Experiment 1
Buffer Overflow Attack

The goal of this experiment is to investigate how to carry out a buffer overflow attack.

Instructor: Dr. Swarup Bhunia
TAs: Shuo Yang and Reiner Dizon

Case Study

The goal of this experiment is to investigate how to carry out a buffer overflow attack.

In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. This is a special case of the violation of memory safety.

Buffer overflows can be triggered by inputs that are designed to execute code or alter the way the program operates. This may result in erratic program behavior, including memory access errors, incorrect results, a crash, or a breach of system security. Thus, they are the basis of many software vulnerabilities and can be maliciously exploited.

Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array (the built-in buffer type) is within the boundaries of that array. Bounds checking can prevent buffer overflows.

Theory Background

Buffer is a contiguous block of computer memory that holds multiple instances of the same data type. Buffer overflow is a kind of attack that is executed by means of software, but it results in a violation of memory safety.

A buffer overflow occurs when data written to a buffer also corrupts data values in memory addresses adjacent to the destination buffer due to insufficient bounds checking. This can occur when copying data from one buffer to another without first checking that the data fits within the destination buffer. Stack overflow, heap overflow and integer overflow are various kinds of buffer overflow attacks.

A simple example of the buffer overflow attack is shown below. A program has two data items which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte integer, B.

```
char           A[8] = "";
unsigned short B    = 1979;
```

Initially, A contains nothing but zero bytes, and B contains the number 1979.

variable name	A	B
value	[null string]	1979

Now, the program does not check the incoming data length to be stored at A. Suppose an attacker sends a 9-character long string that encodes to 10 bytes to store in A, but A can take only 8 bytes. By failing to check the length of the string, it also overwrites the value of B as shown below.

variable name	A								B
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856

Integer overflow is something that occurs when an arithmetic operation tries to create a numeric value that is too large in given (available) storage. Adding 1 to largest value is integer overflow. The result will be the largest negative. Arithmetic overflow occurs when any arithmetic calculation produces a result that is great than storage space. One way is to check overflow bit. Students will get into details of those and through executing code they will get an idea how exactly buffer overflow can be attacked, and malicious code may be injected.

Experiment Set-up: Configuration

- This experiment will require a computer with Linux system.
- You can use either a personal Ubuntu machine or the ECE's Linux server.
- If you're using the ECE's Linux Server [9], you must either be using on campus Internet connection or use the UF VPN [10] if you're accessing it off campus.
  - You can get access to ECE's Linux server through SSH clients, such as "MobaXterm" [11] or "Putty" [12].
  - Download and install an SSH client on your PC and configure as in Figure 1.
  - The host name is "linux.ece.ufl.edu".
  - The user name is your Gatorlink username as shown in Figure 1. You will be required to input your password for your Gatorlink account.
- Download the Experiment_1_code.zip to the Linux machine which you are using and unzip it.

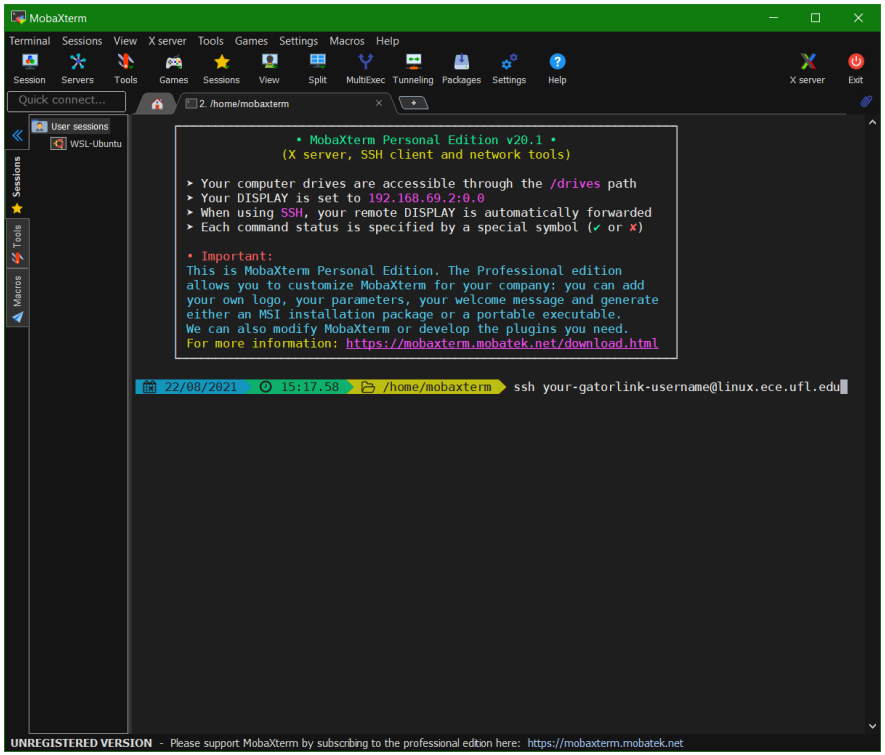

Figure 1 SSH Configuration when using MobaXterm

Experiment Set-up: Instructions

Students will get into details of those and through executing code they will get an idea how exactly buffer overflow can be attacked, and malicious code may be injected.

Part I: Code and test simple examples

Students should code buffer overflow samples for heap overflow, integer overflow and stack overflow. These can be implemented as functions in one C code. Attach this code with your submission.

Step 1: Write your code as mentioned above.

Step 2: In Linux, compile the code using: `gcc -w -fno-stack-protector -Wall -pedantic expt1_1.c -o expt1_1`

Step 3: Run it: `./expt1_1`

The following tasks will surely help you to understand how a buffer can be overflowed and how it can be exploited as an attack. Also, you will see how the execution path of running process may be changed by overwriting the return address with the address that points to the location of a malicious code which gets executed.

Part II: Buffer Overflow Simple Challenge

In these next parts, you will demonstrate what you can do by overflowing the buffer. In this part, overflowing the buffer even if you enter the wrong password may still get you access. It is a perfect example of bad coding, and how one can exploit it.

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`

Note: the key is “gainesville”.

For more information on stack protector, look at the references.

Part III: Buffer Overflow Malicious Code Injection

In this part, we look at how the buffer overflow attack can be used to inject malicious code. Follow the steps below to perform this attack.

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`

Please take a screenshot of a successful attack and attach it as part of your submission.

Measurement, Calculation, and Question

Part I:

1) Submit the code containing the three examples as functions inside one C code.
2) Take screenshots of the output of the three examples.
3) Suggest ways to improve it against this attack implementation.

Part II:

1) What is output when you enter correct key (key = gainesville)?
    a. Take a screenshot of the output.
2) What is output when you enter a wrong key between 10-12 characters? (e.g., key = ghgjhjhjkh)
    a. Take a screenshot of the output.
3) What is output when you enter wrong key longer than 12 characters? (e.g., key = ghgjhkjhkjkhkhkjhkjlkjlkjkjlkjlkjlkjkljkjlkjk)
    a. Take a screenshot of the output.
4) Write your conclusions.
5) Why do all these issues happen?
6) How can you correct them?

Part III:

1) What is the endianness of the address inside the Perl script?
2) Take a screenshot of an successful attack.
3) Write your conclusions.
4) Suggest ways to improve the program against this attack implementation.

Optional Follow-up

Advanced Techniques (optional for students to do: Student will get extra credit if they work on it)

Use of Valgrind tool helps in preventing and detecting buffer overflow attack by proper memory auditing. Students who are interested can follow the user manual in the references, try it, and submit their work.

Prevention Techniques

1) Proper and secure coding
2) Avoid using Library which are unsafe
3) Avoiding Buffer Overflows
    a. Compiler based run time bounds checking
    b. Library based runtime bounds checking
        i. Libsafe
        ii. Libverify
        iii. Libparanoia

☞ Lab Report Guidelines

1. In your report, give the results when you ran the code with different inputs.
2. Include screenshots for all the steps.
3. Give answers to all the questions.
4. Give suggestions for improving the program.

References and Further Reading

[1] http://www.outflux.net/blog/archives/2014/01/27/fstack-protector-strong/

[2] http://www.slideshare.net/aidanshribman/valgrind-29203055

[3] http://pages.cs.wisc.edu/~bart/537/valgrind.html

[4] http://www.freebsd.org/doc/en/books/developers-handbook/secure-bufferov.html

[5] http://www.outflux.net/blog/archives/2014/01/27/fstack-protector-strong/

[6] http://pages.cs.wisc.edu/~bart/537/valgrind.html

[7] http://www.cs.colostate.edu/~massey/Teaching/cs356/RestrictedAccess/Slides/356lecture21.pdf

[8] http://projects.webappsec.org/w/page/13246946/Integer%20Overflows

[9] https://www.ece.ufl.edu/resources/it/ece-student-computing-access/

[10] https://vpn.ufl.edu/

[11] https://mobaxterm.mobatek.net/

[12] https://www.putty.org/