A logo of a state

AI-generated content may be incorrect.

**Name**: Francisco Izaguirre

**Course**: CSCI 4334

**Date**: 05/1/2025

**Instructor**: Wendy A. Sanchez Gomez

**Lab 07 – Code Injection and Buffer Overflow Attacks**

***Introduction:***

This lab focused on understanding buffer overflow vulnerabilities in C programs and how they can be exploited when proper memory safety checks are not in place. I worked through a hands on example involving a vulnerable program that used “strcpy()” to copy user input into a fixed-size buffer without bounds checking. The main objective was to demonstrate how this kind of oversight could be used to redirect the program’s execution flow, specifically to a hidden function within the same program. I also explored how compiler options and secure coding practices like “strncpy” and enabling stack protection, can help defend against suck attacks. Overall, this lab helped reinforce how low level memory handling directly affects the security and behavior of a program.

***Experimental Procedure:***

Step 1) First, we will create a C program that has buffer overflow vulnerability. The program reads user input into a fixed size buffer without checking the length of the input. This will lead to a buffer overflow. This program includes two functions, secret\_function (prints message when called) & vulnerable\_function (contains the buffer and uses strcpy() to copy input into it). It also contains the main() function which checks if input was provided and then calls the vulnerable function.



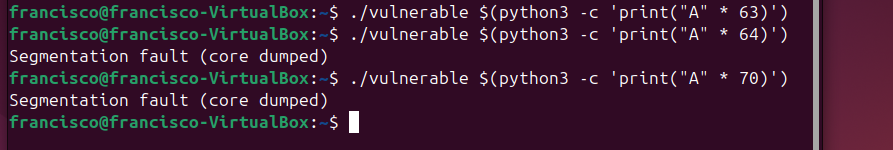
This sets up a classic buffer overflow vulnerability, which can be exploited to change the control of the program.

Step 2) Next, I compiled the program using gcc with no stack protection to intentionally disable security features for testing purposes. The program compiled successfully into an executable named vulnerable. The -fno-stack-protector disables stack protection such as canaries. The –z execstack makes the stack executable. The –no-pie ensures predictable memory addresses.



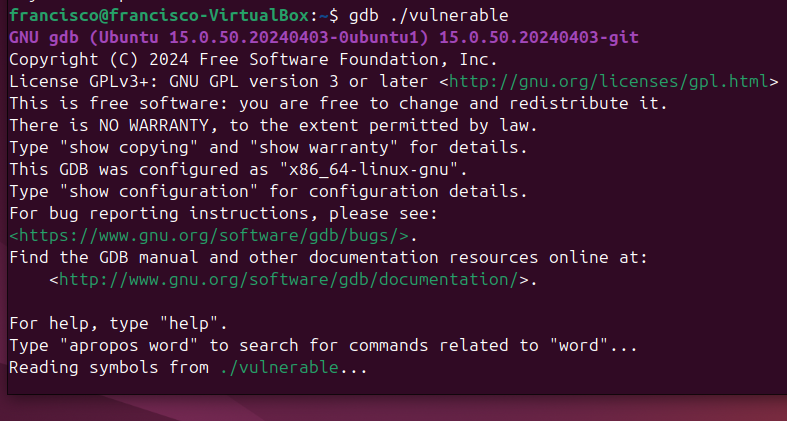
We have now complied a program that is intentionally vulnerable to buffer overflows and can be tested without protections that would normally prevent exploitation.

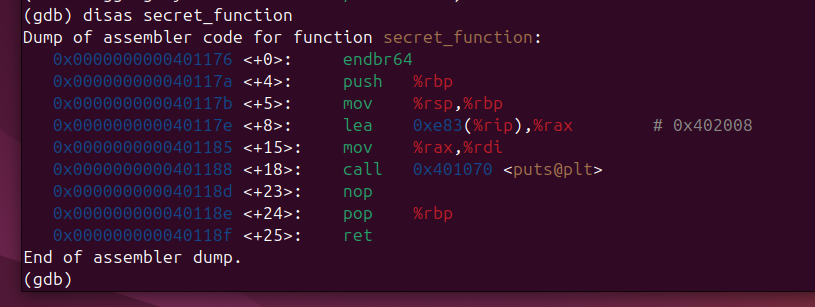
Step 3) Now, we will run the program with inputs that are larger than 64 characters. First we tried 63 to test if it worked in which it did. Then we tried 64 and 70 to show that the buffer overflowed which is indicated by the “Segmentation fault (core dumped). The strcpy() in the code does not check for the length.



We demonstrated that the program is vulnerable to buffer overflow. By passing more than 64 characters, we are able to overwrite memory past the buffer’s boundary, which is the core of this exploit.

Step 4) Next, we are going to identify the address of the secret\_function using gdb and disas in order to inject malicious code into the memory in later steps. We are doing this because the goal of the attacker is to overwrite the return address on the stack to point to the function and then jump to the malicious function instead of returning to main. To use the GNU debugger, we used “gdb ./vulnerable”. Then we entered “disas secret\_function” to print the assembly instructions for the function and get the starting address which should be the first line.





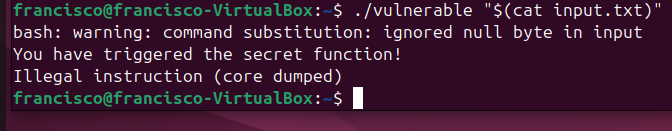
We obtained the exact address of the function (0x0000000000401176) we want to redirect execution. This address will be used to overwrite the saved return address on the stack in the next step, effectively hijacking the programs control.

Step 5) Now, we will create an exploit input with the starting address of secret\_function. We constructed a payload that fills the buffer with 64 A characters and appended the address of the function in little-endian format. This runs a short python script directly from the command line using the “-c” command flag. This redirects the output of the Python command into a file called “input.txt”.



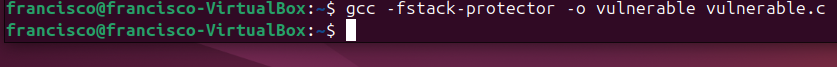
We successfully exploited the buffer overflow by overwriting the return address to point to secret\_function. This demonstrates how insecure memory handling in C can allow attackers to execute arbitrary code.

Step 6) Next, we will now execute the malicious payload. First, we ran the vulnerable program using a specially crafted input file that contains 64-filler bytes followed by a specific memory address that points to secret\_function. The “cat” is a Linux command-line utility that stands for concatenate. It is used to display the contents of a file.



The exploit was successful and we know this because we see the “You have triggered the secret function” which means the program jumped to the function rather than back to main as it should have. This confirms a successful code injection via buffer overflow.

Step 7 & 8) Next, we re-enabled the stack protection by using the “-fstack-protector” option when compiling the program. This adds a layer of protection against buffer overflow by adding a canary value to the stack that will detect if the stack is being overwritten. We will now test the protection and try running the program with malicious input again. This time, the program now crashes with a stack-smashing error displaying the working protection.



This shows us how it terminates before the malicious return address is executed. It also demonstrates a real world mitigation technique that can effectively stop simple buffer overflow attacks.

Step 9) Now, we will see another way in which the attacker can infiltrate. First, we must disable ASLR (address space layout randomization) by using “sudo sysctl –w kernel.randomize\_va\_space=0” for testing purposes. ASLR randomizes the location of the stack, heap, and shared libraries in memory, making it difficult for an attacker to predict the address of the secret\_function. After disabling ASLR, the function becomes predictable and consistent between runs. We will then rerun the malicious input.

After running the input, we see that the attack succeeded. Now, we will re-enable ASLR to see that the attack will fail with it on.

This shows us how disabling ASLR makes exploitation easier and repeatable, because the address used in our payload no longer changes each time the program runs.

Step 10) Next, I will show another protection method. We will recompile the code using “strncpy()” rather than “strcpy()” to prevent overflow. This protects the program because unlike strcpy(), strncpy() limits the number of bytes copied and prevents buffer overflow even if the input is longer. This ensures that the buffer will not overflow and thus adds more protection. We will now re run the program.

This shows us how the exploit failed because strncpy() only copies up to the buffer’s size and prevents writing past its boundary. This step shows how switching to safer function is an effective way to defend against buffer overflow attacks.

***Questions:***

1. **What is buffer overflow vulnerability?** It is where excess input can overwrite important memory locations. (Slide 46)
2. **What is the purpose of the fno-stack-protector flag in the compilation?**

The purpose of the flag is to disable stack protection mechanisms, specifically stack canaries, which are normally added by the compiler to detect overflows.

1. **Which of the following mitigations can prevent a buffer overflow attack?**

Using safe functions, which should replace vulnerable function such as strcpy. Having ASLR enabled which randomizes the location of stack, heap, and shared libraries in memory. Lastly, by enabling stack protection which adds a canary to the program which detects overflows. (Slide 55-58)

1. **What does the ‘secret\_function’ do in the example?**

In the context of our lab, the secret\_function() is a dummy function included as a target for the buffer overflow exploit. It is not called by normal execution bur rather by exploitive execution. It serves as an example for a function that can genuinely be malicious.

1. **What is the role of Address Space Layout Randomization (ASLR) in preventing code injection?**

ASLR randomizes the location of the stack, heap, and shared libraries in memory, making it difficult for an attacker to predict the address of functions. (Slide 57)

***Conclusion:***

The main goal of this lab was to understand how buffer overflows occur and how they can be exploited to manipulate a program’s control flow. By deliberately writing and executing vulnerable code, I was able to see firsthand how an overflow can overwrite a return address and redirect execution to a different function. I also tested how compiler level protections like stack canaries and ASLR work to prevent such attacks, and how replacing unsafe function s like strcpy() with safer alternatives like strncpy() helps improve security. Through this process, I gained a stronger understanding of both the risks involved in unmanaged memory and the importance of writing secure C code. This lab gave me valuable insight into how software vulnerabilities are discovered and how developers can guard against them in real world applications.

**References:**

Sanchez, Wendy. CYBI 3345 – *Operating Systems and Security*

*CSCI 4334 – Operating Systems Security and Protection*. 1 May. 2025. (PowerPoint).