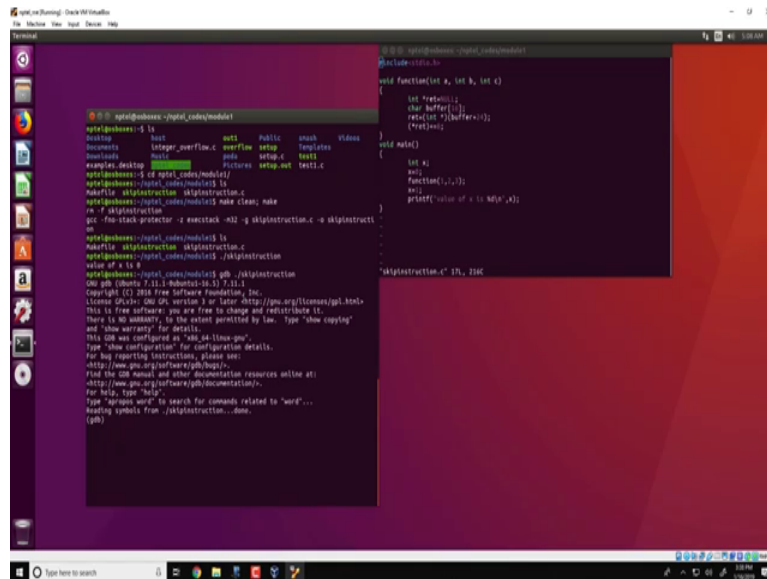


Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Module 1
Lecture 6
Skip Instruction-Demo

So hello and welcome to this demonstration in the course for Secure System Engineering.

(Refer Slide Time: 00:21)



```
mp@ubuntu:~/jupyter_codes/module1$ ls
backlog      test      test1     test2     test3     videos
documents    integer_overflow.c  overflow  setup     test1     test2
examples      multi     picture   setup.out test1.c
mp@ubuntu:~/jupyter_codes/module1$ cd jupyter_codes/module1/
mp@ubuntu:~/jupyter_codes/module1$ ls
Makefile skipinstruction skipinstruction.c
mp@ubuntu:~/jupyter_codes/module1$ make clean; make
rm -f skipinstruction
gcc -fno-stack-protector -c ./skipinstruction.c -o skipinstru
mp@ubuntu:~/jupyter_codes/module1$ ls
Makefile skipinstruction skipinstruction.o
mp@ubuntu:~/jupyter_codes/module1$ ./skipinstruction
value of x is 0
mp@ubuntu:~/jupyter_codes/module1$ gdb ./skipinstruction
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 copyrig
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 ./skipinstruction...done.
(gdb)
```

So in this demo will be showing how a stack can be manipulated to actually skip an instruction. so this demo and the source code is actually present in a virtual chip and actually download so you should have done it with the previous demos and in the week 0 and week 1. So let's actually start of this demo. So the demo is present in this particular directory codes module 1 and it corresponds to this C code skip instruction so let's open this particular C code and open it over here and what we see is two functions, one is a function which takes three parameters a, b and c and it does some simple manipulations on a above her.

Next what we also see is the main function where you have a local variable x which has a value zero then there is an invocationary function with parameters 1, 2 and 3 and then there is a statement for x equal to 1 and then there is a printf value of x is %d, one thing you would actually like to do at this time is to guess what the output of this particular program is. So let us

actually look at it, so one would guess that since x is equal to one over here what would likely be printed on the screen is that the value of x is 1.

So let's see what actually happens, as before we clean and then make and we get the executable skip instruction. Now when we run this particular program what we see is that we print a value of x is zero. In fact, this entire thing of x equal to 1 has not been executed at all. Rather, what has happened to x is that it has somehow managed to obtain a value of zero which corresponds to this particular statement. So this is of course quite intuitive and not what is expected and in order to understand what actually is happening we would have to go into this particular function. So the way we will understand this function is through GDB,

(Refer Slide Time: 03:06)

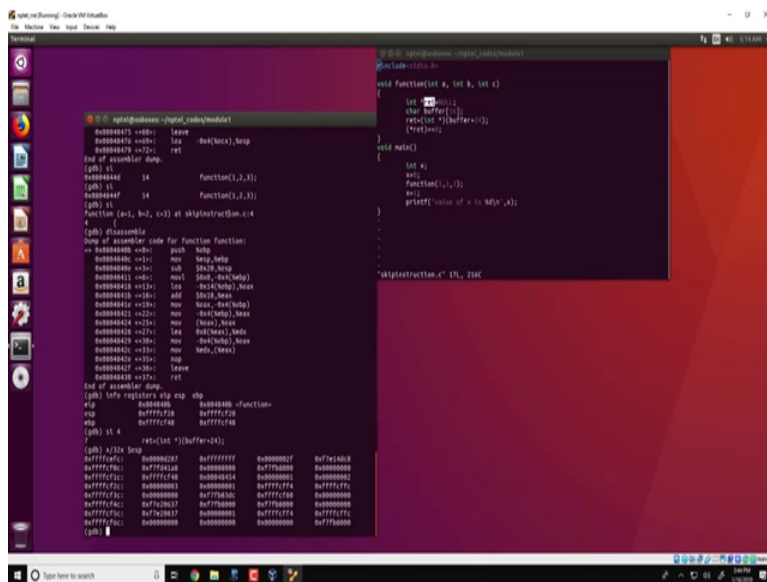
```
Terminal
This file was configured as follows:
Type 'show configuration' for configuration details.
For any reporting instructions, please see:
http://www.gnu.org/help/gdb/html/.
Find the GDB manual and other documentation resources online at:
http://www.gnu.org/manual/gdb.html.
For help, type 'help'.
Recent symbols from 'skipinstruction...done'
(gdb) list
1  #include <stdio.h>
2
3  void function(int a, int b, int c)
4  {
5      int x=1;
6      char buffer[10];
7      printf("value of x is %d\n",x);
8      printf("value of x is %d\n",x);
9  }
10 void main()
11 {
12     int x;
13     x=0;
14     function(1,1,1);
15     printf("value of x is %d\n",x);
16 }
(gdb) b 14
Breakpoint 1 at 0x400444: File skipinstruction.c, line 14.
(gdb) r
Starting program: /home/naga1/naga1_codes/module1/skipinstruction
Breakpoint 1, main () at skipinstruction.c:14
14     function(1,1,1);
(gdb) info locals
x = 0
(gdb) info registers esp ebp ebx
esp 0x7fffd4000000  buffer[0]
ebp 0x7fffd4000000  hardware_externals
ebx 0x7fffd4000000  buffer[0]
(gdb)

skipinstruction.c: 14, 220C
"skipinstruction.c" 171, 220C
```

So let's run for this so we would run GDB dot slash a skip instruction and as we seen before we can list the various contents of the program and we could also set a break point over here say at line number 14, so line number 14 corresponds to the call to this particular function. Now when we run it as we have seen in the previous video a program will execute starting from main and stop due to the break point in line number 14 so we see that the break point has been hit and the execution has stopped just before the function has been envoped.

Now we could actually look at the value of x and rightly enough the value of x will be zero so we could do something like info locals and this has value of x equal to zero this is because x has been initialized in line number 13 to be zero. Now let a single step through this program and see

(Refer Slide Time: 05:25)

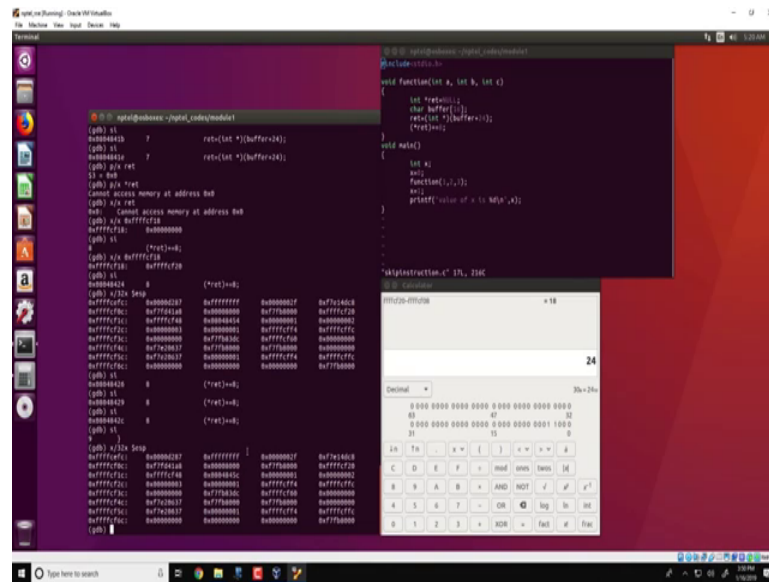


So let us ((5:24) the function and at this point we would also look at the various registers that is the (EIP) ESP and the EBP so what we see here is that we have a stack pointer which is at FFFF CF20 ok and the base pointer is at FFFF CF48 now since this is still at the first line of function the base pointer has not been changed at yet and that new frame corresponding to this function has not been created as yet. So let us single step through a few more instructions let us say the single step through four instructions and look at the contents of the stack.

So the content of the stack as we have seen previously or can be obtained as follows, xx x32x dollar ESP so one can cannot that the written address which should be pushed onto the stack if the call function in name is this so soon after the call gets invoked the next instruction has the address 08048454 now if you actually look at the contents of the stack we see that the return address what we just mentioned is at this location, this has the address FFFF CF1C plus 4 that is FFFF CF20 so in this particular function we have two locals we have pointer to an integer which is known as RET and a buffer which is of 16 characters .

So as we have seen in the previous video we could print the address of this two local variables and what as we would expect this two local variables would be present in the newly formed stacks in for this function.

(Refer Slide Time: 08:13)



So printing the content of register can be done as follows P slash x and for sum thread which is FFFF CF18 so this would be the contents of RET and the contents of buffer is as follows, FFFF CF08 now what you see in that the contents of the buffer starts at FFFF CF08 and extends for 16 bytes.

So anything beyond this 16 bytes would lead to a buffer overflow. Now what we do want over here is that we want to overflow the buffer in such a way so that the return address is modified. Now we would take our calculator we can do this as follows set it to the hexadecimal mode and we would see what is the distance from the buffer to where the return address is stored so we know that the return address is present at the location FFFF CF20 and the buffer is present at this location FFFF CF08 so you subtract this FFFF CF08 so we see that there is an offset of 18 bytes and this 18 is in hexadecimal which corresponds to 24 bytes offset from the start of the buffer to the return address.

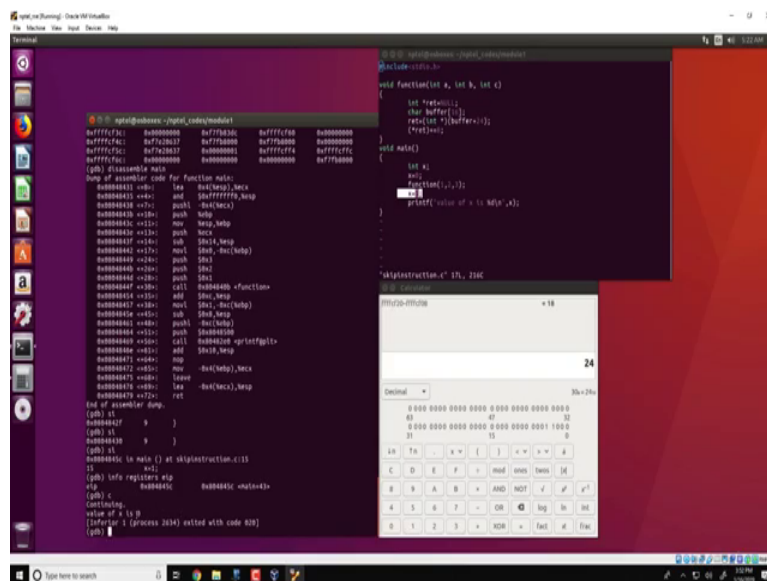
Now what we do in this particular line over here is that at this location buffer plus 24 we obtain a pointer and this particular pointer is stored in this local variable return. So essentially at this particular point what we obtain is that return points to where the return address is stored so we can see this happening by single stepping and looking at the contents of RET so RET is present at FFFF CF18 so we see that RET has a value of zero right now we will execute a single

instruction in which case we have actually executed this particular instruction and changed the value of RET so that it points to the where the return address is present.

So let us now print the contents of the return so x slash x with dumped memory corresponding to this location FFFF CF18 and as we have seen before this particular memory location corresponds to where RET is stored. Now what we see over here is RET has a value FFFF CF20 and this corresponds to this location and this actually is where the return address is stored. Now the next line is very crucial the next line of function increments the value of RET by 8 bytes. Now to understand what this means we would first single step execute a single instruction and see that the contents of the return address has been modified and incremented by 8 bytes.

So not yet let me single instruction ok, so we had to specify four instructions to be executed because this single statement in C corresponds to four instructions in the assembly code so at this point you see that this line of C has completed executing and we would also look at the start and note that the written address has been modified so the return address used to be 08048454 and what it has changed to is 0804845C.

(Refer Slide Time: 13:23)



To understand what is the implication of this change we look at the (())(13:22) of many ok the actual return address is 08048454 and what we have actually changed this to is 08048454C and essentially what it is doing is that it is actually skipping this add instruction and landing somewhere here.

So essentially what would happen when this function completes execution is that this value 0804845C gets taken from the stack and placed into the instruction pointer and execution of the program will continue based on this particular value. So let us single step through this and see that it has come back to main and we would note that the contents of the instruction pointer is 0804845C which essentially means that the add instruction which is specified here has been skipped.

As a result what is happening with respect to the C code is that after a function is invoked the x equal to 1 statement is getting skipped and we are directly going to the printf statement and since this x equal to 1 statement is not being executed the value of x continues to be zero and as a result when we actually continue the execution of this program using the C command which stands for continue to execute then we get that the value of x is zero.

So in this particular video we seen one example of how we could manipulate execution of a program in such a way so that an instruction can be skipped. In the next demonstration what we will see is how we could do something which is more dangerous, we would see how we could actually inject code into a program and force a payload to be executed in the demo that we will look at next. We will take a shell code and we will inject the shell code into a dummy program and then force this shell code to execute, thank you.