Challenge #1 Solution

by Nick Harbour

If you ran the file Flare-On_start_2015.exe which was available for download on the website, and accepted the EULA, it extracted a file named "`i_am_happy_you_are_to_playing_the_flareon_challenge.exe`". If you run this program on the command line it displays a prompt asking for a password, as shown in the figure below.

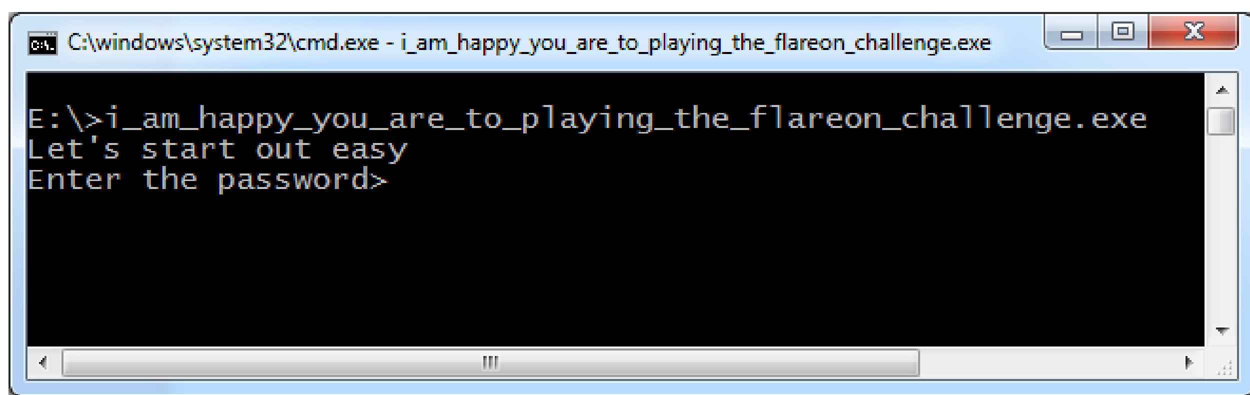

Figure 1: Screenshot of Solution #1 Prompt

If you enter the correct password, it will display the message "You are success" and it will display "You are failure" if you did not enter the correct password. The correct password will be the email address that you will send a message in order to receive the next challenge. Let's break apart the code and learn what that address is!

If you open the program with a disassembler such as IDA Pro (the freeware version will work), you will see that there is only one function in the program. This program is so small that I can include a screenshot showing the program in its entirety below.

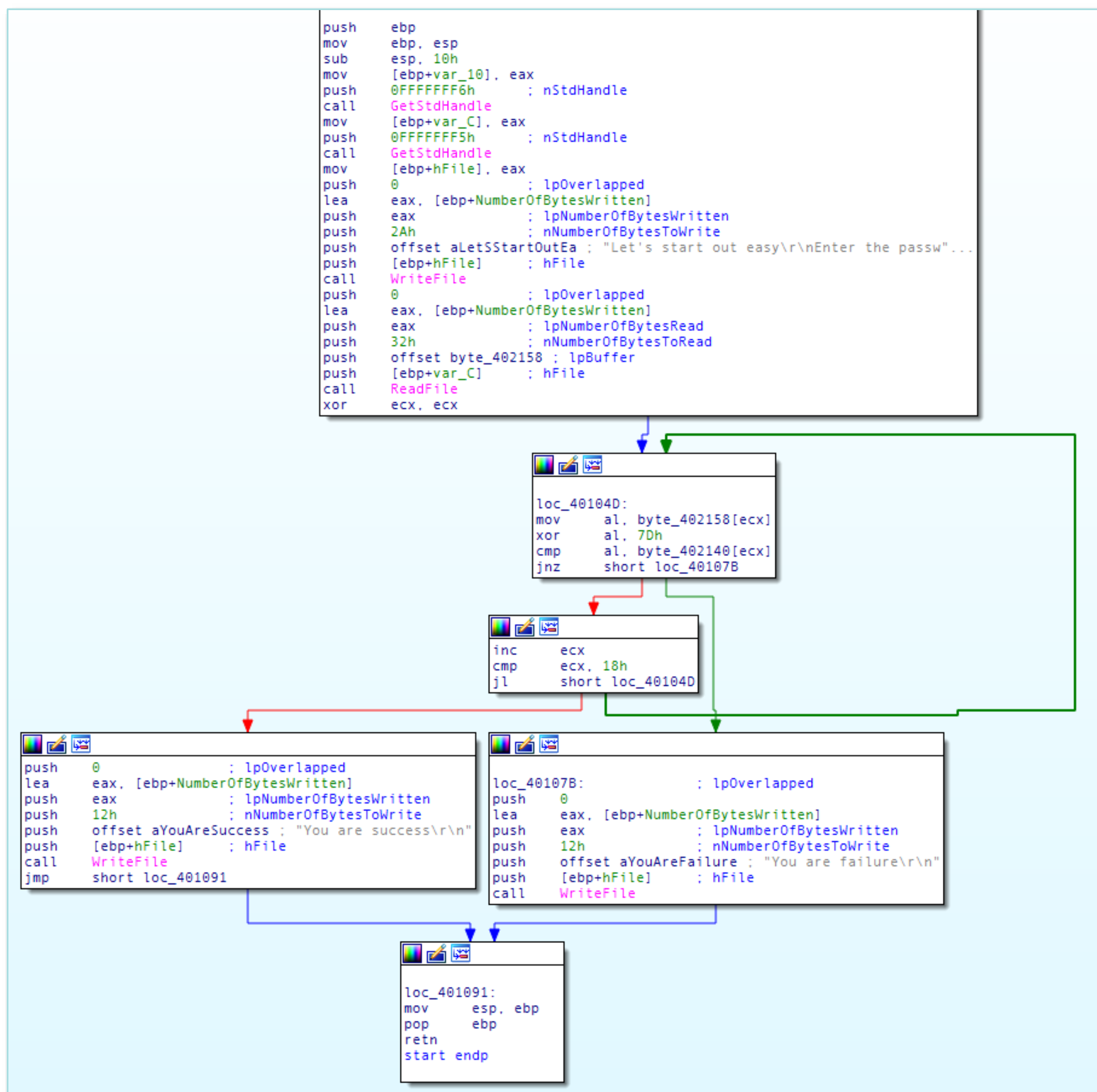

Figure 2: Graph of Solution #1 Disassembly

The first two API calls it makes are to a function called `GetStdHandle()`, which will return a handle to one of the three standard IO streams for the process (standard input, output, and error). If you

select one of the values being passed to this function you can make IDA Pro show you the name of the constant being passed in (right click->Use standard symbolic constant->find the value of the form STD_*_HANDLE). In this case the values are STD_INPUT_HANDLE and STD_OUTPUT_HANDLE, respectively. After each call, the `eax` register is moved to a stack variable. Since `eax` represents the return value from a function after a call, this register contains the handle values for the standard handles after each call, and they are being saved on the stack. If you label the stack variables as well as rename the constants, your disassembly for this section of code will look something like Figure 3 below.

```
push    STD_INPUT_HANDLE ; nStdHandle
call    GetStdHandle
mov     [ebp+hStdInput], eax
push    STD_OUTPUT_HANDLE ; nStdHandle
call    GetStdHandle
mov     [ebp+hStdOutput], eax
```

Figure 3: Marked up calls to GetStdHandle for Solution #1

The next two API functions called in the program are `WriteFile()` and `ReadFile()`. Let's look closer at this section of code to understand its functionality.

```
push    0               ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax             ; lpNumberOfBytesWritten
push    2Ah             ; nNumberOfBytesToWrite
push    offset aLetSStartOutEa ; "Let's start out easy\r\nEnter the passw"...
push    [ebp+hStdOutput] ; hFile
call    WriteFile
push    0               ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax             ; lpNumberOfBytesRead
push    32h             ; nNumberOfBytesToRead
push    offset byte_402158 ; lpBuffer
push    [ebp+hStdInput] ; hFile
call    ReadFile
```

Figure 4: WriteFile and ReadFile calls in Solution #1

The call to `WriteFile()` takes 5 arguments (you can verify this on the MSDN documentation) and these correspond to the 5 `push` instructions preceding the call. This call is simply writing the prompt

message to the standard output handle, which causes it to be displayed to the user in the terminal.

The next call to `ReadFile()` is more interesting. It also takes 5 arguments, starting with the file handle to read from, the buffer to store the data, and the maximum amount to read. The program is reading at most 50 bytes into a buffer at memory address 0x402158. You can right click on the name " byte_402158" and select "rename" to give it a more descriptive name now that we know its purpose in this program. I have chosen to name it "input_buffer".

The code immediately following the `ReadFile()` call we will skip for now, and instead focus on the two blocks of code near the end of the program as shown below in Figure 5.

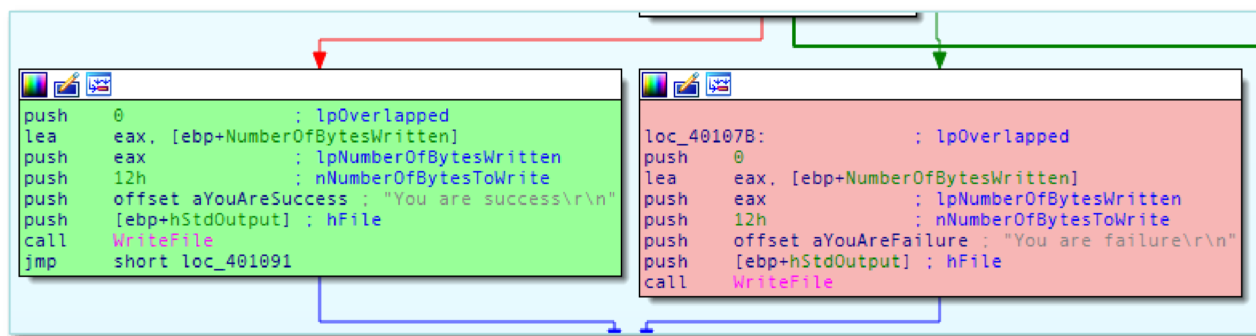

Figure 5: Success and Failure code blocks in Solution #1

I have taken the liberty of color coding these blocks. If the block on the left executes, it will display to the terminal "You are success" so I have colored it mint green. If the block on the right executes, it will display "You are failure" so I have colored it a variant of Coral Pink. You can color a node by right clicking on the node header and selecting the option "Set node color". I recommend choosing colors which both convey meaning and do not interfere with the legibility of the text (as using a color such as pure green or red would have).

The selection of which block to execute occurs by the `jl` instruction (jump if less than) at memory address 0x401061 or potentially by the `jnz` instruction (jump if not zero or not equal) at memory address 0x40105B. Let's examine this code are in more detail.

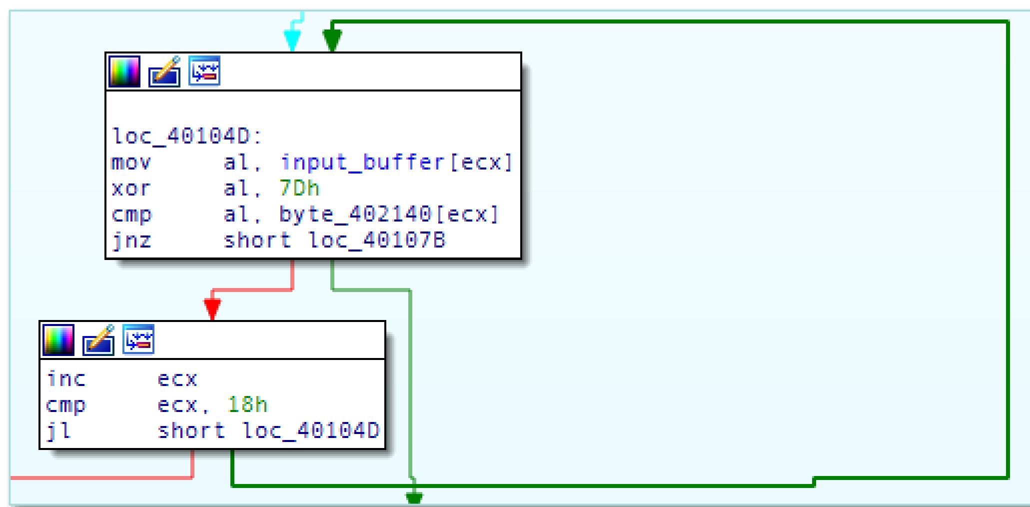

Figure 6: Decoding loop in Solution #1

Not shown in this fragment is the instruction “`xor ecx, ecx`” which simply sets the `ecx` register to zero prior to the beginning of the loop. We can tell that this is a loop because there is a thick, backward pointing arrow in our disassembly graph. The `JL` instruction at the end jumps back to the beginning of the loop if `ecx` is less than 18h (24 in decimal), otherwise the code will execute the success block. The `ecx` register is incremented before this comparison, and it started out the loop as the value 0, so it is likely the loop counter. Therefore this program will only print the success message if this loop iterates a full 24 times.

The loop can short circuit, however. In each iteration of the loop, a byte is loaded from the input buffer in the al register, then XOR’d with the value 0x7D, then compared against a byte in another array at memory address 0x402140. If the bytes match then the zero flag will be set, causing the `jnz` instruction to not branch. If the bytes did not match (i.e. you entered a wrong digit in the password) then the zero flag will not be set, causing the branch to the failure bock to occur.

Since the program is performing a single byte XOR loop. If you then XOR the bytes found at memory address 0x402140 with the value 0x7D, the correct answer will be revealed. Almost any hex editor is

suitable for the task, as is IDA pro itself. Figure 7 below is a small IDA Python script to reveal the correct answer.

```
for i in range(0x00402140, 0x00402158):
      b = 0x7D ^ idc.Byte(i)
      idc.PatchByte(i, b)
```

Figure 7: IDAPython Decoding Script for Solution #1

Running the script reveals the correct input at the memory address of the key in the IDA database:

```
.data:00402140 aBunny_sl0pe@fl db 'bunny_sl0pe@flare-on.com' ; DATA XREF: start+55↑r
```

Figure 8: Decoded Answer for Solution #1

Always confirm your Flare-On solutions by trying the input in the program!

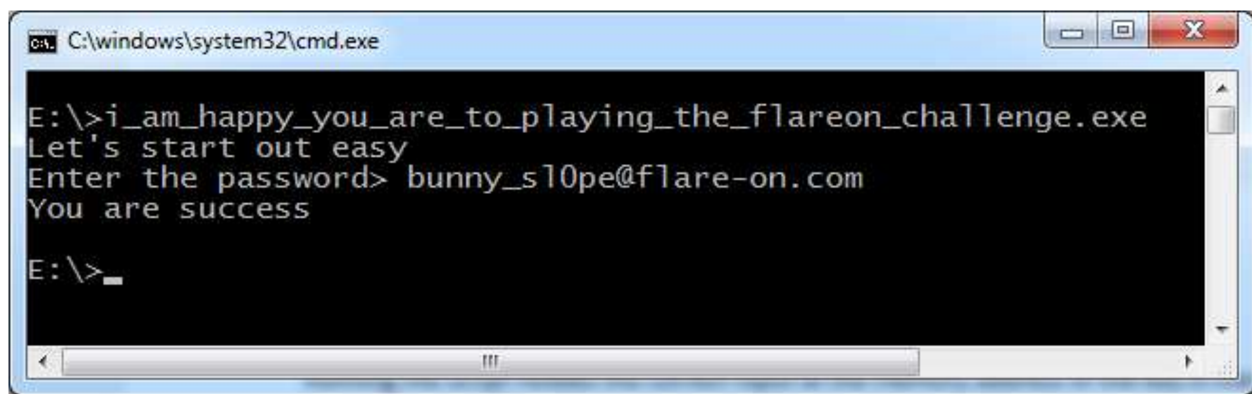

Figure 9: Success Output for Solution #1