PWN: Hi

A good practice to solve this kind of exercises is to understand “what is going on” and to have a clear idea of the execution flow.

We know that in C the function *main* is the starting point of the program execution. The first question we need to answer is: what are the “entering points?”, i.e., the instructions that allow us to interact with the application.

Since the function *main* does not call any other function (only from the stdio library), we can be sure that the only entering point is:

fgets(msg, 0x32, stdin);

The variable *msg* is defined as an array of 32 characters, thus requiring an amount of memory equal to 32 bytes.

The second question is: “is there any vulnerability we can exploit?”. The answer is yes. The vulnerability we can exploit in this exercise is located in the *fgets* function, which copies the first *0x32* bytes of the input provided on *stdin* into *msg.* However, *0x32* is a hexadecimal value, which we can convert into the decimal one with the following code:

print(int(‘0x32’, 16))

This tells us that *0x32* corresponds to *50*, which means that we can provide up to 50 bytes on the *stdin.* However, only the first 32 bytes will fill *msg*, while the remaining 18 bytes can be used to corrupt the memory.

Several defence techniques can be enabled against buffer overflow attacks, and we need to first understand which ones are active. A good practice is to use the *gdb* debugger. We suggest to use *gdb-peda*, a version that eases our security debugging. You can install it from the following link: https://github.com/longld/peda .

Once installed, follow the following instructions:

1. Open a terminal and move to the directory containing the program you want to debug;
2. Type: *gdb <program name>*
3. Now, we are inside the debugger with the program loaded. We can check the security mechanisms of the application by typing the *checksec* command

Most are disabled (e.g., canary), which means that we can overwrite the memory and reach the return address of the *main* function. Our goal is to call the function *print_flag*, which contains the actual flag.

Now, we need to “guess” the distance between *msg* and the return address in order to insert the proper input and take control of the flow. Let’s try to draw the stack:

return address (8 bytes)
base pointer (8 bytes)
msg (32 bytes)

Remember that the allocation of the stack frame goes from higher addresses towards lower ones (i.e., return address → base pointer → msg), while the memory writing follows the opposite order (i.e., msg → base pointer → return address).

Our goal is thus to overwrite the return address with the address of the *print_flag* function, which at this point is unknown to us.
The input we need to send should have the following structure:

[junk] + [print_flag_address] → [32+8 bytes] + [8 bytes]

This could be a good guess, but, however, we cannot trust the compiler, and the stack could contain also something more. So we can continue using our debugger in order to understand what is the actual required offset.

1. Create a pattern for the debugging, e.g., with 100 characters:
   pattern create 100 pat100
2. run the program:
   run < pat100
3. at this stage, the program should crash (Figure 1). We can search for the pattern that did this, and the debugger will give us all the information we need to solve the exercise. Type the following code (Figure 2):
   pattern search <first 8 characters of the pattern that is displayed under the stack section>

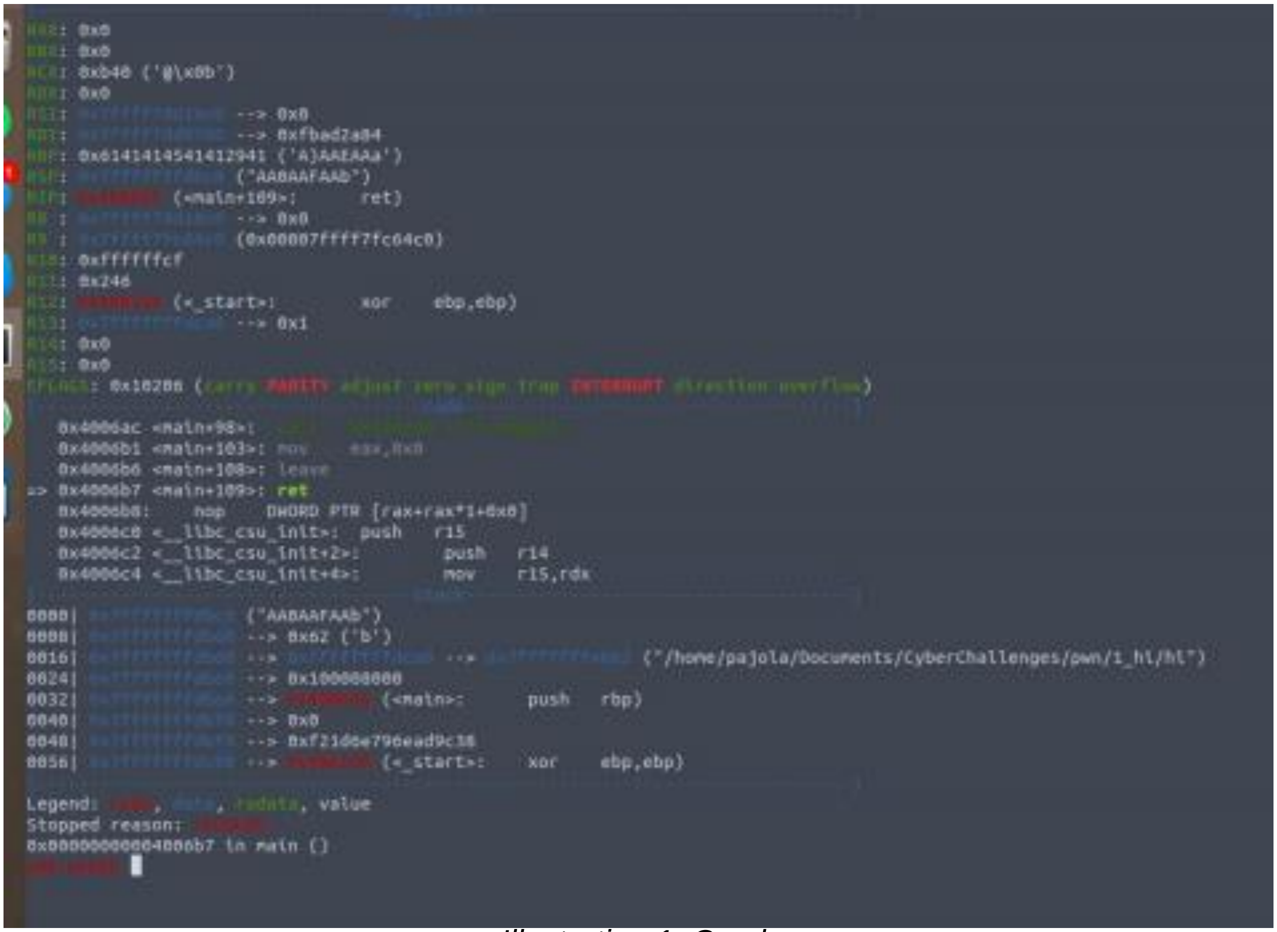

Illustration 1: Crash

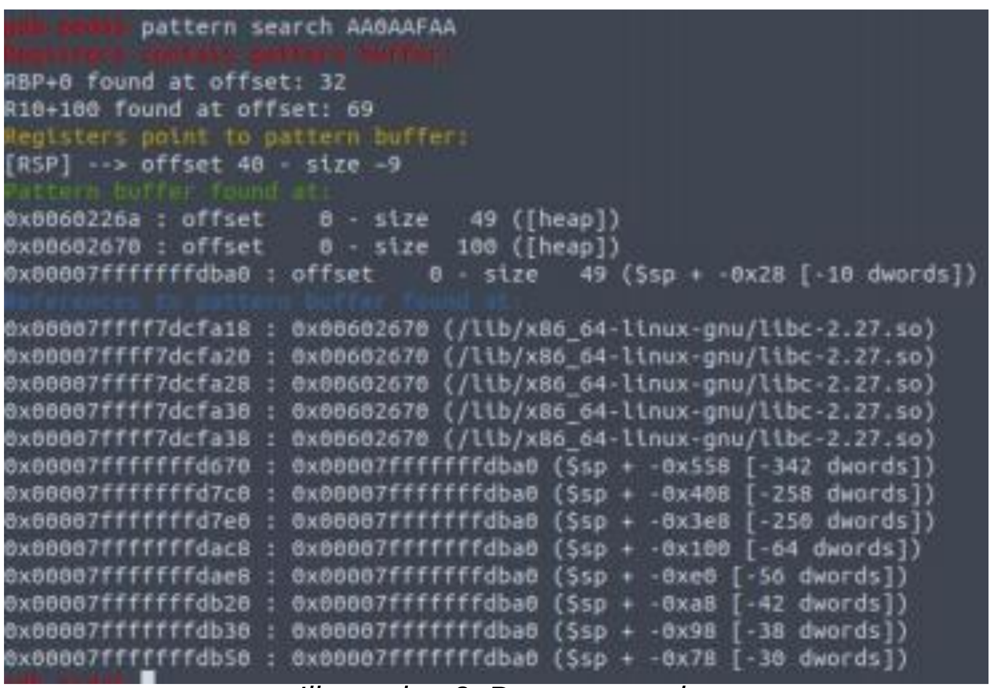

Illustration 2: Pattern search

We can see that RSP (stack pointer - https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture) has 40bytes of offset, which means that our math was right.

We have everything ready now. To solve the exercise, we are going to use the library *pwntools*

First, we need to import the library:

*from pwn import **

Second, we want to find the address of the *print_flag* function:

elf = ELF('./hi')
target_address = p64(elf.symbols['print_flag'])

- *elf.symbols['print_flag']* finds the address of the *print_flag* function
- the *p64* function converts the address in bytes - little endian format

Create the final message:

*garbage = b'a' * (32 + 8)*
msgin = garbage.encode('ascii') + target_address

Thus, *msgin* contains 40 times the character 'a', concatenated with the target address.

Finally, we need to send this message to the process, and we can do it with pwn

functions:

p = process('./hi') #connect with the process
p.sendline(msgin) #send the message
msgout = p.recvall() #receive the output of the execution
print (msgout)

Running the code, you should have something similar to Figure 3.