that `[ebp+var_4]` holds the serial. Offset 1401D loads the serial in `eax`, followed by a call to `sub_13C94`. The routine `sub_13C94` is::

```
CODE:00013C94 ; =============== S U B R O U T I N E =======================================
CODE:00013C94
CODE:00013C94
CODE:00013C94 sub_13C94       proc near               ; CODE XREF: sub_13FD0+50p
CODE:00013C94
CODE:00013C94 var_24          = dword ptr -24h
CODE:00013C94 var_20          = dword ptr -20h
CODE:00013C94 var_1C          = dword ptr -1Ch
CODE:00013C94 var_18          = dword ptr -18h
CODE:00013C94 sign            = dword ptr -14h
CODE:00013C94
CODE:00013C94                 push    ebx
CODE:00013C95                 push    esi
CODE:00013C96                 push    edi
CODE:00013C97                 push    ebp
CODE:00013C98                 add     esp, 0FFFFFFF4h
CODE:00013C9B                 mov     esi, eax
CODE:00013C9D                 mov     [esp+1Ch+var_1C], 0
CODE:00013CA4                 mov     [esp+1Ch+var_18], 0
CODE:00013CAC                 test    esi, esi
CODE:00013CAE                 jz      loc_13D3A
CODE:00013CB4                 mov     [esp+1Ch+sign], 1
CODE:00013CBC                 mov     edi, 1
CODE:00013CC1                 cmp     byte ptr [esi], '-'
CODE:00013CC4                 jnz     short loc_13CCF
CODE:00013CC6                 mov     [esp+1Ch+sign], -1
CODE:00013CCE                 inc     edi
CODE:00013CCF
CODE:00013CCF loc_13CCF:                              ; CODE XREF: sub_13C94+30j
CODE:00013CCF                 mov     eax, esi
CODE:00013CD1                 call    strlen
CODE:00013CD6                 mov     ebp, eax
CODE:00013CD8                 sub     ebp, edi
CODE:00013CDA                 jl      short loc_13D1E
CODE:00013CDC                 inc     ebp
CODE:00013CDD
CODE:00013CDD loc_13CDD:                              ; CODE XREF: sub_13C94+88j
CODE:00013CDD                 mov     bl, [esi+edi-1]
CODE:00013CE1                 cmp     bl, '0'
CODE:00013CE4                 jb      short loc_13D1E
CODE:00013CE6                 cmp     bl, '9'
CODE:00013CE9                 ja      short loc_13D1E
CODE:00013CEB                 push    0
CODE:00013CED                 push    0Ah
CODE:00013CEF                 mov     eax, [esp+24h+var_1C]
CODE:00013CF3                 mov     edx, [esp+24h+var_18]
CODE:00013CF7                 call    sub_131C4
CODE:00013CFC                 push    edx
CODE:00013CFD                 push    eax
```

```
CODE:00013CFE                 xor     eax, eax
CODE:00013D00                 mov     al, bl
CODE:00013D02                 cdq
CODE:00013D03                 add     eax, [esp+24h+var_24]
CODE:00013D06                 adc     edx, [esp+24h+var_20]
CODE:00013D0A                 add     esp, 8
CODE:00013D0D                 sub     eax, 30h
CODE:00013D10                 sbb     edx, 0
CODE:00013D13                 mov     [esp+1Ch+var_1C], eax
CODE:00013D16                 mov     [esp+1Ch+var_18], edx
CODE:00013D1A                 inc     edi
CODE:00013D1B                 dec     ebp
CODE:00013D1C                 jnz     short loc_13CDD
CODE:00013D1E
CODE:00013D1E loc_13D1E:                              ; CODE XREF: sub_13C94+46j
CODE:00013D1E                                         ; sub_13C94+50j ...
CODE:00013D1E                 cmp     [esp+1Ch+sign], 0
CODE:00013D23                 jge     short loc_13D3A
CODE:00013D25                 mov     eax, [esp+1Ch+var_1C]
CODE:00013D28                 mov     edx, [esp+1Ch+var_18]
CODE:00013D2C                 neg     eax
CODE:00013D2E                 adc     edx, 0
CODE:00013D31                 neg     edx
CODE:00013D33                 mov     [esp+1Ch+var_1C], eax
CODE:00013D36                 mov     [esp+1Ch+var_18], edx
CODE:00013D3A
CODE:00013D3A loc_13D3A:                              ; CODE XREF: sub_13C94+1Aj
CODE:00013D3A                                         ; sub_13C94+8Fj
CODE:00013D3A                 mov     eax, [esp+1Ch+var_1C]
CODE:00013D3D                 mov     edx, [esp+1Ch+var_18]
CODE:00013D41                 add     esp, 0Ch
CODE:00013D44                 pop     ebp
CODE:00013D45                 pop     edi
CODE:00013D46                 pop     esi
CODE:00013D47                 pop     ebx
CODE:00013D48                 retn
CODE:00013D48 sub_13C94       endp
```

We immediately recognize the ASCII code for “0”, “9” and “-”: the routine converts strings to (signed) integers, like the C library function `atoi`. The subroutine return the integer in `eax`. The serial as integer is then passed in `edx` to `sub_13E1C`:

```
CODE:00014020 call    sub_13C94
CODE:00014025 mov     edx, eax
CODE:00014027 lea     ecx, [ebp+var_C]
CODE:0001402A pop     eax
CODE:0001402B call    sub_13E1C
```

Before analysing `sub_1314C` let‘s first check the subroutine `sub_1314C` which is called last showing the message box:

```
CODE:00014033 call    sub_1314C
```

The subroutine is very short:

```
CODE:0001314C sub_1314C proc near                      ; CODE XREF: sub_13FD0+36p
CODE:0001314C                                          ; sub_13FD0+63p ...
CODE:0001314C test    eax, eax
CODE:0001314E jz      short loc_13152
CODE:00013150 retn
CODE:00013150 ; ---------------------------------------------------------------------------
CODE:00013151 byte_13151 db 0                        ; DATA XREF: sub_1314C:loc_13152o
CODE:00013152 ; ---------------------------------------------------------------------------
CODE:00013152
CODE:00013152 loc_13152:                               ; CODE XREF: sub_1314C+2j
CODE:00013152 mov     eax, offset byte_13151
CODE:00013157 retn
CODE:00013157 sub_1314C endp
```

All it does is substitute `eax` with `byte_13141 = 0` if and only if `eax` is 0. Apart from potentially changing the flags, the routine does nothing. Our penultimate routine `sub_13E1C` therefore has to do all the decryption. Renaming the subroutines accordingly we have:

```
CODE:0001400B push    eax                              ; lpCaption OK!
CODE:0001400C lea     edx, [ebp+var_10]
CODE:0001400F mov     eax, ds:off_150D0
CODE:00014014 call    sub_13770
CODE:00014019 mov     eax, [ebp+var_10]
CODE:0001401C push    eax
CODE:0001401D mov     eax, [ebp+serial]
CODE:00014020 call    atoi
CODE:00014025 mov     edx, eax
CODE:00014027 lea     ecx, [ebp+var_C]
CODE:0001402A pop     eax
CODE:0001402B call    decrypting
CODE:00014030 mov     eax, [ebp+var_C]
CODE:00014033 call    nop
CODE:00014038 push    eax                              ; lpText
CODE:00014039 push    0                                ; hWnd
CODE:0001403B call    MessageBoxA_0
```

The Decryption Routine

The decryption subroutine is very long:

```
CODE:00013E1C ; =============== S U B R O U T I N E =======================================
CODE:00013E1C
CODE:00013E1C ; Attributes: bp-based frame
CODE:00013E1C
CODE:00013E1C decrypting proc near                     ; CODE XREF: sub_13FD0+5Bp
CODE:00013E1C
CODE:00013E1C set_to_zero= dword ptr -24h
```

```
CODE:00013E1C strlen_ciphertext= dword ptr -20h
CODE:00013E1C j_plus_1= dword ptr -1Ch
CODE:00013E1C plaintext= dword ptr -18h
CODE:00013E1C i_plus_one= dword ptr -14h
CODE:00013E1C var_10= dword ptr -10h
CODE:00013E1C serial_as_int= dword ptr -0Ch
CODE:00013E1C ciphertext= dword ptr -8
CODE:00013E1C var1= dword ptr -4
CODE:00013E1C
CODE:00013E1C push    ebp
CODE:00013E1D mov     ebp, esp
CODE:00013E1F add     esp, 0FFFFFFDCh
CODE:00013E22 push    ebx
CODE:00013E23 xor     ebx, ebx
CODE:00013E25 mov     [ebp+set_to_zero], ebx
CODE:00013E28 mov     [ebp+var_10], ecx
CODE:00013E2B mov     [ebp+serial_as_int], edx
CODE:00013E2E mov     [ebp+ciphertext], eax
CODE:00013E31 mov     eax, [ebp+ciphertext]
CODE:00013E34 call    increment_special_byte
CODE:00013E39 xor     eax, eax
CODE:00013E3B push    ebp
CODE:00013E3C push    offset loc_13F6F
CODE:00013E41 push    dword ptr fs:[eax]
CODE:00013E44 mov     fs:[eax], esp
CODE:00013E47 mov     eax, [ebp+var_10]
CODE:00013E4A call    sub_12FA4
CODE:00013E4F mov     eax, [ebp+ciphertext]
CODE:00013E52 call    strlen
CODE:00013E57 test    eax, eax
CODE:00013E59 jle     loc_13F51
CODE:00013E5F mov     [ebp+strlen_ciphertext], eax
CODE:00013E62 mov     [ebp+i_plus_one], 1
CODE:00013E69
CODE:00013E69 loc_13E69:                              ; CODE XREF: decrypting+12Fj
CODE:00013E69 mov     eax, [ebp+ciphertext]
CODE:00013E6C call    strlen
CODE:00013E71 mov     [ebp+var1], eax
CODE:00013E74 mov     eax, [ebp+ciphertext]
CODE:00013E77 mov     edx, [ebp+i_plus_one]
CODE:00013E7A movzx   eax, byte ptr [eax+edx-1]       ; get ciphertext[i]
CODE:00013E7F sub     eax, 2644h                      ; subtract a
CODE:00013E84 mov     [ebp+plaintext], eax
CODE:00013E87 mov     eax, [ebp+plaintext]
CODE:00013E8A add     eax, eax                        ; multiply by 2
CODE:00013E8C add     [ebp+var1], eax
CODE:00013E8F xor     [ebp+plaintext], 0DEADh
CODE:00013E96 mov     eax, [ebp+var1]
CODE:00013E99 sar     eax, 1                          ; divide by 2
CODE:00013E9B jns     short loc_13EA0
CODE:00013E9D adc     eax, 0
CODE:00013EA0
```

```
CODE:00013EA0 loc_13EA0:                              ; CODE XREF: decrypting+7Fj
CODE:00013EA0 mov     [ebp+var1], eax
CODE:00013EA3 mov     eax, [ebp+serial_as_int]
CODE:00013EA6 add     eax, 10
CODE:00013EA9 add     [ebp+plaintext], eax
CODE:00013EAC add     [ebp+var1], 1337h
CODE:00013EB3 mov     eax, [ebp+serial_as_int]
CODE:00013EB6 add     eax, eax
CODE:00013EB8 mov     edx, [ebp+plaintext]
CODE:00013EBB sub     edx, eax
CODE:00013EBD xor     edx, [ebp+serial_as_int]
CODE:00013EC0 mov     [ebp+plaintext], edx
CODE:00013EC3 mov     eax, [ebp+plaintext]
CODE:00013EC6 imul    [ebp+serial_as_int]
CODE:00013EC9 sub     [ebp+var1], eax
CODE:00013ECC mov     [ebp+j_plus_1], 1
CODE:00013ED3
CODE:00013ED3 loc_13ED3:                              ; CODE XREF: decrypting+E4j
CODE:00013ED3 mov     eax, [ebp+j_plus_1]
CODE:00013ED6 movzx   eax, ds:byte_1509F[eax]
CODE:00013EDD add     eax, eax
CODE:00013EDF xor     [ebp+var1], eax
CODE:00013EE2 mov     eax, [ebp+j_plus_1]
CODE:00013EE5 movzx   eax, ds:byte_1509F[eax]
CODE:00013EEC xor     [ebp+plaintext], eax
CODE:00013EEF mov     eax, [ebp+var1]
CODE:00013EF2 cdq
CODE:00013EF3 idiv    [ebp+j_plus_1]
CODE:00013EF6 mov     [ebp+var1], eax
CODE:00013EF9 inc     [ebp+j_plus_1]
CODE:00013EFC cmp     [ebp+j_plus_1], 37
CODE:00013F00 jnz     short loc_13ED3
CODE:00013F02 add     [ebp+var1], 0ABh
CODE:00013F09 sub     [ebp+plaintext], 100h
CODE:00013F10 mov     eax, [ebp+plaintext]
CODE:00013F13 add     eax, 0DEADh
CODE:00013F18 imul    [ebp+var1]
CODE:00013F1B mov     [ebp+var1], eax
CODE:00013F1E mov     eax, [ebp+var1]
CODE:00013F21 xor     eax, eax
CODE:00013F23 mov     [ebp+var1], eax
CODE:00013F26 mov     eax, [ebp+var1]
CODE:00013F29 add     [ebp+plaintext], eax
CODE:00013F2C lea     eax, [ebp+set_to_zero]
CODE:00013F2F mov     edx, [ebp+plaintext]            ; EDX & 0xFF IS PLAINTEXT CHAR
CODE:00013F32 call    sub_13098
CODE:00013F37 mov     edx, [ebp+set_to_zero]
CODE:00013F3A mov     eax, [ebp+var_10]
CODE:00013F3D call    copy_from_edx_to_plaintext
CODE:00013F42 mov     eax, [ebp+var_10]
CODE:00013F45 inc     [ebp+i_plus_one]
CODE:00013F48 dec     [ebp+strlen_ciphertext]
```

```
CODE:00013F4B jnz     loc_13E69
CODE:00013F51
CODE:00013F51 loc_13F51:                              ; CODE XREF: decrypting+3Dj
CODE:00013F51 xor     eax, eax
CODE:00013F53 pop     edx
CODE:00013F54 pop     ecx
CODE:00013F55 pop     ecx
CODE:00013F56 mov     fs:[eax], edx
CODE:00013F59 push    offset loc_13F76
CODE:00013F5E
CODE:00013F5E loc_13F5E:                              ; CODE XREF: decrypting+158j
CODE:00013F5E lea     eax, [ebp+set_to_zero]
CODE:00013F61 call    sub_12FA4
CODE:00013F66 lea     eax, [ebp+ciphertext]
CODE:00013F69 call    sub_12FA4
CODE:00013F6E retn
CODE:00013F6F ; ---------------------------------------------------------------------------
CODE:00013F6F
CODE:00013F6F loc_13F6F:                              ; DATA XREF: decrypting+20o
CODE:00013F6F jmp     loc_12A40
CODE:00013F74 ; ---------------------------------------------------------------------------
CODE:00013F74 jmp     short loc_13F5E
CODE:00013F76 ; ---------------------------------------------------------------------------
CODE:00013F76
CODE:00013F76 loc_13F76:                              ; CODE XREF: decrypting+152j
CODE:00013F76                                         ; DATA XREF: decrypting+13Do
CODE:00013F76 pop     ebx
CODE:00013F77 mov     esp, ebp
CODE:00013F79 pop     ebp
CODE:00013F7A retn
CODE:00013F7A decrypting endp
```

With help of debugging we find out that after offset 13F2F, the least significant byte of `edx` contains the plaintext characters. The whole routine has many instructions that don‘t affect the plaintext. Removing all unnecessary operations leads to this decryption routine:

```
FUNCTION decrypt(ciphertext, key)
    plaintext = ""
    FORALL characters c IN ciphertext DO
        plaintext_char = c - 0x2644
        plaintext_char ^= 0x0DEAD
        plaintext_char += 10
        plaintext_char -= key
        plaintext_char ^= key
        plaintext += (plaintext_char & 0xFF)
    END FOR
END FUNCTION
```

The ciphertext is passed to the decryption routine as `eax`, which is set to `var_10` here:

```
CODE:00014019 mov     eax, [ebp+var_10]
```

The ciphertext consists of the following 30 bytes:

```
0x74,0x66,0x6f,0x6f,0xc3,0x47,0x6c,0x6d,0x66,0xc2,
0xaf,0xc3,0x60,0x6c,0x6d,0x64,0x71,0x82,0x17,0x16,
0x6f,0x82,0x17,0x6a,0x6c,0x6d,0x70,0xc2,0xc2,0xc2
```

Finding the correct plaintext

I'm using bruteforce to find the correct plaintext, starting with key `0`. From the decryption algorithm we know for any key k, all keys k' with $k' = k \mod 256$ will lead to the same plaintext, i.e.,

$$d_k(c) = d_{k'}(c) \quad \forall\, k` = k \mod 256$$

So we only have to try 256 different keys. We could easily check the 256 potential plaintexts manually to find the most probable one, but I`m using using the following Python script to rate the plaintexts for me:

```
import re

def get_freq():
    frequencies = {}
    with open("ascii_frequencies.txt", "r") as r:
        for f in r:
            m = re.search("(\d+)\s.*\(\s*([\d.]+)", f)
            if m:
                frequencies[int(m.group(1))] = float(m.group(2))
    return frequencies

def rate_plaintext(plaintext, freq):
    score = 1
    for s in plaintext:
        score *= freq.get(ord(s),0)
    return score

freq = get_freq()
ciphertext = [0x74,0x66,0x6f,0x6f,0xc3,0x47,0x6c,0x6d,0x66,0xc2,0xaf,
        0xc3,0x60,0x6c,0x6d,0x64,0x71,0x82,0x17,0x16,0x6f,0x82,0x17,
        0x6a,0x6c,0x6d,0x70,0xc2,0xc2,0xc2]

best_rating = 1
for key in range(0,256):
    plaintext = ""
    for c in ciphertext:
        plaintext_char = c - 0x2644
        plaintext_char ^= 0x0DEAD
        plaintext_char += 10
        plaintext_char -= key
        plaintext_char ^= key
        plaintext += chr(plaintext_char & 0xFF)
```

```
    rating = rate_plaintext(plaintext, freq)
    if rating >= best_rating:
        print("best rating {} for key {} and plaintext:\n{}".format(
            rating, key, plaintext))
        best_rating = rating
```

The script rates plaintexts by multiplying the expected frequencies of all characters. I got the frequencies from a post on fitaly.com[2] The character `e` for example has a frequency of 8.5771%, while the ASCII code 23 (for ETB), has a zero frequency. Running the script returns two keys with the same best plaintext.

```
$ python find_plaintext.py
best rating 329462.124097 for key 126 and plaintext:
Well Done!, Congratulations!!!
best rating 329462.124097 for key 254 and plaintext:
Well Done!, Congratulations!!!
phreak@hp:scripts$
```

The valid keys are therefore $k \in \mathbb{N}$, $k = 126 \mod 128$, or $k = \{126, 254, 382, 510, 638, 766, 894, ...\}$

[2] http://fitaly.com/board/domper3/posts/136.html