**Secure Systems Engineering (CS6570)**

**Assignment-2**

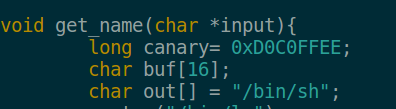
**NAME**: Mohammed Umair Pandit

**ROLL NO:** CS23M037

There are quite a few vulnerabilities in the provided code, there are as follows:

**Constant Canaries:**

These fixed Canaries values are used in stack protection mechanisms to detect buffer overflow attacks. These Canaries are predetermined values placed in memory locations before critical data structures making sure those are not fiddled with or tampered with. During program execution if the canary value is modified, it is indicating that a there is a possibility that a buffer overflow has occurred and the program will terminate or take necessary action to prevent exploitation and send and “stack smashing detected” message to the user.

This is the reason canaries are used, and in the program a fixed canary value is used , which could prove to be less reliant than an canary which is generated by a compiler which is usually unique and random. The fixed canary value here gives and opportunity to manipulate and read the stack even more clearly, since we will not have to search or dig through the stack to identify the canary value.

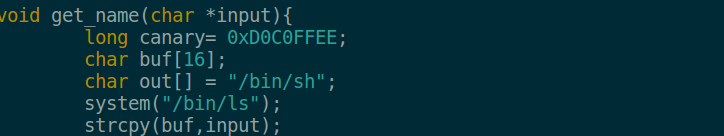


This vulnerability can be fixed by enabling f-stack-protector while compiling the code or using it in the MakeFile, what this flag or command ensures is that it provides canary in the stack and then make sures that the canary value has not changed or any overflow has not occurred, usually it does this by using bit manipulation technique to check before and after compiling canary values.

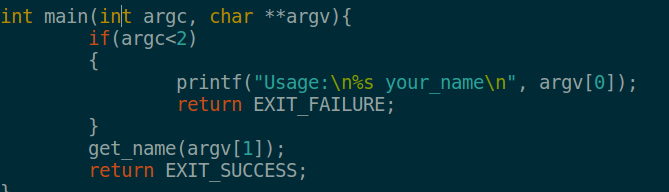
**Shared Library Functions:**

Since the program given is coded in C language, we are bound to use the functionality the language has to offer, and by that reason, the code contains functions which are used from the library of the C language. Now these functions are necessities, some not, but the point is that these functions are needed to use the language effectively, which in turn is the reason these vulnerabilities exist in the first place.

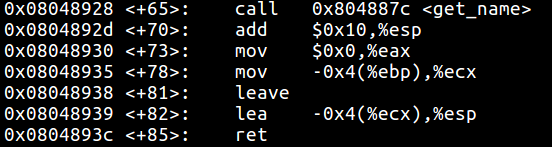
This Vulnerability is famously also known as ‘Return-to-Libc’ attack, where the attacker uses the function of Libc from its library to subvert the execution of the program out of the stack in the code segment , where the attacker can run its exploit/malicious code, and that is because even though we have enabled non-executable stack and made canaries, we are not executing the stack in the first place, just overflowing the return address with the Libc function call to subvert the execution of the program and return to a function in Libc where it can execute its exploit code. And this is exactly what we have carried out in this task as well.

In this program we have used a common Libc function call called ‘System(parameter)’ which takes a pointer to a string as a parameter and executes it.

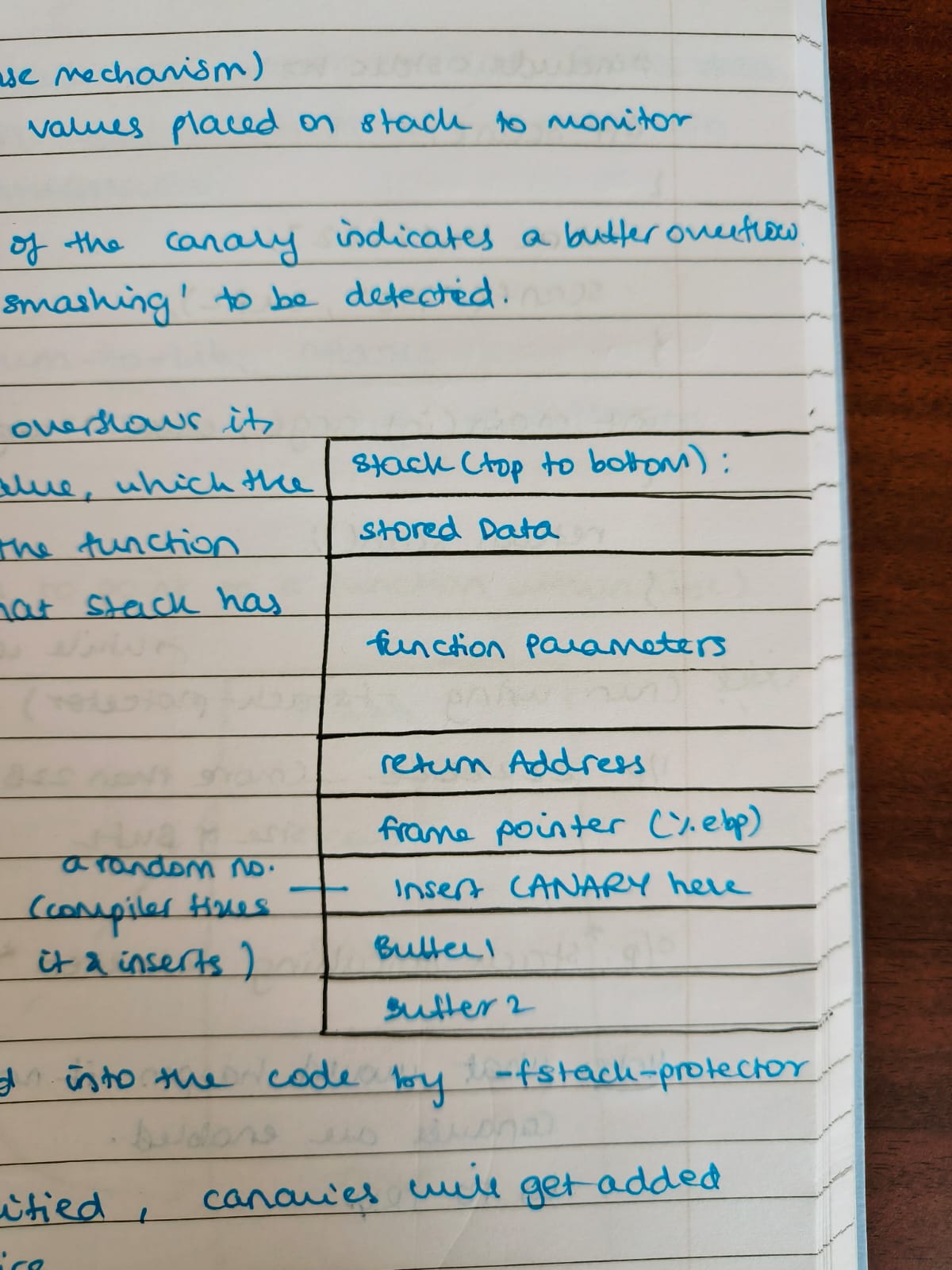


Above is the snippet of the code given, we have retrieved the address of the system by debugging it using GDB and overflowed the return address to the main which is present as shown below after get\_name function. Which subverts to Libc function call and after passing desired parameters (i.e exploit code) exploitation is carried out.









We can summarize our ‘Ret2Libc’ Attack using above stack representation. Since we Know the size of Buffers in our locals of our program we overflow the buffer with some ‘nops’ (“A” in this case) until canary value is met, then leave / replace the same canary value after overflowing the buffers , then insert again a ‘nops’(“A” again) again to fill the frame pointer ‘ebp’ usually it is of 4 Bytes in size, then we input or add the Little Endian format of address of the Libc Function call ‘System’ in place of return address. After the ‘System’ address is overflowed in return address we insert the address of the ‘Exit’ function to make a clean exit out of the program, then we insert the shell code we want to execute after exit address, which the function ‘System’ will take as parameter and then when return address is hit, it subverts to ‘System’ function call where the parameter of shell code passed above it in the ‘function parameters’ is executed by the ‘System’ function call.

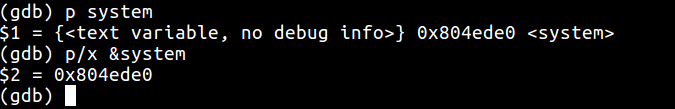
**Buffer Overflow:**

Even though we exploit using Ret2Libc , we still are overflowing the stack using the buffer using function which are stored and that those that access the stack , the locals which are stored in the stack, from where we gain access to the stack and then overflow the adjacent memories or dummy it.

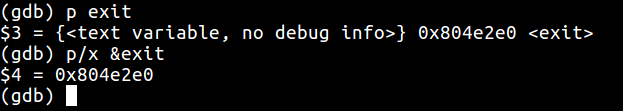
**Addresses and Overview of the exploit:**

Now above methodology which mentioned implemented needs few important address after which we can carry out the exploitation.

Address of the System function call

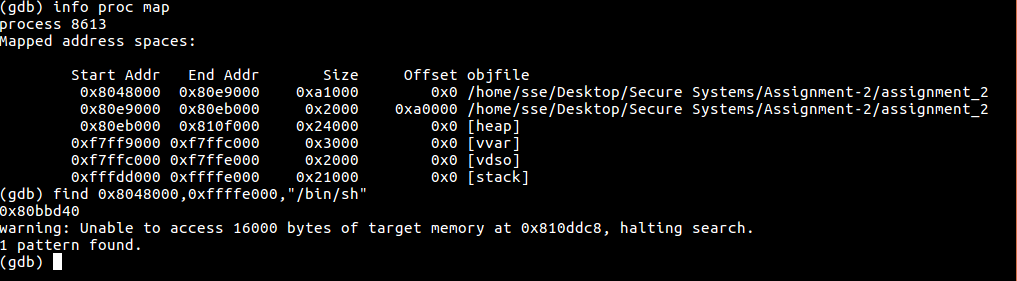
Retrieved as follow:



Address of the Exit Function call retrieved as follows:

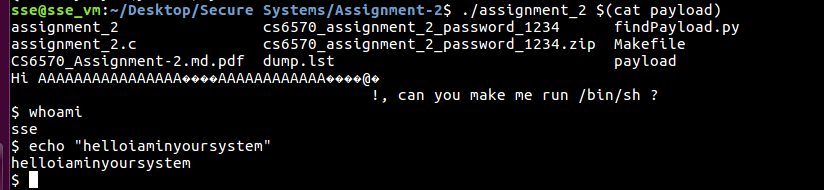


Address of the /bin/sh:

This is present in the stack of the main function stack frame and can be retrieved as follows in command line:



After we retrieve the necessary address we just implement the above approach by overflowing the buffer and replacing the return address with system function call address and inserting the system , exit and shell address.

Eventually we gain access to the shell:



These Ret2Libc Attacks can be prevented by implementing ASLR (Address Space Layout Randomization) which by default is included in current systems. What ASLR does is it randomizes the address space layout of the process , such that each execution would have a different memory map. Thus in turn making it difficult for the attacker to run exploits.

As the attacker knows the address of the ‘System’ function or any other Libc function address in the library, what ASLR does is randomizes these addresses every execution of the process which makes it difficult for the attacker to retrieve the addresses of these Libc functions which is the root cause of this ‘Return to Libc’ Attacks.

**MakeFile Flags:**

The **gcc** compiler flags has a significant role and impact on the security it provides to the compiled binary executable. Flags and their significance are discussed as follows:

**-fstack-protector:** Enables basic stackoverflow protectors, which insert canaries and ensures there are no changes in the canary values which in turn is used in detecting whether any modification to the stack are done or not(a.k.a Stack smashing).

**-z execstack:** Marks the stack as non-executable to prevent shellcode executions.

**-fPIE -pie:** This is a highly significant flag as it prevents the exploit we carried out, it builds the binary as a Position Independent Code (PIC) executable to enable ASLR.