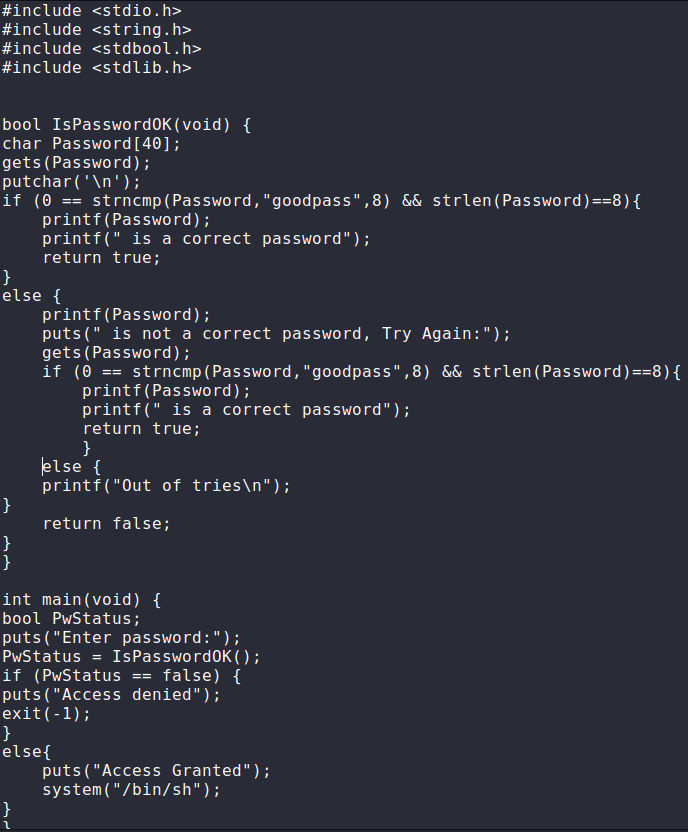
**Buffer Overflow**

In this report, I will be explaining the buffer overflow vulnerability, I will write a vulnerable C code, then compile it, then try to exploit it using only information that can be gained from the binary (No source code examination during exploitation).

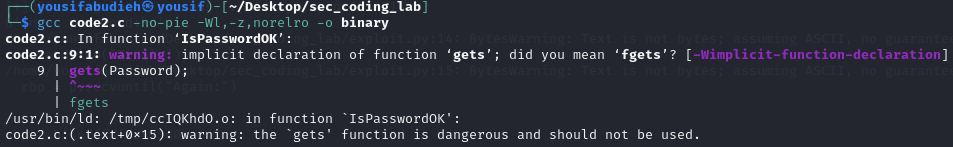
This is the C code



In this code, the user has 2 attempts to enter the password, and if they fail, they will get a “Access Denied” message, otherwise, they will gain access to the system and be able to execute shell commands.

**Note: The code has a vulnerability other than the buffer overflow vulnerability, it is the fact that the password is hard coded, meaning that we can recover the password from the binary simply by using the “strings” built in kali linux command, but in this report, I will be focusing on buffer overflow.**

I compiled the code using gcc code2.c -no-pie -Wl,-z,norelro -o binary



-no-pie : to keep addresses the same with every execution.

-Wl,-z,norelro : to turn off relocation read only

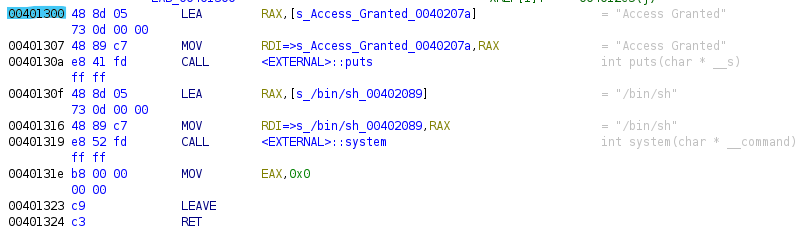
-o binary : name of executable.

I will now conduct the attack using only the binary (no info gathered from source code)

I now opened the binary with ghidra decompiler/disassembler to get the address where the access is granted.

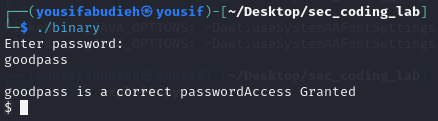
Address I want to go to: 0x00401300

**Note: This instruction copies the address of the string “Access Granted” and places it in the RAX register for it to be passed to the function puts, which will print “Access Granted” to the user before giving them access to execute shell commands. Also Note that these instructions never use the RBP register, keep that in mind, I’ll come back to it later.**

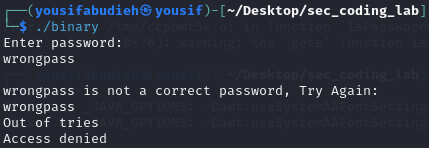


To know how to write the script, we need to know how the program runs in the terminal.

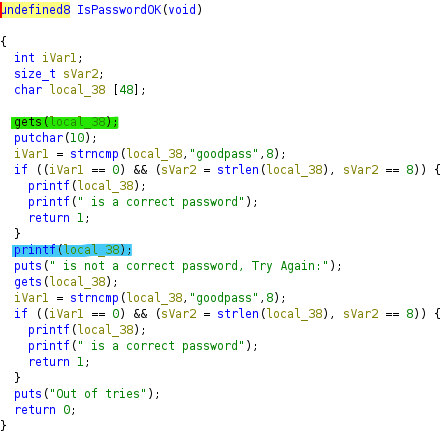
When I enter the correct password:



Wrong password:



Part of decompiled code:



Information gathered:

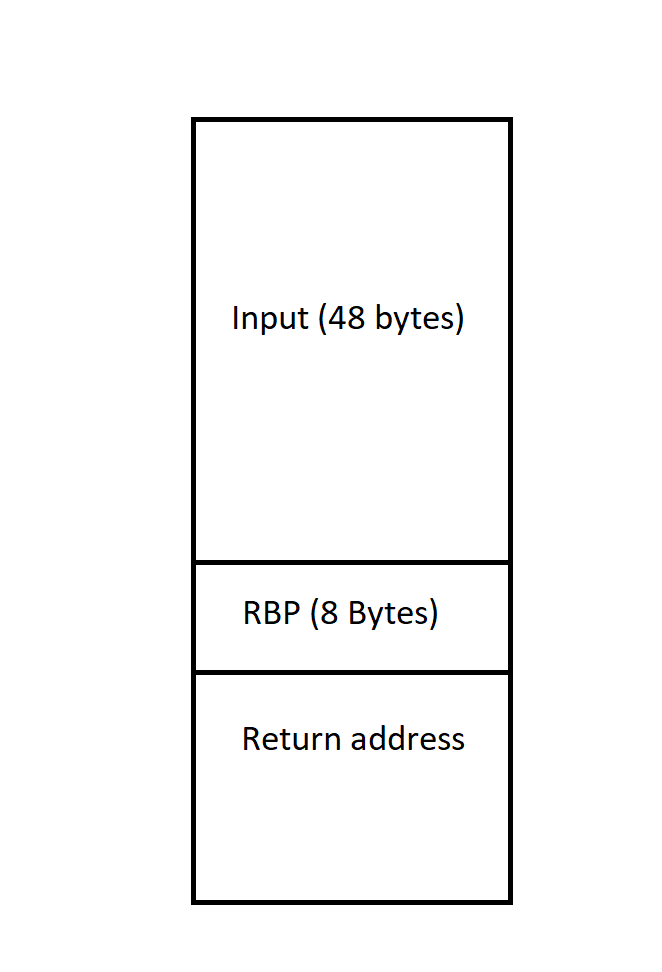
Vulnerability 1: the program uses gets() to read input, which does not do any bounds checking, allowing anyone to overwrite memory.

Vulnerability 2: program uses printf to print my input directly, allowing me to input formatting strings and know values from the stack.

One more piece of information we gathered is the size allocated for our input (48 bytes) as we can see in the variable local\_38.

Last piece of information we gathered from diassembled code is that it uses rax, rsp,rbp,etc… . Meaning it is a 64 bit program, each slot in memory is 8 bytes.

Now we can visualize the stack frame of this function

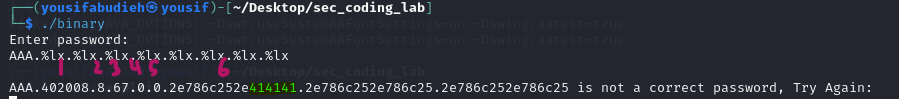


**Note: The RBP stored here is the RBP for the main function’s stack frame, which will be popped when the function IsPasswordOK is done, it is used when the main function wants to access one of its variables on arguments which are stored in the stack.**

**Going back to the note I left on page 2 of this report, we saw that the code segment I want to redirect the program flow to is in the main function, and it does not use the RBP register, meaning that we don’t have to keep RBP the same for those instructions to be normally executed. But in the case that it is used there, we have to try to overwrite the return address on the stack without changing the value of RBP, which is what I will do for demonstration purposes in the next part.**

We can conduct the attack by entering 48 random bytes, then entering the RBP as is (to keep main running normally since it depends on rbp to reach its variables), and then overwrite the return address to point to the first statement after access is granted.

We now need to know how we can get RBP and store it in a variable to send it later, we can use %lx, which prints random values from stack, we will input AAA.%lx.%lx.%lx.%lx.%lx …..  
then we will look for the values 0x414141 which are the values of AAA, and that memory slot will be where our input starts.

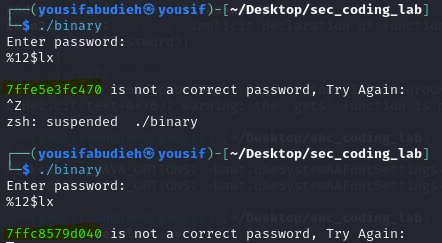


We can see that our input starts in the 6th memory slot, now we want to know where to find RBP, which comes 48 bytes after the start of our input.

48 bytes / 8 bytes per slot = 6 slots after the start of our input.

RBP is in 6th + 6 slots = 12th slot.

We can know specific slots using %12$lx, which shows us contents of 12th slot (RBP).



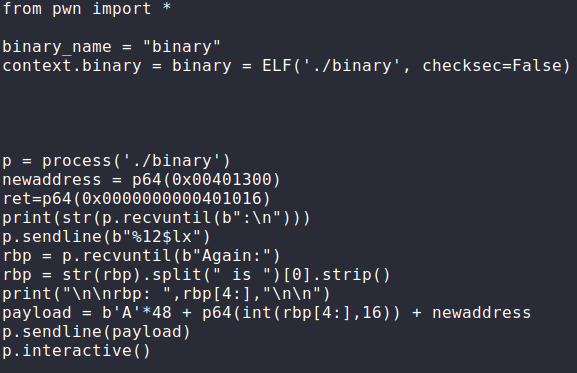
As we can see, the value of RBP changes with every execution, so I have to acquire it and store it in a variable in my script, then send the payload which will look like this:

Payload = 48 random bytes + RBP + access granted address.

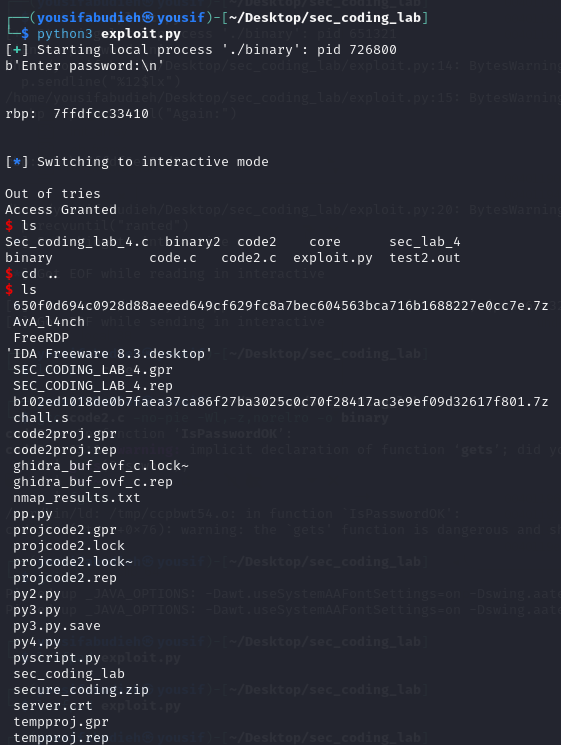
Script pseudocode:

1. Start(binary)
2. Access\_granted\_address = 0x00401300
3. Receive(prompt)
4. Send(“%12$lx”)
5. Rbp = receive(rbpvalue)
6. Receive(prompt)
7. Payload = ‘A’\*48 + Rbp + access\_granted\_address
8. Send(payload)

Here is the python script:



Running the python script:



As we can see, I was able to go to puts(“Access Granted”); without entering correct password, simply by modifying return address.

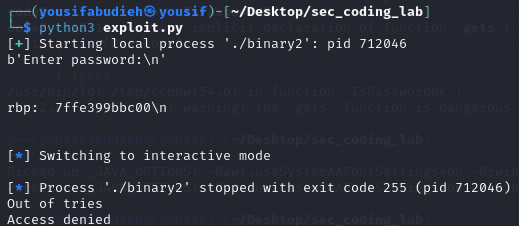
**Note: “Out of tries” was printed since I sent 2 separate payloads, the first one was to get the value of RBP, and the second one was the actual payload that modifies the return address, each of these payloads was treated as a password guess. So after two “guesses” the program printed “Out of tries”, then instead of continuing normally and checking the if condition, it went directly to puts(“Access Granted”)**

**Modifying the code to become more secure:**

Change gets() to fgets() for bounds checking.



Trying to run the script now:



We can see that we were still able to leak the RBP value from the stack, this is because we didn’t change printf(Password) to printf(“%s”,Password).

In this section, I will try to remove the leakage of RBP from the exploit, then run it.

A screen shot of a computer program

AI-generated content may be incorrect.

It still works

A screenshot of a computer

AI-generated content may be incorrect.

However, if I change the C source code to include a variable inside the else segment in the main function, then it will not work.

This is the new main

A screen shot of a computer code

AI-generated content may be incorrect.

After compiling it, here is the section we are aiming for executing:

A screenshot of a computer program

AI-generated content may be incorrect.

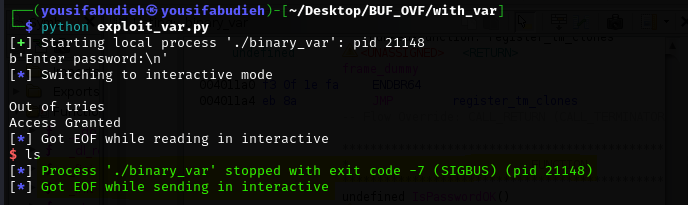
As we can see here, the program is using RBP to reach the local variable testvar, and add 2 to it, meaning that if we put an invalid RBP the program may crash.

Now I will modify a few things with the exploit and try to exploit the program **without** leaking the RBP value.

Here is the modified exploit, I only fixed the binary name to match out new binary, and I also fixed the return address to be the same as shown in the previous photo.



Running the exploit:



The part highlighted in green indicates that our exploit failed, which happened on the assembly instruction ADD dword ptr [RBP+local\_c], 0x2

since the RBP value was invalid.

Fixing the exploit by keeping RBP the same:



Running the exploit:

A screen shot of a computer

AI-generated content may be incorrect.

It worked as expected.