Crackmes.de – san01suke's SomeCrypto~02

Posted By Johannes On July 15, 2014 @ 18:33 In information security | No Comments

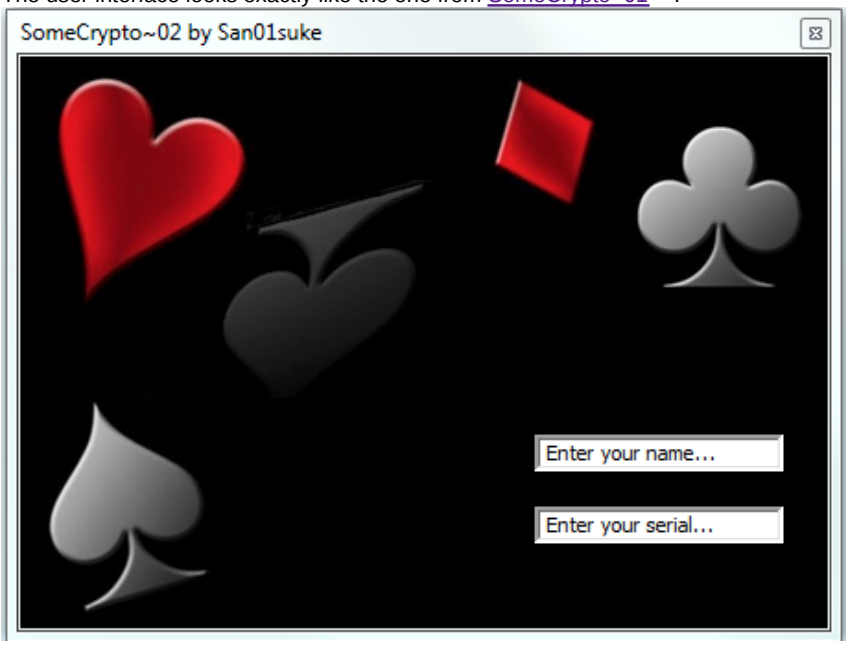
The author *san01suke* submitted three crackmes to www.crackmes.de [1] on July 1st. This is my attempt to solve the second one, called **SomeCrypto~02**. You can view and download the crackme here [2]. The short description simply says:

> "Just write a valid keygen for this crackme."
>
> − san01suke

Reverse Engineering the Crackme

Getting the Disassembly

The user interface looks exactly like the one from SomeCrypto~01 [3]:



[4]

Let's open the code in OllyDbg. At the entry point we get the following picture:

```
004014B7    20              DB 20                   CHAR ' '
004014B8    4A              DB 4A                   CHAR 'J'
004014B9    20              DB 20                   CHAR ' '
004014BA    48              DB 48                   CHAR 'H'
004014BB    80              DB 80
004014BC    32              DB 32                   CHAR '2'
004014BD    60              DB 60                   CHAR '`'
004014BE    20              DB 20                   CHAR ' '
004014BF    4A              DB 4A                   CHAR 'J'
004014C0    20              DB 20                   CHAR ' '
004014C1    4A              DB 4A                   CHAR 'J'
004014C2    45              DB 45                   CHAR 'E'
004014C3    4A              DB 4A                   CHAR 'J'
004014C4    20              DB 20                   CHAR ' '
004014C5    83              DB 83
004014C6    4C              DB 4C                   CHAR 'L'
004014C7    12              DB 12
004014C8    60              DB 60                   CHAR '`'
004014C9    20              DB 20                   CHAR ' '
004014CA    DF              DB DF
004014CB  . 35 64 00        ASCII "5d",0
004014CE    60              DB 60                   CHAR '`'
004014CF    20              DB 20                   CHAR ' '
004014D0    13              DB 13
004014D1    E0              DB E0
004014D2    E3              DB E3
004014D3  $ B0 20           MOV AL,20
004014D5  . B9 D3040000     MOV ECX,4D3
004014DA  . BB 00104000     MOV EBX,SomeCryp.00401000
004014DF  > 3E:8A13         MOV DL,BYTE PTR DS:[EBX]
004014E2  . 32D0            XOR DL,AL
004014E4  . 3E:8813         MOV BYTE PTR DS:[EBX],DL
004014E7  . 43              INC EBX
004014E8  .^E2 F5           LOOPD SHORT SomeCryp.004014DF
004014EA  . B8 B0144000     MOV EAX,SomeCryp.004014B0
004014EF  . FFD0            CALL EAX
004014F1  . 0000            ADD BYTE PTR DS:[EAX],AL
004014F3  . 0000            ADD BYTE PTR DS:[EAX],AL
```

[5]

One can spot two interesting things:

- The code above 4014D3 could not be disassembled and looks obfuscated
- The code that follows the entry point clearly modifies the code. Starting at location `401000`, the snippet XORs exactly `4D3h` bytes with 20h.

Let's run the code up to `4014DA` to get the changed code section. The bytes clearly changed

```
004014B7    00              DB 00
004014B8    6A              DB 6A                   CHAR 'j'
004014B9    00              DB 00
004014BA    68              DB 68                   CHAR 'h'
004014BB    A0              DB A0
004014BC    12              DB 12
004014BD    40              DB 40                   CHAR '@'
004014BE    00              DB 00
004014BF    6A              DB 6A                   CHAR 'j'
004014C0    00              DB 00
004014C1    6A              DB 6A                   CHAR 'j'
004014C2    65              DB 65                   CHAR 'e'
004014C3    6A              DB 6A                   CHAR 'j'
004014C4    00              DB 00
004014C5    A3              DB A3
004014C6    6C              DB 6C                   CHAR 'l'
004014C7    32              DB 32                   CHAR '2'
004014C8    40              DB 40                   CHAR '@'
004014C9    00              DB 00
004014CA    FF              DB FF
004014CB  . 15 44 20        ASCII "$D "
004014CE    40              DB 40                   CHAR '@'
004014CF    00              DB 00
004014D0    33              DB 33                   CHAR '3'
004014D1    C0              DB C0
004014D2    C3              DB C3
004014D3  $ B0 20           MOV AL,20
004014D5  . B9 D3040000     MOV ECX,4D3
004014DA  . BB 00104000     MOV EBX,SomeCryp.00401000
004014DF  > 3E:8A13         MOV DL,BYTE PTR DS:[EBX]
004014E2  . 32D0            XOR DL,AL
004014E4  . 3E:8813         MOV BYTE PTR DS:[EBX],DL
004014E7  . 43              INC EBX
004014E8  .^E2 F5           LOOPD SHORT SomeCryp.004014DF
004014EA  . B8 B0144000     MOV EAX,SomeCryp.004014B0
004014EF  . FFD0            CALL EAX                SomeCryp.004014B0
004014F1  . 0000            ADD BYTE PTR DS:[EAX],AL
```

[6]

but OllyDbg doesn't automatically reanalyze the section and still displays the section as data bytes. Hit `CTRL+A` to analyze the code again and now you should see meaningful disassembly:

```
004014A7  .  8BE5          MOV ESP,EBP
004014A9  .  5D            POP EBP
004014AA  .  C2 1000       RETN 10
004014AD     CC            INT3
004014AE     CC            INT3
004014AF     CC            INT3
004014B0  .  6A 00         PUSH 0                              pModule = NULL
004014B2  .  FF15 18204000 CALL DWORD PTR DS:[<&KERNEL32.GetModule  GetModuleHandleA
004014B8  .  6A 00         PUSH 0                              lParam = NULL
004014BA  .  68 A0124000   PUSH SomeCryp.004012A0              DlgProc = SomeCryp.004012A0
004014BF  .  6A 00         PUSH 0                              hOwner = NULL
004014C1  .  6A 65         PUSH 65                             pTemplate = 65
004014C3  .  6A 00         PUSH 0                              hInst = NULL
004014C5  .  A3 6C324000   MOV DWORD PTR DS:[40326C],EAX
004014CA  .  FF15 44204000 CALL DWORD PTR DS:[<&USER32.DialogBoxPa  DialogBoxParamA
004014D0  .  33C0          XOR EAX,EAX
004014D2  .  C3            RETN
004014D3  $  B0 20         MOV AL,20
004014D5  .  B9 D3040000   MOV ECX,4D3
004014DA  .  BB 00104000   MOV EBX,SomeCryp.00401000           Entry address
004014DF  >  3E:8A13       MOV DL,BYTE PTR DS:[EBX]
004014E2  .  32D0          XOR DL,AL
004014E4  .  3E:8813       MOV BYTE PTR DS:[EBX],DL
004014E7  .  43            INC EBX
004014E8  .^E2 F5          LOOPD SHORT SomeCryp.004014DF
004014EA  .  B8 B0144000   MOV EAX,SomeCryp.004014B0
004014EF  .  FFD0          CALL EAX                            SomeCryp.004014B0
```
[7].

I then created a patched version of *SomeCrypto~02* with the bytes from 401000 to 4014D2 XORed with 20h. This allowed me to get the correct disassembly in IDA Pro.

sub_4011E0 – Part 1

I then opened the patched executable in IDA Pro and went to the strings subview to follow the *"Success"* string (like in SomeCrypto~01 [3]). The good boy message is produced here:

```
1  .text:00401475          call    sub_4011E0
2  .text:0040147A          add     esp, 8
3  .text:0040147D          mov     byte_403270, al
4  .text:00401482          test    al, al
5  .text:00401484          jz      short loc_4014A2
6  .text:00401486          mov     edx, [esp+0Ch]
7  .text:0040148A          push    0
8  .text:0040148C          push    offset aSuccess ; "Success"
9  .text:00401491          push    edx
10 .text:00401492          push    esi
11 .text:00401493          call    ds:MessageBoxA
12 .text:00401499          push    0
```

We will see the "Success"-message when `sub_4011E0` returns a non zero value in `eax`. Let's walk through subroutine `sub_4011E0`

```
1  .text:004011E0 ; =============== S U B R O U T I N E
   =======================================
2  .text:004011E0
3  .text:004011E0 ; Attributes: bp-based frame
4  .text:004011E0
5  .text:004011E0 sub_4011E0      proc near               ; CODE XREF: .text:00401475p
6  .text:004011E0
7  .text:004011E0 var_1C          = byte ptr -1Ch
8  .text:004011E0 arg_0           = dword ptr  8
9  .text:004011E0 arg_4           = dword ptr  0Ch
10 .text:004011E0
11 .text:004011E0                 push    ebp
12 .text:004011E1                 mov     ebp, esp
13 .text:004011E3                 xor     eax, eax
14 .text:004011E5                 sub     esp, 1Ch
15 .text:004011E8                 cmp     [edi], al
16 .text:004011EA                 jz      loc_401296
17 .text:004011F0
```

In OllyDbg we see that `edi` contains the serial:

```
Registers (FPU)
EAX 0018FB64
ECX 0018FBD8 ASCII "Enter your name..."
EDX 00000030
EBX 00000001
ESP 0018FB4C
EBP 0018FC58
ESI 00060452
EDI 0018FC18 ASCII "1234567"
EIP 004011E0 SomeCryp.004011E0
```
[8]

The snippet translate to the following pseudo code:

```
1 serial = edi
2 IF strlen(serial) == 0 THEN
3     RETURN 0 // loc_401296
4 END
```

This is the code at `loc_401296` that returns 0:

```
86 .text:00401296 loc_401296:                     ; CODE XREF: sub_4011E0+Aj
87 .text:00401296                                 ; sub_4011E0+1Aj ...
88 .text:00401296                 xor     al, al
89 .text:00401298                 mov     esp, ebp
90 .text:0040129A                 pop     ebp
91 .text:0040129B                 retn
92 .text:0040129B sub_4011E0      endp
```

If the serial is not empty the next section is executed:

```
18 .text:004011F0 loc_4011F0:                     ; CODE XREF: sub_4011E0+15j
19 .text:004011F0                 inc     eax
20 .text:004011F1                 cmp     byte ptr [eax+edi], 0
21 .text:004011F5                 jnz     short loc_4011F0
22 .text:004011F7                 cmp     eax, 7
23 .text:004011FA                 jnz     loc_401296
```

The lines calculate the length of the serial and return 0 if the length is not 7:

```
1 serial_length = strlen(serial)
2 IF serial_length != 7 THEN
3     RETURN 0 // loc_401296
4 END
```

Next follows a call to another subroutine:

```
24 .text:00401200                 mov     edx, [ebp+arg_0]
25 .text:00401203                 lea     eax, [ebp+var_1C]
26 .text:00401206                 call    sub_401000
```

`eax` is a local variable. The memory location `ebp+arg_0 = ebp+8` points to the content of the name input box, as can be seen in OllyDbg:

```
$-8     0018FC58
$-4     00401461 RETURN to SomeCryp.00401461
$ ==>   0018FC58
$+4     0040147A RETURN to SomeCryp.0040147A from SomeCryp.004011E0
$+8     0018FBD8 ASCII "Enter your name..."
$+C     0018FB64
```
[9]

So the call boils down to:

```
1 unknown_type var_1C;
2 sub_401000(&var_1C, name)
```

sub_401000

The subroutine `sub_401000` looks as follows:

```
1  .text:00401000 ; =============== S U B R O U T I N E
2  =======================================
3  .text:00401000
4  .text:00401000
```

```
.text:00401000 sub_401000      proc near               ; CODE XREF: sub_4011E0+26p
.text:00401000                                         ; DATA XREF: .text:004014DAo
.text:00401000                 mov     dword ptr [eax], 0
.text:00401006                 mov     dword ptr [eax+4], 1
.text:0040100D                 mov     dword ptr [eax+8], 2
.text:00401014                 mov     dword ptr [eax+0Ch], 3
.text:0040101B                 mov     dword ptr [eax+10h], 4
.text:00401022                 mov     dword ptr [eax+14h], 5
.text:00401029                 mov     dword ptr [eax+18h], 6
.text:00401030                 mov     cl, [edx]
.text:00401032                 test    cl, cl
.text:00401034                 jz      short locret_40108C
.text:00401036                 push    esi
.text:00401037                 jmp     short loc_401040
.text:00401037 ; ---------------------------------------------------------------------------
.text:00401039                 align 10h
.text:00401040
.text:00401040 loc_401040:                             ; CODE XREF: sub_401000+37j
.text:00401040                                         ; sub_401000+89j
.text:00401040                 movsx   ecx, cl
.text:00401043                 and     ecx, 80000001h
.text:00401049                 jns     short loc_401050
.text:0040104B                 dec     ecx
.text:0040104C                 or      ecx, 0FFFFFFFEh
.text:0040104F                 inc     ecx
.text:00401050
.text:00401050 loc_401050:                             ; CODE XREF: sub_401000+49j
.text:00401050                 jz      short loc_40105C
.text:00401052                 mov     esi, [eax+4]
.text:00401055                 mov     ecx, [eax]
.text:00401057                 mov     [eax], esi
.text:00401059                 mov     [eax+4], ecx
.text:0040105C
.text:0040105C loc_40105C:                             ; CODE XREF: sub_401000:loc_401050j
.text:0040105C                 mov     ecx, [eax]
.text:0040105E                 mov     esi, [eax+4]
.text:00401061                 mov     [eax], esi
.text:00401063                 mov     esi, [eax+8]
.text:00401066                 mov     [eax+4], esi
.text:00401069                 mov     esi, [eax+0Ch]
.text:0040106C                 mov     [eax+8], esi
.text:0040106F                 mov     esi, [eax+10h]
.text:00401072                 mov     [eax+0Ch], esi
.text:00401075                 mov     esi, [eax+14h]
.text:00401078                 mov     [eax+10h], esi
.text:0040107B                 mov     esi, [eax+18h]
.text:0040107E                 mov     [eax+14h], esi
.text:00401081                 inc     edx
.text:00401082                 mov     [eax+18h], ecx
.text:00401085                 mov     cl, [edx]
.text:00401087                 test    cl, cl
.text:00401089                 jnz     short loc_401040
.text:0040108B                 pop     esi
.text:0040108C
.text:0040108C locret_40108C:                          ; CODE XREF: sub_401000+34j
.text:0040108C                 retn
.text:0040108C sub_401000      endp
```

The code translates to the following pseudo code:

```
1 FUNCTION sub_401000(mapping<var_1C>, name)
2    mapping = {0,1,2,3,4,5,6} // in eax = &var_1C
3    FOR character IN name DO
4       IF character % 2 != 0 DO
5          swap(mapping[0], mapping[1])
6       ENDIF
7       circular_left_shift(mapping)
```

```
8     ENDFOR
9 END
```

`swap(mapping[0], mapping[1])` means `{0,1,2,3,4,5,6}` would become `{1,0,2,3,4,5,6}`.
`circular_left_shift(mapping)` means, `{0,1,2,3,4,5,6}` becomes `{1,2,3,4,5,6,0}`.

sub_4011E0 – Part 2

Let's go back to the caller where we continue with line 27:

```
27 .text:0040120B              xor     eax, eax
28 .text:0040120D              lea     ecx, [ecx+0]
29 .text:00401210
30 .text:00401210 loc_401210:                          ; CODE XREF: sub_4011E0+3Fj
31 .text:00401210              mov     cl, byte_403010[eax]
32 .text:00401216              mov     byte_403140[eax], cl
33 .text:0040121C              inc     eax
34 .text:0040121D              test    cl, cl
35 .text:0040121F              jnz     short loc_401210
36 .text:00401221              push    esi
37 .text:00401222              xor     esi, esi
38 .text:00401224              cmp     byte_403140, 0
39 .text:0040122B              jz      short loc_40123A
```

These line simply copy the hard coded null-terminated string at `byte_403010` to `byte_403140` and check if the destination address is not null:

```
1 STRCPY(byte_403140, byte_403010) // copy string byte_403010 to byte_403140
2 IF byte_403140 == NULL THEN
3     GOTO loc_40123A \\ should never happen
4 ENDIF
```

Next up is another small loop:

```
40 .text:0040122D              lea     ecx, [ecx+0]
41 .text:00401230
42 .text:00401230 loc_401230:                          ; CODE XREF: sub_4011E0+58j
43 .text:00401230              inc     esi
44 .text:00401231              cmp     byte_403140[esi], 0
45 .text:00401238              jnz     short loc_401230
46 .text:0040123A
```

The code searches the null-byte in string `byte_403140`. When the null byte is found, then the index in `esi` corresponds to the length of the string at `byte_403140`:

```
1 esi = strlen(byte_403140)
```

Another subroutine call follows:

```
47 .text:0040123A loc_40123A:                          ; CODE XREF: sub_4011E0+4Bj
48 .text:0040123A              push    esi
49 .text:0040123B              lea     eax, [ebp+var_1C]
50 .text:0040123E              call    sub_401110
```

The only parameter is `var_1C`, which we know contains the scrambled sequence of numbers 0 to 7 that `sub_401000` generated: sub_401110(mapping)

sub_401110

The subroutine is a little hard to read, because it uses a few local variables to shuffle characters. Here's the disassembly:

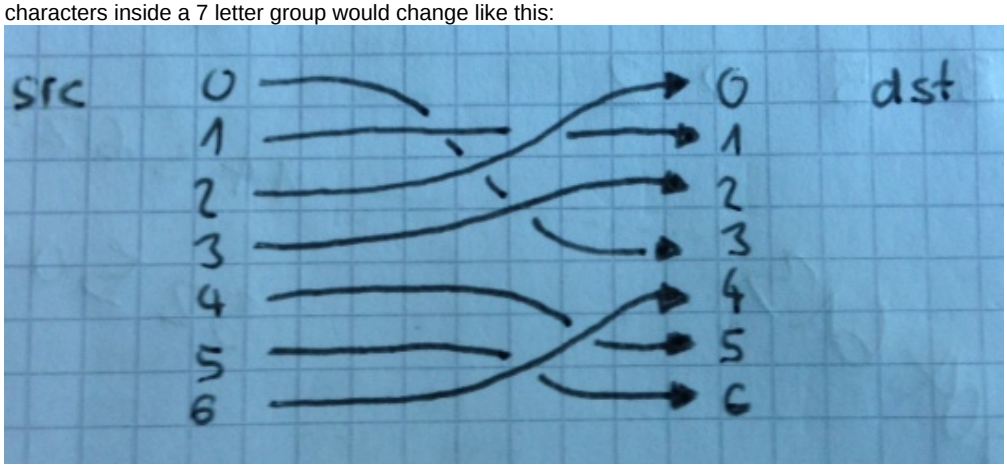
```
1  .text:00401110 sub_401110      proc near               ; CODE XREF: sub_4011E0+5Ep
2  .text:00401110                                         ; sub_4011E0+71p
3  .text:00401110
4  .text:00401110 var_1C          = dword ptr -1Ch
5  .text:00401110 var_18          = word ptr -18h
6  .text:00401110 var_16          = byte ptr -16h
7  .text:00401110 var_14          = dword ptr -14h
8  .text:00401110 var_10          = dword ptr -10h
9  .text:00401110 var_C           = dword ptr -0Ch
10 .text:00401110 var_8           = dword ptr -8
11 .text:00401110 var_4           = dword ptr -4
12 .text:00401110 arg_0           = dword ptr  8
13 .text:00401110
14 .text:00401110                 push    ebp
15 .text:00401111                 mov     ebp, esp
16 .text:00401113                 mov     ecx, 7
17 .text:00401118                 sub     esp, 1Ch
18 .text:0040111B                 cmp     [ebp+arg_0], ecx
19 .text:0040111E                 jle     loc_4011CB
20 .text:00401124                 mov     edx, [eax+8]
21 .text:00401127                 lea     edx, [ebp+edx+var_1C]
22 .text:0040112B                 mov     [ebp+var_4], edx
23 .text:0040112E                 mov     edx, [eax+0Ch]
24 .text:00401131                 lea     edx, [ebp+edx+var_1C]
25 .text:00401135                 mov     [ebp+var_8], edx
26 .text:00401138                 mov     edx, [eax+10h]
27 .text:0040113B                 lea     edx, [ebp+edx+var_1C]
28 .text:0040113F                 push    ebx
29 .text:00401140                 push    esi
30 .text:00401141                 mov     esi, [eax]
31 .text:00401143                 mov     [ebp+var_C], edx
32 .text:00401146                 mov     edx, [eax+14h]
33 .text:00401149                 push    edi
34 .text:0040114A                 mov     edi, [eax+4]
35 .text:0040114D                 mov     eax, [eax+18h]
36 .text:00401150                 lea     edx, [ebp+edx+var_1C]
37 .text:00401154                 mov     [ebp+var_10], edx
38 .text:00401157                 lea     edx, [ebp+eax+var_1C]
39 .text:0040115B                 mov     eax, offset unk_403142
40 .text:00401160                 lea     esi, [ebp+esi+var_1C]
41 .text:00401164                 lea     edi, [ebp+edi+var_1C]
42 .text:00401168                 mov     [ebp+var_14], edx
43 .text:0040116B                 sub     ecx, eax
44 .text:0040116D                 lea     ecx, [ecx+0]
45 .text:00401170
46 .text:00401170 loc_401170:                             ; CODE XREF: sub_401110+B6j
47 .text:00401170                 movzx   edx, byte ptr [eax-2]
48 .text:00401174                 mov     ebx, [ebp+var_4]
49 .text:00401177                 mov     [esi], dl
50 .text:00401179                 movzx   edx, byte ptr [eax-1]
51 .text:0040117D                 mov     [edi], dl
52 .text:0040117F                 movzx   edx, byte ptr [eax]
53 .text:00401182                 mov     [ebx], dl
54 .text:00401184                 movzx   edx, byte ptr [eax+1]
55 .text:00401188                 mov     ebx, [ebp+var_8]
56 .text:0040118B                 mov     [ebx], dl
57 .text:0040118D                 movzx   edx, byte ptr [eax+2]
58 .text:00401191                 mov     ebx, [ebp+var_C]
59 .text:00401194                 mov     [ebx], dl
60 .text:00401196                 movzx   edx, byte ptr [eax+3]
61 .text:0040119A                 mov     ebx, [ebp+var_10]
62 .text:0040119D                 mov     [ebx], dl
63 .text:0040119F                 movzx   edx, byte ptr [eax+4]
64 .text:004011A3                 mov     ebx, [ebp+var_14]
65 .text:004011A6                 mov     [ebx], dl
66 .text:004011A8                 mov     edx, [ebp+var_1C]
67 .text:004011AB                 mov     [eax-2], edx
68 .text:004011AE                 mov     dx, [ebp+var_18]
```

```
69 .text:004011B2                mov     [eax+2], dx
70 .text:004011B6                movzx   edx, [ebp+var_16]
71 .text:004011BA                mov     [eax+4], dl
72 .text:004011BD                add     eax, 7
73 .text:004011C0                lea     edx, [ecx+eax]
74 .text:004011C3                cmp     edx, [ebp+arg_0]
75 .text:004011C6                jl      short loc_401170
76 .text:004011C8                pop     edi
77 .text:004011C9                pop     esi
78 .text:004011CA                pop     ebx
79 .text:004011CB
80 .text:004011CB loc_4011CB:                          ; CODE XREF: sub_401110+Ej
81 .text:004011CB                mov     esp, ebp
82 .text:004011CD                pop     ebp
83 .text:004011CE                retn    4
84 .text:004011CE sub_401110     endp
```

All the code does is shuffle the characters in the string `byte_403140` based on the mapping that `sub_401000` generated:

```
1   FUNCTION sub_401110(mapping)
2      i = 0
3      message = byte_403140
4      WHILE i+7 < strlen(mapping) DO
5         tmp[7]
6         FOR j=0 TO 6 DO:
7            tmp[j] = message[i + mapping[j]]
8         ENDFOR
9         FOR j=0 TO 6 DO:
10           message[j] = tmp[j]
11        ENDFOR
12     END
13 END
```

The subroutine applies a permutation to the message in `byte_403140`. It permutates blocks of 7 characters based on the mapping that the subroutine `sub_401000` generated. Let's say the mapping is {3, 1, 0, 2, 5, 6, 4}, then the characters inside a 7 letter group would change like this:



[10]

sub_4011E0 – Part 3

Back at caller we see yet another subroutine:

```
51 .text:00401243                mov     ecx, edi
52 .text:00401245                lea     eax, [ebp+var_1C]
53 .text:00401248                call    sub_401090
```

The subroutine operates on `[ebp+var_1C]` which we know contains the `mapping`, and on `ecx = edi` which holds

the text from the serial input box:

1 sub_401090(mapping, serial)

sub_401090

This is `sub_401090`:

```
1   .text:00401090 ; =============== S U B R O U T I N E
2   =======================================
3   .text:00401090
4   .text:00401090
5   .text:00401090 sub_401090      proc near               ; CODE XREF: sub_4011E0+68p
6   .text:00401090                 movsx   edx, byte ptr [ecx]
7   .text:00401093                 add     edx, 0FFFFFFD0h
8   .text:00401096                 push    esi
9   .text:00401097                 xor     esi, esi
10  .text:00401099                 mov     [eax], edx
11  .text:0040109B                 cmp     edx, 7
12  .text:0040109E                 jb      short loc_4010A2
13  .text:004010A0                 mov     [eax], esi
14  .text:004010A2
15  .text:004010A2 loc_4010A2:                             ; CODE XREF: sub_401090+Ej
16  .text:004010A2                 movsx   edx, byte ptr [ecx+1]
17  .text:004010A6                 add     edx, 0FFFFFFD0h
18  .text:004010A9                 mov     [eax+4], edx
19  .text:004010AC                 cmp     edx, 7
20  .text:004010AF                 jb      short loc_4010B4
21  .text:004010B1                 mov     [eax+4], esi
22  .text:004010B4
23  .text:004010B4 loc_4010B4:                             ; CODE XREF: sub_401090+1Fj
24  .text:004010B4                 movsx   edx, byte ptr [ecx+2]
25  .text:004010B8                 add     edx, 0FFFFFFD0h
26  .text:004010BB                 mov     [eax+8], edx
27  .text:004010BE                 cmp     edx, 7
28  .text:004010C1                 jb      short loc_4010C6
29  .text:004010C3                 mov     [eax+8], esi
30  .text:004010C6
31  .text:004010C6 loc_4010C6:                             ; CODE XREF: sub_401090+31j
32  .text:004010C6                 movsx   edx, byte ptr [ecx+3]
33  .text:004010CA                 add     edx, 0FFFFFFD0h
34  .text:004010CD                 mov     [eax+0Ch], edx
35  .text:004010D0                 cmp     edx, 7
36  .text:004010D3                 jb      short loc_4010D8
37  .text:004010D5                 mov     [eax+0Ch], esi
38  .text:004010D8
39  .text:004010D8 loc_4010D8:                             ; CODE XREF: sub_401090+43j
40  .text:004010D8                 movsx   edx, byte ptr [ecx+4]
41  .text:004010DC                 add     edx, 0FFFFFFD0h
42  .text:004010DF                 mov     [eax+10h], edx
43  .text:004010E2                 cmp     edx, 7
44  .text:004010E5                 jb      short loc_4010EA
45  .text:004010E7                 mov     [eax+10h], esi
46  .text:004010EA
47  .text:004010EA loc_4010EA:                             ; CODE XREF: sub_401090+55j
48  .text:004010EA                 movsx   edx, byte ptr [ecx+5]
49  .text:004010EE                 add     edx, 0FFFFFFD0h
50  .text:004010F1                 mov     [eax+14h], edx
51  .text:004010F4                 cmp     edx, 7
52  .text:004010F7                 jb      short loc_4010FC
53  .text:004010F9                 mov     [eax+14h], esi
54  .text:004010FC
55  .text:004010FC loc_4010FC:                             ; CODE XREF: sub_401090+67j
56  .text:004010FC                 movsx   ecx, byte ptr [ecx+6]
57  .text:00401100                 add     ecx, 0FFFFFFD0h
58  .text:00401103                 mov     [eax+18h], ecx
59  .text:00401106                 cmp     ecx, 7
```

```
.text:00401109                jb      short loc_40110E
.text:0040110B                mov     [eax+18h], esi
.text:0040110E
.text:0040110E loc_40110E:                            ; CODE XREF: sub_401090+79j
.text:0040110E                pop     esi
.text:0040110F                retn
.text:0040110F sub_401090     endp
```

The code is quite long because the assembler did loop unwinding. The underlying algorithm is very simple though:

```
FUNCTION sub_401090(mapping, serial)
    FOR i = 0 TO 6 DO
        nr = serial[i] - '0'
        IF nr >= 7 THEN
            mapping[i] = 0
        ELSE
            mapping[i] = nr
        ENDIF
    ENDFOR
END
```

So the serial number (7 letters) is converted to 7 integers that are stored in `mapping` (as long as numbers are smaller than 7):

```
FUNCTION sub_401090(mapping, serial)
    FOR i = 0 TO 6 DO
        nr = serial[i] - '0'
        IF nr >= 7 THEN
            mapping[i] = 0
        ELSE
            mapping[i] = nr
        ENDIF
    ENDFOR
END
```

sub_4011E0 – Part 4

After `sub_401090` loaded the serial into the mapping we find a second call to the permutation routine `sub_401110`

```
.text:0040124D                push    esi
.text:0040124E                lea     eax, [ebp+var_1C]
.text:00401251                call    sub_401110
```

The routine finishes up with some code that calculates a hash of `byte_403140`. If the hash matches `0B45D7873h` we get the success message:

```
.text:00401256                or      eax, 0FFFFFFFFh
.text:00401259                mov     ecx, esi
.text:0040125B                mov     edx, offset byte_403140
.text:00401260                test    esi, esi
.text:00401262                jz      short loc_40127D
.text:00401264
.text:00401264 loc_401264:                            ; CODE XREF: sub_4011E0+9Bj
.text:00401264                movzx   esi, byte ptr [edx]
.text:00401267                xor     esi, eax
.text:00401269                and     esi, 0FFh
.text:0040126F                shr     eax, 8
.text:00401272                xor     eax, ds:dword_402058[esi*4]
.text:00401279                inc     edx
.text:0040127A                dec     ecx
.text:0040127B                jnz     short loc_401264
.text:0040127D
.text:0040127D loc_40127D:                            ; CODE XREF: sub_4011E0+82j
.text:0040127D                not     eax
```

```
.text:0040127F                pop     esi
.text:00401280                cmp     eax, 0B45D7873h
.text:00401285                jnz     short loc_401296
.text:00401287                mov     eax, [ebp+arg_4]
.text:0040128A                mov     dword ptr [eax], offset byte_403140
.text:00401290                mov     al, 1
.text:00401292                mov     esp, ebp
.text:00401294                pop     ebp
.text:00401295                retn
.text:00401296 ; ---------------------------------------------------------------------------
.text:00401296
.text:00401296 loc_401296:                            ; CODE XREF: sub_4011E0+Aj
.text:00401296                                        ; sub_4011E0+1Aj ...
.text:00401296                xor     al, al
.text:00401298                mov     esp, ebp
.text:0040129A                pop     ebp
.text:0040129B                retn
.text:0040129B sub_4011E0     endp
```

In pseudo code:

```
some_hash = some_hash_routine(message)
IF some_hash = '0B45D7873h' THEN
    RETURN 1 // success
ELSE
    RETURN 0 // failure
ENDIF
```

Pseudo-Code

To summarize, here's the cleaned up pseudo code:

```
FUNCTION name_mapping(name)
    mapping = {0,1,2,3,4,5,6} // in eax = &var_1C
    FOR character IN name DO
        IF character % 2 != 0 DO
            swap(mapping[0], mapping[1])
        ENDIF
        circular_left_shift(mapping)
    ENDFOR
    RETURN mapping
END

FUNCTION serial_mapping(serial)
    message[7]
    FOR i = 0 TO 6 DO
        nr = serial[i] - '0'
        IF nr >= 7 THEN
            mapping[i] = 0
        ELSE
            mapping[i] = nr
        ENDIF
    ENDFOR
    RETURN message
END

FUNCTION permutation(message, mapping)
    i = 0
    message = byte_403140
    WHILE i+7 < strlen(mapping) DO
        tmp[7]
        FOR j=0 TO 6 DO:
            tmp[j] = message[i + mapping[j]]
        ENDFOR
        FOR j=0 TO 6 DO:
            message[j] = tmp[j]
```

```
35        ENDFOR
36    END
37    RETURN message
38 END
39
40 FUNCTION CHECK_SERIAL(serial, name)
41    IF strlen(serial) != 7 THEN
42        RETURN 0
43    END
44    mapping = name_mapping(name)
45
46    STRCPY(message, byte_403010) // hard coded message
47    IF message == NULL THEN
48        GOTO loc_40123A \\ should never happen
49    ENDIF
50    message = permutation(message, mapping)
51    mapping = serial_mapping(serial)
52    message = permutation(message, mapping)
53    some_hash = some_hash_routine(message)
54    IF some_hash = '0B45D7873h' THEN
55        RETURN 1 // success
56    ELSE
57        RETURN 0 // failure
58    ENDIF
59 END
60
61 CHECK_SERIAL(serial, name)
```

Cracking the Code

Now that we know how the algorithm works, we need to first figure out which permutation would produce the correct plaintext message. The encrypted message in `byte_403010` is:

> prncyl In cryp haorptg e ,apy iamttru onbxo b Po -(r ix so)o t ehami fbdofu- hftss i gulnpod t e tr
> ueemnrr tao bep s sorat cisbSs -osn xs ioer,us ptnntii eauf if gdh inw soatl r ienssoi npg.

To crack the code, it is enough to find the permutation for one 7 letter block. The first 7 letters of the ciphertext are:

```
1 prncyl
```

(there's a space at the end). The capital letter `I` comes first, the rest isn't too hard to guess either:

```
1 In cryp
```

So the correct decryption mapping is: 6 4 1 3 5 0 2. (the first letter goes to the 6th position, the second letter to the 4th, etc.).

Keygen

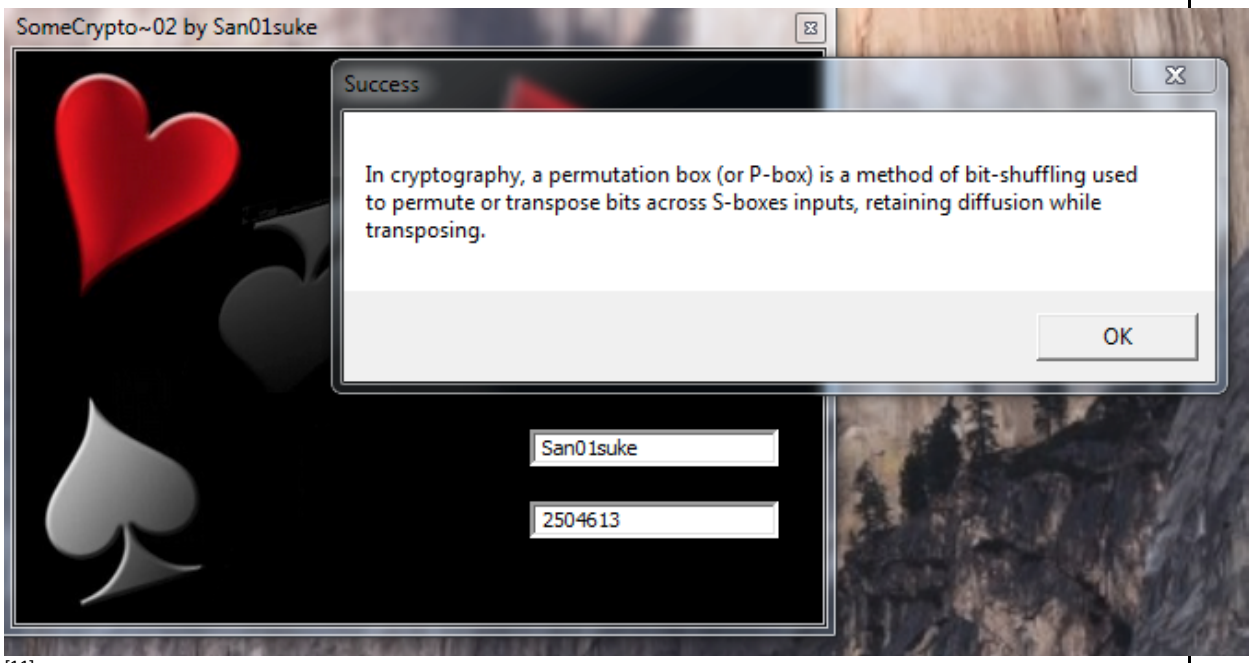
The `SomeCrypt~02` applies two permutations to the ciphertext. The first is based on the name, the second is given by the serial. To write a keygen for a given name we need to calculate the resulting mapping by running the `name_mapping` routine. Then we can determine which second mapping, when combined with the name mapping, results in the correct mapping 6 4 1 3 5 0 2:

```
1  import argparse
2  from collections import deque
3
4  parser = argparse.ArgumentParser(description="SomeCrypto~02 keygen")
5  parser.add_argument('name')
6  args = parser.parse_args()
7  name = args.name
8
9  correct_key = [6, 4, 1, 3, 5, 0, 2]
10 cypher = deque(list(range(7)))
```

```
11
12 for c in name:
13     if ord(c) % 2:
14         cypher[0], cypher[1] = cypher[1], cypher[0]
15     cypher.rotate(-1)
16
17 serial = 7*[None]
18 for c, k in zip(cypher, correct_key):
19     serial[c] = k
20
21 print('serial: ' + ''.join(str(s) for s in serial))
```

Testing:

```
1 $ python keygen.py San01suke
2 serial: 2504613
```



[11]
