Write-up for SuperSecureVault
Dhruv Meena

The Challenge

I think this was the simplest challenge in CTM_Assignment_2024-25. This was my first reverse engineering challenge that I completed by myself. I knew what ***ghidra*** was before I began the assignment and so I instantly knew that I had to use it, although it was not the first tool that I used. The biggest struggle I had at first was to run the `vaultdoor` file. It kept saying **`permission denied`** whenever I ran `./vaultdoor`. Alas I solved this error and then the entire challenge.

Running the file

To fix the **`permission denied`** error, I remembered something from a *John Hammond* youtube video. We have to make it executable by running **`chmod +x <file_name>`** on it. And voila! It works! Now all we need to do is analyze the binary in ghidra and then somehow find the flag in it, nothing too difficult.

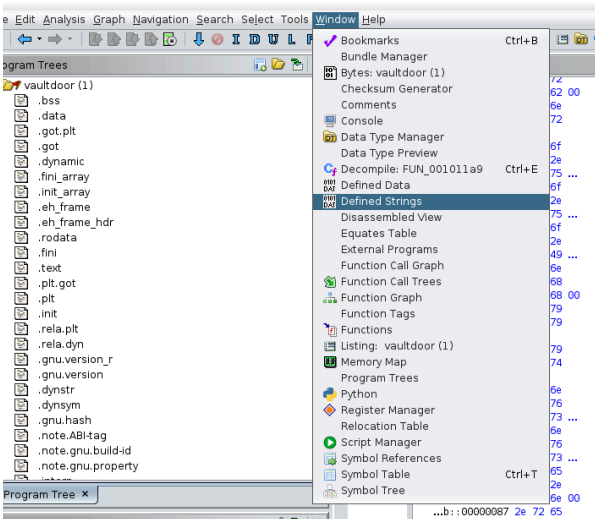
Ghidra

Before I tried ghidra, I ran **`file vaultdoor`** to check what sort of file it was.

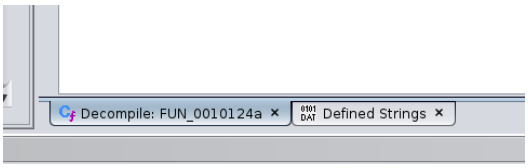
```
root@byzantiumvirus-exe:/csec/supersecurevault# file vaultdoor
vaultdoor: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
 BuildID[sha1]=1ee5bda013b310bbffa10d6b8499f53d975aeb6d, for GNU/Linux 3.2.0, stripped
```

It turned out to be an ELF (Executable and Linkable Format) as expected but also a PIE which is a position independent executable. I have no idea what that means so moving on. I ran **`readelf -a`** on the executable because I had seen someone do it in a youtube video. I did not find anything of importance in all the junk that displayed on the terminal.

Finally, I downloaded ghidra, made a project, imported the binary and started exploring around. Again, I took the help of John's tutorial on youtube to guide me. When the binary was loaded on to ghidra, we run the auto-analyzer with default settings and then go to **Windows>Defined Strings**.On the right-hand side a panel pulls up which as all the defined strings. I found the most important string of them all, "Enter the password" and clicked to see where it was in the code.



This bit is not of much use to me since I can not understand it very well so I clicked on the *decompile* tab on the right-hand side panel to see the code in, what I guess, is *C* language.



Decompile: FUN_0010124a - (vaultdoor (1))

```
1
2  undefined8 FUN_0010124a(void)
3
4  {
5    int iVar1;
6    size_t sVar2;
7    char local_58 [72];
8    char *local_10;
9
10   puts("Enter the password");
11   fgets(local_58,0x40,stdin);
12   sVar2 = strcspn(local_58,"\n");
13   local_58[sVar2] = '\0';
14   iVar1 = strcmp(local_58,"Sup3rS3cr3tP@ass");
15   if (iVar1 == 0) {
16     local_10 = (char *)FUN_001011a9();
17   }
18   else {
19     local_10 = "Noob";
20   }
21   puts(local_10);
22   free(local_10);
23   return 0;
24 }
```

I found something interesting here. `Sup3rS3cr3tP@ass` seems like the password but is definitely a **red herring**. I ran `vaultdoor` and put this in as the password. As expected, no flag came out (although the program did call me a noob).

A bunch of local variables are declared within this function and a mention of another interesting and suspicious function, `FUN_001011a9();`

I clicked on this function to see what it was. Lo and behold! I found a way to extract the flag!

Let's explore what this program is doing. First thing to notice is that it doesn't take in any arguments. Then we have 2 void pointers declared and an integer variable `local_c`. There is memory being allocated of the size `0x1b` bytes which are being pointed to by both `__dest` and `pvVar1` it seems. Ignoring the if statement we go directly to the `memcpy` function call. There's a reference to a memory

Decompile: FUN_001011a9 - (vaultdoor (1))

```
1
2  char * FUN_001011a9(void)
3
4  {
5    void *__dest;
6    void *pvVar1;
7    int local_c;
8
9    __dest = malloc(0x1b);
10   if (__dest == (void *)0x0) {
11     puts("Memory allocation failed.");
12                     /* WARNING: Subroutine does not return */
13     exit(1);
14   }
15   pvVar1 = malloc(0x1b);
16   memcpy(__dest,&DAT_00102010,0x1b);
17   for (local_c = 0; local_c < 0x1b; local_c = local_c + 1) {
18     *(byte *)((long)pvVar1 + (long)local_c) = *(byte *)((long)__dest + (long)local_c) ^ 0x6b;
19   }
20   return "tomper!";
21 }
22
```

location by `DAT_0010201e` which is where we will find our password or flag.

My Struggle and its result

This is where I spent most of my time. I got to the `DAT_0010201e` location by following the youtube tutorial but this is as far as it went. By this point, my guy John Hammond had already found his flag so I was wondering what I did wrong. It was only after 2 hours, when the sun came out again and my eyes were heavy with sleep that I noticed the for loop. All this time I kept looking for the flag in the assembly code, around the data reference point and the function's location.

The for loop runs `0x1b` times. And in each iteration it takes in individual bytes at the location pointed to by `pvVar1` and `__dest`, which are offset by `local_c` to iterate through all the bytes at those locations and then XOR them with `0x6b`. I took the data at the pointed destination and XORed it myself using a python script,

```
ENCRYPTED_DATA = [
      0x0d, 0x07, 0x0a, 0x0c, 0x10, 0x0c, 0x5b, 0x5b, 0x0f, 0x34, 0x01,
0x5b, 0x09, 0x34, 0x2b, 0x1f, 0x3f, 0x1e, 0x05, 0x1b, 0x2b, 0x08, 0x00,
0x5a, 0x05, 0x0c, 0x16, 0x00
]

XOR_VALUE = 0x6b

FLAG = ''.join(chr(BYTE ^ XOR_VALUE) for BYTE in ENCRYPTED_DATA)

print(FLAG)
```

After XOR operation, the `chr` function converts it to characters. Running this on terminal gives the flag as output.

Flag: **`flag{g00d_j0b_@tTunp@ck1ng}`**