

After completing the TryHackMe Ghidra tutorial, I decided I wanted to continue down the path of reverse engineering and test my skills. This box was the perfect intro to crackme's and allowed me to continue familiarizing myself with both Ghidra and reverse engineering. Note: this writeup will not contain any of the flags needed to complete the room.

Link to room: <https://tryhackme.com/room/basicmalwarere>

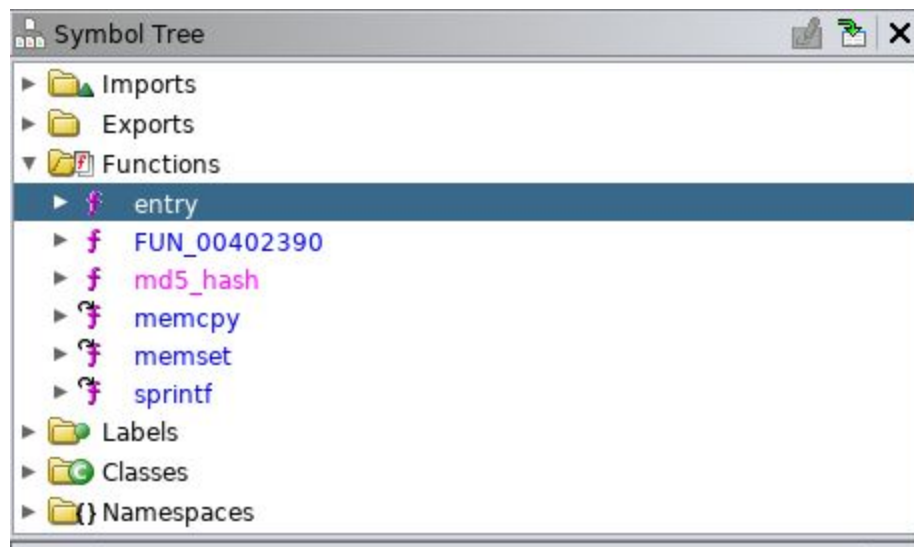
Strings1.exe

Disclaimer: no pictures have been included for this level, because they would very quickly give away the flag.

1. The first thing I did was create a new Ghidra project and import this binary into it (obviously)
2. Next, I went ahead and looked at the functions. One of interests that stuck out to me was a function called entry, I double clicked it to have it opened in the decompile window.
3. Looking at the C code, we see a character pointer named lpText. This character pointer is instantiated and then set equal to an md5_hash. The md5_hash is run on a defined string.
4. Double clicking the defined sting, we are taken to the spot in the code where it is defined and BOOM we have our first flag.
 - a. Quick side note, you input the flag as is into the room. This should be obvious, but at first I actually ran it through an online md5 hash converter lol.

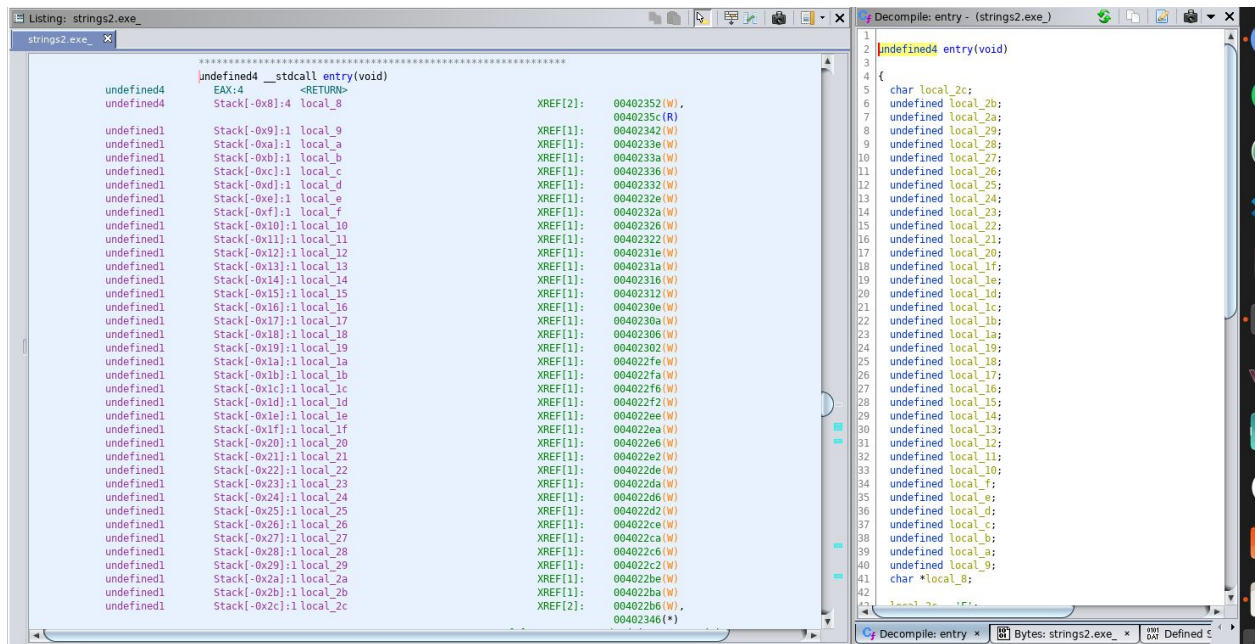
Strings2.exe

1. Again, the first thing I did was import this binary into my current project and analyze it.
2. A quick look at the functions in the symbol tree shows a very similar layout to that of the first executable. Since entry was where the magic happened last time, I head over there



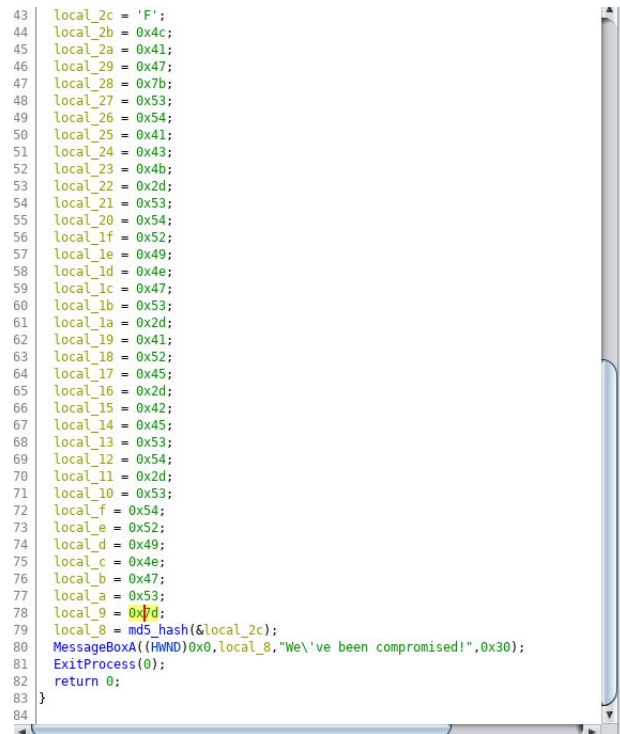
3. Now this part requires a little bit of assembly and stack knowledge. Looking at the C code we see a bunch of variables defined. First we have a char and then the rest are

undefined (except the last one, we will get to that in a minute). If we take a look at the main window, we can see how this definitions are added to the stack, growing up with each new definition.



4. Next, each of these variables is given an assignments. The first char is assigned to 'F' and all of the undefined are given hex values. A little intuition implies that these Hex values are most likely ASCII characters since each one is exactly one byte. As you can see, the last defined variable on the stack is a character pointer. It is then assigned to be a Hash of the string starting at the memory address of the char local_2c. This means

that it will hash the entire string starting at the bottom of the stack and moving up.



```

43 | local_2c = 'F';
44 | local_2b = 0x4c;
45 | local_2a = 0x41;
46 | local_29 = 0x47;
47 | local_28 = 0x7b;
48 | local_27 = 0x53;
49 | local_26 = 0x54;
50 | local_25 = 0x41;
51 | local_24 = 0x43;
52 | local_23 = 0x4b;
53 | local_22 = 0x2d;
54 | local_21 = 0x53;
55 | local_20 = 0x54;
56 | local_1f = 0x52;
57 | local_1e = 0x49;
58 | local_1d = 0x4e;
59 | local_1c = 0x47;
60 | local_1b = 0x53;
61 | local_1a = 0x2d;
62 | local_19 = 0x41;
63 | local_18 = 0x52;
64 | local_17 = 0x45;
65 | local_16 = 0x2d;
66 | local_15 = 0x42;
67 | local_14 = 0x45;
68 | local_13 = 0x53;
69 | local_12 = 0x54;
70 | local_11 = 0x2d;
71 | local_10 = 0x53;
72 | local_9 = 0x54;
73 | local_8 = 0x52;
74 | local_7 = 0x49;
75 | local_6 = 0x4e;
76 | local_5 = 0x47;
77 | local_4 = 0x53;
78 | local_3 = 0x7d;
79 | local_8 = md5_hash(&local_2c);
80 | MessageBoxA((HWND)0x0, local_8, "We've been compromised!", 0x30);
81 | ExitProcess(0);
82 | return 0;
83 | }
84 |

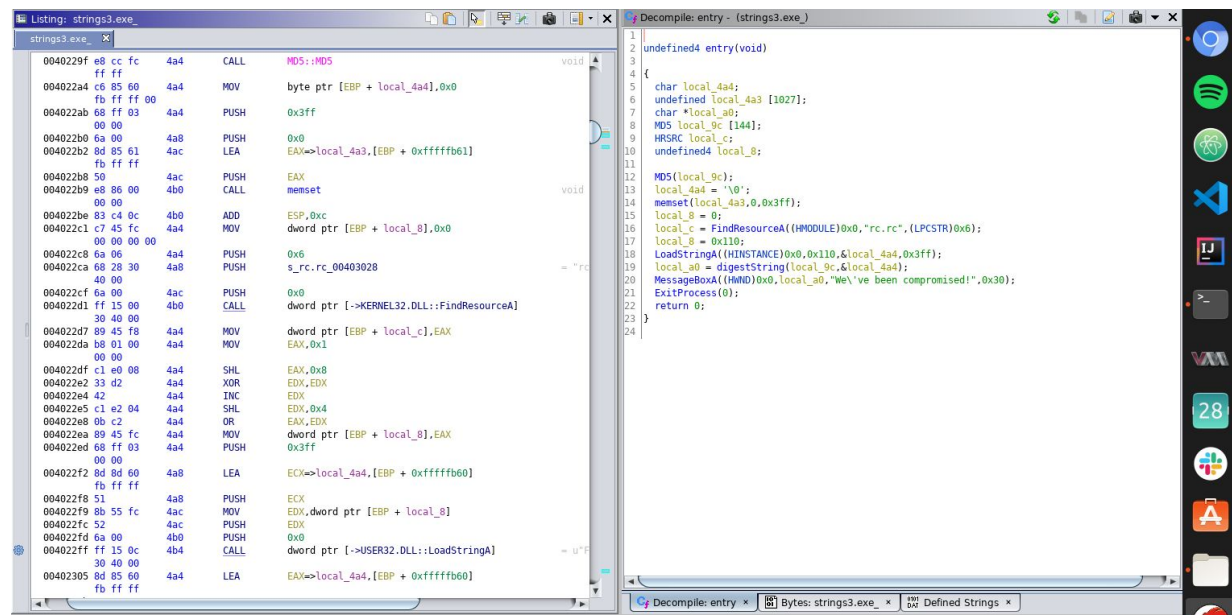
```

5. So, if we start with the character 'F' and then translate each of the hex values into their respective characters, we will get the string that is being hashed, aka our flag.

Strings3.exe

1. Open this project, go the functions, see entry and double click it. Same as before.
2. This time, entry is a bit different and maybe more complicated
 - a. Note: I am going to do my best to explain what happens here but to be honest, Ghidra analyzes it and puts the flag right there in the main window. Kinda crazy how well it works, and you could get away with this one without any knowledge of what is going on.
3. Okay, so first we have quite a few defined variables. None of these were super interesting. I was hopeful that maybe one of them would reference a defined string, but no dice.
4. Next I look at the body of the function, and we see a call another functions FindResourceA() and LoadStringA(). A quick look at the assembly and we see FindResourceA() is from the Kernel32 Library and LoadStringA() is from User32 Library. I quickly found [this](#) reference to try and gain an understanding of LoadStringA(). The most relevant thing it seems is that the string is being loaded from its reference and stored at the variable local_4a4.
 - a. The assembly code shows the LoadStringA function, and then right next to it shows what it actually loads (aka the flag). I have cut this out for obvious

reasons.



- We know the identifier for the string is hex value 0x110 as seen in the parameters for the function call. Now if we open up the defined strings and click on the first one, we can see the start of the flag table. Included on the far right is the string id. 0x110 translates to 272 in decimal. Scroll down to 272 and that is your final flag :)

