Blue Lock

(from bi0sCTF 2022)

Intro

Chall comes with two files: enc_file and malware.exe. By glancing at enc_file, we notice that it is a little sus.


```
f600 7204 4d 5a 90 0 3 0 2 de 0 33 0 4 0 0 0 ff ab ff b0 0 0 b8 0 0 0 0 f7 8f 0 0 0 40 0 83 58 0 0 0 0 0 0 0 0 26 0 0 0 8a ec 0 0 0 0 ac 0 de 0 5b 0 0 0 58 db 0 a2 0 0 0 0 c5 af 0 0
0 0 9e 0 0 0 0 0 0 1 99 0 32 8c b6 0 e 1f c1 ba e 0 17 37 53 b4 a3 86 9 df 8b cd 21 b8 1 7e c5 4c 16 cd b6 21 54 f6 68 69 d1 73 56 20 21 70 72 6f 67 72 72 61 6d ec 20 d4 87 bb 53 63
a2 ec 61 6e 6e 6f a1 74 20 62 86 65 2e 20 a 72 75 46 91 6e 58 20 69 6e 20 44 4f 53 20 6d 6f d5 64 65 2e d d d6 a 5d b7 3a 24 0 c8 0 54 0 0 e1 0 0 0 d9 7c b1 cc 79 b8 f5 ad 17 f8 aa
eb 50 f5 ad 17 eb f5 ad 17 eb fc d5 84 eb e7 c8 ad 17 eb a1 84 ce b3 97 d5 91 13 ea 63 46 55 ff 52 ad 32 17 eb 97 f0 35 6d 89 d5 14 ea f1 ad 17 eb 97 d5 12 93 ea ee ad 17 eb 27 3f 97
d5 16 ea f3 ad 17 eb e1 c6 4d 16 ea 24 f6 ad 17 eb 4f f5 ad 16 eb b4 62 d2 ad 17 ab 4c 5c eb 75 d4 12 ea f7 ad 17 ae 59 eb 75 6b d4 e8 eb f4 f2 ad c4 17 20 eb 75 d4 15 2a ea f4 ad 17
eb 52 69 f5 63 68 2e f5 ad 17 eb 0 b2 cc 0 0 b9 0 0 0 0 cd 0 0 0 0 44 0 0 0 0 0 0 0 0 0 0 1a 0 0 0 50 45 aa 19 0 0 f 64 d6 86 6 0 c1 91 2c a4 7 e8 c9 63 0 47 37 0 0 0 0 fe 0 0 a5 0
f0 ce b8 0 22 b8 71 b5 e6 0 b f 2 e 1f 87 75 0 a0 47 0 e3 0 0 5a 0 94 32 0 0 0 0 7e 0 3d d ac e9 1b 95 9f b1 0 0 0 10 0 0 6e 0 0 0 40 1 0 0 0 53 1e 0 d5 10 0 0 29 0 2 0 0 6 0 0 0 26
8f cd 0 0 20 0 0 7d 6 0 50 0 0 71 0 0 6e b5 0 0 53 0 30 1 17 0 0 4 0 0 6 0 0 0 0 3 0 25 ff 60 f6 81 0 0 10 0 5f f7 0 27 0 0 0 0 10 5c 0 0 0 ae 0 0 0 e6 0 43 0 65 10 c0 0 0 0 0 0 40 0
44 10 0 0 0 0 93 0 66 0 0 c4 0 0 ca 0 10 0 f4 0 20 0 0 0 0 0 0 7 0 0 0 fc ce 0 0 56 18 37 1 0 0 0 10 1 0 e0 96 1 0 0 0 0 1 65 0 40 27 e 0 0 0 0 0 0 0 a6 62 6e 0 0 d7 0 ab c3 0 83 3b
36 36 51 38 b 20 1 24 0 a8 19 0 d6 0 0 9f 9 40 3d b9 0 c6 44 fd 0 70 0 0 82 a7 0 0 0 0 3 0 0 0 0 4e bf 0 0 0 d0 0 ef d 8e 14 0 0 0 5f 0 0 0 0 0 0 0 0 0 c2 0 f3 0 5c b8 0 0 40 e2 3e 1
0 0 4b 0 0 25 0 0 0 0 0 0 0 79 b0 0 5f 3c 0 f8 4 c1 0 de 0 0 0 0 0 0 0 a0 0 0 0 0 4c 0 0 0 6c 0 e7 76 eb 1 0 0 0 0 0 0 0 79 0 0 45 56 0 2e b0 74 65 78 13 74 f5 78 0 0 0 7f 9f 0 0 0
10 0 0 d8 0 80 a0 0 0 5a 0 4 ba 0 0 0 1b 0 b2 0 0 7a ab b4 62 db 30 ea 0 0 0 77 0 0 38 0 4c 0 0 20 58 0 58 19 c3 0 60 2e 72 64 c 61 b8 74 61 0 16 f8 0 38 18 be 39 0 ef 3d 1f 0 0 b0 0
0 0 2f 3a 0 0 0 1a 95 a4 0 0 ec 0 0 0 7b 0 0 c2 0 0 0 c4 0 7d cf 26 55 0 0 0 40 df 0 0 40 af f2 2e 64 61 74 61 a1 0 8a 0 0 50 6c b 0 0 0 f0 0 0 0 4 0 0 e3 0 21 5 de 0 0 0 0 0 0 0 0 0
0 0 0 0 b0 0 21 40 94 77 0 c9 3e 0 c0 2e 44 70 64 61 74 61 0 0 40 91 e 0 0 0 42 0 1 0 6d 0 10 0 0 0 e2 0 0 90 0 0 f9 0 0 0 0 0 29 0 81 50 0 7c 9b f2 0 63 0 0 40 0 0 40 b0 2e 72 73 59
72 63 0 0 0 e0 1 0 0 0 10 90 1 0 0 2 9f 0 0 0 f2 0 6 0 0 0 83 0 be 0 0 57 0 0 0 0 0 7e fd 0 0 40 77 0 0 40 2e 2a 63 72 65 6c 6f 63 0 0 a8 42 7c 0 0 0 0 20 1 9 0 0 c 2 0 0 a9 b0 cf 0
```

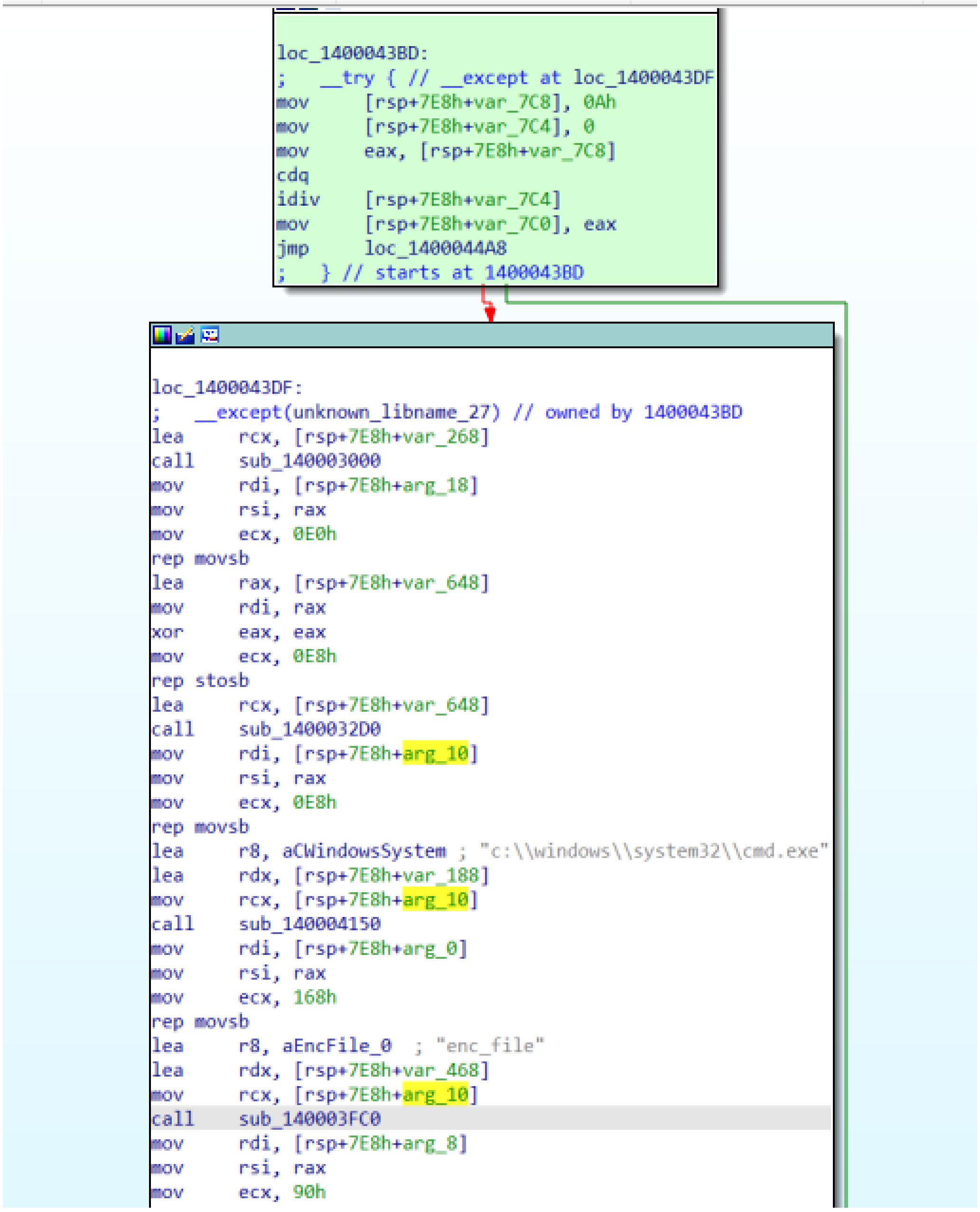

It consists of bytes (except first two hex) and specifically from the third number on (4d) it resembles some exe file(4d, 5a = MZ → DOS Header). If we try to write our bytes to a exe file the resulting file will be corrupted.

The Malware

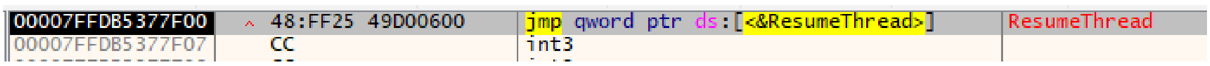
After some initial static analysis with IDA, we realize that the malware intentionally throws some exceptions:

```
__int64 sub_140004540()
{
  return 1600 / 0;
}
```

which makes decompilation harder.
The exception handlers can be viewed in disassembly and that is where the main activity is taking place.



So, after playing around a little bit, and by using some dynamic analysis, we notice that the malware opens cmd.exe and enc_file, reads from the latter and writes data to the first one. It then tries to give control to a thread in cmd.exe running the malicious code. That is a good point to intervene. So we intercept the cmd.exe process before being resumed and then step over ResumeThread.



Ok so, now we are in the cmd process and we notice yet another exception as well as two handlers

```
48:83EC 38              sub rsp,38
48:8D15 D5DAFFFF        lea rdx,qword ptr ds:[7FF68D6C6F50]
B9 01000000             mov ecx,1
FF15 9A1B0000           call qword ptr ds:[<&RtlAddVectoredExcep
48:8D15 83FDFFFF        lea rdx,qword ptr ds:[7FF68D6C9210]
33C9                    xor ecx,ecx
FF15 8B1B0000           call qword ptr ds:[<&RtlAddVectoredExcep
C74424 20 05000000      mov dword ptr ss:[rsp+20],5
8B4424 20               mov eax,dword ptr ss:[rsp+20]
99                      cdq
33C9                    xor ecx,ecx
F7F9                    idiv ecx
```

ok so the first one checks for existence of a file called flag and if it doesn't exist it outputs error and exits:

```
mov rax,qword ptr ds:[rax]
cmp dword ptr ds:[rax],C0000094
jne cmd.7FF68D6C706B
lea r8,qword ptr ds:[7FF68D6CB5A0]       00007FF68D6CB5A0:"rb"
lea rdx,qword ptr ds:[7FF68D6CB5A4]      00007FF68D6CB5A4:"flag"
lea rcx,qword ptr ss:[rsp+28]
call qword ptr ds:[<&fopen_s>]
test eax,eax
je cmd.7FF68D6C6FBE
mov ecx,2
call qword ptr ds:[<&__acrt_iob_func>]
lea rdx,qword ptr ds:[7FF68D6CB5AC]      00007FF68D6CB5AC:"Error\n"
mov rcx,rax
```

Else, it reads the contents into memory.

Now the second handler is more complicated:

In short words, it generates a key for encryption and then uses that to encrypt the data read from flag.txt. The encryption algorithm used is xxtea but i didn't figure that out at the time so I crafted my own sloppy decryptor.

This is a snapshot of my decryptor for when i needed to decipher the flag payload:

```python
for l in range(33):
    data = open("flag",'rb').read()

    data = [data[i:i+4] for i in range(0,len(data),4)]
    data = [data[i][::-1] for i in range(len(data))]
    if (l == 6):
        print(data[-1])

    data = [int(data[i].hex(),16) for i in range(len(data))]
    random_start = (0x9E3779B9 * l) & 0xffffffff
    random_3 = (random_start >> 2) & 3

    a3 = [0X35343736,0x31323131,0x36323735,0x36323439]
    for i in range(l):


        data[-1] -= ((( data[-2] ^ (a3[(random_3 ^ (len(data) -1) & 3)]) ) + (data[0] ^ random_start) ) ^ (((16*data[-2]) ^ (data[0] >> 3) ) + ((4 * data[0]  ) ^ (data[-2] >> 5) )) )
        data[-1] &= 0xffffffff


        for j in range(len(data)-2,-1,-1):
            data[j] -= (((data[j-1] ^   (a3[(random_3 ^ j & 3)] ) ) + (data[j+1] ^ random_start)) ^ (((16 * data[j-1])^ (data[j+1]>>3)) + ((4* data[j+1]) ^ (data[j-1] >> 5))))
            data[j] &= 0xffffffff

        random_start -= 0x9E3779B9 & 0xffffffff
        random_3 = (random_start >> 2) & 3
    lol = [format(da, '08x') for da in data]
    if (l == 6 ):
        print(lol)

        epitelous = (bytes.fromhex("".join([bytes.fromhex(lol[i])[::-1].hex() for i in range(len(lol))])))
        print(epitelous)
        f = open("hah.png","wb")
        f.write(epitelous)
```

(Short note here: In order to get the key of the encryption, i had to attach to cmd.exe after it executed the key generation algorithm, i have no idea why. I 've talked to one creator who had to say this to help me:

> The source code has std::string cast, I'm still unaware why it changes, but i suspect there is some underlying permission defined where it states that heap regions must be destroyed for a windows program while a debugger is attached to them
>
> I'm yet to find proof for this hunch tho

)

Next up, the encrypted data gets written to something like a copy of cmd, and the output is stored in a similar fashion to a new enc_file. Now, it is easier to find out what the first two numbers in enc_file mean.
The first number (e.g. 0xf600) is the actual useful size of the file. The second one (eg. 0x7204) is the superfluous bytes added randomly(?) by the malware running on cmd.exe

Putting it all together

So, we know that whoever created the enc_file, used a 0xf600 bytes program and then used the malware in some fashion to encrypt 0x7204 other bytes to it. If only we knew the order and positions of that bytes. But wait a minute we know, because in order for the enc_file to be written to cmd.exe and be functional, it needs to have its gibberish removed somehow. After careful examination, we find a function that does exactly that, store the position of superfluous bytes, e.g.:

```
C78424 48C30100 0364( mov dword ptr ss:[rsp+1C348],16403
C78424 4CC30100 0664( mov dword ptr ss:[rsp+1C34C],16406
C78424 50C30100 0764( mov dword ptr ss:[rsp+1C350],16407
C78424 54C30100 0A64( mov dword ptr ss:[rsp+1C354],1640A
C78424 58C30100 0B64( mov dword ptr ss:[rsp+1C358],1640B
C78424 5CC30100 0D64( mov dword ptr ss:[rsp+1C35C],1640D
C78424 60C30100 0E64( mov dword ptr ss:[rsp+1C360],1640E
C78424 64C30100 1364( mov dword ptr ss:[rsp+1C364],16413
C78424 68C30100 1464( mov dword ptr ss:[rsp+1C368],16414
C78424 6CC30100 1564( mov dword ptr ss:[rsp+1C36C],16415
C78424 70C30100 1664( mov dword ptr ss:[rsp+1C370],16416
C78424 74C30100 1964( mov dword ptr ss:[rsp+1C374],16419
C78424 78C30100 1A64( mov dword ptr ss:[rsp+1C378],1641A
C78424 7CC30100 1B64( mov dword ptr ss:[rsp+1C37C],1641B
C78424 80C30100 1E64( mov dword ptr ss:[rsp+1C380],1641E
C78424 84C30100 2164( mov dword ptr ss:[rsp+1C384],16421
C78424 88C30100 2564( mov dword ptr ss:[rsp+1C388],16425
C78424 8CC30100 2664( mov dword ptr ss:[rsp+1C38C],16426
C78424 90C30100 2764( mov dword ptr ss:[rsp+1C390],16427
C78424 94C30100 2A64( mov dword ptr ss:[rsp+1C394],1642A
C78424 98C30100 2D64( mov dword ptr ss:[rsp+1C398],1642D
C78424 9CC30100 2E64( mov dword ptr ss:[rsp+1C39C],1642E
C78424 A0C30100 2F64( mov dword ptr ss:[rsp+1C3A0],1642F
C78424 A4C30100 3064( mov dword ptr ss:[rsp+1C3A4],16430
C78424 A8C30100 3264( mov dword ptr ss:[rsp+1C3A8],16432
C78424 ACC30100 3A64( mov dword ptr ss:[rsp+1C3AC],1643A
C78424 B0C30100 3C64( mov dword ptr ss:[rsp+1C3B0],1643C
C78424 B4C30100 3D64( mov dword ptr ss:[rsp+1C3B4],1643D
C78424 B8C30100 4364( mov dword ptr ss:[rsp+1C3B8],16443
C78424 BCC30100 4A64( mov dword ptr ss:[rsp+1C3BC],1644A
C78424 C0C30100 4B64( mov dword ptr ss:[rsp+1C3C0],1644B
C78424 C4C30100 4D64( mov dword ptr ss:[rsp+1C3C4],1644D
```

The bytes are copied to stack so they can be obtained and isolated. We then extract the bytes at the positions we got earlier.
Now we have our encrypted bytes and a decryptor at our disposal, so we combine those two and get the following png:

We got lucky here, as the order of the bytes wasn't changed.
Overall a great challenge!!