Experiment 1: Buffer Overflow Attacks

Hardware Security Lab

EEL:5934 Section: 1265

Submitted by -

**Atul Prasad Deb Nath  
UF ID: 09635509**

**Sarah Amir**

**UF ID: 67935829**

Electrical and Computer Engineering (ECE) Department

University of Florida

August 31, 2016

**Abstract:**

The primary objectives of the experiment are to carry out buffer overflow attacks, observe the consequences, and learn about different correction as well as prevention techniques. The experiment consisted of two parts. In the first part, it is demonstrated that how buffer overflow can cause security breach and provide unauthorized access to users with malicious intents. A sample C code with no buffer overflow protection is exploited for the demonstration. While observing the anomaly caused by buffer overrun, different solutions are contemplated to fix the security threat. Consequently, the solution is implemented by using secure functions in the C code replacing the unsafe functions. As for the second part of the experiment, the tasks consisted exploitation of buffer overflow situation and injection of malicious code. A sample C code is used to demonstrate how buffer overflow attack can be utilized to execute arbitrary code. The vulnerability caused by using an unsafe function in the sample code is exploited to overwrite the return address of the good function and activate malicious function. The security issue is solved by replacing the dangerous function with a safe one. In an additional part of the experiment, the Valgrind tool is used to detect and prevent potential buffer overflow vulnerability.

# Experiment Details:

## Goal:

The goal of this experiment is to observe how buffer overflow attack is carried out and learn about different countermeasures to detect and prevent such attacks.

## Experimental Setup:

* The experiment is performed on Linux terminal with GCC compiler.
* For the given code and set of arguments, it is preferable to use a 32-bit system. In case of a 64-bit system, the argument set would need to be changed (in the given pearl script).

## Part 1:

## Experiment Steps:

1. A sample C code is given, named ccode.c.
2. The code is compiled with GCC in Linux terminal, explicitly turning off the stack protector of GCC, with *-fno-stack-protector*.
3. The executable is run with desired arguments. The key argument is provided as
   1. Right key
   2. Wrong key
   3. Long wrong key

# Result and Observation:

1. **Right key**

For right key, the access is given as expected. Figure 1 shows the result.

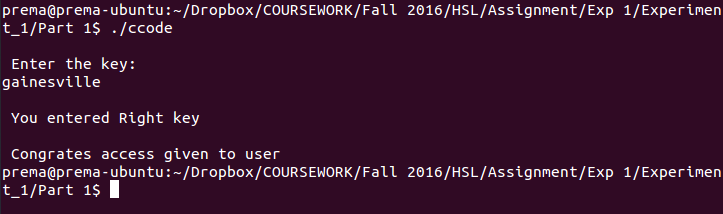


Figure 1 : The execution of code ccode.c with proper key

1. **Wrong key**

For any wrong key smaller than the buffer size, the access is not given. Figure 2 shows the result for this condition.

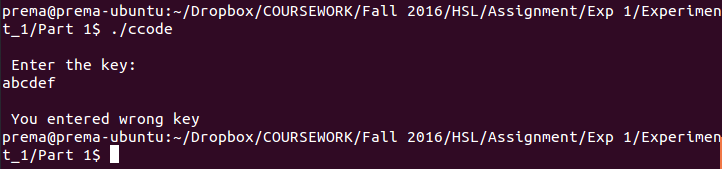


Figure 2 : The execution of code ccode.c with small wrong key

1. **Long wrong key**

For any wrong key longer than buffer size, the code is capable of detecting the wrong key. However, it gives access to the attacker due to buffer overflow. Figure 3 shows the terminal window of this execution.

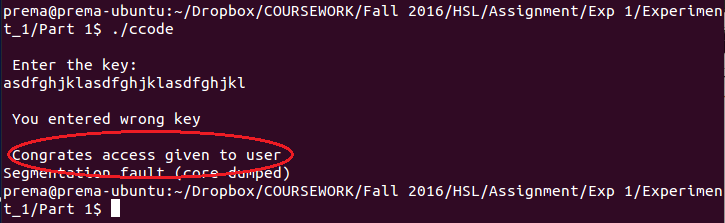


Figure 3 : The execution of code ccode.c with long wrong key

1. **Our conclusion**

From the observations made by compiling and executing the sample code, it can be concluded that if a long wrong password is provided by the user, he or she will be granted access to the system despite the fact that the password is wrong. This erroneous situation is caused by buffer overrun.

1. **Explanation for the anomaly**

In the sample C code, the *gets()* function is used. However, the *gets()* function does not inspect the bounds of the array. As a result, it can write strings of larger length than the buffer size. Consequently, when the long wrong password is entered, it exceeded the buffer size and the adjacent memory is overwritten. In the case of provided sample code, the memory of integer xyz is overwritten and the value become non-zero despite a wrong password is entered. Thus, the access is granted to the attacker who took advantage of the buffer overflow scenario.

1. **Remedy for the situation**

As a remedy for the current C code,

* 1. The *gets()* function should be replaced with *fgets()* function and the *strncmp()* function should be entered in place of *strcmp()* functions. Thus, the buffer overflow situation can be corrected. Code 1 given below shows such correction.
  2. The *getchar()* function can also be used in a loop. The number of loops can be controlled by the length of the allocated location.
  3. String lengths can be compared before using *strcmp()* function to avoid accidental boundary breaching.

**Code 1 : Corrected code for part 1**



**Result provided by correct code:** Figure 4 shows the execution of the corrected code for right key, wrong key and long wrong key.

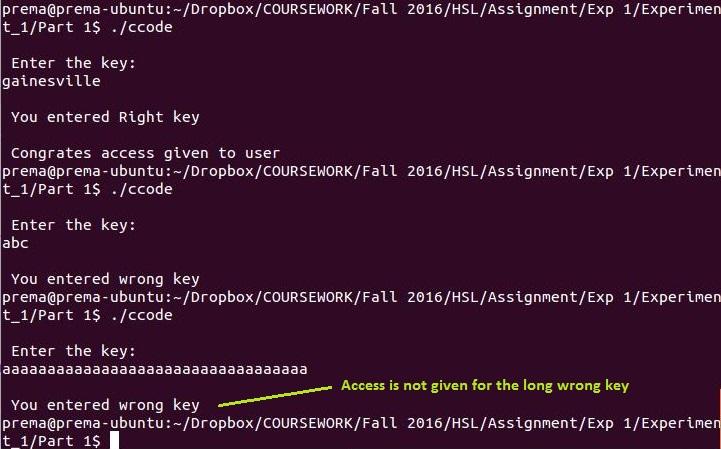


Figure 4 : The execution of corrected ccode.c with right key, short wrong key and long wrong key

## Part 2:

## Experiment Steps:

1. A sample C code is given, named bo\_test.c
2. The code is compiled with GCC in Linux terminal, explicitly turning off the stack protector of GCC, with *-fno-stack-protector*.
3. The executable is run with desired arguments. At first, it is run with a key with length less than 10 characters. The code runs normally and prints the address of malicious code.The address of the malicious code is used as an argument for the second run.
4. To run the attack and bypass the segmentation fault detection, the arguments are written in a perl script. The argument is comprised of ten keys concatenated with the address of the malicious code. The perl script is run in the terminal.

# Result and Observation:

1. In the first step, the code runs normally. As written in the code, it displays the addresses of the good function and malicious function.

Figure 5 shows the output for the first run. The malicious address marked by the green circle is copied for the next step of the experiment.

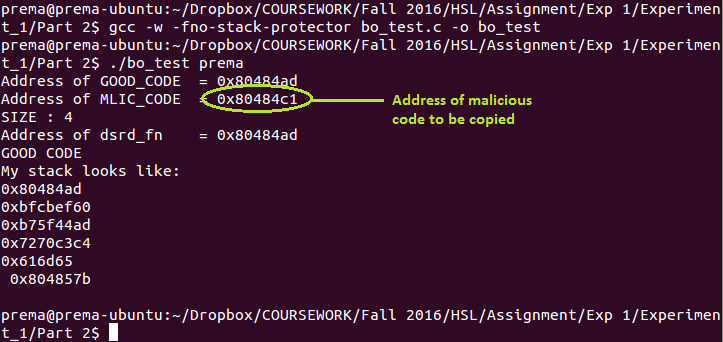
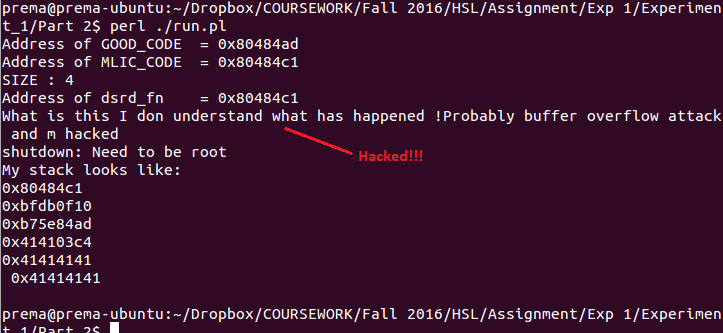


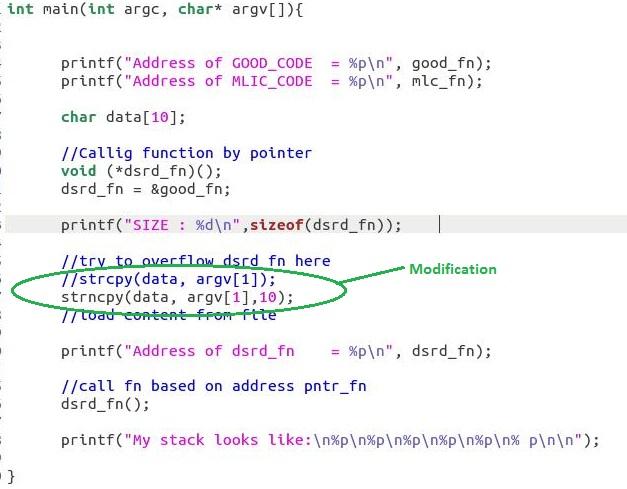
Figure 5 : First execution of the code bo\_test in terminal. It gives the address for the malicious code.

1. In the second step, because of the arguments, Buffer Overflow happens and the malicious function is called instead of the good function. Figure 6 shows the output.

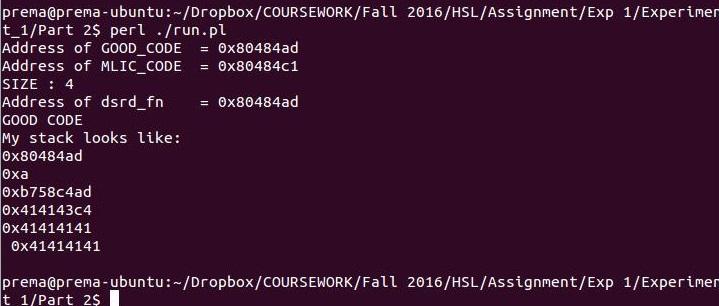
  
Figure 6 : The execution of the malicious code due to Buffer Overflow

1. The buffer overflow occurred because the the array to take input, named data[10] was defined and just after that, the pointer *dsrd\_fn* was defined. In the stack, the space reserved for data[10] ends, and the space for the pointer to save the address starts. The *strcpy()* function do not check if the prior string location has fewer space defined than the later one or not. It only copies the context. Because of that, the argument. which has 10 characters and the replacement address at first fills up the allocation for data with the 10 characters. Then it rewrites the address of the good function in the space allocated for pointer *dsrd\_fn*. For this, when the desired function is called by the pointer *dsrd\_fn*, the malicious function is called. It is because now the pointer has the address of the malicious function.
2. Replacing the vulnerable function strcpy() with strncpy protects the code from overflowing the buffer. The strncpy() function has a integer argument which says it how many characters are to be copied. Code 2 shows the modification made. Following that, in figure 7, the result for this modified code is presented.

**Code 2: Corrected code for part 2**



**Result provided by the corrected code:**

  
Figure 7 : The execution of the modified bo\_test program no longer have the buffer overflow vulnerability

# Optional Follow-up Section:

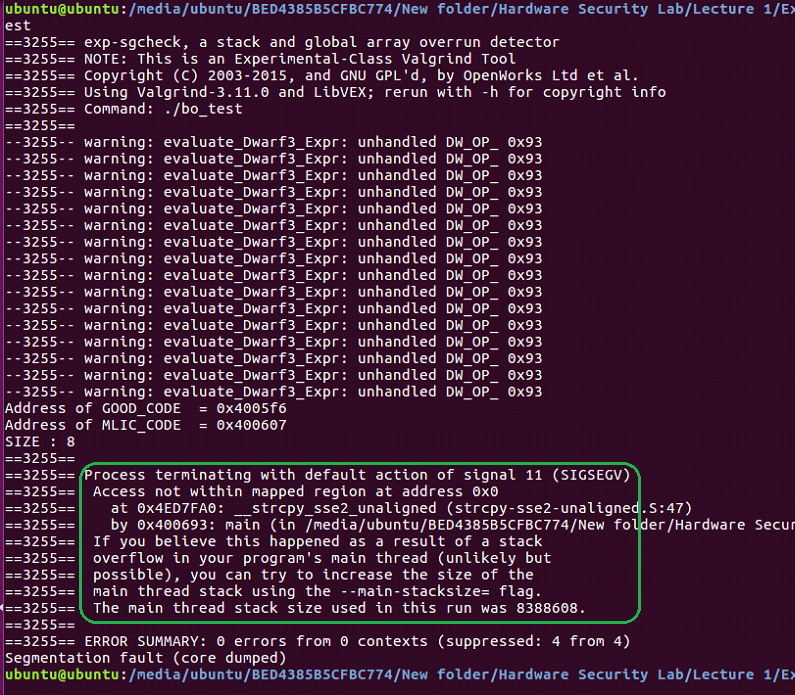
The Valgrind is basically an instrumentation framework that provides the opportunity for developing dynamic analysis tools. It comes along with a set of tools that help debugging, profiling, or performing other similar tasks to improve the programs. In the option follow-up section of experiment 1, two different tools of Valgrind are utilized to detect the buffer overrun vulnerability in the provided sample C code. 

Figure 8 : The error report produced by the *sgcheck* tool of **Valgrind**

1. At first, the tool *memcheck* is used to identify the buffer overrun possibility of the given program. The error report produced by the *memcheck* tool is provided in figure 9.
2. Another tool named *sgcheck* is used to check the buffer overflow possibility in the sample code. The error report generated by the *sgcheck* tool of Valgrind is given in figure 8.

Based on the error reports of the tools of Valgrind, it can be concluded that Valgrind is successful in detecting and preventing buffer overflow attacks by proper memory auditing.

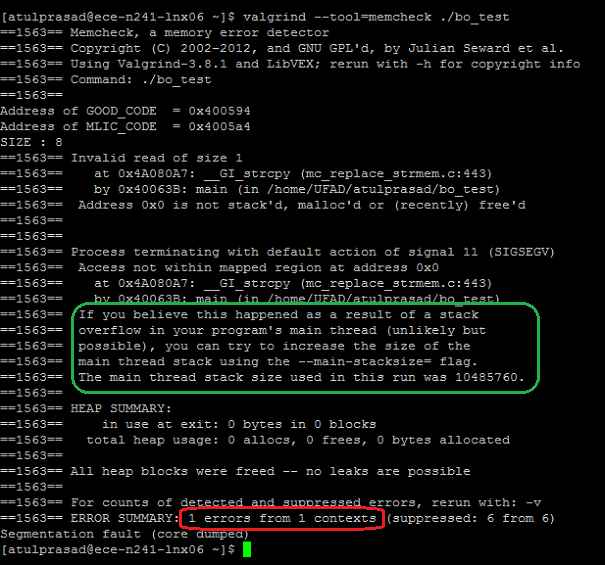


Figure 9 : The error report produced by *memcheck* tool of **Valgrind**

# Discussion and Summary

## Interesting Observations:

If the program bo\_test is called from the terminal directly, with the same argument, it ends up with “segmentation fault”, instead of buffer overflowing and calling the malicious function. The phenomenon implies GCC has a buffer overflow protection but Perl does not have such.

## Aspects of Experiment:

The experiment was very helpful to observe and learn how buffer overflow attack exploits a vulnerable code. Though the code itself is not malicious, the lack of countermeasure for buffer overflow attack lets hackers have access and run malicious activity.

## Summary:

In the first part of this experiment, it is demonstrated that how buffer overflow can cause security breach and provide unauthorized access to users with malicious intents. A sample C code, vulnerable to Buffer Overflow Attack, is given along with instruction for the demonstration of the attack. The code is written to restrict an access to some resource without a correct password. Due to Buffer Overflow, if the wrong password is larger than the buffer size, the program malfunctions and gives wrongful access to attacker. As for the second part of the experiment, the tasks consisted exploitation of buffer overflow situation to exploit the program to run a malicious one. The provided C code demonstrates how the the buffer size definition and pointer definition can cause unintended call to some other memory location. As an optional follow up task, proper memory auditing is done on the sample codes using the Valgrind tool. It helps to detect and prevent buffer overrun attacks launched on vulnerable codes.

## Countermeasures:

Buffer overflow can be avoided in many ways. Several are listed below:

* Avoiding using dangerous functions can be a solution. The gets() function should be replaced with fgets() function and the strncmp() function should be entered in place of strcmp() functions.
* The external input can be checked if that is longer than the allocated area before casting.
* Compiler should always check destination area before casting or saving data.
* Valgrind tools like memcheck or sgcheck can be used to determine and protect buffer overflow situations.
* As it was suggested in the instruction of this experiments, library functions like *Libsafe*, *Libverify* or *Libparanoia* can help to check the boundary conditions.