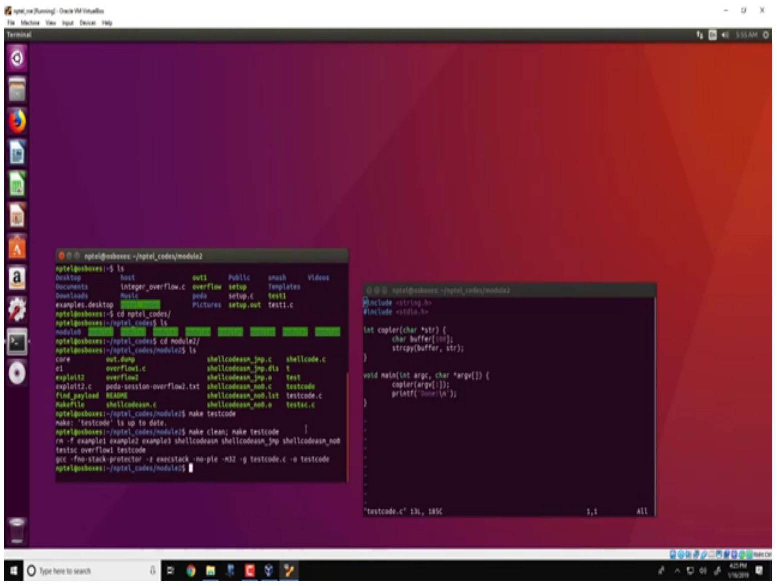
Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Module 1
Lecture 7
Buffer Overflow-Demo

Hello and welcome to this demonstration in the NPTEL curse for Secure System Engineering, in this demonstration and the one that follows we will be seeing how we could inject a payload into a program and cause that payload to execute in particular we will be looking at the payload in detail and we will be seeing essentially where and how that payload can be modified.

(Refer Slide Time: 00:42)



```
nptel@osboxes:~$ ls
Desktop          host                out1      Public     snash      Videos
Documents        integer_overflow.c  overflow  setup      Templates
Downloads        Music               peda      setup.c    test1
examples.desktop [illegible]         Pictures  setup.out  test1.c
nptel@osboxes:~$ cd nptel_codes/
nptel@osboxes:~/nptel_codes$ ls
module0  [illegible]
nptel@osboxes:~/nptel_codes$ cd module2/
nptel@osboxes:~/nptel_codes/module2$ ls
core        out.dump                    shellcodeasm_jmp.c    shellcode.c
c1          overflow1.c                 shellcodeasm_jmp.dis  t
exploit2    overflow2                   shellcodeasm_jmp.o    test
exploit2.c  peda-session-overflow2.txt  shellcodeasm_no0.c    testcode
find_payload README                     shellcodeasm_no0.lst  testcode.c
Makefile    shellcodeasm.c              shellcodeasm_no0.o    testsc.c
nptel@osboxes:~/nptel_codes/module2$ make testcode
make: 'testcode' is up to date.
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f example1 example2 example3 shellcodeasm shellcodeasm_jmp shellcodeasm_no0
testsc overflow1 testcode
gcc -fno-stack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$
```

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

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

void main(int argc, char *argv[]) {
        copier(argv[1]);
        printf("Done!\n");
}
~
"testcode.c" 13L, 185C                                  1,1           All
```

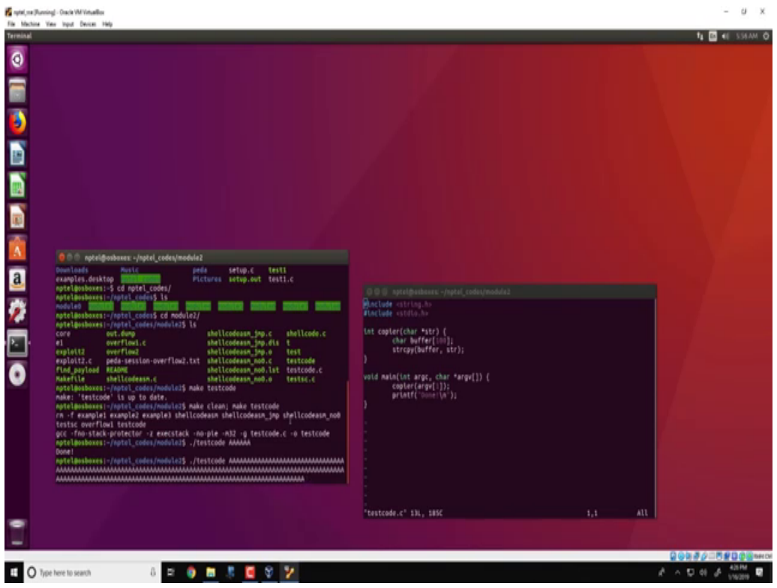
So this code is present in the big bucket depository just like all the other videos that we have done. Ok so we go to (())(00:54) codes see the module 2 and we take this particular written program which is in this file called we will be looking at file which is called VIN test code dot C so as you can see over here this is a really small C program it comprises of two functions the first is the copier function and the main function.

The main function invokes the copier function and passes it argument 1 so argue 1 is what is obtained from the mains command line and copier then executes it takes the character pointer

STR it creates a local buffer called buffer and then invokes string copy to copy from STR into buffer so note that string copy will continue to execute copying character by character from STR into buffer until a slash zero or a null character is obtained in STR. The first thing to note in this particular program is the vulnerability. The vulnerability comes due to string copy operation so note that argue 1 could be of any arbitrary size.

So it is in control of the user of this particular program so this user could specify any length for argue 1 which could be much larger than 100 bytes and if there is no null termination character within the 100 bytes string copy will continue to execute an overflow buffer. So we will first see this in action. We first make test code and as so and as we have seen before the GCC with this various options about generate the executable dot test code.

(Refer Slide Time: 03:11)



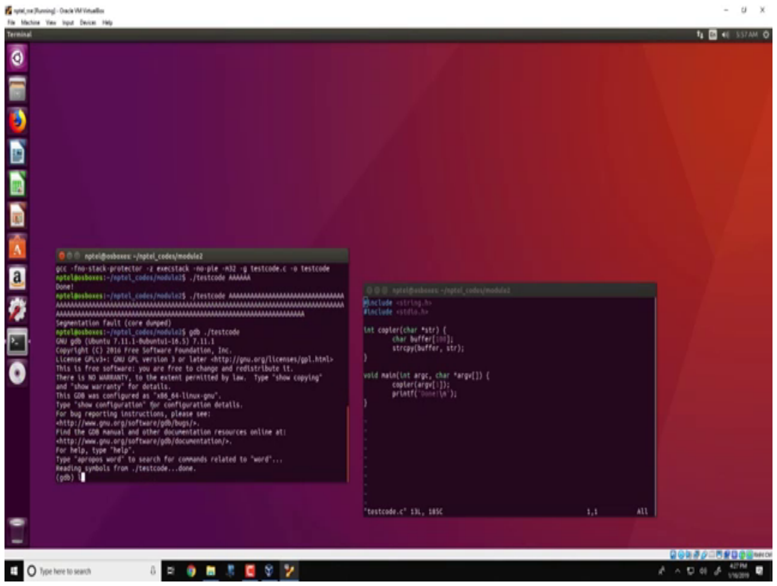
```
nptel@osboxes: ~/nptel_codes/module2
Downloads          Music          peda      setup.c    test1
examples.desktop   [illegible]    Pictures  setup.out  test1.c
nptel@osboxes:~$ cd nptel_codes/
nptel@osboxes:~/nptel_codes$ ls
module0  [illegible]
nptel@osboxes:~/nptel_codes$ cd module2/
nptel@osboxes:~/nptel_codes/module2$ ls
core         out.dump                    shellcodeasm_jmp.c    shellcode.c
e1           overflow1.c                 shellcodeasm_jmp.dis  t
exploit2     overflow2                   shellcodeasm_jmp.o    test
exploit2.c   peda-session-overflow2.txt  shellcodeasm_no0.c    testcode
find_payload README                      shellcodeasm_no0.lst  testcode.c
Makefile     shellcodeasm.c              shellcodeasm_no0.o    testsc.c
nptel@osboxes:~/nptel_codes/module2$ make testcode
make: 'testcode' is up to date.
nptel@osboxes:~/nptel_codes/module2$ make clean; make testcode
rm -f example1 example2 example3 shellcodeasm shellcodeasm_jmp shellcodeasm_no0
testsc overflow1 testcode
gcc -fno-stack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode AAAAAA
Done!
nptel@osboxes:~/nptel_codes/module2$ ./testcode AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
nptel@osboxes: ~/nptel_codes/module2
#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");
}
~
"testcode.c" 13L, 185C                              1,1           All
```

Now we would run test code with a valid input for example 5 is as we see the program execute and we get a done displayed on the screen, so what is happening here is that is which is present in STR is copied into buffer.

The copier function then returns and printf done is executed. Now look what happens when we increase the length of the input so we say test code and increased the length of the input considerably much larger than the 100 bytes and what we see here is that the program terminates with a segmentation fault. First investigate what causes this segmentation fault and as we have seen before we would use GDB to make this investigation.

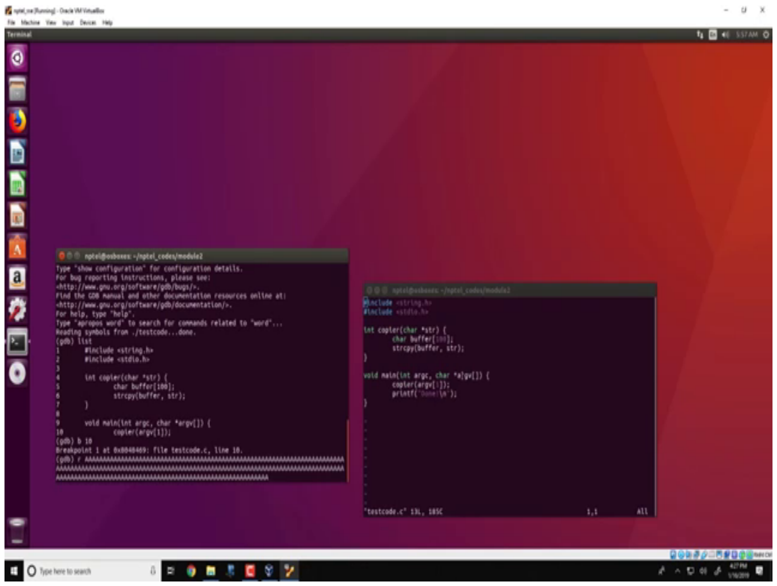
(Refer Slide Time: 04:12)



```
nptel@osboxes: ~/nptel_codes/module2
gcc -fno-stack-protector -z execstack -no-pie -m32 -g testcode.c -o testcode
nptel@osboxes:~/nptel_codes/module2$ ./testcode AAAAAA
Done!
nptel@osboxes:~/nptel_codes/module2$ ./testcode AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
nptel@osboxes:~/nptel_codes/module2$ gdb ./testcode
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 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 "x86_64-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 ./testcode...done.
(gdb) l
```

```
#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");
}

"testcode.c" 13L, 185C                              1,1           All
```

(Refer Slide Time: 04:19)



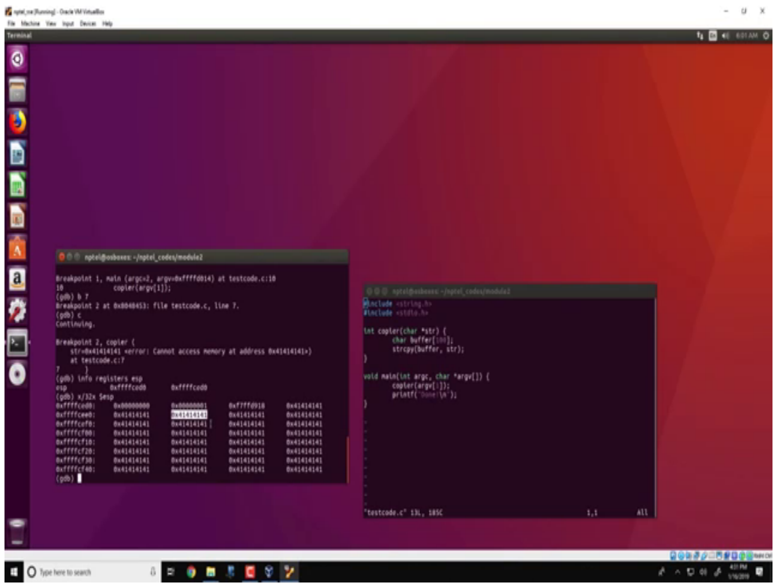
```
nptel@osboxes: ~/nptel_codes/module2
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./testcode...done.
(gdb) list
1       #include <string.h>
2       #include <stdio.h>
3
4       int copier(char *str) {
5               char buffer[100];
6               strcpy(buffer, str);
7       }
8
9       void main(int argc, char *argv[]) {
10              copier(argv[1]);
(gdb) b 10
Breakpoint 1 at 0x8048469: file testcode.c, line 10.
(gdb) r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
#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");
}

"testcode.c" 13L, 185C                              1,1           All
```

So we run GDB as follows we could list the program and put a break point in line number 10 right and then we would run this code through GDB also specify the inputs and we specify a very large input through argue 1.

So argue1 is the series of is which is then passed into the main function through this second parameter P1 ok so what's going to happen here is that since we are running through GDB and have a break point at line number 10.

(Refer Slide Time: 05:11)



```
nptel@osboxes: ~/nptel_codes/module2

Breakpoint 1, main (argc=2, argv=0xffffd014) at testcode.c:10
10              copier(argv[1]);
(gdb) b 7
Breakpoint 2 at 0x8048453: file testcode.c, line 7.
(gdb) c
Continuing.

Breakpoint 2, copier (
    str=0x41414141 <error: Cannot access memory at address 0x41414141>)
    at testcode.c:7
7       }
(gdb) info registers esp
esp            0xffffced0       0xffffced0
(gdb) x/32x $esp
0xffffced0:     0x00000000      0x00000001      0xf7ffd918      0x41414141
0xffffcee0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcef0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf00:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf10:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf20:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf30:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcf40:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

```
#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");
}

"testcode.c" 13L, 185C                                  1,1           All
```

So we would hit the break point over here and then we would put an other break point in line number 7 so line number 7 comprises of the end of the copier function line number 7 gets is reached after string copy completes its execution, so we could break line number 7 and in order to continue the execution we put the command C.

So what happens now is that we see an error over here stating that STR equal to 41414141 the error is it cannot access memory at address 0X41414141. Now we could investigate a little bit in detail and see what actually is happening. So we first see that the entire stack obtained from in four registers ESP this is the stack and x slash x 32x dollar ESP would give you the contents of the stack. So what you notice over here is that the entire stack is filled with 41's now 41 is essentially the ASCI value for the character A, so what has happened is that buffer is over flowed and the entire stack is filled with the contents of A return address which was present on the stack is also over written and what it is replaced with is these values 41414141.

Now at the end of execution of string copy when this particular function copier tries to return it picks the return address from the expected location on the stack that what it would obtain is this value 41414141 and during their RET instruction that copier executes it tries to change the instruction pointer to this location. Now further execution from this location would result in a segmentation fault because this is an invalid location. So in the next video what we will see is that we will take the same test code and will see how we could exploit this test code and enforce this test code to execute a particular payload.

So the next video we will see that instead of just crashing the program we will force one particular payload of our choice to execute from this program, thank you.