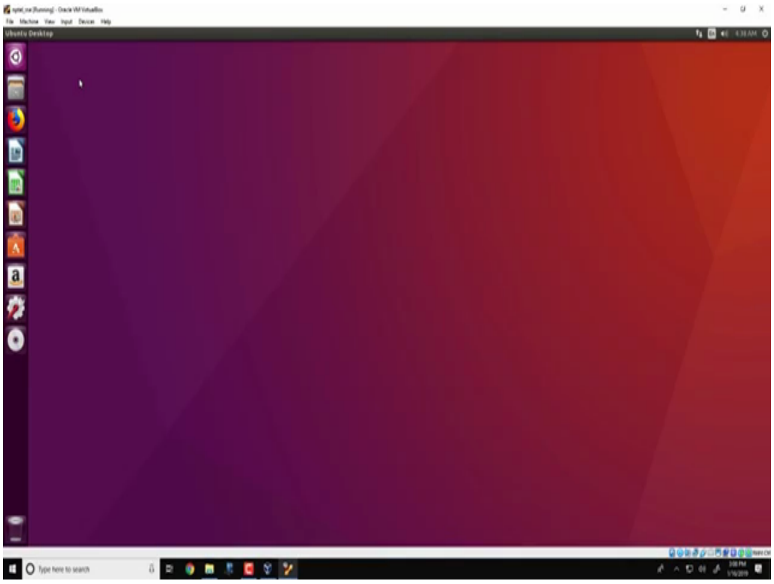
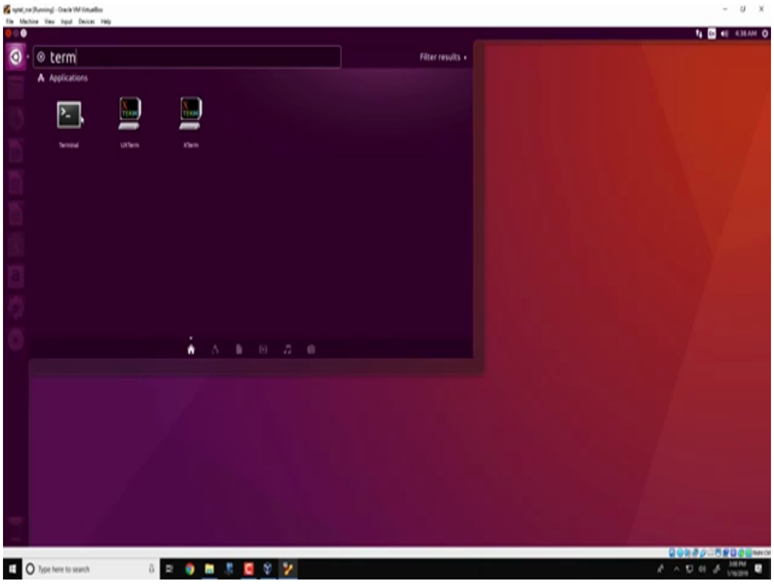
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Gdb-Demo

(Refer Slide Time: 0:22)



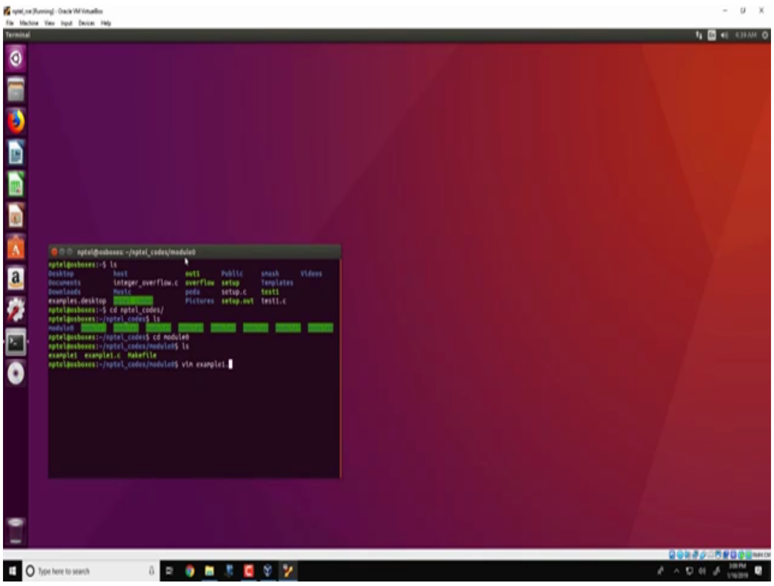
Hello, and welcome to this lecture so this is the demo in the course for in secure systems engineering. So I hope by now that you are actually install the virtual box and you are actually have this Ubuntu running as shown over here. So in this first demo we will actually look at the basics we will see how this stack is presented in a program and it also uses a tool called gdb which can be used to actually debug these programs.

(Refer Slide Time: 0:48)



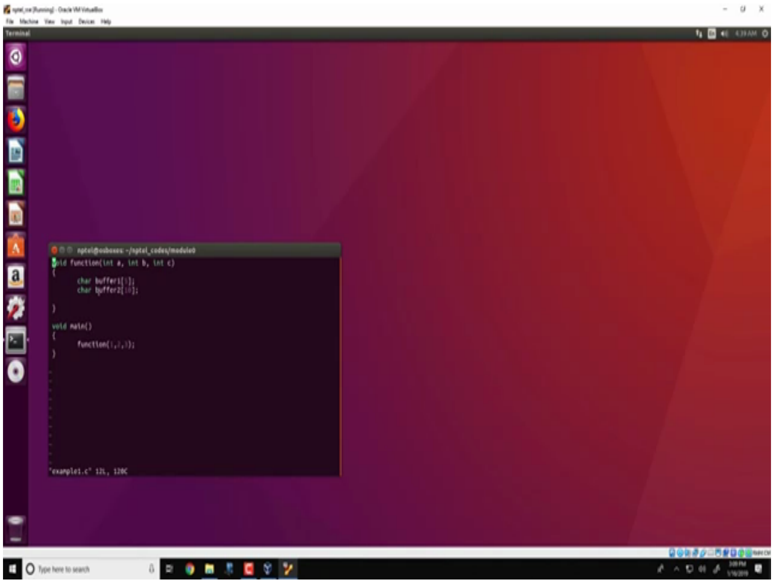
So the first thing to do is to start the terminal, so you could do it like this (())(0:52) terminal.

(Refer Slide Time: 0:57)



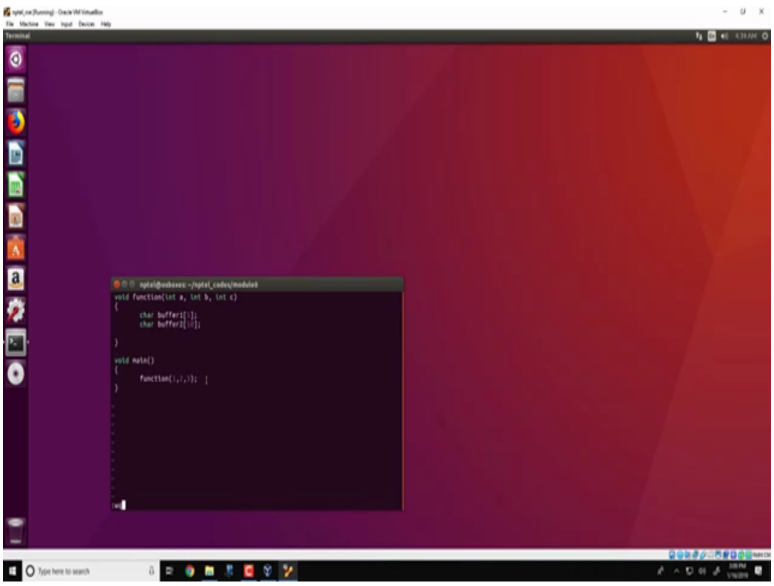
If you had downloaded the codes from the website that we specified you would actually see that there is a directory called NPTEL codes, go there and in this video we will be looking at module 0, okay. So this particular module (())(1:15) to the code that we have seen in the presentation.

(Refer Slide Time: 1:22)



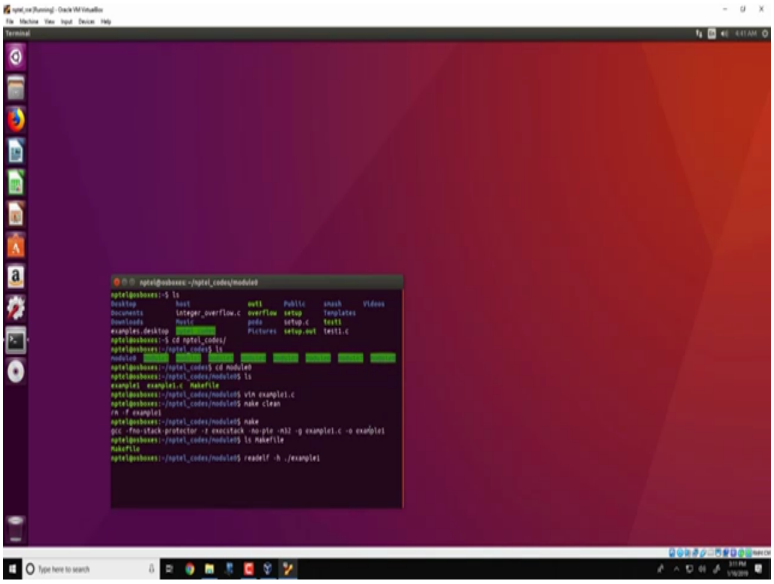
So what it does is that it has a main function and an evocation to this function which is passed parameters 1, 2 and 3 and as we have seen in the previous videos this function just defines two buffers, buffer 1 of 5 bytes and buffer 2 of 10 bytes.

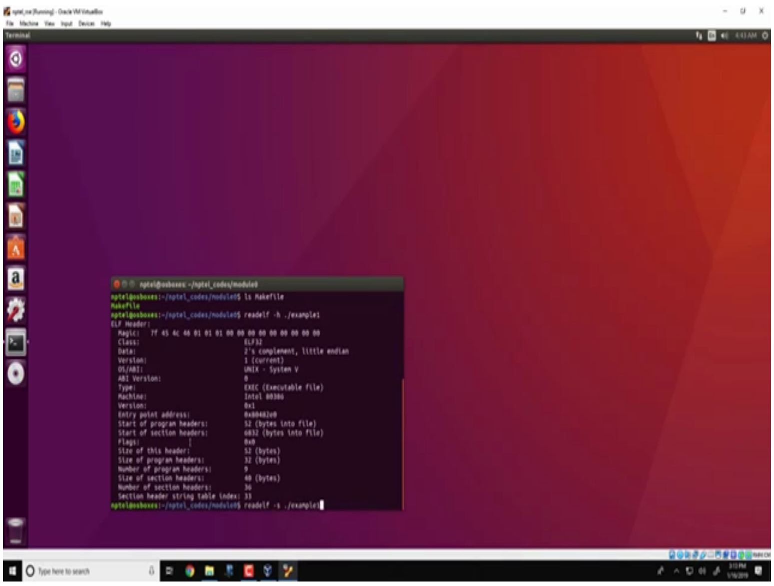
(Refer Slide Time: 1:40)



Now what we will see in this particular video is how we could actually analyse the stack for this particular program.

(Refer Slide Time: 1:50)





Okay, the first thing we do is to actually create the executable for this particular program we do that by running make. So this make depends on what is known as a make file and it is just a script we will not go into the details about what is present in a make file but it is just a script that would commit us to invoke gcc specify various parameters for gcc and finally obtain the corresponding executable.

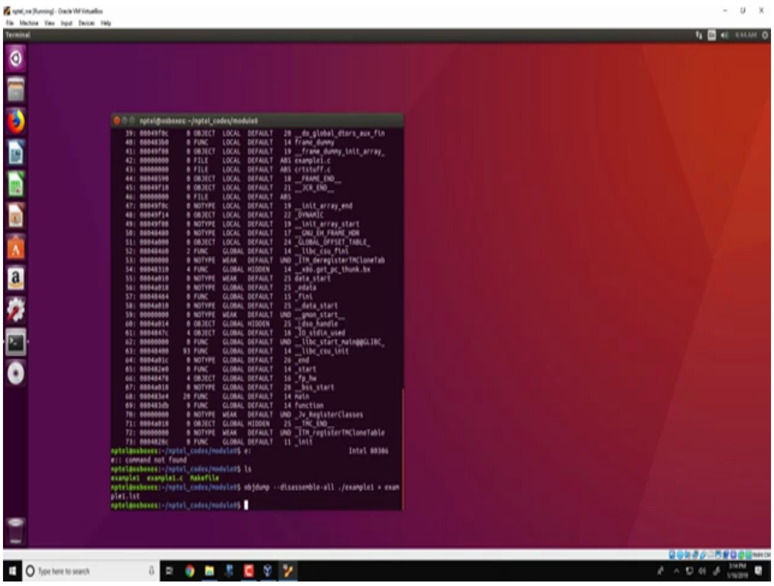
So this particular line is what the make file executes and it has created an elf file example 1, so some of these parameters for gcc like from minus F no stack protector, minus Z X 6 stack and so on. We will be actually seeing in a later video, for this particular video we would explain two specific parameters at minus M32 and minus G, so minus M32 indicates that the executable that is getting created in this case example 1 is a 32 bit executable and since this is an Intel virtual box or Intel machine therefore, the executable that gets generated is a Intel 32 bit executable.

The minus G option that we specified over here ensures that debugging symbols get added into the executable. Now as we seen in the second video today this particular file the example 1 is an elf executable, so we have seen commands such as read elf minus H and we can look at the header of this particular executable and as we have seen in the previous video we would obtain the header for example 1.

So as seen this is the identifier 7f 45 4c 46 which actually transforms to elf in (())(4:01). Other things to actually note is that for the Intel 80386 which indicates that it is a 32 bit executable, so as we have seen the entry point for this executable is at the location 0x8048ce0 and we have also seen the various other aspects such as the section headers and so on. So in particular we can also use other read elf features such as read elf minus S to get more information about the section headers.

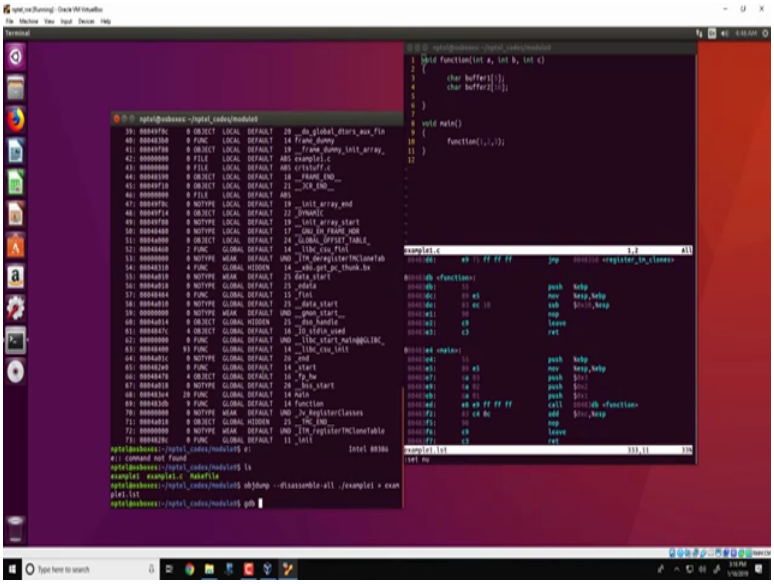
So read elf minus S dot slash example 1 would give you more information about the various sections that are present in this particular executable, okay.

(Refer Slide Time: 4:58)



Another important tool that we will actually look at to go more into detail about the example 1 executable is this objdump, so if I do something like objdump minus minus is assemble all example 1 and it would actually store the output in example 1 dot list, what this particular tool would do is that it would disassemble example 1 and store the disassembly in this file called example 1 dot list.

(Refer Slide Time: 5:45)

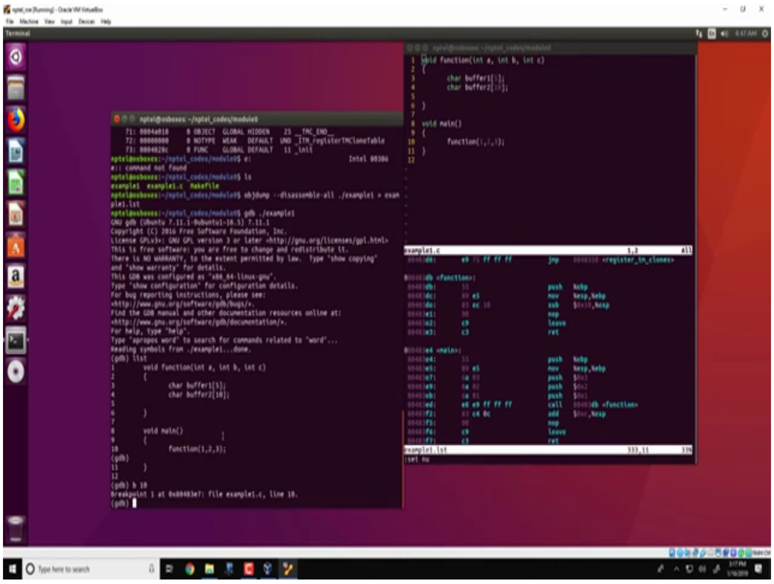


So we could open another tab and look at example 1 dot list which looks like this we can search for main, it is here which is the main function and we could also open the corresponding C code and C that in mean there is a invocation to this function which is essentially done by this particular instruction, in this location 80483ED. So these (())(6:32) here is the machine code for this instruction call to function.

And as we have seen in the previous videos the three parameters 1, 2 and 3 which is passed to the function a pushed on to the stack, so it is pushed from the right that is push 3, push 2 and push 1, over here we see the disassembly for this particular function and as we know there is a preamble in the function where the stack frame is created for this particular function, there are space for the local variables over here and find P the function returns.

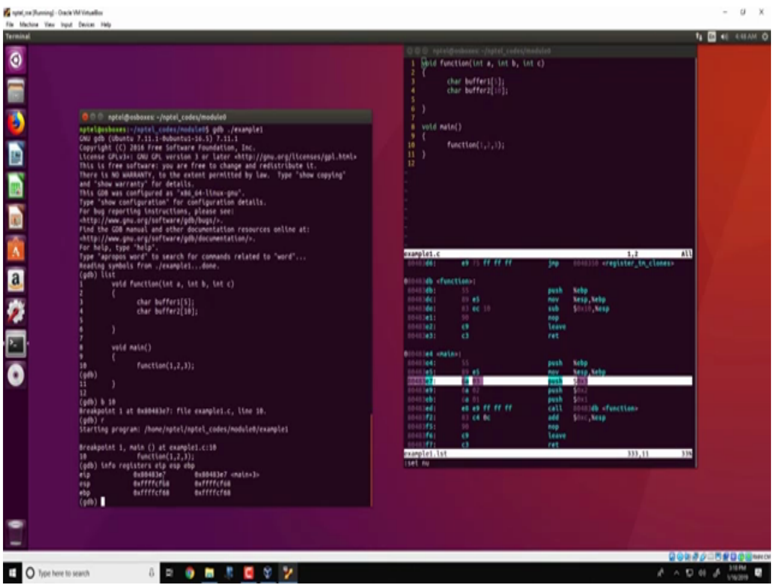
So in this particular instruction we see that there are 0 X1 0 that is 16 bytes for the stack frame, so this is because the total size for the local variables is 16 bytes. The next tool that we will actually look at is the most important so that is known as the gdb gnu debugger and this tool can be used to actually debug this program look at the various registers, look at the stack contents and so on.

(Refer Slide Time: 7:48)



So the way to actually start this tool is to provide gdb with the executable example 1, so gdb would provide you a prompt like this and through this prompt you could actually send various commands such as the list command which would list the entire source code like this. Also you could send break points such as b to line number 10, so what this means is that the program will execute until the break point 10 act line number 10 is obtained. So this could mean that the program will execute until line number 10 and then it will wait for further action.

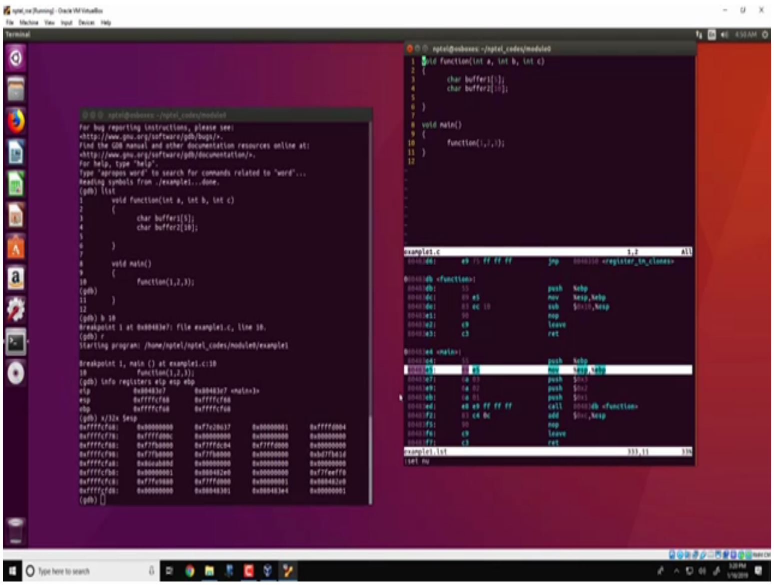
(Refer Slide Time: 8:38)



So let us now run this particular program, we run it by this command called r and what you see is that gdb tells you that break point 1 at line 10 is reached. So we can look into the various registers which are present, now at this particular point we can look at the various contents of the various registers and this can be done with a command like info registers eip which stands for the instruction pointer, esp which stands for the stack pointer register and the ebp which is the frame pointer in the stack.

So we look at this and we see that eip is pointing to a location over here that is 80483e7, now if we go back to this code we see that the eip is actually pointing to this location over here essentially this instruction has to be executed. So we also see that the stack pointer is at the location 0xffffcf68.

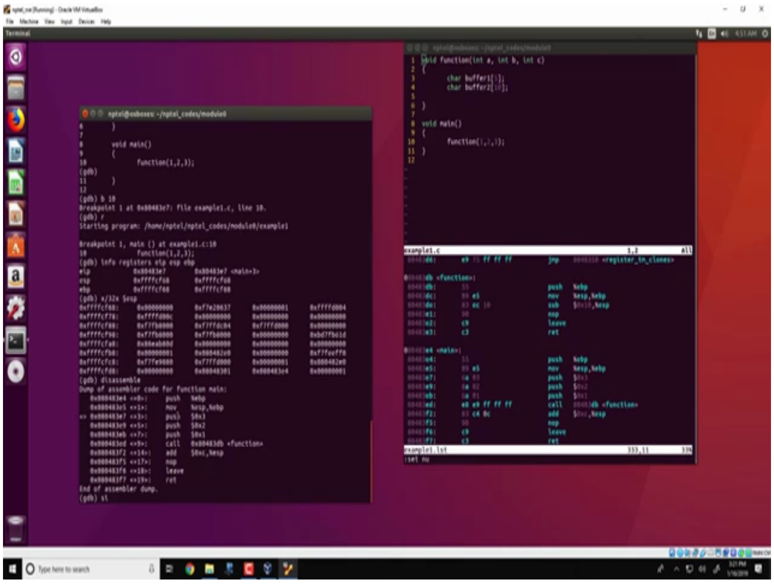
(Refer Slide Time: 9:54)



So what we could do is we could actually print the contents of the stack with a command like this x slash 32x dollar esp and what this command does is that it essentially dumps the memory starting at the location specified by the stack pointer which in this case is ffffcf68 and this 32 over here indicates a number of memory locations that needs to be displayed, the second x over here specifies that the display should be in hexa decimal values.

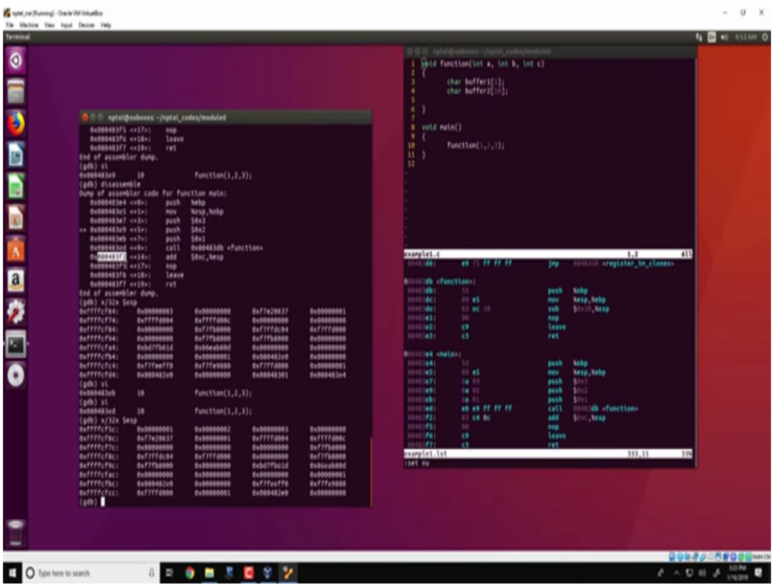
So now we have seen that there is the instruction pointer pointing to this location push 03 which has not yet been executed, we have a stack pointer which is pointing to the location ffffcf68 and the base pointer in this particular case is also ffffcf68 the reason for this is that the stack pointer over here in this instruction move esp to be ebp the stack pointer is in fact copied to be base pointer and therefore the stack pointer and the base pointer are the same.

(Refer Slide Time: 11:12)



Now this similar disassembly what we see using objdump can be also obtained with gdb, so all we need to do is specify this command called disassemble and it would give us the exact same details. So we see that the instruction specified over here in gdb is the same instructions that are specified in this listing which we have obtained from objdump, what is specified in gdb because gdb tells you the run time status of the particular program is this additional point over here which tells you the current location where the program counter is present.

(Refer Slide Time: 12:12)

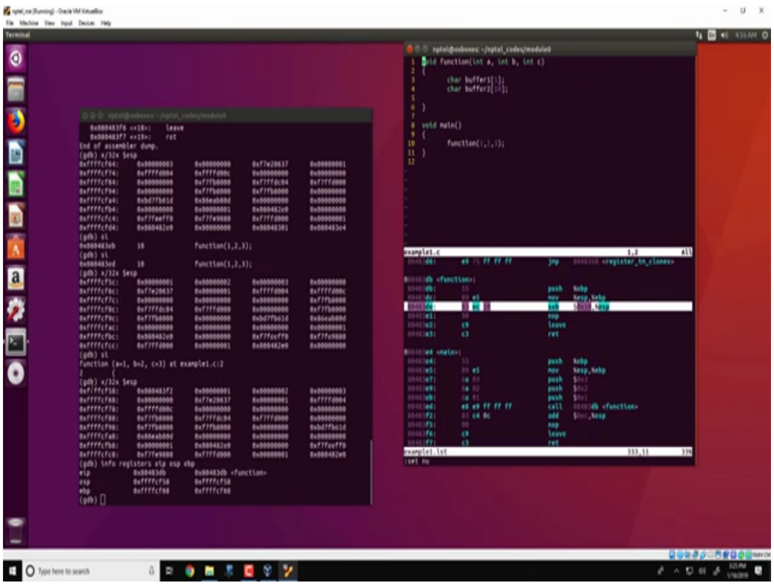


Now we could execute each instruction individually by a command known as single step or single instruction and we shorten this command like si so when we specify si a single instruction in this case the push instruction would get executed and we will actually look at this, so it says that this instruction which ends in 4a3e7 has executed and we could also see if we disassemble again that the program counter or the instruction pointer has moved to the next instruction.

We can also see the effect on the stack by dumping the stack contents as we have done before so that we have done by this x 32x followed 3esp and we see that there is a difference here the contents of 3 have been pushed on to the stack, single step through the next instruction like this and one more instruction like this and then look at the stack again and we see that all the parameters 1, 2 and 3 have been pushed on to the stack. The instruction pointer now is pointing to this call instruction.

So as we have seen in the previous video when there is a call instruction what is done is that this particular address 80483db is moved into the instruction pointer at the same time the instruction of the next address that is 080483f2 gets pushed on to the stack.

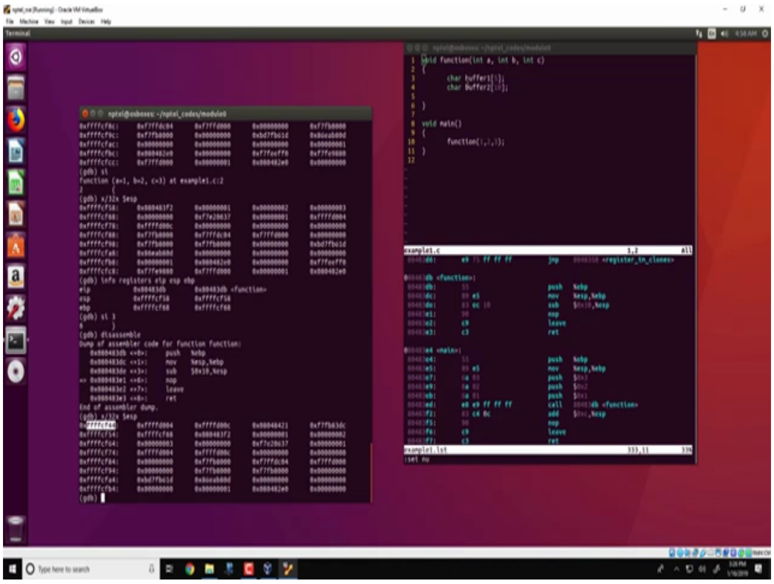
(Refer Slide Time: 13:40)



So let us see how this happens with a single instruction, okay. So we would look at the stack again and we will see that the next instruction 08483f2 which is the return address is pushed on to the stack, at the same time if you look at the contents of the registers we see that the instruction pointer that is the eip is pointing to a location 80483db which is the start of this particular function.

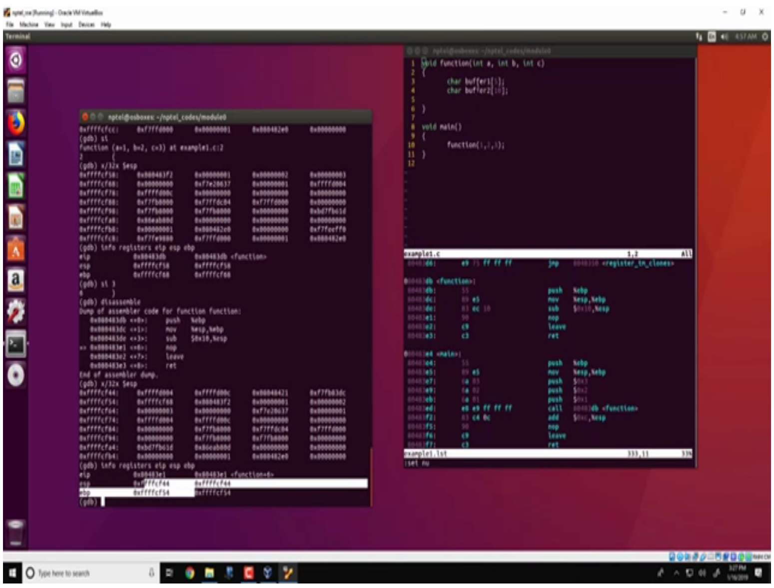
Now as we have seen in the previous video what this function initially does in its preamble is that it pushes into the stack that is the previous frame pointer into the stack, it moves the stack pointer to the base pointer and then this subtract instruction allocate 16 bytes of data in the stack.

(Refer Slide Time: 14:48)



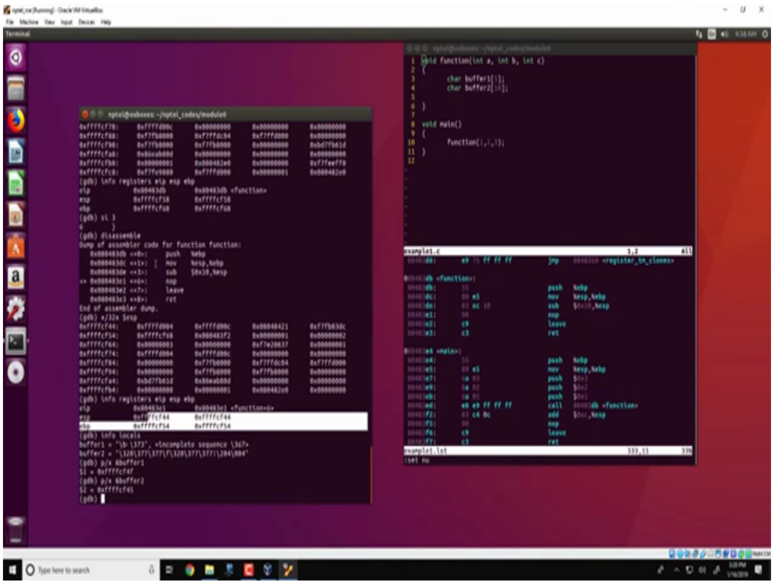
So let us see this happening using 3 instructions, so I can just specify si3 which means that 3 instructions have to be executed, okay and if I disassemble it over here I see that these 3 instructions push ebp move esp to ebp and subtracts 16 from esp has been executed. So I could look at the contents of the stack again are like follows and what we see is that now the stack has actually gone down to ffffcf44 and the reason for this is because of this subtract instruction which is essentially allocating 16 bytes of data who hold buffer 1 and buffer 2.

(Refer Slide Time: 15:48)



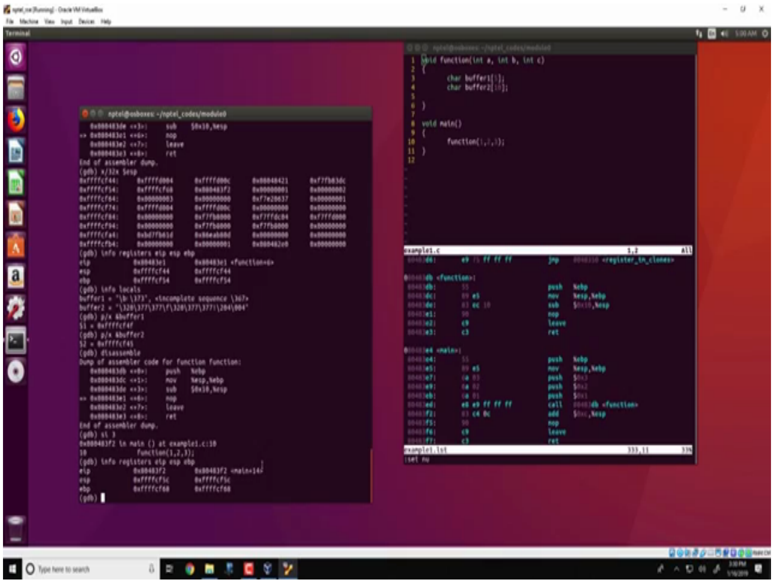
So if I look at the contents of the registers info registers eip which is (())(16:00) ebp, you see that there is a distance between the stack pointer and the corresponding base pointer, so this region between the stack pointer and the base pointer is the active frame for that particular function. Now within this particular region between ffffcf44 and ffffcf54 is where the locals of this particular function resides.

(Refer Slide Time: 16:30)



So we can look at the locals for this function using this command info locals and we see that there are 2 locals specified in this function, so buffer 1 and buffer 2. We can also look at the address of buffer 1 and buffer 2 as follows at p which is a print slash x and ampersand buffer 1 would provide the address for buffer 1. Similarly p slash x and change it to buffer 2 which specify the address for buffer 2 and what we see over here is that buffer 1 and buffer 2 is both within the stack frame that is both are within the stack pointer and the base pointer.

(Refer Slide Time: 17:25)



Now let us get back to the disassembly and what we see is that there are 2 instructions remaining one is the leave instruction and then the return instruction, the leave instruction essentially restores the stack frame such that the old frame corresponding to the previous function in this case the main function is restored and the return would then return back to the main function.

So let us see this happening, so I can do single instruction 3 which should execute 3 instructions and what we see is that the execution has gone back to the main function. Now as before we could also verify this, we could look at the various registers and we see that the instruction pointer now points to somewhere in main in fact it is pointing to the function soon after the call which is this which corresponds to the add function and what we also see is that the stack pointer and the base pointer have moved up in the stack.

So now we have completed executing execution of that particular function and since we have moved back to the main function, so the active frame now is corresponding to that for the main function. So all the local variables if any that are present in the main would come back and would actually be active over here. So gdb is one of the best debugging softwares that are available, it is (())(19:01) and what we actually capture is just the small glimpse of what gdb can do.

So you could also search and we will also try to share a document which comprises of all the various commands the gdb has and you can actually try various things with gdb and also try to do such debugging with various other programs using gdb, look at the various registers and see how the stack and other local variables are maintained, thank you.