CDA 4205L Lab #13: Buffer Overflow Attacks

University of South Florida

Lab Date: April 16 (Last Lab for Spring 2025)

Report due: One week after the lab section, see Canvas

This is a group lab. *One submission per group is sufficient.*

There is no in-class work!

Welcome to CDA 4205L Lab #13! The goal of this lab is to give you an example of how understanding topics in computer architecture – in this case, ISAs and memory organization – can expose software-level security issues/vulnerabilities. In particular, we will focus on a *buffer overflow attack* in RISC-V.

Prelab

Stack, Buffers, and Buffer Overflow

Earlier this semester, we discussed how a program uses different regions of memory for storing program data. In particular, we covered the use of the *stack* and the *heap*. When writing low-level programs, e.g., using C or directly using assembly, management of memory is generally left to the programmer. You've already seen how, when writing a procedure in RISC-V assembly, you are responsible for moving the stack pointer to allocate space, preserving saved registers (`s0`, `s1`, etc.), and popping the stack before leaving the procedure code. You've also seen how space can be allocated on the stack for static data (in the `.data` segment in the code).

Now consider this scenario. In the `.data` segment, your program allocates 1) space for a 4-word array, followed by 2) a word representing a constant (integer) in your code. This is the structure in memory:

Variable Name	**A**				**B**
Value	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Another word for (1) is a *buffer*, a contiguous block of computer memory that holds multiple instances of the same data type. When writing to a buffer, it is important to know the size of the buffer. If we write the "`ComputerArchLab`" to the start address of **A** and 4205_{10} to **B**, we get the following:

Variable Name	**A**				**B**
Value	0x706D6F43	0x72657475	0x68637241	0x0062614C	0x0000106D

In this example, if there are no protections in place, writing "`ComputerArchitecture`" words beginning at the start address of **A** would result in overwriting the adjacent memory location (**B**), resulting in the following:

Variable Name	**A**				**B**
Value	0x706D6F43	0x72757465	0x68637241	0x63657469	0x65727574

While this is technically correct, you have overwritten variable **B** (which was supposed to be an integer). Your code will still treat **B** as an integer. Instead of 4205_{10}, it is now 1701999988_{10}, which is certainly not the course number.

Buffer Overflow Attack

This was a fairly benign example, but the possibility of overwriting data on the stack is a very serious security issue. Consider the following, *less* benign example.

Variable Name	InputBuffer		Password	
Value	0x00000000	0x00000000	0x34616463	0x00353032

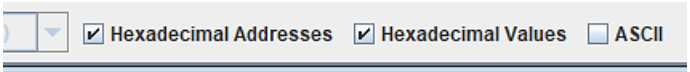
Here, there is an 8-byte input buffer, followed by an 8-byte password buffer. The program will compare the value of **InputBuffer** with **Password** and, if they match, will allow access to the "secure" system. If, however, you manage to overwrite the two words beginning at **Password** with your own data, you could then ensure any text you input will match; e.g., for an input of "aaaabbbbaaaabbbb", the memory contents would look like this:

Variable Name	InputBuffer		Password	
Value	0x61616161	0x62626262	0x61616161	0x62626262

Therefore, if your code compares byte-by-byte, or word-by-word, in either case, the **InputBuffer** would match the **Password**, bypassing the security measure.

Modern compilers will use different techniques to prevent this, and modern programming languages will also use safer routines for taking user input (specifically, ones that truncate the size of the input to match the specified buffer size). Still, it is important to be aware of the potential vulnerability and ensure you use safe coding practices if working with low-level languages.

Lab

1. Download the **buffer_overflow.asm** file from Canvas. This RISC-V code simulates a scenario where a buffer overflow is possible, as described in the pre-lab.
2. Compile and run the code. A dialog box will open asking for you to enter a password. The correct password is "cda4205L". Entering this should grant you access to the system.
3. Re-run and enter an incorrect password, e.g. "cda3201". This should deny access, as the password is incorrect.
4. Observe the data memory view. You can check the **ASCII** box to show the actual characters stored. You should be able to see the text you entered as well as the correct password.
5. 

6. Run the code again, but this time, enter some repeating characters, e.g. "aaaaaa…..a". Observe the data memory view to see where the entered characters were stored. Re-run/repeat this process until you have successfully gained access to the system.

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?
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.

Final Report - Zip your report pdf with answers to all questions and screenshots. Your .zip file per group contains only a PDF report file.