Crackmes.de – browsesan01suke's SomeCrypto~01

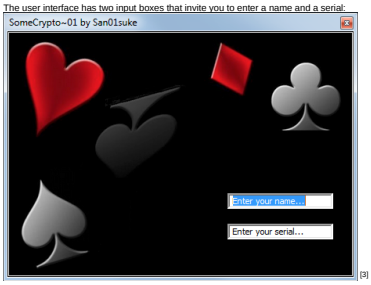
Posted By Johannes On July 14, 2014 @ 19:39 In information security | No Comments

The author *san01suke* submitted three crackmes to www.crackmes.de [1] on July 1st. This is my attempt to solve the first one, called **SomeCrypto~01**. You can view and download the crackme here [2]. The short description simply says:

> "Just solve this simple crackme"
>
> – san01suke

The User Interace

The user interface has two input boxes that invite you to enter a name and a serial:

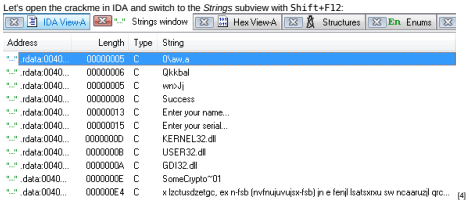


[3]

There are no buttons to validate the input; the crackme probably checks the input whenever the content of the text boxes changes or periodically with a timer event. There is no feedback when you enter a wrong name/serial combination.

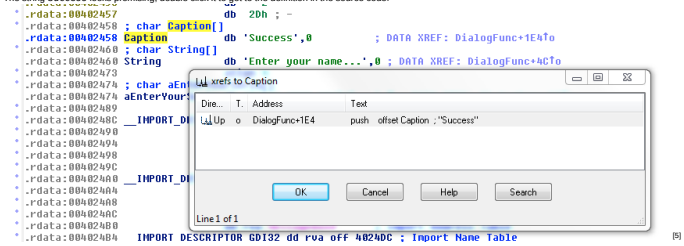
Looking for the Good Boy Message

Let's open the crackme in IDA and switch to the *Strings* subview with `Shift+F12`:



[4]

The string `Success` looks promising, double click it to get to the definition in the source code:



[5]

Putting the cursor on the variable name and hitting X brings us to the only code location that references the success string:

```
1 .text:0040129D            call    loc_401000
2 .text:004012A2            add     esp, 4
3 .text:004012A5            mov     byte_403270, al
4 .text:004012AA            test    al, al
5 .text:004012AC            jz      short loc_4012CA
6 .text:004012AE            mov     ecx, [esp+100h+lpText]
7 .text:004012B2            push    0               ; uType
8 .text:004012B4            push    offset Caption  ; "Success"
```

Lines 4 and 5 check if `al` is zero and jump over the "Success" part if `al` is non zero. The register `eax` is probably set by the subroutine `loc_401000`, at least it is the common register to hold return values. So the task is to find name/serial combinations for which the subroutines `loc_401000` does not return null.

A First Look at loc_401000

Double clicking the label `loc_401000` brings us to the `loc_401000` subroutine. Let's have a quick look at the entire code before starting to analyze it in detail:

```
1  loc_401000:                             ; CODE XREF: DialogFunc+1CDp
2  .text:00401000              push    ebp
3  .text:00401001              mov     ebp, esp
4  .text:00401003              mov     al, [ecx]
5  .text:00401005              sub     esp, 20h
6  .text:00401008              push    esi
7  .text:00401009              xor     esi, esi
8  .text:0040100B              test    al, al
9  .text:0040100D              jz      loc_4010C6
10 .text:00401013              lea     edx, [ebp-20h]
11 .text:00401016              sub     edx, ecx
12 .text:00401018
13 .text:00401018 loc_401018:                          ; CODE XREF: .text:00401032j
14 .text:00401018              cmp     al, 61h
15 .text:0040101A              jl      loc_4010C6
16 .text:00401020              cmp     al, 7Ah
17 .text:00401022              jg      loc_4010C6
18 .text:00401028              mov     [edx+ecx], al
19 .text:0040102B              mov     al, [ecx+1]
20 .text:0040102E              inc     ecx
21 .text:0040102F              inc     esi
22 .text:00401030              test    al, al
23 .text:00401032              jnz     short loc_401018
24 .text:00401034              cmp     esi, 1Ah
25 .text:00401037              jnz     loc_4010C6
26 .text:0040103D              xor     eax, eax
27 .text:0040103F              nop
28 .text:00401040
29 .text:00401040 loc_401040:                          ; CODE XREF: .text:0040104Fj
30 .text:00401040              mov     cl, byte_403010[eax]
31 .text:00401046              mov     byte_403140[eax], cl
32 .text:0040104C              inc     eax
33 .text:0040104D              test    cl, cl
34 .text:0040104F              jnz     short loc_401040
35 .text:00401051              xor     ecx, ecx
36 .text:00401053              cmp     byte_403140, cl
37 .text:00401059              jz      short loc_401088
38 .text:0040105B              jmp     short loc_401060
39 .text:0040105B ; ---------------------------------------------------------------------------
40 .text:0040105D              align 10h
41 .text:00401060
42 .text:00401060 loc_401060:                          ; CODE XREF: .text:0040105Bj
43 .text:00401060                                      ; .text:00401086j
44 .text:00401060              mov     al, byte_403140[ecx]
45 .text:00401066              cmp     al, 61h
46 .text:00401068              jl      short loc_40107E
47 .text:0040106A              cmp     al, 7Ah
48 .text:0040106C              jg      short loc_40107E
49 .text:0040106E
50 .text:0040106E loc_40106E:                          ; DATA XREF: start:loc_4012D5w
```

```
51 .text:0040106E                push    cs
52 .text:0040106F                mov     esi, 5948AC0h
53 .text:00401074
54 .text:00401074 loc_401074:                          ; CODE XREF: .text:loc_401074j
55 .text:00401074                jg      short near ptr loc_401074+1
56 .text:00401074 ; ---------------------------------------------------------------------------
57 .text:00401076                dw 0FFFFh
58 .text:00401078 ; ---------------------------------------------------------------------------
59 .text:00401078                mov     byte_403140[ecx], dl
60 .text:0040107E
61 .text:0040107E loc_40107E:                          ; CODE XREF: .text:00401068j
62 .text:0040107E                                      ; .text:0040106Cj
63 .text:0040107E                inc     ecx
64 .text:0040107F                cmp     byte_403140[ecx], 0
65 .text:00401086                jnz     short loc_401060
66 .text:00401088
67 .text:00401088 loc_401088:                          ; CODE XREF: .text:00401059j
68 .text:00401088                or      eax, 0FFFFFFFFh
69 .text:0040108B                mov     edx, offset byte_403140
70 .text:00401090                test    ecx, ecx
71 .text:00401092                jz      short loc_4010AD
72 .text:00401094
73 .text:00401094 loc_401094:                          ; CODE XREF: .text:004010ABj
74 .text:00401094                movzx   esi, byte ptr [edx]
75 .text:00401097                xor     esi, eax
76 .text:00401099                and     esi, 0FFh
77 .text:0040109F                shr     eax, 8
78 .text:004010A2                xor     eax, ds:dword_402058[esi*4]
79 .text:004010A9                inc     edx
80 .text:004010AA                dec     ecx
81 .text:004010AB                jnz     short loc_401094
82 .text:004010AD
83 .text:004010AD loc_4010AD:                          ; CODE XREF: .text:00401092j
84 .text:004010AD                not     eax
85 .text:004010AF                cmp     eax, 0F891B218h
86 .text:004010B4                jnz     short loc_4010C6
87 .text:004010B6                mov     eax, [ebp+8]
88 .text:004010B9                mov     dword ptr [eax], offset byte_403140
89 .text:004010BF                mov     al, 1
90 .text:004010C1                pop     esi
91 .text:004010C2                mov     esp, ebp
92 .text:004010C4                pop     ebp
93 .text:004010C5                retn
```

The code isn't overly long and looks reasonable. There's one exception though in lines 56 to 58:

```
1 ; ---------------------------------------------------------------------------
2 .text:00401076                dw 0FFFFh
3 .text:00401078 ; ---------------------------------------------------------------------------
```

IDA wasn't able to convert these two bytes to code and displays it as data. So either the snippet has a data section inside the code, or the code is self-modifying. We will come to this later.

The Function Parameter

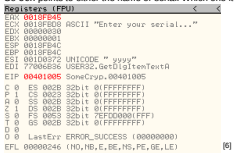
The first nine lines of the function are:

```
1 loc_401000:                           ; CODE XREF: DialogFunc+1CDp
2 .text:00401000                push    ebp
3 .text:00401001                mov     ebp, esp
4 .text:00401003                mov     al, [ecx]
5 .text:00401005                sub     esp, 20h
6 .text:00401008                push    esi
7 .text:00401009                xor     esi, esi
8 .text:0040100B                test    al, al
9 .text:0040100D                jz      loc_4010C6
```

After the standard function prologue we find a reference to register `ecx`. Since the register wasn't set within the subroutine, it must be a function argument. If we go back to the caller we see that ecx holds the address `esp+104h+var_80`. This address is also used as the `lpString` argument to the `GetDlgItemTextA` call (line 5):

```
1 .text:00401281                lea     edx, [esp+104h+var_80]
2 .text:00401288                push    edx             ; lpString
3 .text:00401289                push    3EAh            ; nIDDlgItem
4 .text:0040128E                push    esi             ; hDlg
5 .text:0040128F                call    edi ; GetDlgItemTextA
6 .text:00401291                lea     eax, [esp+100h+lpText]
7 .text:00401295                push    eax
8 .text:00401296                lea     ecx, [esp+104h+var_80]
9 .text:0040129D                call    loc_401000
```

So `ecx` points to either the name or serial. Which one it is can be determined with OllyDbg. Set a breakpoint at `0040128F`, then run the executable and inspect the `ecx` register:



So `ecx` points to the `serial`. The subroutine `loc_401000` then creates a stack frame for local variables in line 5. After that it checks whether the first character of the serial number is 0, i.e., *if* the serial number is an empty string. If the serial is empty, the subroutine jumps to `loc_4010C6` and returns 0 (which means we failed). Let's continue assuming the serial is a non empty string.

Valid Serial Characters

These are the next few lines of the subroutine:

```
10 .text:00401013                lea     edx, [ebp-20h]
11 .text:00401016                sub     edx, ecx
12 .text:00401018
13 .text:00401018 loc_401018:                          ; CODE XREF: .text:00401032j
14 .text:00401018                cmp     al, 61h
15 .text:0040101A                jl      loc_4010C6
16 .text:00401020                cmp     al, 7Ah
17 .text:00401022                jg      loc_4010C6
18 .text:00401028                mov     [edx+ecx], al
19 .text:0040102B                mov     al, [ecx+1]
20 .text:0040102E                inc     ecx
21 .text:0040102F                inc     esi
22 .text:00401030                test    al, al
23 .text:00401032                jnz     short loc_401018
```

The code first loads the address of the beginning of the stack frame to `edx`, and then subtracts `ecx`. After that follow two comparisons of `al` with constants `61h` and `7Ah`. The register `al` was set in line 4 (`mov al, [ecx]`) and holds the first character of the serial. The two constants are the ASCII codes for a and z respectively. So the two checks make sure that the character in `al` is one of the 26 lowercase letters. If not, the code jumps `loc_4010C6` which we already know returns 0 (failure). After the two checks, the code copies the character in `al` to `[edx+ecx]` (line 18). From lines 10 and 11 we know that this memory location is in fact `[ebp-20h]`. Line 19 loads the next character from the serial, and line 20 sets the pointer `ecx` to this character. Line 21 increments `esi` which was set to 0 in line 7. Line 22 is a check to see if the character is the null-byte. If not, the code jumps back to loc_401018 for a next iteration. The snippet implements the loop:

```
1  char serial_copy[32] // in [ebp-20h]
2  esi = 0
3  DO
4      c = serial[esi]
5      IF NOT 'a' <= c <= 'z' THEN
6          RETURN 0 // failure
7      ENDIF
8      serial_copy[esi] = c
9      esi += 1
10 WHILE c != '\0'
```

To summarize: the snippet copies the serial to `[ebp-20h]`. It also makes sure that **the serial only contains lower case characters**.

Valid Serial Length

These lines come next:

```
24 .text:00401034                cmp     esi, 1Ah
25 .text:00401037                jnz     loc_4010C6
```

The first line compares `esi` to `1Ah` = 26. The register `esi` holds the index of the null byte into the serial string, which corresponds to the length of the string. So the snippet **checks if the serial has exactly 26 letters. If it doesn't, the snippet jumps to the well known failure location `loc_4010C6`**:

```
1 IF len(serial) != 26 THEN
2     RETURN 0 // failure
3 ENDIF
```

Copy String *byte_403010* to *byte_403140*

The next lines are easy to decompile:

```
26 .text:0040103D                xor     eax, eax
27 .text:0040103F                nop
28 .text:00401040
29 .text:00401040 loc_401040:                            ; CODE XREF: .text:0040104Fj
30 .text:00401040                mov     cl, byte_403010[eax]
31 .text:00401046                mov     byte_403140[eax], cl
32 .text:0040104C                inc     eax
33 .text:0040104D                test    cl, cl
34 .text:0040104F                jnz     short loc_401040
35 .text:00401051                xor     ecx, ecx
36 .text:00401053                cmp     byte_403140, cl
37 .text:00401059                jz      short loc_401088
```

The snippet simply copies the null-terminated string in `byte_403010` to `byte_403140`. Lines 36 and 37 also check if the first character in byte_403140 is a null byte, i.e., if the string is empty. If it is, then the code jumps to `loc_401088`. Since `byte_403140` is hard coded and is not a null byte, we can assume the jump is not taken. This is the pseudo-code for the snippet:

```
1 STRCPY(byte_403140, byte_403010) // copy string byte_403010 to byte_403140
2 IF byte_403140[0] == '\0' THEN
3     GOTO loc_401088 \\ should never happen
4 ENDIF
```

Check Character(s) in *byte_403140*

This snippet comes next:

```
38 .text:0040105B                jmp     short loc_401060
39 .text:0040105B ; ---------------------------------------------------------------------------
40 .text:0040105D                align 10h
41 .text:00401060
42 .text:00401060 loc_401060:                            ; CODE XREF: .text:0040105Bj
43 .text:00401060                                        ; .text:00401086j
44 .text:00401060                mov     al, byte_403140[ecx]
45 .text:00401066                cmp     al, 61h
46 .text:00401068                jl      short loc_40107E
47 .text:0040106A                cmp     al, 7Ah
48 .text:0040106C                jg      short loc_40107E
49 .text:0040106E
```

Register `ecx` was set to 0 in line 35. So `al` is the first character in `byte_403140`. Lines 45 to 48 check if it is a lower case character and jump to loc_40107E if not:

```
1 IF NOT 'a' <= byte_403140[ecx] <= 'z'  THEN
2     GOTO loc_40107E
3 ENDIF
```

Substitution Cipher (Very Gently Obfuscated)

If the jump is not take, we continue with these very interesting lines:

```
49 .text:0040106E
50 .text:0040106E loc_40106E:                            ; DATA XREF: start:loc_4012D5w
51 .text:0040106E                push    cs
52 .text:0040106F                mov     esi, 5948AC0h
53 .text:00401074
54 .text:00401074 loc_401074:                            ; CODE XREF: .text:loc_401074j
55 .text:00401074                jg      short near ptr loc_401074+1
56 .text:00401074 ; ---------------------------------------------------------------------------
57 .text:00401076                dw 0FFFFh
58 .text:00401078 ; ---------------------------------------------------------------------------
59 .text:00401078                mov     byte_403140[ecx], dl
```

As noted before, the disassembly looks very odd:

- `push cs` doesn't make sense at this point
- The constant in `mov esi, 5948AC0h` looks very arbitrary
- The jump in `jg short near ptr loc_401074+1` has an invalid target
- The data section `dw 0FFFFh` is unexpected

So probably this part is modified during runtime to something more meaningful. A quick way to verify this is to set a breakpoint at `.text:0040106E` in OllyDbg and check how the code looks at runtime. Make sure you enter a valid serial (26 lower case characters), otherwise you won't even arrive at the breakpoint:

```
0040106C  .v7F 10         JG SHORT SoneCryp.0040107E
0040106E  . 0FBEC0        MOVSX EAX,AL
00401071  ? 8094C6 7FFFFF MOV CL,BYTE PTR SS:[EBP+EAX-81]
00401078  . 8891 40314000 MOV BYTE PTR DS:[ECX+403140],DL
0040107E  > 41            INC ECX
0040107F  . 80B9 40314000 CMP BYTE PTR DS:[ECX+403140],0
00401086  .^75 E8         JNZ SHORT SoneCryp.00401060
00401088  > 83C8 FF       OR EAX,FFFFFFFF
0040108B  . BA 40314000   MOV EDX,SoneCryp.00403140
00401090  . 85C9          TEST ECX,ECX
00401092  .v7F 13         JG SHORT SoneCryp.004010A7
00401094  > 0FB632        MOVZX ESI,BYTE PTR DS:[EDX]                    [7]
00401097  . 33F0          XOR ESI,EAX
```

This confirms that the code looks different during runtime. The same location before running the exe looks like this:

```
0040106C  .v7F 10         JG SHORT SoneCryp.0040107E
0040106E  . 0E            PUSH CS
0040106F  . BE C0809405   MOV ESI,5948AC0
00401074  .v7F FF         JG SHORT SoneCryp.00401075
00401076    FF            DB FF
00401077    FF            DB FF
00401078  . 8891 40314000 MOV BYTE PTR DS:[ECX+403140],DL
0040107E  > 41            INC ECX                                        [8]
```

So bytes `0F BE C0` are changed to `0E BE C0` during runtime. Let's set a breakpoint at `0040106E` with *"Memory, on write"* to see which part of the code modifies the code. This is the location were the memory is modified:

```
004012D1  [. 5D           POP EBP
004012D2  . C2 1000       RETN 10
004012D5  > FE05 6E104000 INC BYTE PTR DS:[40106E]
004012DB  .v EB 03        JMP SHORT SoneCryp.004012E0
004012DD  . CC            INT3
004012DE  > EB F5         JMP SHORT SoneCryp.004012D5                    [9]
```

The crackme uses a simple increment instruction to modify one byte inside our subroutine. Newer versions of IDA Pro allow you to patch the code to get the correct code:

```
49 .text:0040106E
50 .text:0040106E loc_40106E:                            ; DATA XREF: start:loc_4012D5w
51 .text:0040106E                movsx   eax, al
52 .text:00401071                mov     dl, ebp[eax-81h]
53 .text:00401078                mov     byte_403140[ecx], dl
```

The memory location `[ebp+eax-81h]` looks strange at first. But we know that at `[ebp-20h]` there is a copy of our serial. So we can rewrite the location as `serial_copy[eax - 61h]`. Furthermore, 61h = 97 is the ASCII code of letter "`a`", which leads to the following pseudo code:

```
1 // al = byte_403140[ecx]
2 dl = serial_copy[eax - 'a']
```

So **dl holds the *n*th character of the serial, where n is 0 for eax = 'a', 1 for 'b', ..., 26 for 'z'**. This also shows why the serial needs to have 26 characters: it serves as the key for a substitution cipher.

Wrapping up the Encryption Loop

The next couple of lines finish the encryption loop. The lines simply increase the index in `ecx` and loop back to `loc_401060` if the next character in `byte_403140` is not the null byte:

```
60 .text:0040107E
61 .text:0040107E loc_40107E:                            ; CODE XREF: .text:00401068j
62 .text:0040107E                                        ; .text:0040106Cj
63 .text:0040107E                inc     ecx
64 .text:0040107F                cmp     byte_403140[ecx], 0
65 .text:00401086                jnz     short loc_401060
```

Here's the entire decryption routine in pseudo code:

```
1 FOR i = 0 TO LEN(byte_403140) DO
2     ch = byte_403140[i]
3     IF 'a' <= ch <= 'z'  THEN
4         byte_403140[i] = serial_copy[ch - 'a']
5     ENDIF
6 END FOR
```

All characters in byte_403140 that are not lowercase letters are left alone. All lower case letters are replaced with the character from the serial at index *n*, where *n* is zero if the character is 'a', is 1 if the character is 'b', etc.

Hashing the Plaintext Message

The next lines look quite complicated:

```
67 00401088 loc_401088:                            ; CODE XREF: .text:00401059j
68 .text:00401088                or      eax, 0FFFFFFFFh
69 .text:0040108B                mov     edx, offset byte_403140
70 .text:00401090                test    ecx, ecx
71 .text:00401092                jz      short loc_4010AD
72 .text:00401094
73 .text:00401094 loc_401094:                            ; CODE XREF: .text:004010ABj
74 .text:00401094                movzx   esi, byte ptr [edx]
75 .text:00401097                xor     esi, eax
76 .text:00401099                and     esi, 0FFh
77 .text:0040109F                shr     eax, 8
78 .text:004010A2                xor     eax, ds:dword_402058[esi*4]
79 .text:004010A9                inc     edx
80 .text:004010AA                dec     ecx
81 .text:004010AB                jnz     short loc_401094
82 .text:004010AD
83 .text:004010AD loc_4010AD:                            ; CODE XREF: .text:00401092j
84 .text:004010AD                not     eax
85 .text:004010AF                cmp     eax, 0F891B218h
```

```
.text:004010B4          jnz     short loc_4010C6
.text:004010B6          mov     eax, [ebp+8]
.text:004010B9          mov     dword ptr [eax], offset byte_403140
.text:004010BF          mov     al, 1
.text:004010C1          pop     esi
.text:004010C2          mov     esp, ebp
.text:004010C4          pop     ebp
.text:004010C5          retn
```

What the lines basically do is to calculate a hash of the string in `byte_403140` (the plaintext message). If the hash is equal to `0F891B218h` (see line 85), then we get the success message, otherwise we get the failure code at `loc_4010C6`. We can only assume that the hash `0F891B218h` is produced when the decryption leads to the correct plaintext.

Finding the Plaintext Message

It's time to have a look at the string in byte_403010. The string is hardcoded and has the following value:

> lx lzctusdzetgc, ex n-fsb (nvfnujuvujsx-fsb) jn e fenjl lsatsxrxu sw ncaaruzjl qrc ehdszjugan pgjlg trzwszan nvfnujuvujsx. lx fhslq ljtgrzn, ugrc ezr uctjlehhc vnrm us sfnlvzr ugr zrheujsxngjt fruprrx ugr qrc exm ugr ljtgrzurbu.

We know that all lowercase character in this strings stem from a simple substitution cipher. Entering the correct key as the *serial* should produce a meaningful plaintext and hopefully lead to the success message. Breaking substitution ciphers is pretty easy. I'm using the Python script `break_simplesub` from the Practical Cryptography webpage [10]. The code uses a random optimization algorithm and the output varies each time you run the code. Here's my output:

```
$ python break_simplesub.py
Substitution Cipher solver, you may have to wait several iterations
for the correct result. Press ctrl+c to exit program.

best score so far: -1001.47696195 on iteration 1
    best key: RATQNHWGEDMLCZJBKFXUSIPOVY
    plaintext:
VSLNMCTUJNICHMISERUPEYRETOTYTOUSRUPOEIRIEOLLUBCUSASTUGEMBBATNOLDAMIFJUNOTHBEWHOLHCANGUNBEEYRETOTYTOUSVSRFULDLOCHANETHAMINATMCOLIFFMYEAKTUURELYNATHANAFITOUSEHOCRATWAASTHADAMISKTHALOCHANT

best score so far: -996.55579631 on iteration 3
    best key: UAESNQPXZIBFDJLVYGCRTWMKHO
    plaintext:
JHOISUADMICURSCHELDKEPLEANAPANDHLDKNECLCENOODBUDHTHADVESBBTAINOFTSCYMDINARBEGRNORUTIVDIBEEPLEANAPANDHJHLYDOFONURTIEARTSCITASUNOCYYSPETWADDLEOPITARTITYCANDHERNULTAGTTHARTFTSCHWARTONURTIATK

best score so far: -827.410210054 on iteration 9
    best key: EFLMRWDGJYQHAXSTOZNUVIPBCK
    plaintext:
VNCRYPTOGRAPHYANSBOXSUBSTITUTIONBOXISABASICCOMPONENTOFSYMMETRICKEYALGORITHMSWHICHPERFORMSSUBSTITUTIONVNBLOCKCIPHERSTHEYARETYPICALLYUSEDTOOBSCURETHERELATIONSHIPBETWEENTHEKEYANDTHECIPHERTEXT
```

The plaintext is not very readable because the script does not tackle special characters like spaces. However, you can clearly recognize meaningful text. The key to encrypt the message therefore **EFLMRWDGJYQHAXSTOZNUVIPBCK**. To decrypt it, we must enter the reverse of the key as the serial. The following Python script generates the decryption key, and also shows the resulting plaintext:

```
import string

crypt = """lx lzctusdzetgc, ex n-fsb (nvfnujuvujsx-fsb) jn e fenjl lsatsxrxu sw
ncaaruzjl qrc ehdszjugan pgjlg trzwszan nvfnujuvujsx. lx fhslq ljtgrzn, ugrc
ezr uctjlehhc vnrm us sfnlvzr ugr zrheujsxngjt fruprrx ugr qrc exm ugr
ljtgrzurbu.""".replace('\n', '')

key = 'EFLMRWDGJIQHAXSTOZNUVYPBCK'.lower()
mapping = {}
for k, c in zip(key, string.lowercase):
    mapping[k] = c

msg = ""
for c in crypt:
    msg += mapping.get(c, c)

print("the key is: {}".format(''.join([mapping[x] for x in string.lowercase])))
print("the plaintext is: {}".format(msg))
```

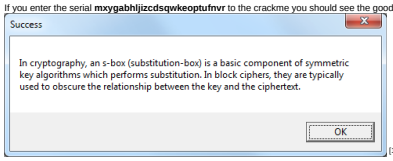
It produces the following output:

```
$ python decrypt.py
the key is: mxygabhljizcdsqwkeoptufnvr
the plaintext is: in cryptography, an s-box (substitution-box) is a basic component of symmetric key algorithms which performs substitution. In block ciphers, they are typically used to obscure the relationship between the key and the ciphertext.
```

The message therefore is

> In cryptography, an s-box (substitution-box) is a basic component of symmetric key algorithms which performs substitution. In block ciphers, they are typically used to obscure the relationship between the key and the ciphertext.

If you enter the serial **mxygabhljizcdsqwkeoptufnvr** to the crackme you should see the good boy message:



[11]

URLs in this post:

[1] www.crackmes.de: **http://www.crackmes.de**
[2] here: **http://www.crackmes.de/users/san01suke/somecrypto01/**
[3] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/SomeCrypto-01-screenshot.png**
[4] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/ida_strings.png**
[5] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/ida_success_declaration.png**
[6] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/s1_1.png**
[7] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/self_modifying_part.png**
[8] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/self_modifying_2.png**
[9] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/self_modifying_3.png**
[10] Practical Cryptography webpage: **http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-simple-substitution-cipher/**
[11] Image: **http://www.johannesbader.ch/wp-content/uploads/2014/07/success_message.png**