Experiment 1. Documentation

Part I:

Since this computer required a Linux system, I was not able to do it using the Leased laptop so I collaborated this with my friend who had Ubuntu system present in his windows already.

To start off the project we opened up the Ubuntu and started coding. It took us a bit of time for this part but it ended up compiling. So we used the code.

Part II:

For part II we followed the given instructions   
  
Step 1: Navigate to Part 2 directory inside the Experiment\_1\_code directory.   
Step 2: In Linux, compile the code using: gcc -w -fno-stack-protector ccode.c -o ccode   
Step 3: Run it: ./ccode

Part III:  
  
This part is where we inject malicious code. This part was straight forward as given in the. powerpoint.

Step 1: Navigate to Part 3 directory inside the Experiment\_1\_code directory.   
Step 2: In Linux compile the code using: gcc -w -fno-stack-protector bo\_test.c -o bo\_test Note: If you’re running this experiment on an Ubuntu machine, you may need to disable the Address Space Layout Randomization (ASLR) functionality by running the following terminal command: echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space   
Step 3: Run the program with any 5-letter argument or any number of letters less than 10. For example: ./bo\_test abcde   
Step 4: Get stack address of the good code and the malicious code. Copy the hex address of the malicious code.   
Step 5: Inside the run.pl Perl script, there is an address on the first line. Note that it is grouped by bytes, where each group is preceded by a \x character. There are four bytes there, but your copied malicious code may not be of that length which you should keep intact. Group the address of the malicious code from the previous step in the same manner as in the Perl script. Be mindful of the endianness of the address (i.e., most significant byte first OR least significant byte first).   
Step 6: Run the Perl code: perl ./run.pl  
  
Questions:

Part I:

Suggest ways to improve it against this attack implementation.  
- Improving the code against buffer overflow attacks generally involves several layers of defense. Here are some strategies:

1. Proper Input Validation: Ensure that the input length does not exceed the buffer size.

2. Use Safe Functions: Avoid using functions known to be vulnerable to buffer overflows.

3. Stack Protection: Compile the code with stack protection enabled. In GCC, this is done by removing the `-fno-stack-protector` flag.

4. Address Space Layout Randomization : Enable ASLR, which randomizes the memory addresses used by a process, making it harder for an attacker to predict where to inject malicious code.

5. Regular Updates and Patches: Keep the system and all its components updated with the latest security patches.

6. Security Training: Ensure that all programmers and relevant staff are trained in secure coding practices.

By combining these strategies, we could significantly reduce the risk of a buffer overflow attack. However, security is a constantly evolving field, and even with these measures in place, it's important to stay informed about new vulnerabilities and attack techniques.

PartII:

1. What is output when you enter correct key (key = gainesville)? A computer screen with text on it

   Description automatically generated
2. What is output when you enter a wrong key between 10-12 characters? (e.g., key = ghgjhjhjkh)  
   A computer screen with a green circle and blue text

   Description automatically generated

3. What is output when you enter wrong key longer than 12 characters? (e.g., key = ghgjhkjhkjkhkhkjhkjlkjlkjkjlkjlkjlkjkljkjlkjk)  
A computer screen with a green circle and black text

Description automatically generated

4. Write your conclusions,

- Based on the scenarios, it is a buffer overflow vulnerability, particularly evident when incorrect keys of various lengths are entered. When the correct key "gainesville" is used, the program behaves as intended, granting access. However, with incorrect keys, especially those exceeding the buffer size, the program still grants access, indicating a serious security flaw. This flaw is likely due to inadequate validation logic and improper handling of buffer sizes. The program fails to correctly manage and validate input lengths, resulting in a buffer overflow when excessively long keys are input. This overflow can overwrite adjacent memory spaces, leading to unintended behavior such as unauthorized access. The vulnerability is further highlighted when keys of 10-12 characters, although incorrect, also grant access, pointing to flawed validation logic. To address these issues, implementing robust input validation, ensuring proper buffer size and boundary checking, using safer functions for string operations, and adhering to security best practices in coding are essential steps. Correcting these vulnerabilities is crucial to prevent potential security breaches and maintain the integrity of the system.

5. Why do all these issues happen?  
- These issues arise due to a buffer overflow, a common security vulnerability that occurs when more data is written to a buffer, or a temporary storage area, than it can hold. In this case, when a key longer than what the buffer is allocated to store is entered, it overflows the buffer and overwrites adjacent memory locations. This overflow can disrupt the normal flow of the program, often altering control flow data, such as return addresses or pointers, leading to unpredictable behavior. In your examples, this results in unauthorized access being granted. The problem is compounded by insufficient or incorrect input validation, where the program fails to adequately check the length and content of the input before processing it. This oversight allows users to input data that exceeds the buffer's capacity, triggering the overflow. Buffer overflows are particularly dangerous as they can be exploited to execute arbitrary code, leading to serious security breaches.  
  
6. How can you correct them?  
- To correct buffer overflow vulnerabilities, a multifaceted approach focused on secure coding practices is required. First and foremost, implement rigorous input validation to ensure that all incoming data, especially from user inputs, is within expected bounds and conforms to the expected format. Use safer string and memory handling functions that include explicit length parameters, such as `strncpy` instead of `strcpy`, to avoid writing data beyond the buffer's capacity. Additionally, adopting bounds checking mechanisms can prevent data from exceeding the buffer's allocated size. It's also vital to initialize buffers to a known state and clear them after use to prevent data leakage. Employing modern programming languages or compilers that inherently manage memory and include built-in protections against buffer overflows can further bolster security. Regular code reviews and dynamic analysis tools, such as fuzz testing, can identify potential vulnerabilities. Lastly, educating developers about secure coding practices and common vulnerabilities like buffer overflows is crucial in preventing such issues from occurring in the first place.

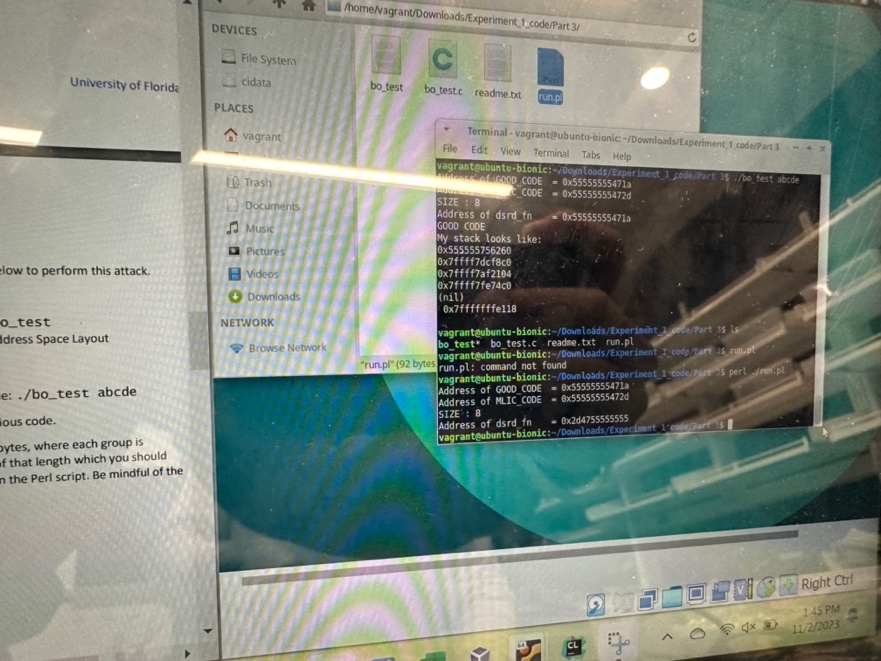
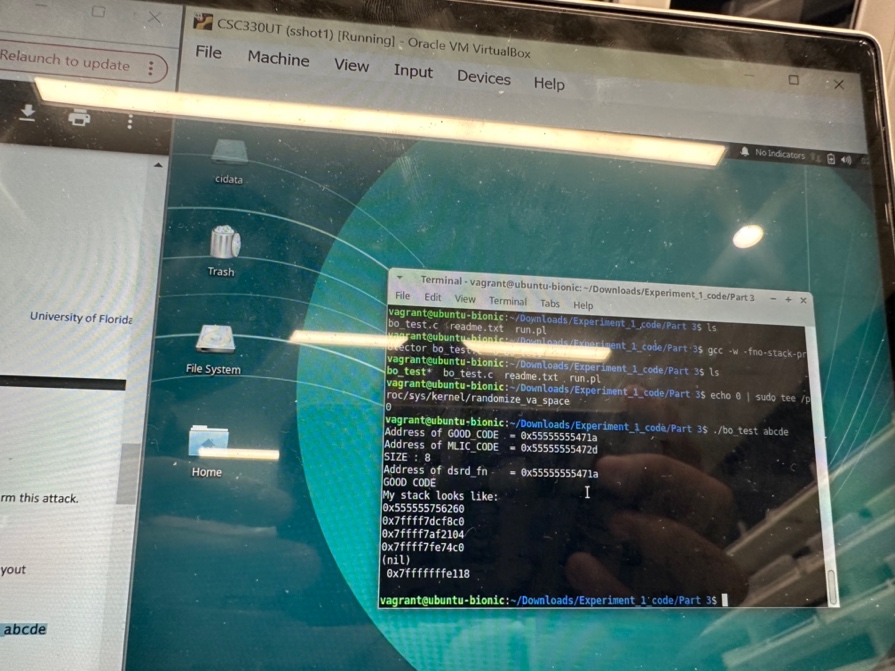
PartIII:

1. What is the endianness of the address inside the Perl script?  
   - In Perl, the endianness of an address, or any other data, is dependent on the underlying hardware architecture of the system on which the Perl script is running, rather than on Perl itself. Endianness refers to the order in which bytes are arranged in memory. There are mainly two types:

1. \*\*Big-Endian\*\*: In a big-endian system, the most significant byte (the "big end") of a word is stored at the smallest memory address and the least significant byte at the highest.

2. \*\*Little-Endian\*\*: In a little-endian system, the least significant byte is stored at the smallest address and the most significant byte at the highest.

1. Take a screenshot of an successful attack.



1. Write your conclusions.

* There are memory addresses displayed, which likely correspond to specific locations in memory where a buffer overflow could be exploited. The Perl script named `run.pl` is being used, presumably to automate the process of testing or demonstrating the buffer overflow. The output indicates that the program has a section labeled "GOOD CODE" and "MILC\_CODE", which could be the parts of the program's memory that the exercise is targeting for the buffer overflow. The memory addresses and values like `0x555555555555` and `0x7fffffffdfc8` indicate that the system is using a 64-bit architecture based on the length of the addresses.

The mention of the stack and address space layout randomization (ASLR) suggests that the exercise may also be demonstrating how modern operating systems use ASLR to make exploiting buffer overflows more difficult. The goal of the exercise is lto show how a buffer overflow can be used to manipulate a program's execution flow, and how to protect against such vulnerabilities..

1. Suggest ways to improve the program against this attack implementation.

* To improve the program and protect it against buffer overflow attacks, as seemingly demonstrated in the images, we can take several steps:

1. Bounds Checking: Implement explicit bounds checking for all buffer operations. This ensures that data written to a buffer does not exceed its allocated size.

2. Safe Functions: Replace unsafe C functions.

3. Stack Protection\*\*: Utilize compiler-provided stack protection mechanisms.

4. Address Space Layout Randomization: Enable ASLR, which randomizes the memory address space layout of a program's process, making it difficult for an attacker to predict the location of specific processes.

References

1. <https://www.geeksforgeeks.org/courses?itm_source=geeksforgeeks&itm_medium=main_header&itm_campaign=courses&source=google&medium=cpc&device=c&keyword=programming%20classes%20online&matchtype=b&campaignid=20775210381&adgroup=153007046582&gad_source=1&gclid=CjwKCAiAjfyqBhAsEiwA-UdzJI6Z_CA57HY_jO2GneI1lFN6D3gH3xKBUNNaTiP6owk4jNZeUqTw2BoC64EQAvD_BwE>