Flare-On 4: Challenge #2 Solution – IgniteMe.exe

Challenge Author: Nhan Huynh

`IgniteMe.exe` expects to run without any command line argument. It asks the player to input the flag, which is then verified. When an incorrect flag is given, `IgiteMe.exe` prints out an error message then exits. Figure 1 shows an example of supplying an incorrect password to `IgniteMe.exe` and its error message.

```
C:\Users\user\Desktop>IgniteMe.exe
G1v3 m3 t3h fl4g:  Test
N0t t00 h0t R we? 7ry 4ga1nz plzzz!
C:\Users\user\Desktop>
```

Figure 1: Failsauce!

When running strings on the binary, we find very little output. Some of the interesting strings are input prompt, a success message, and an error message, as shown in Figure 2.

```
G1v3 m3 t3h fl4g:
G00d j0b!
N0t t00 h0t R we? 7ry 4ga1nz plzzz!
```

Figure 2: Input prompt, success message and error message from strings

The `l33t` talk indicates that this is a serious challenge, so we proceed on opening `IgniteMe.exe` with IDA. At first glance, we find out that this challenge is tiny, containing only five functions. Looking at the graph view of the program entry point, which IDA has nicely named `start`. I have included a screenshot of this function in its entirety in Figure 3. In this function we see that `sub_401050` is called right before a condition branch to an error message or a success message. It appears that `sub_401050` is the flag validation function. Right above the call to `sub_401050` is the call to `sub_4010F0` after the prompt. We can conclude that `sub_4010F0` is likely used to read the player's input.

At the lines `004011BD` and `004011CA` are two calls to `GetStdHandle` with `STD_INPUT_HANDLE` and `STD_OUTPUT_HANDLE` respectively. The program stores the standard input handle at address `0x403070` (on line 004011C3), and the standard output handle to address `0x403074` (one line 004011D0).

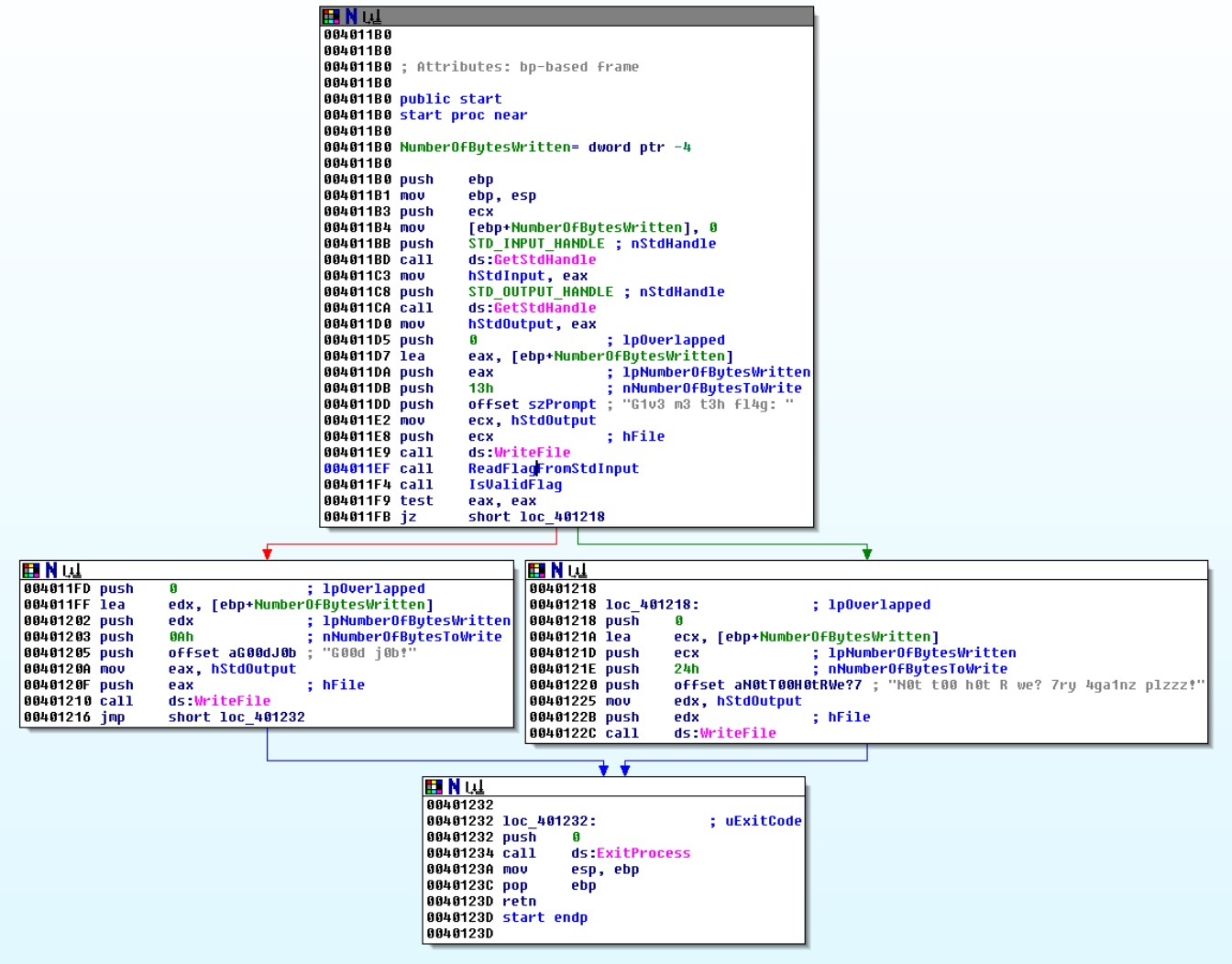

Figure 3: Marked up graph view of Entry Point

The function `sub_4010F0` reads the standard input into a local variable at the call to `ReadFile` at `0x401145`. The player's input is then copied to a global variable at address `0x40119E`. We can rename the global variable from `byte_403078` to `szUserInput`.

We then look at the flag validation function (`sub_401050`), focusing on code block shown in Figure 4, between 0x401088 and 0x4010AD:

```
.text:00401079
.text:00401079 loc_401079:
.text:00401079                 mov     ecx, [ebp+index]
.text:0040107C                 sub     ecx, 1
.text:0040107F                 mov     [ebp+index], ecx
.text:00401082
.text:00401082 loc_401082:
.text:00401082                 cmp     [ebp+index], 0
.text:00401086                 jl      short loc_4010AF
.text:00401088                 mov     edx, [ebp+index]
.text:0040108B                 movsx   eax, szUserInput[edx]
.text:00401092                 movzx   ecx, [ebp+key]
.text:00401096                 xor     eax, ecx
.text:00401098                 mov     edx, [ebp+index]
.text:0040109B                 mov     szEncodedUserInput[edx], al
.text:004010A1                 mov     eax, [ebp+index]
.text:004010A4                 mov     cl, szUserInput[eax]
.text:004010AA                 mov     [ebp+key], cl
.text:004010AD                 jmp     short loc_401079
```

Figure 4: Flag Validation Function

We can see that at address `0x40108B`, the user input is copied into eax register, which is then XOR-ed with `var_1`. The result is stored at `byte_403180` at address `0x40109B`. The user input at the current index is then moved to `var_1`. The block between address `0x401079` and `0x40107F` decrments the current index. We can see that the the user input is XOR encoded, one byte at a time, in reverse order. The global variable `byte_403180` stores the encoded user input. The buffer is later checked one byte at a time against the global varialbe `byte_403000` between `0x4010C7` and `0x4010DD`. We can conclude that `byte_403000` holds the encoded flag that the player must enter.

To decode the flag, we must figure out what is the value of `var_1` at the first iteration through the enconding routine. At address `0x40106B`, `var_1` is assigned the value of `al`, which is the one byte return value from `sub_401000`. Looking at the assembly seems fairly intimidating with all the big number and arithmatic operations. But, let's walk through the function to fully understand what it does in Figure 5.

```
.text:00401000 sub_401000 proc near
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
.text:00401003       mov     eax, 80070057h
.text:00401008       mov     edx, eax
.text:0040100A       xor     ax, dx
```

```
.text:0040100D      rol     eax, 4
.text:00401010      shr     ax, 1
.text:00401013      pop     ebp
.text:00401014      retn
.text:00401014 sub_401000 end
```

Figure 5: sub_401000

First, the magic number `0x80070057` is assigned to both eax and edx. Then it XOR's ax with dx. Since both ax and dx holds the same value, the XOR result is 0, and thus the ax register is set to 0. The new value of eax is `0x80070000`. After rotating eax 4 times, we end up with `0x07000008`. Shifiting all the bits to the right one time results in `0x038000004`. The one byte return value of `sub_401000` is `0x04`.

If we pay attention to the assembly, we can also see that all these operations happen with static values. We can also run `IgniteMe.exe` in a debugger one time to see the return value of `0x401000`, which confirms our calculation. Now that we know the initial value of `var_1`, we can attempt to decode the flag. The python script in Figure 6 implements the decoding routine for the encoded flag at `byte_403000`.

```
def main(argv):
    encoded = '\x0d\x26\x49\x45\x2A\x17\x78\x44\x2B\x6C\x5D\x5E\x45'
    encoded+= '\x12\x2f\x17\x2B\x44\x6F\x6E\x56\x09\x5F\x45\x47\x73'
    encoded+= '\x26\x0a\x0D\x13\x17\x48\x42\x01\x40\x4D\x0c\x02\x69'
    # Since the encoding routine is done in reverse, we start with
    # reversing the encoded string
    encoded = encoded[::-1]
    key = 0x04  # Initial key
    decoded = list()
    for c in encoded:
        encoded_char = ord(c)
        decoded_char = encoded_char ^ key
        decoded.append(chr(decoded_char))

        # the encoding routine replace the key with the original input,
        # meaning we must replace the key with the decoded char and not
        # the encoded char
        key = decoded_char

    # We reverse the list to the original order.
    decoded = decoded[::-1]
    print 'decoded:', ''.join(decoded)

if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Figure 6: Decode python script

Running the python script yeids “R_y0u_H0t_3n0ugH_t0_1gn1t3@flare-on.com”. Figure 7 shows the success message when inputing the correct flag. Easy peasy!

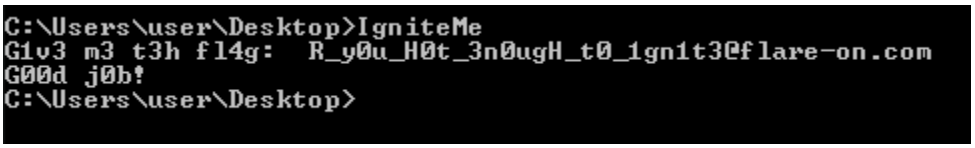

Figure 7: We're hot enough