University of South Florida

CDA 4205L Computer Architecture Lab

Lab Report

Semester: Spring 2025		
Experiment	*Number:*	13
	Date:	April 16, 2025
Lab	*Section:*	004
	Lab TA:	Rupal Agarwal
Report	*Due Date:*	April 23, 2025
Group	*Member #1 name:*	Claude Watson
	Member #2 name:	Anirudh Sesham

ABSTRACT:

The purpose of this lab was to investigate buffer overflow attacks in computer architecture, using RISC-V memory organization. The objective was to demonstrate how improper management of memory buffers in low-level assembly code can lead to serious software vulnerabilities. In this lab, we executed a RISC-V program intentionally designed with a buffer overflow vulnerability. By experimenting with different input lengths, we observed how exceeding the input buffer's capacity allowed us to overwrite an adjacent password variable in memory, granting unauthorized access to the system. Through this process, we learned how buffer overflows occur, how to detect and exploit them, and why it is critical to implement secure coding practices to prevent such attacks. In conclusion, this lab highlighted the importance of careful memory management and input validation to protect software systems from common security threats.

INTRODUCTION:

Understanding software security vulnerabilities is an essential aspect of computer architecture, particularly of low-level programming where direct memory management is required. Among the most common and critical vulnerabilities is the buffer overflow attack, which exploits improper handling of memory buffers, potentially allowing attackers to overwrite important program data and compromise system security. This vulnerability is particularly relevant when working with RISC-V, where the programmer has direct control over memory allocation and must manually manage buffers on the stack or in the data segment.

In this lab, we explored buffer overflow attacks by analyzing and experimenting with a RISC-V assembly program specifically designed to contain a buffer overflow vulnerability. The experiment involved inputting various character strings to observe how exceeding the input buffer's capacity could overwrite a password stored in an adjacent memory region. By monitoring the memory layout before and after the overflow, we were able to directly observe the consequences of insecure memory handling. This hands-on approach reinforced the importance of implementing safe coding practices, such as proper bounds checking, to prevent such vulnerabilities in real-world software systems.

METHOD:

Software Tools Used:

- **RARS (RISC-V Assembler and Runtime Simulator):**
  Used to assemble, run, and debug the RISC-V assembly code. RARS provides both a console for program interaction and a memory viewer to observe buffer overflow effects.

Files:

- **buffer_overflow.asm:**
  This RISC-V assembly script simulates a buffer overflow vulnerability by accepting user

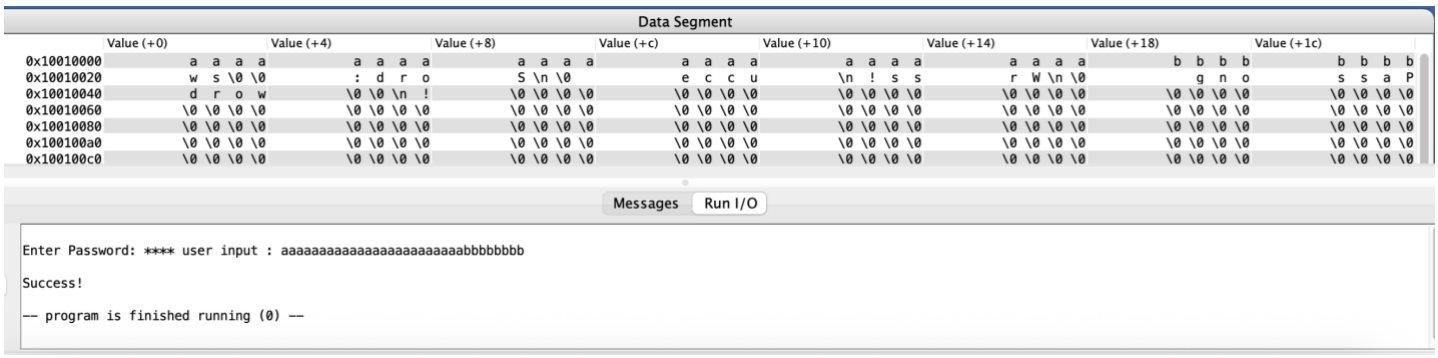
input and storing it in a fixed-size buffer adjacent to a password variable. Used for demonstration and experimentation with buffer overflows.

RESULTS:

T1: How many characters do you need to enter to reach the region of memory where the password is stored? How many total characters do you need to enter to overwrite the password?

Through experimentation, we determined that it takes 24 characters to reach the region of memory where the password is stored. This was observed by entering a repeating pattern of characters and monitoring the data segment: after inputting 24 characters, the next character entered appears at the beginning of the password variable's memory location. To completely overwrite the entire password, a total of 32 characters must be entered. This process demonstrates how a buffer overflow can allow input data to spill over into adjacent sensitive variables, such as passwords, in memory.

T2: Take a screenshot of the console output showing how you gained access to the system with an incorrect password.



T3: How can the code be modified to prevent your attack? Describe what changes you would make to the code to prevent unauthorized access.

The primary way to prevent a buffer overflow attack in this scenario is to ensure that the program never writes more data to the input buffer than it can safely hold. This can be achieved by implementing input bounds checking, which restricts the number of characters read from the user to the exact size of the buffer. For example, if the input buffer is 24 bytes, the code should be modified to only read a maximum of 24 characters, ignoring any extra input.

DISCUSSION:

This lab demonstrates the serious security risks posed by buffer overflow vulnerabilities in low-level programming environments. By intentionally exploiting a buffer overflow in RISC-V assembly, we were able to overwrite a password stored in memory and gain unauthorized access,

despite not knowing the correct password. This highlights how a lack of proper input validation and bounds checking can lead to critical software vulnerabilities that can be easily exploited. The experiment also reinforced the importance of understanding computer architecture concepts, such as stack and memory organization, in designing secure programs. Our results showed that even a single oversight in memory management can compromise the integrity and security of an entire system.

Looking ahead, future work could involve implementing and testing more advanced protection mechanisms, such as stack canaries, or hardware-enforced memory protections, to see how well these techniques defend against buffer overflow attacks in practice. Additionally, exploring real-world examples in C or other low-level languages could further illustrate the prevalence and danger of such vulnerabilities. A limitation of our design is that it focuses on a controlled, instructional setting and does not account for more sophisticated attack vectors or multi-layered system defenses found in modern computing environments. Nevertheless, this exercise underscores the need for rigorous input validation and safe programming practices at all levels of software development to prevent similar security breaches.

CONCLUSION:

This lab focused on understanding and exploiting buffer overflow vulnerabilities in RISC-V assembly to demonstrate the importance of secure memory management in low-level programming. By experimenting with user input that exceeded the bounds of a fixed-size buffer, we were able to overwrite an adjacent password variable and gain unauthorized access, effectively illustrating how such vulnerabilities can compromise system security. The primary challenge encountered was ensuring precise tracking of memory offsets to observe and trigger the overflow, which emphasized the need for careful programming and thorough validation. Overall, the lab reinforced the critical role of input validation and proper bounds checking in preventing buffer overflow attacks and highlighted how small implementation oversights can lead to significant security risks.