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, IgniteMe.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.

![Example of supplying an incorrect flag to IgniteMe.exe](C:\Users\user\Desktop\IgniteMe.exe)

Figure 1: FailSafe!

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.

![Example of the output prompt, success message, and error message](Giv3 m3 t3h fl4g:
G00d j0b!
N0t t00 h0t R we? 7ry 4ga1nz p1zzz!)

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).
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:
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 decrements the current index. We can see that 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 variable 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 encoding 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 arithmetic operations. But, let’s walk through the function to fully understand what it does in Figure 5.
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 0x07000000. Shifting all the bits to the right one time results in 0x03800000. 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\x56\x09\x5F\x45\x47\x73'
    encoded+= '\x26\x0a\x0D\x13\x17\x48\x42\x01\x40\x4D\x0c\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))
    # 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 5: sub_401000

Figure 6: Decode python script
Running the python script yields “R_y0u_H0t_3n0ugH_t0_1gn1t3@flare-on.com”. Figure 7 shows the success message when inputing the correct flag. Easy peasy!

```
C:\Users\user\Desktop>IgniteMe
Give me the flag: R_y0u_H0t_3n0ugH_t0_1gn1t3@flare-on.com
Good job!
C:\Users\user\Desktop>
```

*Figure 7: We’re hot enough*