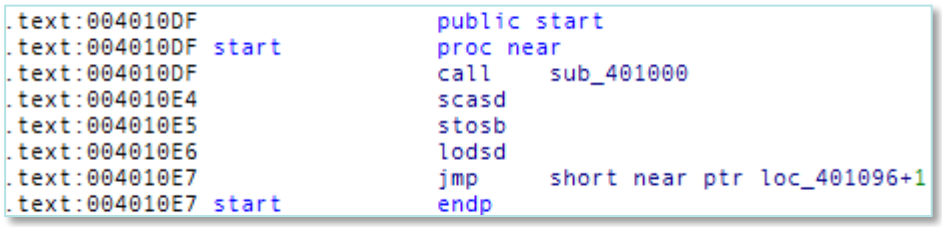
Challenge #2 Solution

by Nick Harbour

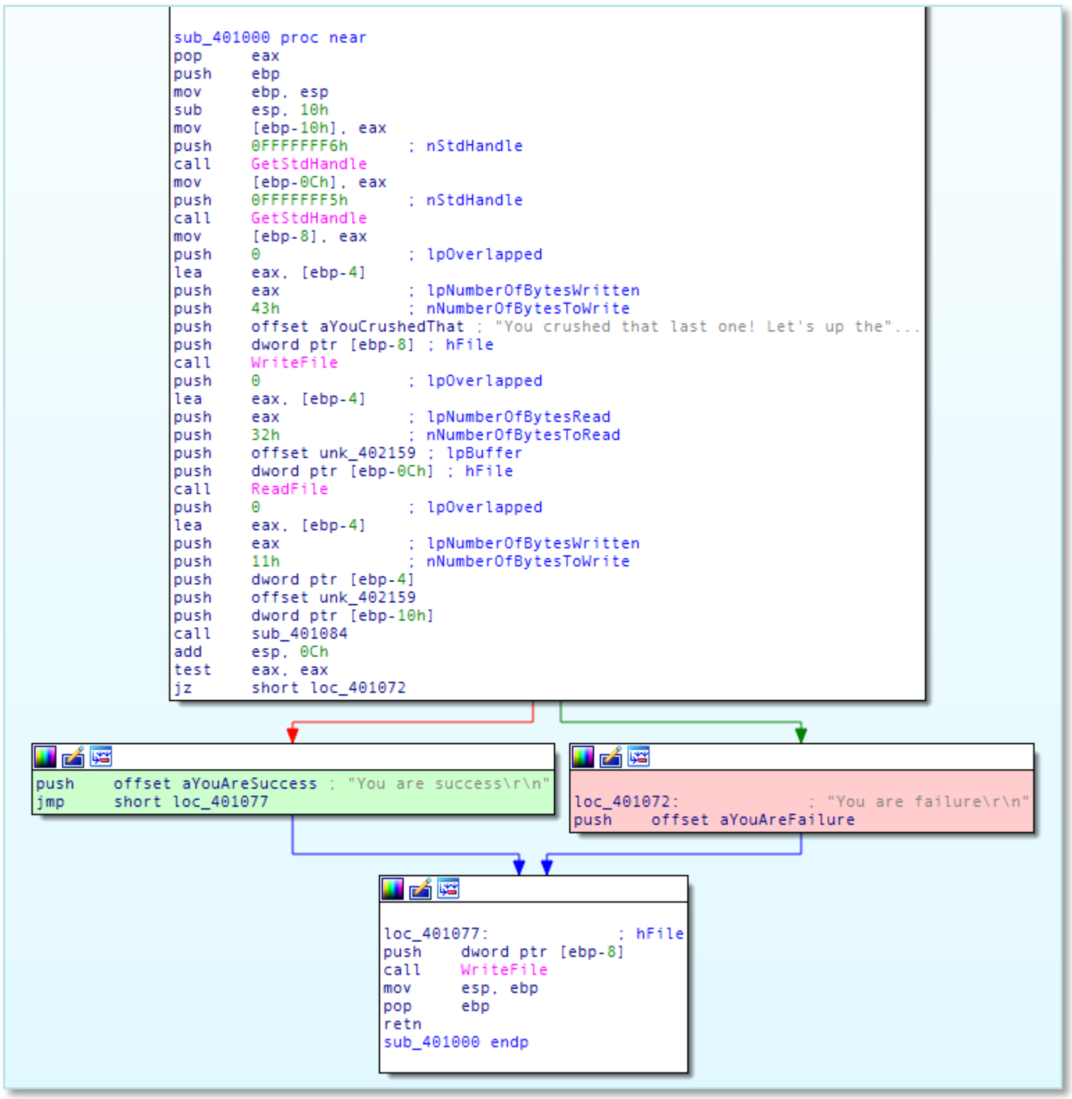
The overall structure of #2 is very similar to the first challenge as they both display a prompt, allow you to type input, and display a success or failure message depending on whether the key entered was correct. Many code structures are identical as well, so we will not spend much time discussing the reverse engineering aspects which are the same as those covered by the last solution. Let's jump into the code in IDA Pro!



```
.text:004010DF                         public start
.text:004010DF start                   proc near
.text:004010DF                         call    sub_401000
.text:004010E4                         scasd
.text:004010E5                         stosb
.text:004010E6                         lodsd
.text:004010E7                         jmp     short near ptr loc_401096+1
.text:004010E7 start                   endp
```

Figure 1: Initial call instruction for Solution #2

The first instruction in the program is a call instruction at address 0x4010DF. This calls the actual start of the program but execution never returns to the area after this call. IDA Pro has mislabeled the data that follows as instructions. The result of this instruction is merely to push the value 0x4010E4 on the stack. Now let's focus on the code at 0x401000.



```
sub_401000 proc near
pop     eax
push    ebp
mov     ebp, esp
sub     esp, 10h
mov     [ebp-10h], eax
push    0FFFFFFF6h      ; nStdHandle
call    GetStdHandle
mov     [ebp-0Ch], eax
push    0FFFFFFF5h      ; nStdHandle
call    GetStdHandle
mov     [ebp-8], eax
push    0               ; lpOverlapped
lea     eax, [ebp-4]
push    eax             ; lpNumberOfBytesWritten
push    43h             ; nNumberOfBytesToWrite
push    offset aYouCrushedThat ; "You crushed that last one! Let's up the"...
push    dword ptr [ebp-8] ; hFile
call    WriteFile
push    0               ; lpOverlapped
lea     eax, [ebp-4]
push    eax             ; lpNumberOfBytesRead
push    32h             ; nNumberOfBytesToRead
push    offset unk_402159 ; lpBuffer
push    dword ptr [ebp-0Ch] ; hFile
call    ReadFile
push    0               ; lpOverlapped
lea     eax, [ebp-4]
push    eax             ; lpNumberOfBytesWritten
push    11h             ; nNumberOfBytesToWrite
push    dword ptr [ebp-4]
push    offset unk_402159
push    dword ptr [ebp-10h]
call    sub_401084
add     esp, 0Ch
test    eax, eax
jz      short loc_401072
```

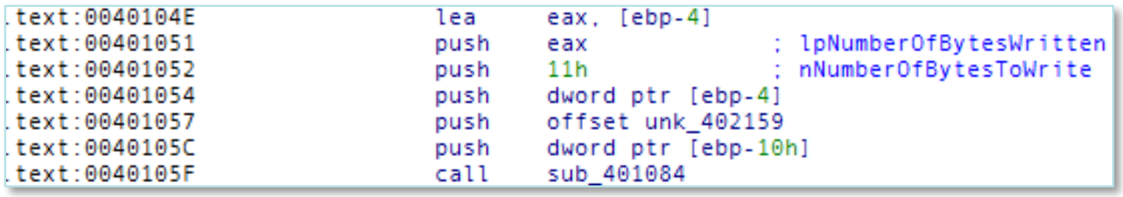
```
push    offset aYouAreSuccess ; "You are success\r\n"
jmp     short loc_401077
```

```
loc_401072:             ; "You are failure\r\n"
push    offset aYouAreFailure
```

```
loc_401077:             ; hFile
push    dword ptr [ebp-8]
call    WriteFile
mov     esp, ebp
pop     ebp
retn
sub_401000 endp
```

Figure 2: Main Function disassembly graph for Solution #2

The structure of this code looks very similar to the last challenge, with the two calls to `GetStdHandle()`, a `WriteFile()` call to display the prompt, and a `ReadFile()` call to collect the input to a buffer (at 0x402159 in this case). Where the two programs differ is that this challenge performs a function call (at address 0x40105F) to determine its success or failure condition whereas the last challenge performed a loop directly in this this function.

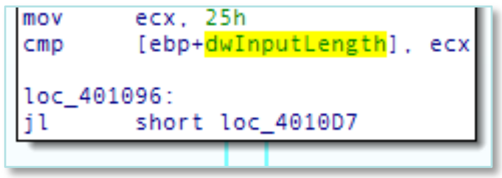


```
.text:0040104E                 lea     eax, [ebp-4]
.text:00401051                 push    eax             ; lpNumberOfBytesWritten
.text:00401052                 push    11h             ; nNumberOfBytesToWrite
.text:00401054                 push    dword ptr [ebp-4]
.text:00401057                 push    offset unk_402159
.text:0040105C                 push    dword ptr [ebp-10h]
.text:0040105F                 call    sub_401084
```

Figure 3: Dissassembly of the call to check function for Solution #2

In this block of code we see the call to the function whose return value (zero or non-zero) will determine if the program displays the success or failure message. Please note that it is not simply sufficient to patch the program to always display the success message, as even though it will display success you still will not know the email address to send to in order to receive the next challenge. This function takes 3 arguments: The pointer value (0x4010E4) from the first call instruction (stored at `ebp-10h`), a pointer to buffer of input (0x402149), and the number of bytes entered as input (stored in `ebp-4`). Let's take a look at that function!

Since we have already determined what the function parameters are I took the liberty of renaming them in the check function to "key", "input", and "dwInputLength". The first sequence of instructions of note in this function begin at 0x40108E:

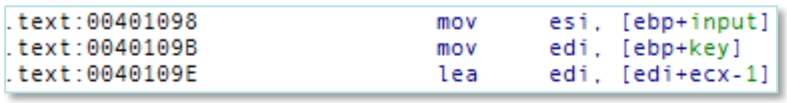


```
mov     ecx, 25h
cmp     [ebp+dwInputLength], ecx

loc_401096:
jl      short loc_4010D7
```

Figure 4: Input Length check for Solution #2

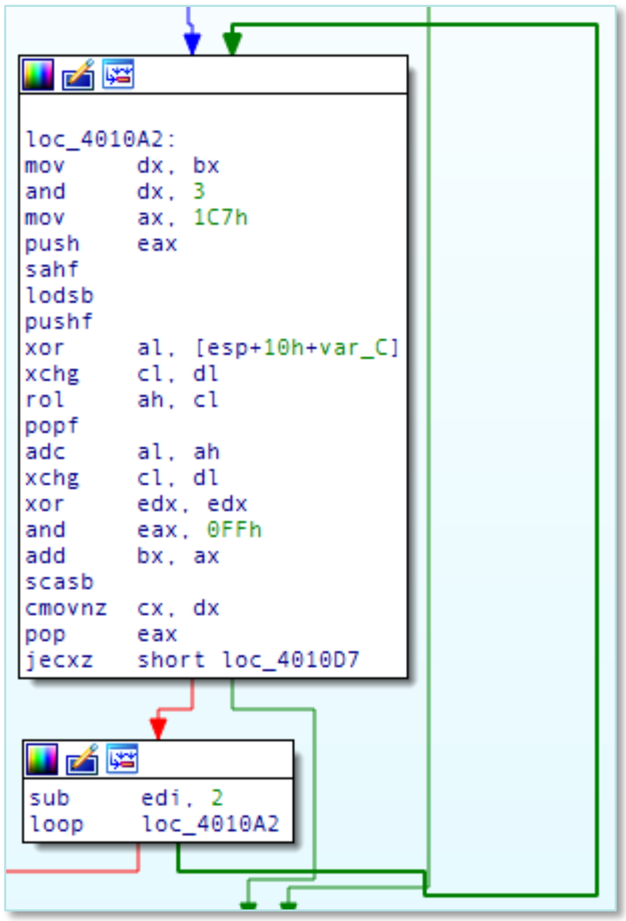
This sequence simply checks the length of the entered input and exits (with failure) if it is less than 37. It just so happens that the length of the correct answer is 37, but we'll get to that later. The next sequence of instructions just loads the function parameters into registers, `esi` for the input and `edi` for the key.



```
.text:00401098                 mov     esi, [ebp+input]
.text:0040109B                 mov     edi, [ebp+key]
.text:0040109E                 lea     edi, [edi+ecx-1]
```

Figure 5: Loop Initialization for Solution #2

The last instruction in this sequence overwrites `edi` by adding `ecx` (which is 0x25 at this point in the program) and -1 to it. This effectively sets `edi` to point to the end of the key instead of the beginning. The two blocks following this in the IDA Pro graph represent the core of the loop which is going to check the values of the input against the key values, albeit in a slightly bizarre fashion.



```
loc_4010A2:
mov     dx, bx
and     dx, 3
mov     ax, 1C7h
push    eax
sahf
lodsb
pushf
xor     al, [esp+10h+var_C]
xchg    cl, dl
rol     ah, cl
popf
adc     al, ah
xchg    cl, dl
xor     edx, edx
and     eax, 0FFh
add     bx, ax
scasb
cmovnz  cx, dx
pop     eax
jecxz   short loc_4010D7
```

```
sub     edi, 2
loop    loc_4010A2
```

Figure 6: Loop Core for Solution #2

Notice that the end of the loop structure is a subtraction against `edi` (which points to the key) instead of a more common increment. This along with the fact that `edi` is set to the end of the key at the beginning of the loop is a sign that the key is stored in reverse order. The BX register maintains a rolling sum throughout the loop, which is factored into the computation of each byte. At the beginning of the loop, BX is copied to DX where the AND instruction clears all but the bottom 2 bits. Next, the value 0x1C7 is saved into AX, then stored on the stack, then the SAHF instruction loads the high byte of that value (which is 0x01) into the EFLAGS register. The result of this is that the Carry flag is set. The LODSB instruction loads a byte of the input, pointed to by ESI into AL and the PUSHF instruction stores the EFLAGS to the stack.

The input byte is XOR'd with a value from the stack, which if you inspect it closely or with a debugger you'll see that the value it is XORing against is 0xC7, which was saved on the stack with the previous "push eax" instruction. The contents of AH is still set to 0x01 and is then rotated (left) by the amount specified by the lowest 2 bits of the rolling sum value. Then the POPF instruction restores the EFLAGS register with the saved flags on the stack (carry flag set). The ADC instruction that follows is just like a normal ADD instruction but it also adds the carry flag, which is always set in this case. The next few instructions are straight forward, ending with the low byte of eax being added to the rolling sum in bx. The SCASB will then compare this low byte with the key value that is pointed to by EDI, as well as increment EDI. If the values do not match then the CMOVNZ instruction will move a zero value stored in DX to the loop counter CX (part of ECX), effectively terminating the loop prematurely. The last JECXZ instruction checks to see if the loop counter is zero, which signifies a premature termination and a key mismatch, and jumps to the failure code path. If the loop iterates a full 37 times with no mismatches between the input and the key, then the LOOP instruction will stop branching when it decrements ECX to zero and the code will follow the success path.

Deducing the inverse to this function is not straight-forward and it may be easier to simply place a breakpoint at the SCASB instruction and repeatedly try values until one produces the correct value needed to match the key. Figure 7 below is an IDA Python script which will print the correct answer to the IDA Python window.

```python
def decode2_ida():
    keystart = 0x4010E4
    keylen = 37
    index = keylen - 1
    rolling_sum = 0
    result = []
    while index >= 0:
        addval = ROL(1, rolling_sum & 3) + 1
        newval = idc.Byte(keystart + index)
        rolling_sum += newval
        newval -= addval
        newval &= 0xFF
        newval ^= 0xC7
        result.append(chr(newval))
        index -= 1
    print ''.join(result)

def ROR(x, n, bits = 32):
    n = n % bits
    mask = (2L**n) - 1
    mask_bits = x & mask
    return (x >> n) | (mask_bits << (bits - n))

def ROL(x, n, bits=32):
    n = n % bits
    return ROR(x, bits - n, bits)
```

```
decode2_ida()
```

Figure 7: IDAPython decoding script for Solution #2