Solution to SomeCrypto~03 by San01Suke

This crackme is very similar to SomeCrypto~02, please solve this crackme first or read the solution. This is the relevant subroutine:

```
.text:00401258 ; ---------------------------------------------------------------------------
.text:00401259                 align 10h
.text:00401260
.text:00401260 ; =============== S U B R O U T I N E =======================================
.text:00401260
.text:00401260 ; Attributes: bp-based frame
.text:00401260
.text:00401260 sub_401260      proc near               ; CODE XREF: 1DDp
.text:00401260
.text:00401260 permutation     = byte ptr -2Ch
.text:00401260 arg_0           = dword ptr  8
.text:00401260
.text:00401260 name_length = eax
.text:00401260 serial = esi
.text:00401260                 push    ebp
.text:00401261                 mov     ebp, esp
.text:00401263                 sub     esp, 30h
.text:00401266                 push    serial
.text:00401267                 mov     serial, name_length
.text:00401269                 xor     name_length, name_length
.text:0040126B                 cmp     [edx], al
.text:0040126D                 jz      failed          ; if name empty jump
.text:00401273
.text:00401273 loc_401273:                             ; CODE XREF: sub_401260+18j
.text:00401273                 inc     name_length
.text:00401274                 cmp     byte ptr [name_length+edx], 0 ; name[index] == '\0'
.text:00401278                 jnz     short loc_401273
.text:0040127A                 cmp     name_length, 4  ; -> name length needs to >= 4
.text:0040127D                 jl      failed
.text:00401283                 xor     eax, eax
.text:00401285                 cmp     [serial], al    ; if empty serial jump
.text:00401287                 jz      failed
.text:0040128D                 lea     ecx, [ecx+0]    ; ecx = 7743008E
.text:00401290
.text:00401290 loc_401290:                             ; CODE XREF: sub_401260+35j
.text:00401290                 inc     eax
.text:00401291                 cmp     byte ptr [eax+serial+0], 0
.text:00401295                 jnz     short loc_401290
.text:00401297                 cmp     eax, 10         ; serial length needs to be 10
.text:0040129A                 jnz     failed
.text:004012A0                 lea     eax, [ebp+permutation] ; eax will hold permutation
.text:004012A3                 call    permutation_name ; generate permutation based on name
.text:004012A8                 mov     al, byte_403010
.text:004012AD                 mov     byte_403080, al
.text:004012B2                 mov     edx, 17
.text:004012B7                 jmp     short loc_4012C0
.text:004012B7 ; ---------------------------------------------------------------------------
.text:004012B9                 align 10h
.text:004012C0
.text:004012C0 loc_4012C0:                             ; CODE XREF: sub_401260+57j
.text:004012C0                                         ; sub_401260+79j
```

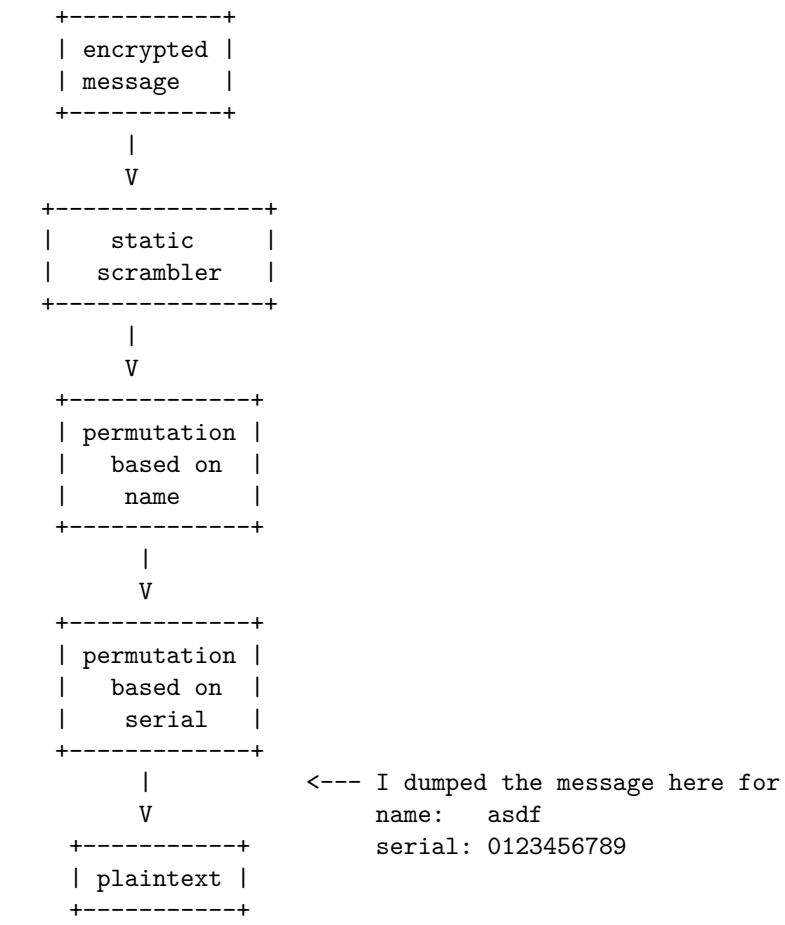
```
.text:004012C0                 mov     cl, byte_403010[edx]
.text:004012C6                 mov     byte_403080[edx], cl
.text:004012CC                 lea     eax, [edx+17]
.text:004012CF                 cdq
.text:004012D0                 mov     ecx, 103
.text:004012D5                 idiv    ecx             ; eax = edx/103, edx = edx % 103
.text:004012D7                 test    edx, edx
.text:004012D9                 jnz     short loc_4012C0 ; loop while edx != 0
.text:004012DB                 mov     edx, offset byte_403080
.text:004012E0                 lea     eax, [ebp+permutation]
.text:004012E3                 call    map_rotate
.text:004012E8                 mov     ecx, serial
.text:004012EA                 lea     eax, [ebp+permutation]
.text:004012ED                 call    mapping_is_serial
.text:004012F2                 mov     edx, offset byte_403080
.text:004012F7                 lea     eax, [ebp+permutation]
.text:004012FA                 call    map_rotate
.text:004012FF encrypted_msg = edx
.text:004012FF                 mov     ecx, 67h
.text:00401304                 mov     encrypted_msg, offset byte_403080
.text:00401309                 or      eax, 0FFFFFFFFh
.text:0040130C                 lea     esp, [esp+0]
.text:00401310
.text:00401310 loc_401310:                             ; CODE XREF: sub_401260+C7j
.text:00401310                 movzx   esi, byte ptr [encrypted_msg]
.text:00401313                 xor     esi, eax
.text:00401315                 and     esi, 0FFh
.text:0040131B                 shr     eax, 8
.text:0040131E                 xor     eax, ds:dword_402060[esi*4]
.text:00401325                 inc     encrypted_msg
.text:00401326                 dec     ecx
.text:00401327                 jnz     short loc_401310
.text:00401329                 not     eax
.text:0040132B                 cmp     eax, 72DD193Dh
.text:00401330                 jnz     short failed
.text:00401332                 mov     encrypted_msg, [ebp+arg_0]
.text:00401335                 mov     eax, ds:MessageBoxA
.text:0040133A                 push    encrypted_msg
.text:0040133B                 push    offset aSuccess ; "Success"
.text:00401340                 push    eax
.text:00401341                 mov     ecx, offset byte_403080
.text:00401346                 mov     byte ptr unk_403078, 1
.text:0040134D                 call    ecx ; byte_403080
.text:0040134F                 add     esp, 0Ch
.text:00401352                 mov     al, 1
.text:00401354                 pop     esi
.text:00401355                 mov     esp, ebp
.text:00401357                 pop     ebp
.text:00401358                 retn
.text:00401359 ; ---------------------------------------------------------------------------
```

The performed steps are almost the same as in *SomeCrypto~02*:

- There is a hardcoded, encrypted message at `byte_403080`
- The message is first shuffled around in a static fashion, independent of the name, serial or any other input. This part does not exist in *SomeCrypto~02*, but it doesn't make the problem any harder because we just need to dump the message *after* this routine.

- The code then permutates the message in blocks of 10. The permutation is based on the *name* field.
- The resulting message is then permutated once more, this time the mapping corresponds to the *serial* input. Both the *name* and the *serial* permutation are calculated as in *SomeCrypto~03*, except this time the block size is 10 Bytes rather than 7.

Here's a visualization of the different steps:



The steps to crack this problem are the same as in *SomeCrypto~02*:

1. Find out which permutation generates the correct plaintext from the encrypted message (*after* it was shuffled around).
2. Given the correct permutation, write a keygen that gives serials which - combined with the permutation of the name - produce the desired permutation from step 1.

In *SomeCrypto~02* the desired plaintext was pretty obvious, because it was an English sentence encoded in ASCII. This time, most bytes are not ASCII characters (at least not printable ones that is). Heres a dump of location `403080` (the message) just before the correctness check in line `004012FF`. I used the name `asdf` and serial `0123456789`:

```
EC EC 83 EC 59 45 C6 55 8B 14
C6 C6 6F ED 45 75 EE C6 45 45
F1 F0 45 C6 61 45 C6 EF 20 68
C6 C6 76 F2 45 65 F3 C6 45 45
F6 F5 45 C6 6F 45 C6 F4 20 64
C6 C6 6E F7 45 65 F8 C6 45 45
FB FA 45 C6 74 45 C6 F9 20 69
```

```
6A C6 21 FC 00 00 FD C6 45 45
8B 8D 50 0C 55 51 EC 8B 45 4D
8B 08 55 FF E5 10 C4 10 52 83
5D C3 55
```

So if the message isn't text, what is it? Looking at the code at the end of the snippets reveals the purpose of 403080:

```
.text:00401341                mov     ecx, offset byte_403080
.text:00401346                mov     byte ptr unk_403078, 1
.text:0040134D                call    ecx ; byte_403080
```

The message isn't text, but an executable subroutine! In other words: the correct permutation will generate a sequence of bytes that can be disassembled into meaningful code. While it was easy to guess the correct permutation for the English sentence from *SomeCrypto~02*, this task is much harder because it isn't obvious what sequence of bytes produce reasonable disassembly. However, there are certain constructs that many subroutines share, e.g., the function prologue and epilogue. This is the common function prologue:

The usual prologue is:

```
55          PUSH EBP
8B EC       MOV EBP, ESP
83 EC <nr>  SUB ESP <nr>
```

Let's compare that with the first block of our message:

```
 0  1  2  3  4  5  6  7  8  9
EC EC 83 EC 59 45 C6 55 8B 14
```

Since the subroutine probably starts with `PUSH EBP` (opcode `55`), this means that the byte at 7 should go to place 0. Doing this for all bytes in the prologue gives the following incomplete permutation:

```
7 -> 0
8 -> 1
0/1/3 -> 2,4
2 -> 3
9 -> 5 ?
```

The last one is only a guess, based on the fact that only 14h and ECh are divisible by 4, and EC seems awfully large for a stack frame for this small function snippet.

Let's look at the usual epilogue:

```
8B E5   MOV ESP, EBP
5D      POP EBP
C3      RETN
```

The last (incomplete) block of the message is:

```
5D C3 55
```

The code doesn't permutate the last block if it hasn't 10 bytes, so the bytes should be in the correct order - which they are. Byte 55 is an extra byte after the `RETN` instruction. The second to last line is:

```
 0  1  2  3  4  5  6  7  8  9
8B 08 55 FF E5 10 C4 10 52 83
```

We need to have `MOV ESP, EBP` as the last to bytes, which gives:

```
0 -> 8
4 -> 9
```

Combine this with the mapping from the prologue we get:

```
7   -> 0
8   -> 1
1/3 -> 2
2   -> 3
1/3 -> 4
9   -> 5
5/6 -> 6
5/6 -> 7
0   -> 8
4   -> 9
```

We still have 4 potential permutations. Lets got back to the prologue, where the next instruction follows at byte 6. From the previous mapping we know that the next bytes are either `45 C6 EC 59` or `C6 45 EC 59`. The first is an `inc` instruction, the second a `mov` instruction. I used the Online Disassembler to check which sequence makes more sense. Only `C6 45 EC 59` gave a plausible instruction:

```
    MOV BYTE PTR [EBP-0x14], 0x59
```

So the mapping is almost done:

```
7   -> 0
8   -> 1
1/3 -> 2
2   -> 3
1/3 -> 4
9   -> 5
6   -> 6
5   -> 7
0   -> 8
4   -> 9
```

The second block of the message is:

```
 0  1  2  3  4  5  6  7  8  9
C6 C6 6F ED 45 75 EE C6 45 45
```

which - according to our permutation - reads either as:

```
C6 45 C6 6F ED 45 EE 75 C6 45
```

or

```
C6 45 ED 6F C6 45 EE 75 C6 45
```

The first version gives:

```
.data:0x00000000    c645c66f    mov BYTE PTR [ebp-0x3a],0x6f
.data:0x00000004    ed          in eax,dx
.data:0x00000005    45          inc ebp
.data:0x00000006    ee          out dx,al
.data:0x00000007    75c6        jne 0xffffffcf
.data:0x00000009    45          inc ebp
```

This doesn't seem quite right, the `in` instruction is very uncommon. The second disassembly is way better:

```
.data:0x00000000    c645ed6f    mov BYTE PTR [ebp-0x13],0x6f
.data:0x00000004    c645ee75    mov BYTE PTR [ebp-0x12],0x75
.data:0x00000008    c6          .byte 0xc6
.data:0x00000009    45          inc ebp
```

The first two instructions make perfect sense, and the last two instruction are only broken because the bytes from the next block are missing. So this has to be the final mapping is:

```
7   ->  0
8   ->  1
3   ->  2
2   ->  3
1   ->  4
9   ->  5
6   ->  6
5   ->  7
0   ->  8
4   ->  9
```

The reverse of this mapping is our serial for the name `asdf`:

```
8432976015
```

While I entered the identity permutation for the serial `0123456789`, I forgot to do the same for the name. Using the routine from *SomeCrypto~02* I got the permutation for `asdf`:

```
from collections import deque

name = 'asdf'
cypher = deque(list(range(10)))
for c in name:
    if ord(c) % 2:
        cypher[0], cypher[1] = cypher[1], cypher[0]
    cypher.rotate(-1)
print(cypher)

$ python name_erm.py
deque([4, 5, 6, 7, 8, 9, 1, 2, 0, 3])
```

To undo it I applied it to `8432976015` and got:

```
9780154382
```

(9 is the 4th digit into `8432976015`, 7 is the 5th digit, and so on).

The keygen is the same as in `SomeCrypto~02`, except for the change in block size and the new mapping of course:

```
import argparse
from collections import deque

parser = argparse.ArgumentParser(description="SomeCrypto~03 keygen")
parser.add_argument('name')
args = parser.parse_args()
name = args.name

correct_key = [9,7,6,0,1,5,4,3,8,2]
cypher = deque(list(range(10)))

for c in name:
    if ord(c) % 2:
        cypher[0], cypher[1] = cypher[1], cypher[0]
    cypher.rotate(-1)
serial = 10*[None]
for c, k in zip(cypher, correct_key):
    serial[c] = k

print('serial: ' + ''.join(str(s) for s in serial))


$ python keygen.py tristana
serial: 6041583927
```

If you enter valid name/serial combinations, you are greeted with the good boy message: