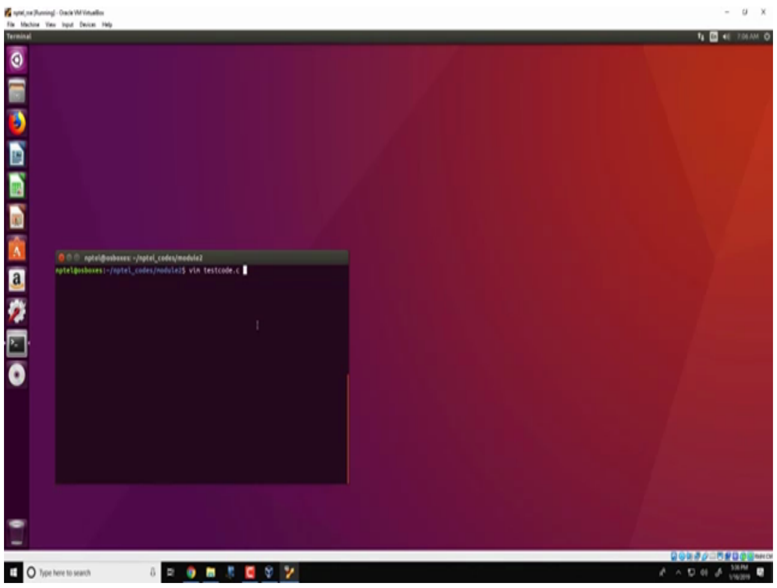
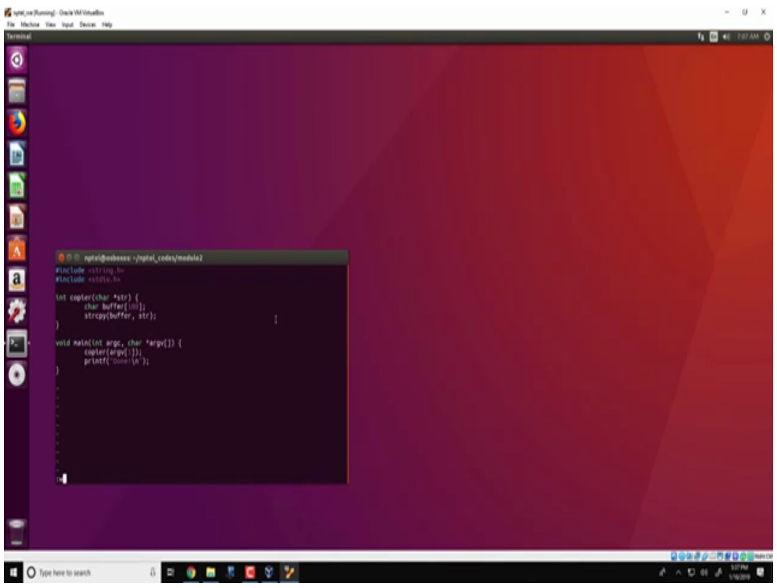
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Demonstration of Canaries, W^X, and ASLR to prevent Buffer Overflow Attacks

Hello and welcome to this demonstration in the course for secure system engineering. So in the previous demo what we have seen is that we overflowed a buffer and we were able to execute a payload and we this payload actually created a shell and what we mentioned is that if an attacker creates such a shell forcing a particular application to be subverted from its execution, the attacker would be able to run whatever is possible from that shell.

(Refer Slide Time: 0:52)



```
nptel@osboxes:~/nptel_codes/module2$ vim testcode.c
```



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

int copier(char *str) {
        char buffer[100];
        strcpy(buffer, str);
}

void main(int argc, char *argv[]) {
        copier(argv[1]);
        printf("Done!\n");
}
```

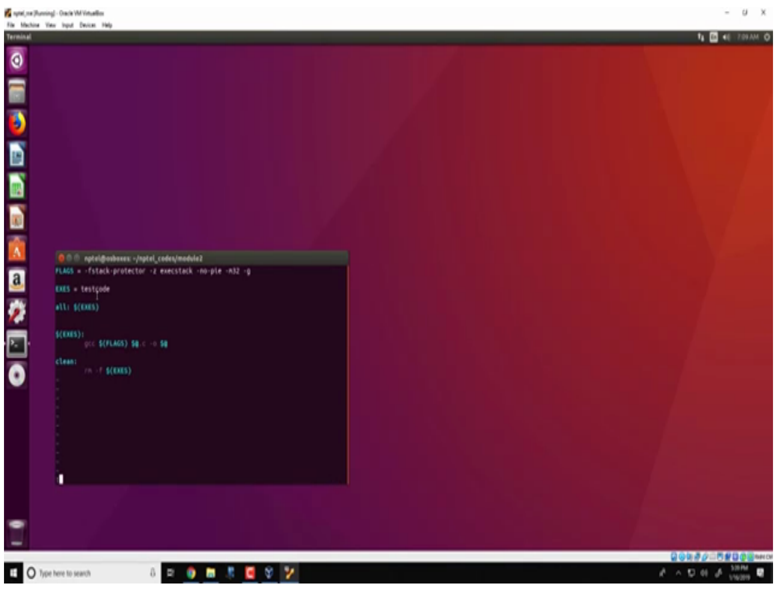
So in the other videos that we have looked at we have seen that there are several countermeasures that have been implemented in standard systems. So in fact we had looked at three different countermeasures one is the NX bit or the WX or X bit which is present in all Intel AMD processors as well as many of the microcontrollers as well. So what we have seen is that, this bit would ensure that a particular page in memory is either executable or is writeable.

So therefore the example in the stack this particular bit would ensure that you cannot execute code from the stack and other countermeasure that is implemented by some of the modern day compilers is by the use of canaries the canaries present in each stack frame would detect that a buffer is overflowing and crossing that particular stack frame and this would be caught during the function return and the subversion of the execution is prevented.

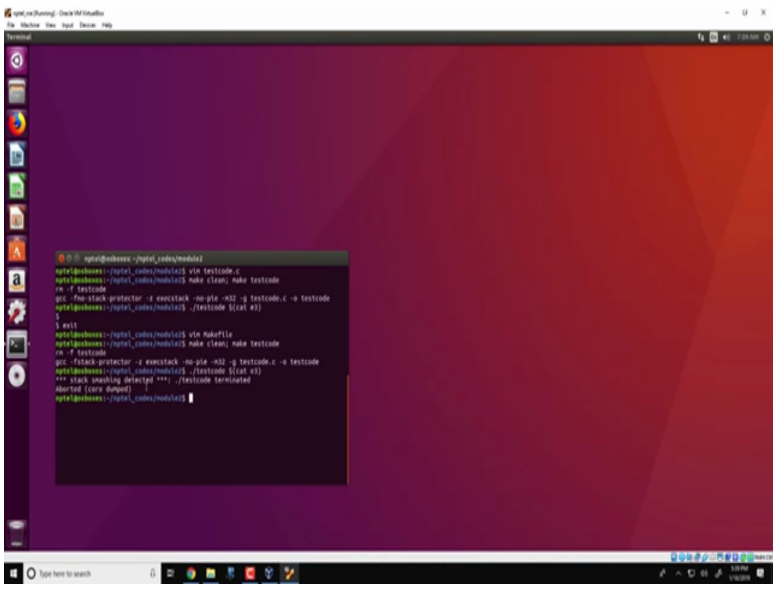
The third thing that we also looked at was address space layout randomization or ASLR with this a countermeasure, what was possible was that the locations of the various modules within a particular program is randomized at each run. Therefore, the attacker would not be able to specify where the subversion should occur and to which location should the return be present, on other words the attacker will find it difficult to actually identify the address at which the payload would be present.

So to demonstrate these three countermeasures we look at the previous example that we took of the buffer overflow. So what we do is that we take the same example as we have done in the previous video which is test code dot code dot c, which comprises of these two functions and as we seen in the previous video we overflow this buffer this local buffer and using the vulnerability in the string copy forces a payload which would create should shell to execute.

(Refer Slide Time: 3:21)



```
FLAGS = -fstack-protector -z execstack -no-pie -m32 -g

EXES = testcode

all: $(EXES)


$(EXES):
	gcc $(FLAGS) $@.c -o $@

clean:
	rm -f $(EXES)
```



```
nptel@osboxes:~/nptel_codes/module2$ vim testcode.c
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fno-stack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
$
$ exit
nptel@osboxes:~/nptel_codes/module2$ vim Makefile
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fstack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
*** stack smashing detected ***: ./testcode terminated
Aborted (core dumped)
nptel@osboxes:~/nptel_codes/module2$ 
```

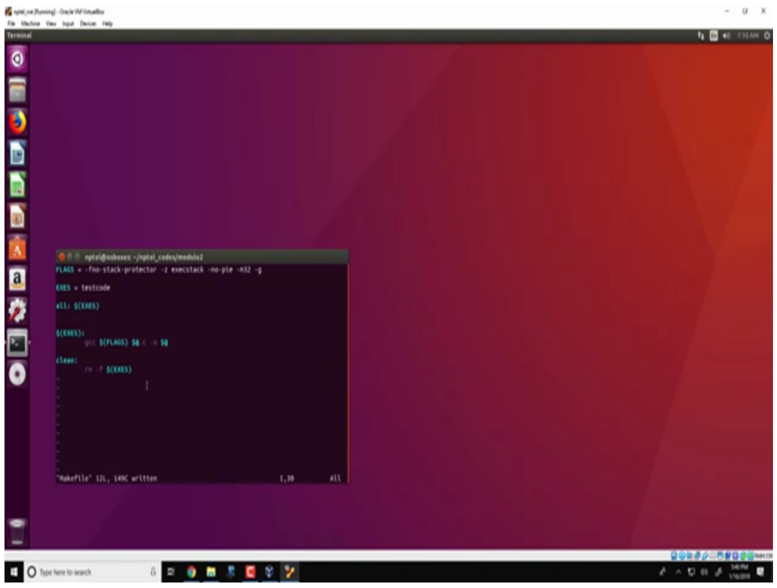
So we will see an example of this working again, so we first make this as follows, okay and then we can run this particular code using this argument. So what we are passing here is the file E3 which comprises of the payload comprising of the shell code that we want to execute and when we run this what we see is that it successfully creates a shell. So this particular program is running because we have disabled all the countermeasures so we have disabled ASLR, we have disabled canaries, as well as we have disabled the stack protection.

So what we will do is that we will enable each of these one by one and then we will see how this shell code is prevented. So the first thing we will look at is that of canaries. Now canaries is added by the compiler by default and what we have done previously was that we have specified this compiler parameter minus F no stack protector which would actually disable the canaries.

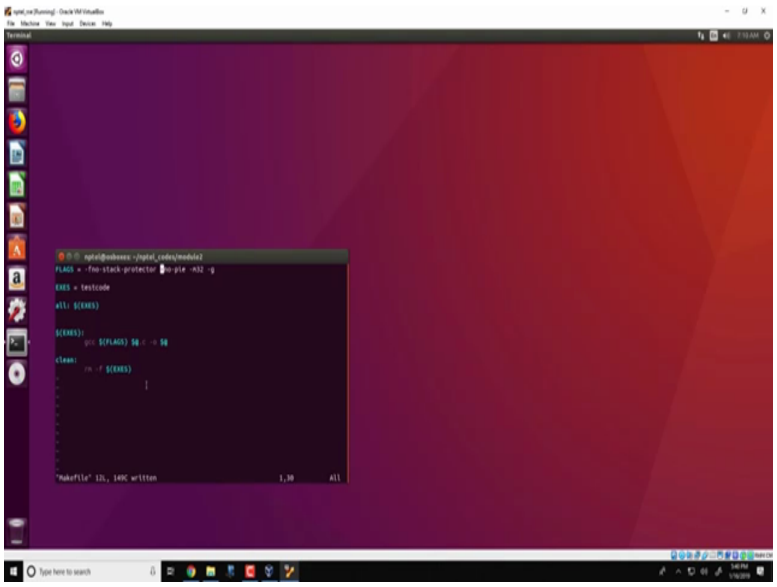
Now suppose we enable canaries let us see what would happen. So instead of minus F no stack protector we would change this to minus F stack protector which forces that canaries be present in every function. So we would compile the code again, notice that it is compiled now such that canaries would be present in the functions. So now if you run the same executable giving the same payload, what we obtain is that stacks machine gets detected.

So what has happened here is due to the edition of the canaries in each functions stack the buffer overflows due to the string copy gets detected and caught by these canaries and therefore before subversion actually obtain is done the program terminates with a stack smashing detection.

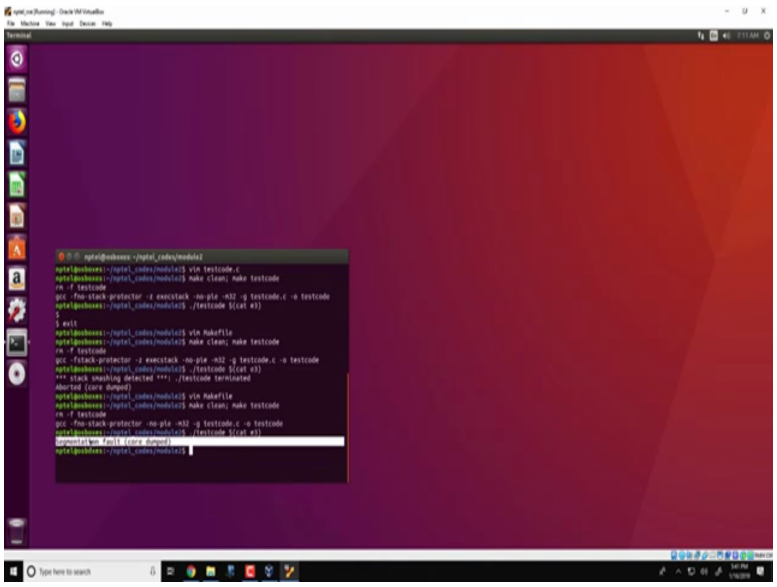
(Refer Slide Time: 5:52)



```
nptel@osboxes: ~/nptel_codes/module2
FLAGS = -fno-stack-protector -z execstack -no-pie -m32 -g

EXES = testcode

all: $(EXES)


$(EXES):
        gcc $(FLAGS) $@.c -o $@

clean:
        rm -f $(EXES)
~
~
~
~
~
~
~
~
~
~
"Makefile" 12L, 149C written                          1,30          All
```



```
FLAGS = -fno-stack-protector -no-pie -m32 -g

EXES = testcode

all: $(EXES)


$(EXES):
	gcc $(FLAGS) $@.c -o $@

clean:
	rm -f $(EXES)
~
~
~
~
~
~
~
~
~
~
"Makefile" 12L, 149C written                                  1,30          All
```

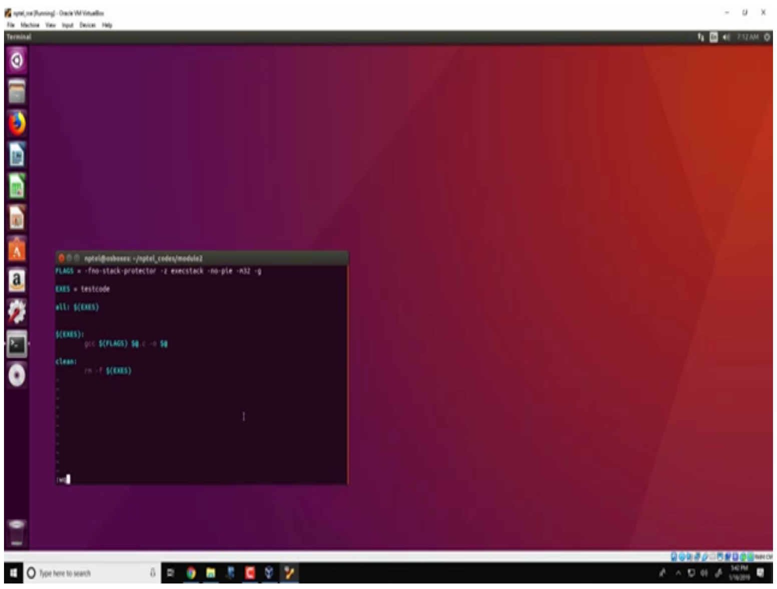


```
nptel@osboxes:~/nptel_codes/module2$ vim testcode.c
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fno-stack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
$
$ exit
nptel@osboxes:~/nptel_codes/module2$ vim Makefile
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fstack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
*** stack smashing detected ***: ./testcode terminated
Aborted (core dumped)
nptel@osboxes:~/nptel_codes/module2$ vim Makefile
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fno-stack-protector -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
Segmentation fault (core dumped)
nptel@osboxes:~/nptel_codes/module2$
```

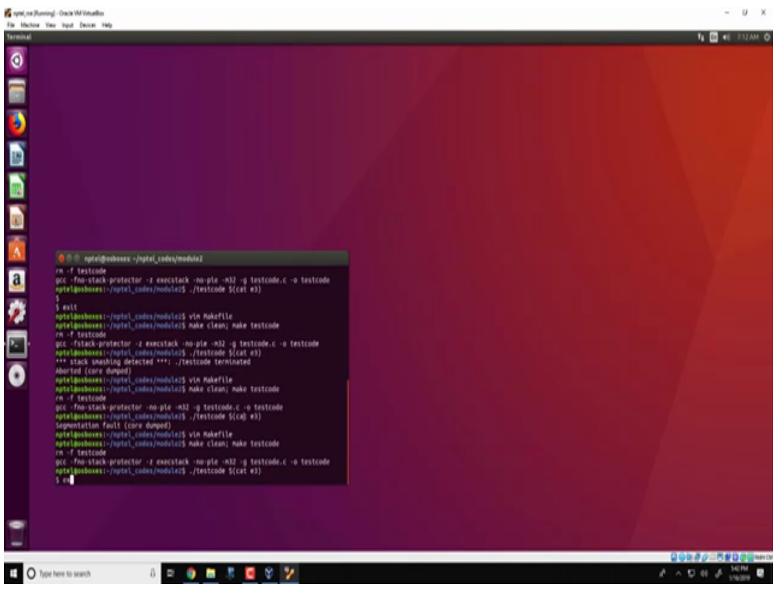
So let us add the canaries again just to get things back and look at the next aspect. So the next thing which we will look at is the WX or X bit. Now the WX or X bit by default is enabled in every Linux system and what we have done previously was to disable this WX or X setting with this minus Z exec stack. So if I remove this particular command line parameter, compile the program as before and execute the program, what happens is that instead of subverting execution and creating the payload we get a segmentation for it.

The reason being is that this payload (has) would successfully overflow the buffer and it will overflow the return address and modify the return address and act the return of the function it would try to execute code from the stack. Now since this particular function by default would have its stack as non-executable therefore the processor detects this as an illegal execution being made and falls the program and therefore the program terminates.

(Refer Slide Time: 7:32)



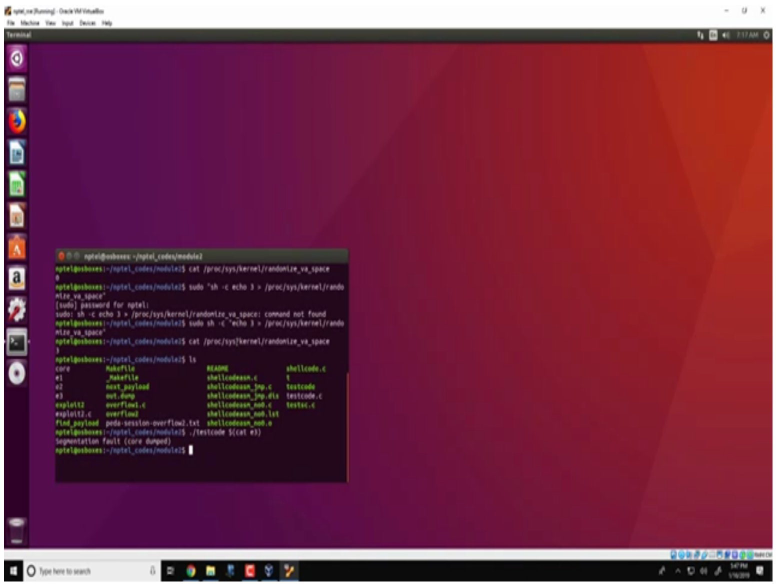
```
FLAGS = -fno-stack-protector -z execstack -no-pie -m32 -g

EXES = testcode

all: $(EXES)


$(EXES):
	gcc $(FLAGS) $@.c -o $@

clean:
	rm -f $(EXES)
~
~
~
~
~
~
~
~
~
~
~
:wq
```



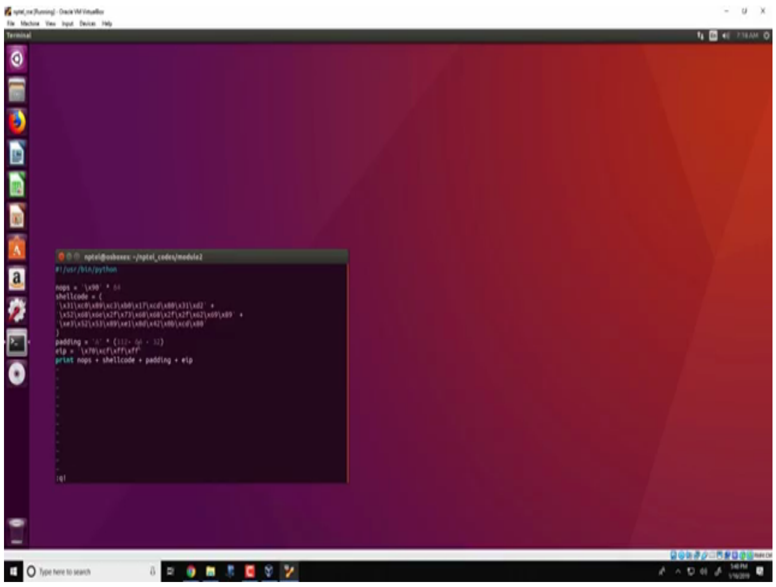
```
rm -f testcode
gcc -fno-stack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
$
$ exit
nptel@osboxes:~/nptel_codes/module2$ vim Makefile
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fstack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
*** stack smashing detected ***: ./testcode terminated
Aborted (core dumped)
nptel@osboxes:~/nptel_codes/module2$ vim Makefile
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fno-stack-protector -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
Segmentation fault (core dumped)
nptel@osboxes:~/nptel_codes/module2$ vim Makefile
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f testcode
gcc -fno-stack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
$ ex
```

So let us get this back and we will put this command line parameter again to ensure that the stack becomes executable and we are able to run a payload from the stack and therefore we are essentially disabling not just the canaries but also disabling the check for execution from the stack, we run this code again and we see that we obtain the stack due to the two countermeasures being disabled.

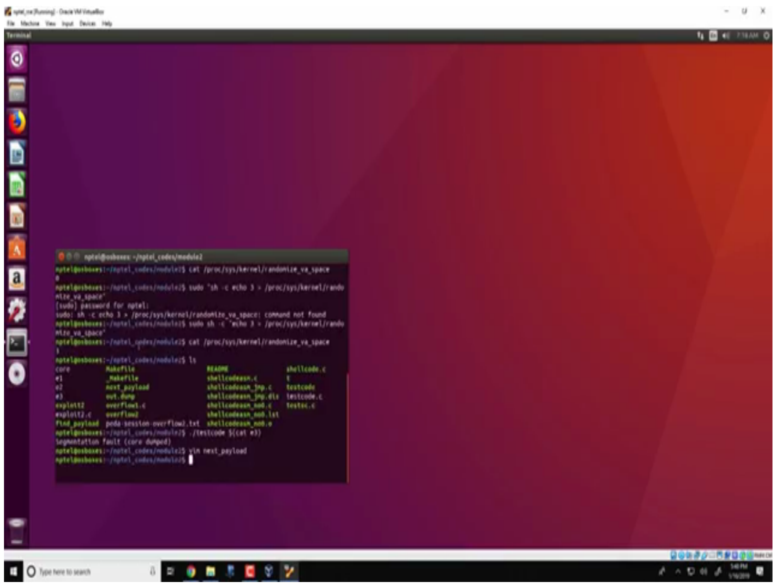
(Refer Slide Time: 8:16)



```
nptel@osboxes:~/nptel_codes/module2$ cat /proc/sys/kernel/randomize_va_space
0
nptel@osboxes:~/nptel_codes/module2$ sudo "sh -c echo 3 > /proc/sys/kernel/rando
mize_va_space"
[sudo] password for nptel:
sudo: sh -c echo 3 > /proc/sys/kernel/randomize_va_space: command not found
nptel@osboxes:~/nptel_codes/module2$ sudo sh -c "echo 3 > /proc/sys/kernel/rando
mize_va_space"
nptel@osboxes:~/nptel_codes/module2$ cat /proc/sys/kernel/randomize_va_space
3
nptel@osboxes:~/nptel_codes/module2$ ls
core          Makefile                  README               shellcode.c
e1            _Makefile                 shellcodeasm.c       t
e2            next_payload              shellcodeasm_jmp.c   testcode
e3            out.dump                  shellcodeasm_jmp.dis testcode.c
exploit2      overflow1.c               shellcodeasm_no0.c   testsc.c
exploit2.c    overflow2                 shellcodeasm_no0.lst
find_payload  peda-session-overflow2.txt shellcodeasm_no0.o
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
Segmentation fault (core dumped)
nptel@osboxes:~/nptel_codes/module2$ 
```



```
#!/usr/bin/python

nops = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112- 64 - 32)
eip = '\x70\xcf\xff\xff'
print nops + shellcode + padding + eip
~
~
~
~
~
~
~
~
~
~
~
~
:q!
```



```
nptel@osboxes:~/nptel_codes/module2$ cat /proc/sys/kernel/randomize_va_space
0
nptel@osboxes:~/nptel_codes/module2$ sudo "sh -c echo 3 > /proc/sys/kernel/rando
mize_va_space"
[sudo] password for nptel:
sudo: sh -c echo 3 > /proc/sys/kernel/randomize_va_space: command not found
nptel@osboxes:~/nptel_codes/module2$ sudo sh -c "echo 3 > /proc/sys/kernel/rando
mize_va_space"
nptel@osboxes:~/nptel_codes/module2$ cat /proc/sys/kernel/randomize_va_space
3
nptel@osboxes:~/nptel_codes/module2$ ls
core          Makefile                  README               shellcode.c
e1            _Makefile                 shellcodeasm.c       t
e2            next_payload              shellcodeasm_jmp.c   testcode
e3            out.dump                  shellcodeasm_jmp.dis testcode.c
exploit2      overflow1.c               shellcodeasm_no0.c   testsc.c
exploit2.c    overflow2                 shellcodeasm_no0.lst
find_payload  peda-session-overflow2.txt shellcodeasm_no0.o
nptel@osboxes:~/nptel_codes/module2$ ./testcode $(cat e3)
Segmentation fault (core dumped)
nptel@osboxes:~/nptel_codes/module2$ vim next_payload
nptel@osboxes:~/nptel_codes/module2$ 
```

So now let us look at the final countermeasure which is ASLR, now the status of ASLR on Linux systems can be obtained from this particular file randomize underscore VA underscore space which is present in this directory slash proc slash sys kernel randomize underscore VA space. A value of 0 in this file indicates that ASLR is disabled on this system. In order to enable ASLR on this particular system temporally what we need to do is change the contents of this file to either 1, 2, or 3.

So let us say that we actually change this value to 3, we can do so by specifying sudo because changing the file requires sudo, sudo sh minus c echo 3 and this you redirect the output into slash proc sys kernel (())(9:30) slash randomize VA underscore space, so we need the password which is which are 1, 2, 3. So in this way we have changed the value of randomize underscore VA underscore space from 0 to 3.

So we can verify that it indeed has changed by printing out the result the contents of that particular file. So what we have done here now is we have enabled ASLR, so with this enabling let us see if the program still works. So we would run the same program and we see that it has successfully prevented the attack. So because ASLR is enabled therefore we get a segmentation fault.

The reason being that the return address which we have overwritten next (())(10:29) to the location ffffcf70 would no longer have the stack the randomization would ensure that the location of the stack would be changed with every execution and this would prevent the attack, thank you.