<u>Warkah Terlarang (Malware) Write up.</u>

Created By : Muhammad Hafiz Izzuddin Bin Elias (2023436426)

Challenge Name : Warkah Terlarang

Category : Reverse Engineer

Level : Medium

File : warkah.exe

Descriptions:

The challenge binary prompts the user to input a flag.

If the correct flag is provided, the program prints “Correct!”, otherwise “Wrong!”.

The flag is not stored in plaintext and must be recovered via static analysis.

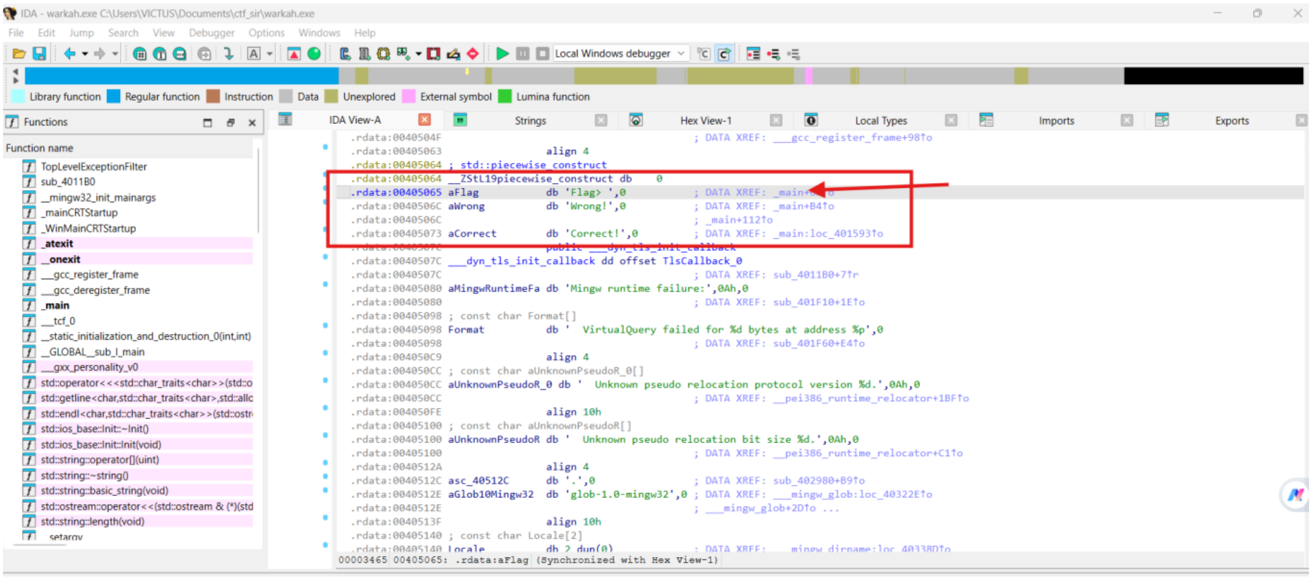
Write up:

```
hafiz@LAPTOP-MJFKGEDL:/mnt/c/Users/VICTUS/Documents/ctf_sir$ file warkah.exe
warkah.exe: PE32 executable (console) Intel 80386, for MS Windows, 13 sections
```

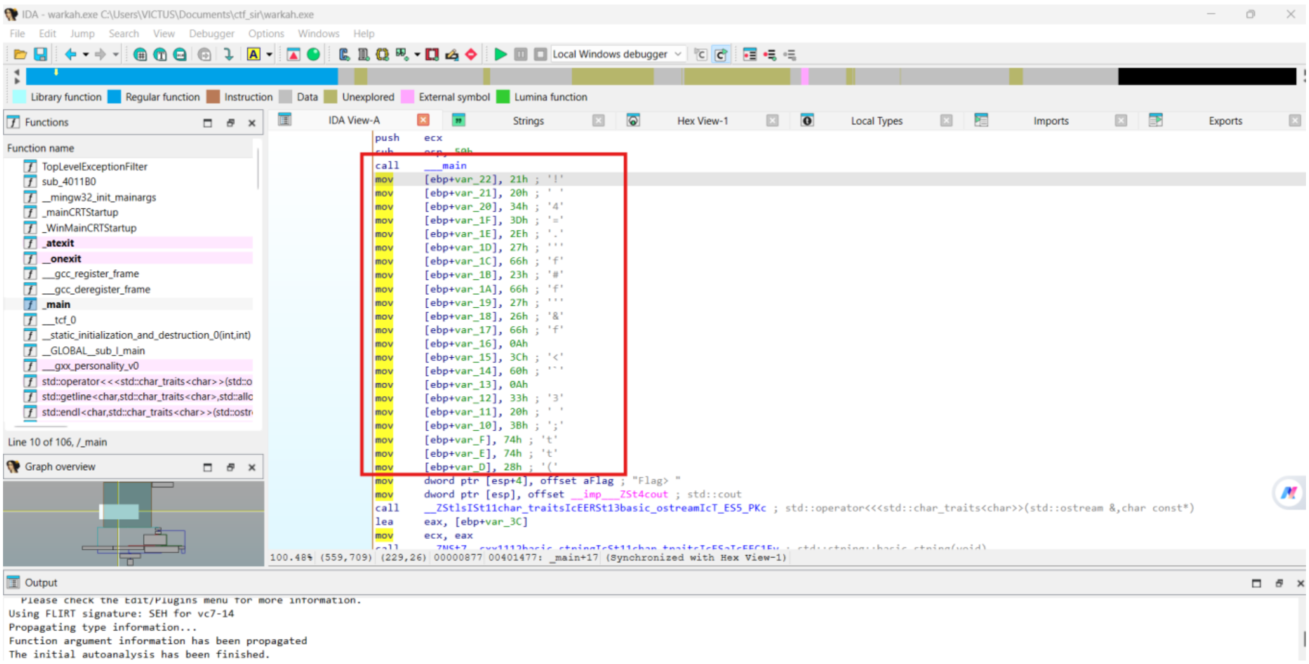
We first identify the binary using the *file* command. The output shows that warkah.exe is a 32-bit Windows PE console executable for Intel x86 architecture. This indicates that the program should be analyzed using 32-bit reverse-engineering tools such as IDA Free or Ghidra. Since it is a console application, we expect user input and output through standard I/O.

```
>.Q@
[^_]
>.Q@
5lp@
[^_]
UWVS
[^_]
libgcc_s_dw2-1.dll
__register_frame_info
__deregister_frame_info
libgcj-16.dll
_Jv_RegisterClasses
Flag>
Wrong!
Correct!
Mingw runtime failure:
  VirtualQuery failed for %d bytes at address %p
  Unknown pseudo relocation protocol version %d.
  Unknown pseudo relocation bit size %d.
glob-1.0-mingw32
GCC: (GNU) 6.3.0
```

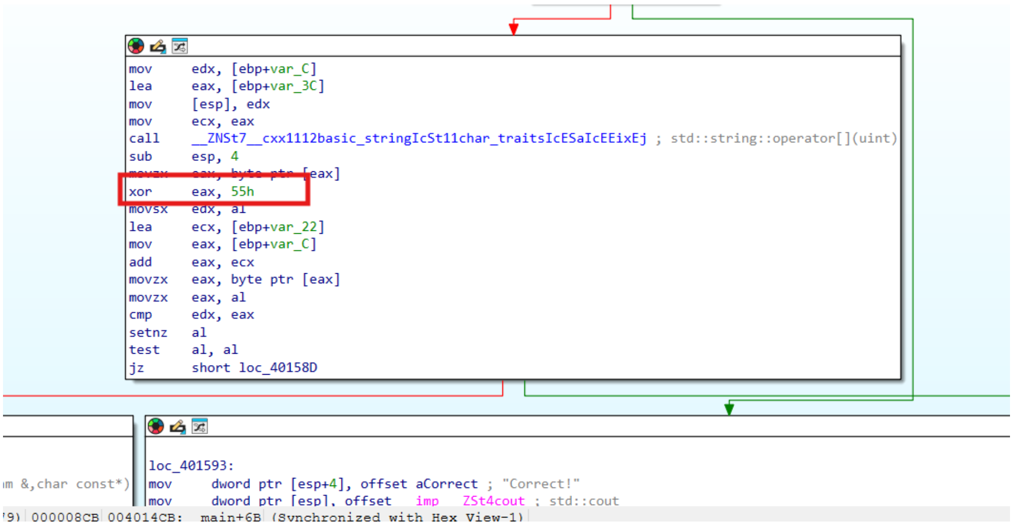
Running *strings* command on the binary reveals only basic runtime libraries and user interaction messages such as Flag>, Wrong!, and Correct!. No plaintext flag or obvious encoded data is present, indicating that the flag is validated programmatically rather than stored directly. This suggests further static analysis is required.



After load warkah.exe into IDA, we can see the strings **Flag>** has been refer to *main* function. Therefore we can see how the code works by using IDA flowchart.



From main function, we can see some data have been move to the memory.



After analysing main function, we got something interesting here. This file are using xor operation with hex 55.

So what we can do here? We actually can reverse the data that have been move to memory with a xor operation hex 55 and maybe we can uncover what the flag is supposed to be.

Now we can use IDA features that convert the flowchart int pseudocode, so it became more easier for us to understand.

```
  _BYTE v6[26]; // [esp+0h] [ebp-3Ch] BYREF
  _BYTE v7[22]; // [esp+1Ah] [ebp-22h] BYREF
  int i; // [esp+30h] [ebp-Ch]
  int *p_argc; // [esp+34h] [ebp-8h]

  p_argc = &argc;
  __main();
  qmemcpy(v7, "! 4=.'f#f'&f\n<`\n3 ;tt(", sizeof(v7));
  std::operator<<<std::char_traits<char>>(&std::cout, "Flag> ");
  std::string::basic_string(v6);
  std::getline<char,std::char_traits<char>,std::allocator<char>>(&std::cin, v6);
  if ( std::string::length(v6) == 22 )
  {
    for ( i = 0; i <= 21; ++i )
    {
      v3 = (_BYTE *)std::string::operator[](v6, i);
      if ( (char)(*v3 ^ 0x55) != (unsigned __int8)v7[i] )
        goto LABEL_2;
    }
    v4 = std::operator<<<std::char_traits<char>>(&std::cout, "Correct!");
    std::ostream::operator<<(v4, &std::endl<char,std::char_traits<char>>);
  }
  else
  {
LABEL_2:
    std::operator<<<std::char_traits<char>>(&std::cout, "Wrong!");
  }
  std::string::~string(v6);
  return 0;
}
```

After observing the pseudocode we can conclude:

1. The required flag length is 22 characters

2. The array v7 contains encoded bytes

3. Each input character is XORed with 0x55

4. The result is compared directly against the stored bytes

This means the validation logic is :

```
(flag[i] ^ 0x55) == encoded[i]
```

And we can reverse it like this :

```
(encoded[i] ^ 0x55) == flag[i]
```

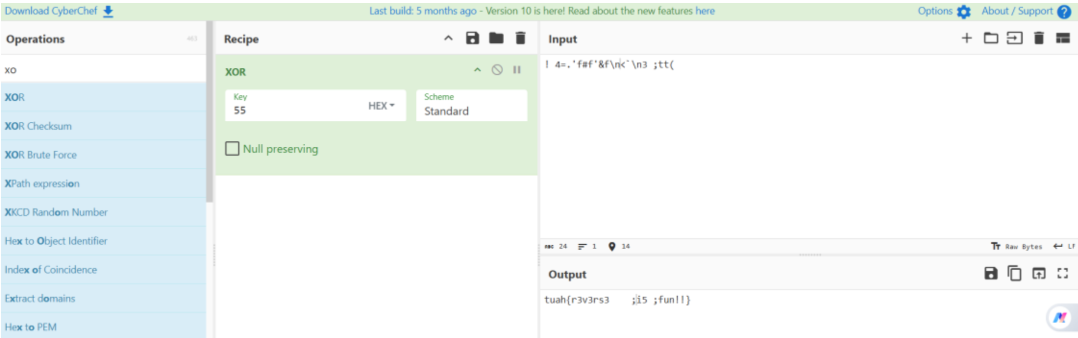
IDA show a encoded bytes as strings that actually represent as raw bytes value :

"! 4=.'f#f'&f\n<`\n3 ;tt("

Next step is to uncover the flag, there are 2 options here which is using online tool like CyberChef or manually reverse the bytes value using python. I will show you both solution.

CyberChef

CyberChef is pretty straight forward, just select xor operation and put the hex value key to 55 which we got it from static analysis before.



And that's it, we got the flag which is **tuah{r3v3rs3_i5_fun!!}**

Solve.py

Let's move on to another approach. We can manually reverse the bytes value using python. In my case I used this following code.

```
data = "! 4=.'f#f'&f\n<`\n3 ;tt("

flag = bytearray()

for ch in data:
    decoded = ord(ch) ^ 0x55
    flag.append(decoded)

print(flag.decode("utf-8"))
```

Then we got the flag **tuah{r3v3rs3_i5_fun!!}.**