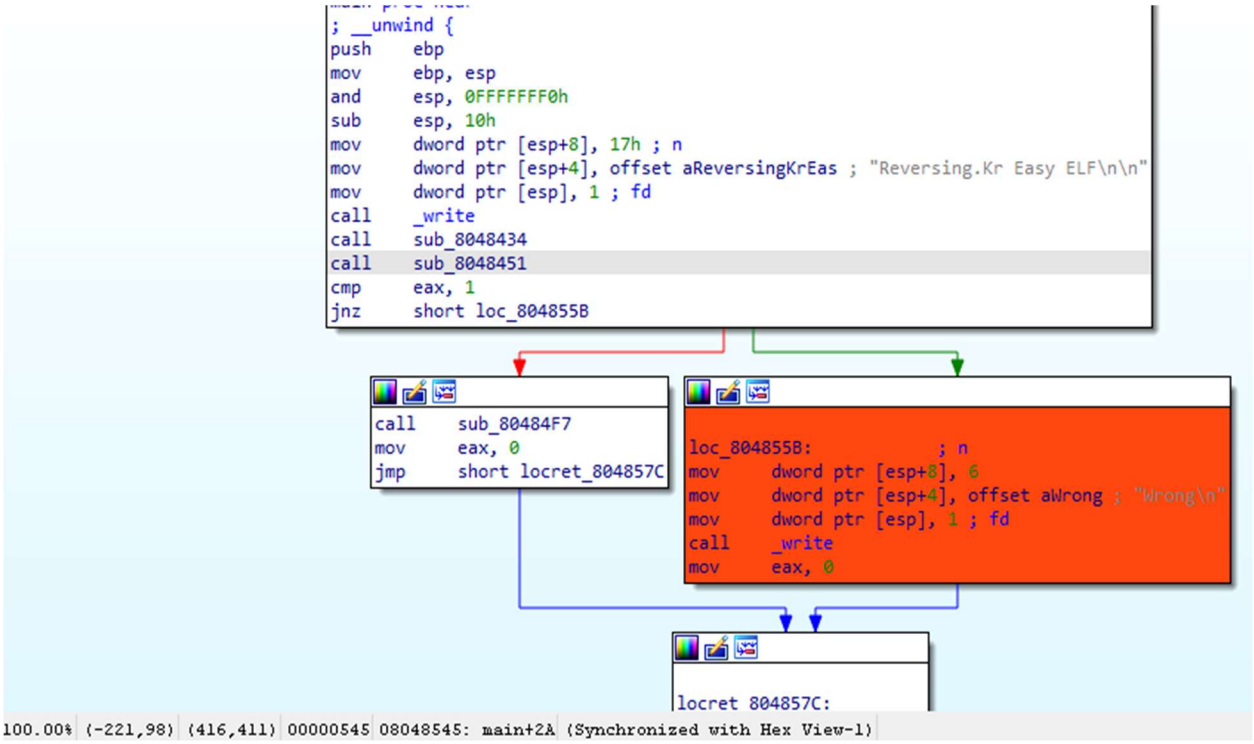
Reversing.kr write-ups

These are some of the write-ups I have written for the challenges I have solved in the ***reversing.kr*** website. The goal is to learn and improve the reverse engineering skills. The challenges rare a mix of Linux and windows executables.

1. Easy elf

The goal is to get the correct password.

We drag our binary in IDA.



From the above main function we can see we are calling some function. Our goal is to control our flow to the *sub_80484F7* which prints correct when provided with the right password.

The call to sub_8048451 function is the password checking algorithm. We can decompile the code.

```
1  _BOOL4 sub_8048451()
2  {
3    if ( byte_804A021 != '1' )
4      return 0;
5    byte_804A020 ^= 0x34u;
6    byte_804A022 ^= 0x32u;
7    byte_804A023 ^= 0x88u;
8    if ( byte_804A024 != 88 )
9      return 0;
10   if ( byte_804A025 )
11     return 0;
12   if ( byte_804A022 != 124 )
13     return 0;
14   if ( byte_804A020 == 120 )
15     return byte_804A023 == -35;
16   return 0;
17 }
```

From the decompiled code we can see we are doing some xoring of the use input and comparing with some value.

From the decompile code we can deduce the length of the password is starting from byte_804A020 to byte_804A025 buffer. We can write a simple python script and print the correct password.

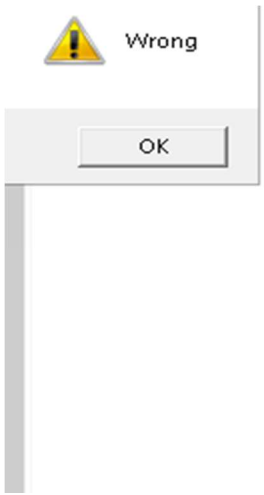
```
2   byte20 = 120 ^ 0x34
3   byte21= 0x31
4   byte22 = 124 ^ 0x32
5   byte23 = 0x88 ^ 0xDD
6   byte24 = 88
7
8   pass1 = chr(byte20)
9   pass2 = chr(byte21)
10  pass3 = chr(byte22)
11  pass4 = chr(byte23)
12  pass5 = chr(byte24)
13
14  print(pass1 + pass2 + pass3 +pass4 + pass5)
```

Running the above code we get the correct password. The correct password is ***L1NUX***

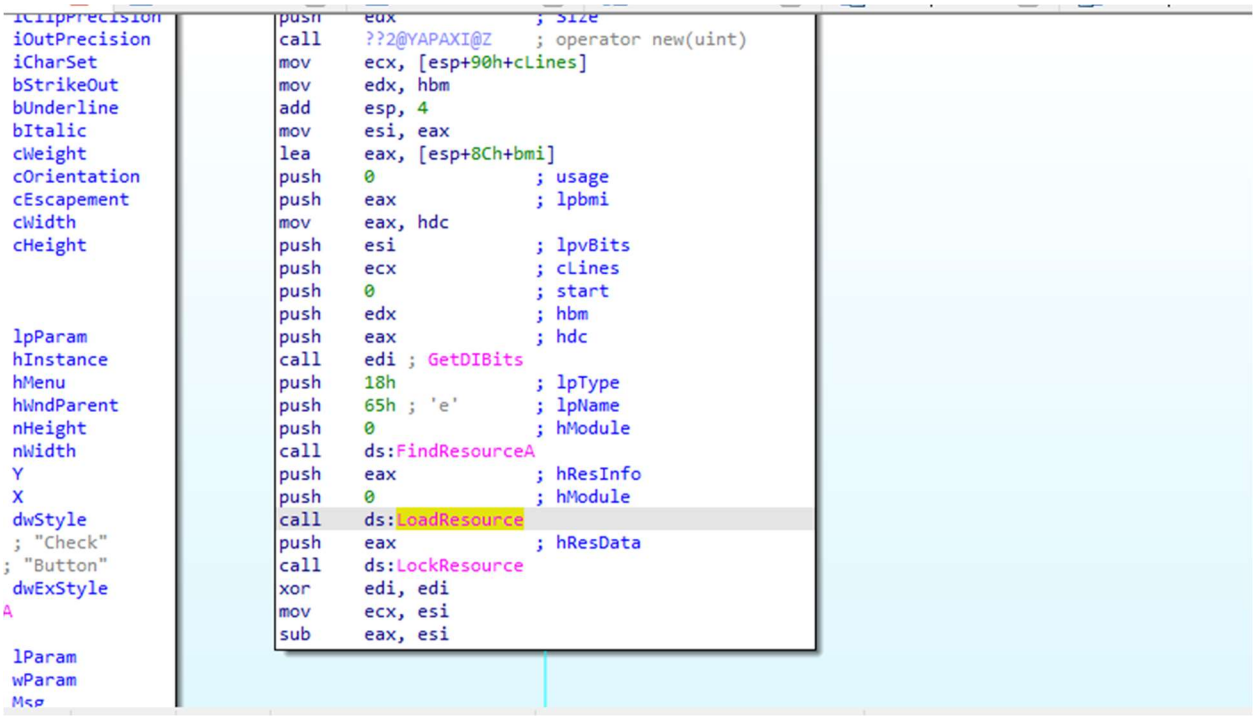
```
C:\Users\admin\Desktop
λ python easyelf.py
L1NUX

C:\Users\admin\Desktop
λ
```

2. Imageprc

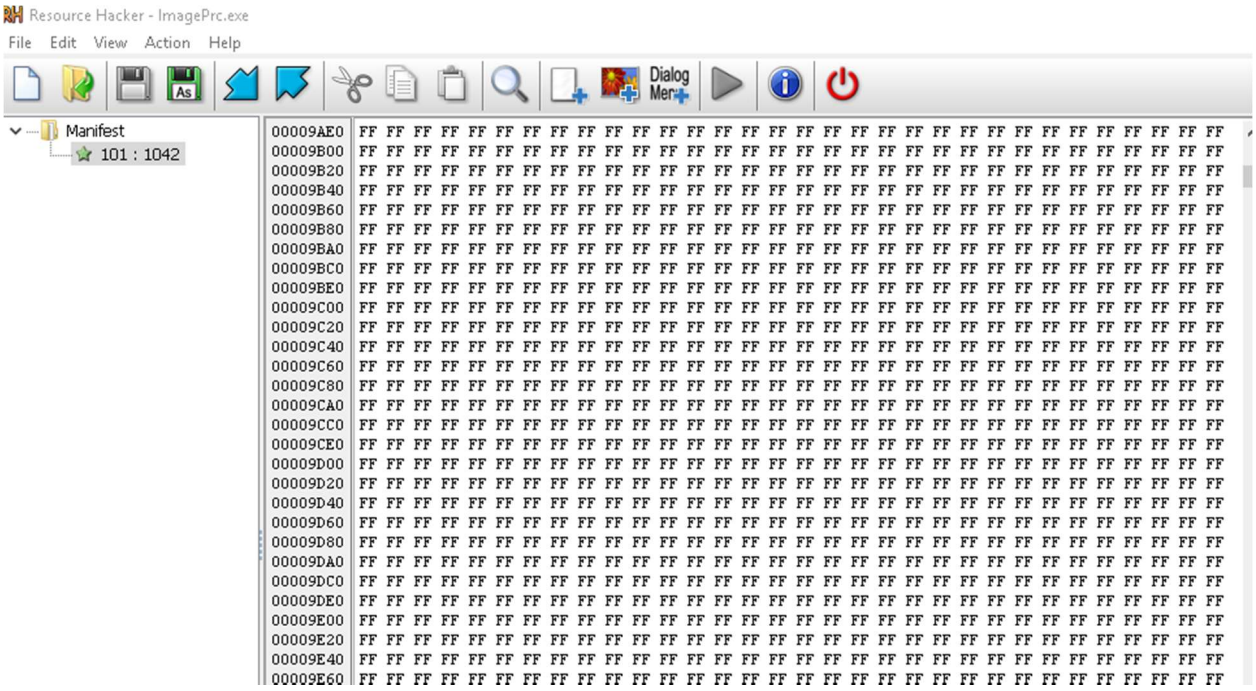
This was an interesting challenge. The goal was to draw a pixel in order to get the correct password. Guessing the password, dialog box displays wrong message.



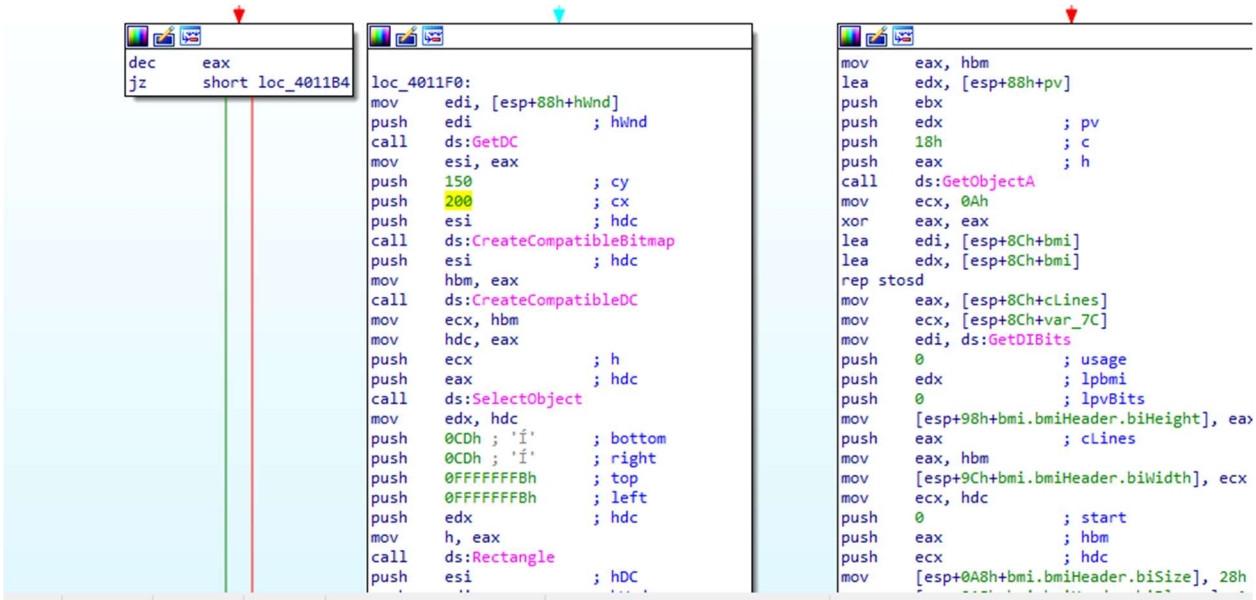
We drag binary in IDA Pro for further analysis.



From the above disassembly we can see the file is loading some resources for later use in the resource section. We can use resource hacker to extract the resource from the binary and save it as *.bin. Most malwares use resource section to store second stage dropper or load another executable. Resources can also be used to store strings that are accessed during runtime by the binary. I.e. string challenge by malwaretech.

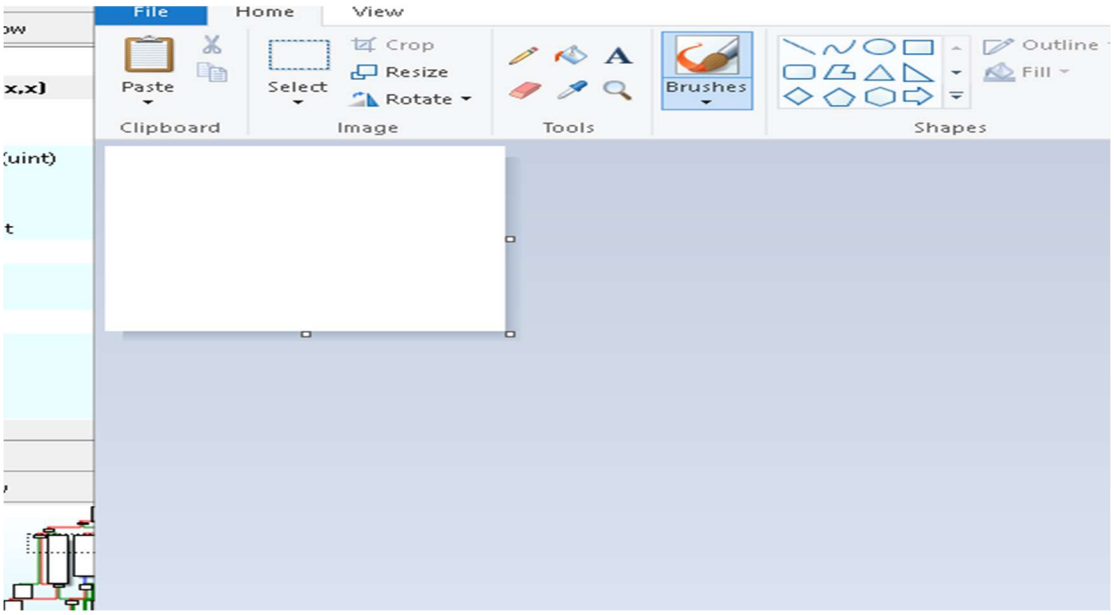


For further analysis we can see we are creating a bmp image of pixel of width 200 and height 150 pixel.



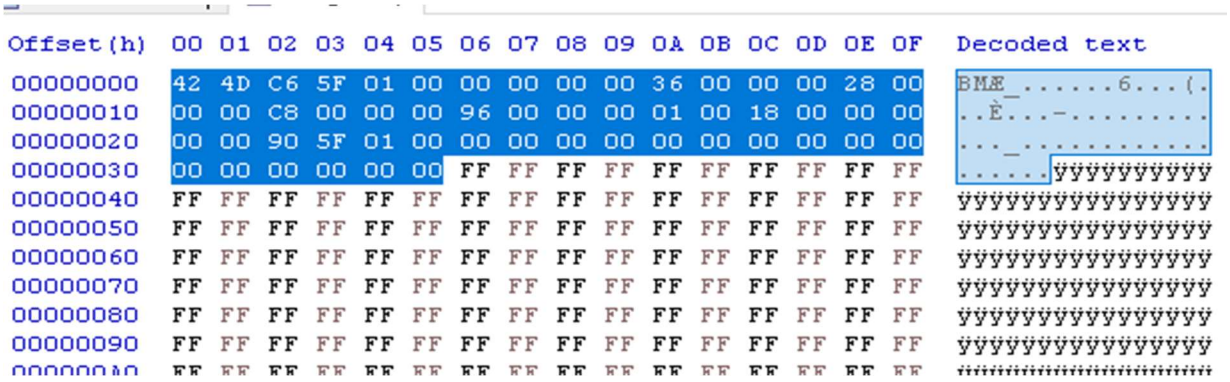
In our understanding of the disassembled code above we can create a bitmap using windows paint application and extract the file header and then insert to the resource section we extracted from the binary.

We will use hex editor for editing and fixing another bitmap.



We load both extracted resource and created bitmap in the HXD editor.

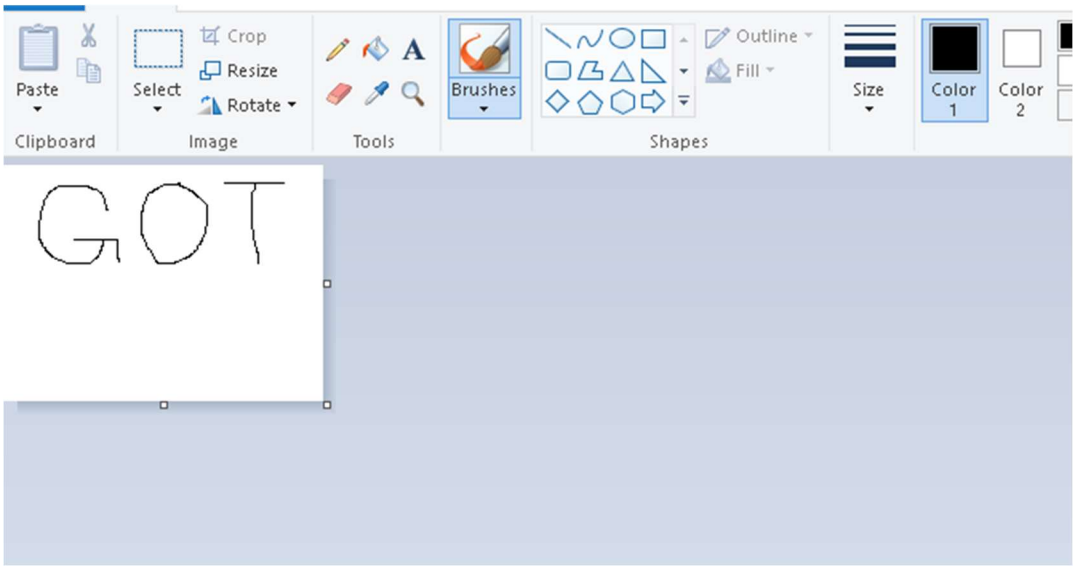
We extract the file header from the bitmap we have created.



We copy the selected file header section highlighted above and insert it in the extracted resource and save it as imagefixed.bmp image.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	42	4D	C6	5F	01	00	00	00	00	00	36	00	00	00	28	00	BMÆ_......6...(.
00000010	00	00	C8	00	00	00	96	00	00	00	01	00	18	00	00	00	..È...-.........
00000020	00	00	90	5F	01	00	00	00	00	00	00	00	00	00	00	00	..._............
00000030	00	00	00	00	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	......ÿÿÿÿÿÿÿÿÿÿ
00000040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
000000E0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ

Opening the resulting image we get a pixel image which is the correct password.



The password or the flag is ***GOT***. This challenge was good for understanding how resource section is used for storing other binaries or resource needed by the binary during runtime.