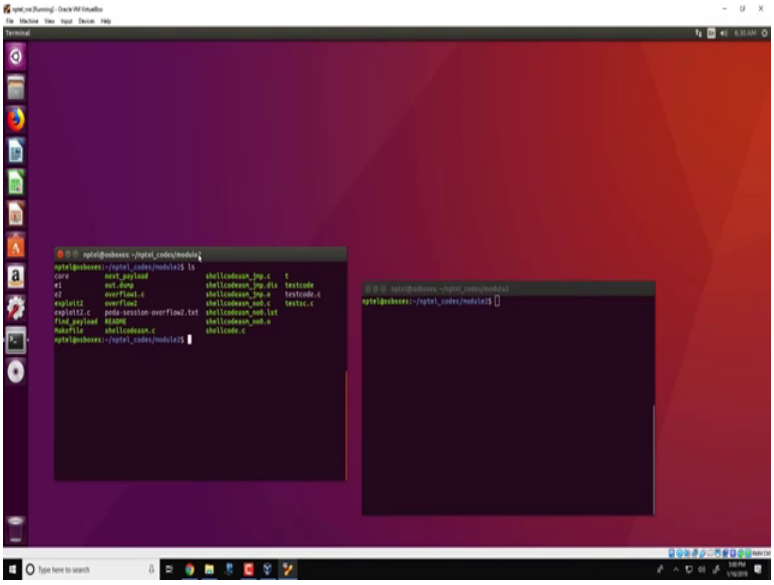
Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Buffer overflow (create a shell)- Demo

Hello and welcome to this demonstration as part of the course Secure Systems Engineering.

(Refer Slide Time: 00:31)

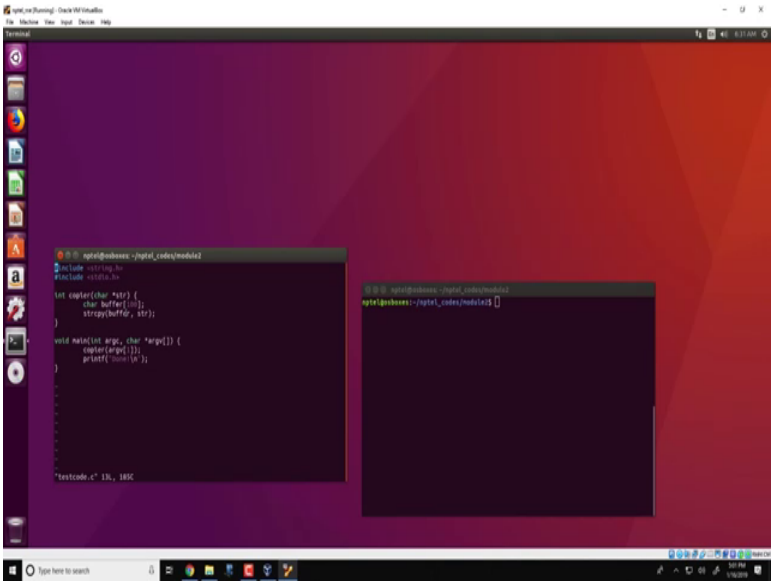


So in the previous demonstration we had taken a particular code a program in C and we had actually looked at a vulnerability that was occurring due to string copy and we seen how overflowing of particular local buffer more than 100 bytes could actually create a segmentation fault and we used GDB during that demonstration to actually see that the return address was overwritten due to the buffer overflow. In this particular demonstration we will work from there and we will see how we can enhance that attack and inject a particular payload into that program and force that payload to execute.

So the payload that we will actually consider is to create a shell now such payloads where a shell is created is very popular in this kind of examples because once an attacker has a shell he has quite a better control on the system, suppose an attacker actually is able to obtain a shell in a system and if he assume that the shell has (())(01:32) then the attacker has super user privileges

in that system and can be anything that every user can do. So as before we will be using the codes that is available with (())(01:46) depository.

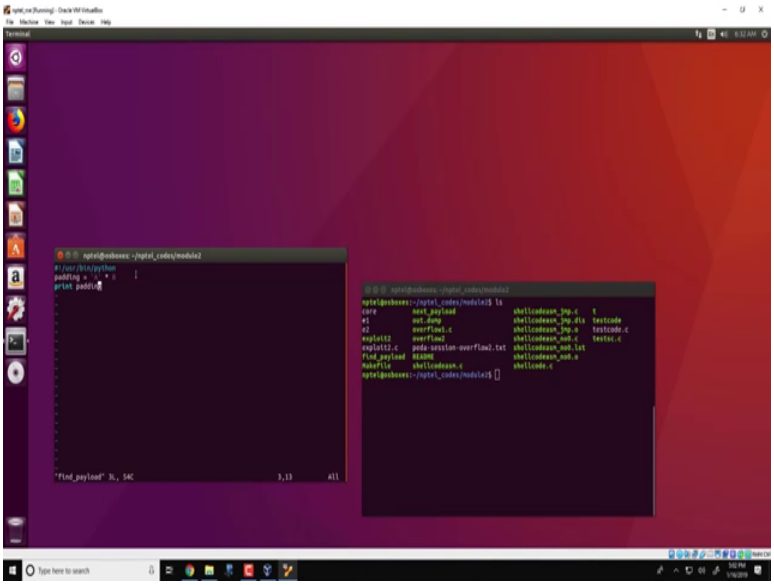
(Refer Slide Time: 02:00)



So the this codes are present in NPTL codes module 2 look at this and the code that we will be looking at is the test code so recollect that the test code had two functions a copier function and a main function, the main function took the argue as command line arguments and it passed argue 1 to the copier function, the copier function had a character pointer STR which have this pointer to this particular string and then invoke string copy taking character by character from STR into this buffer. Now note that the vulnerability occurs over here because buffer is of just 100 bytes while string copy would continue to copy into buffer until slash zero or a null character is obtained in STR.

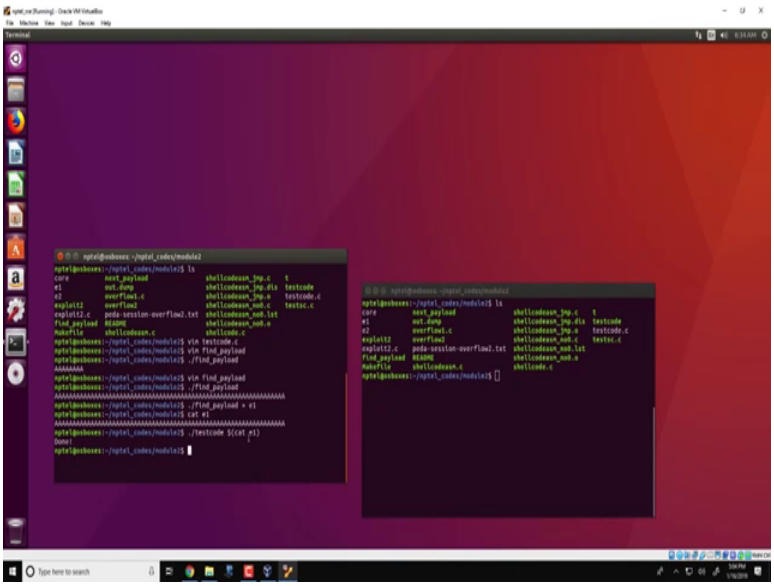
So as long as there is no null termination character STR copy will continue to execute copying the byte by byte into buffer and resulting in a buffer overflow. The first thing to actually do when creating this expert is to identify at what point during execution or what payload would actually cause this test code to stop executing. So in order to do that we use this small script called find payload.

(Refer Slide Time: 03:21)



So find payload looks like this it is a small python script and it essentially creates a string S in this particular case a known eight times. So for example if I add see 8 over here then print (panting) or padding could actually print (A) 8 times as follows.

(Refer Slide Time: 03:46)

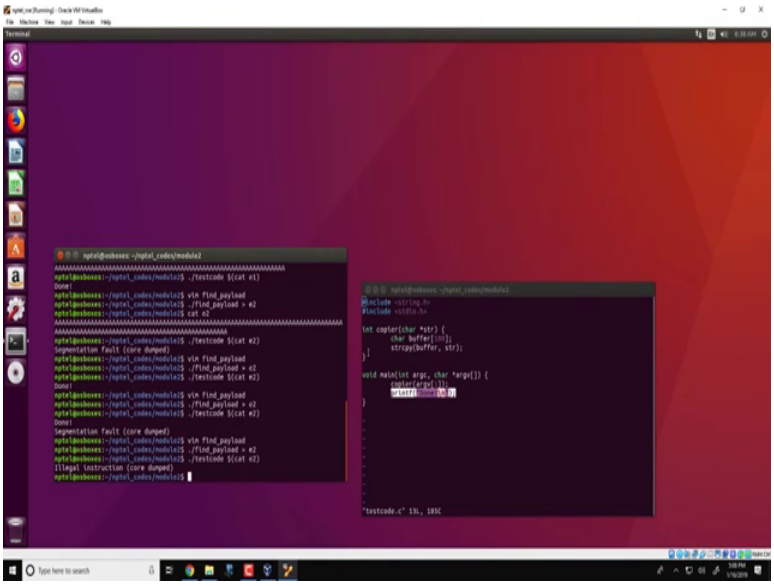


So find payload if print A 8 times similarly by changing this over here to say 64 I will get a string of length 64 now in order to pass this as the command line argument to test code we do the following we run test code as follows the first thing we do is to put the output from finely two let

into E1 into some file E1 so fat E1 comprises of 64 A's the second thing is to sent E1 as an input or to test code this is done as follows.

Fat E1 and we see what happens here is that test code takes the input from E1 which is 64 A's and executes since 64 is less than the size of the buffer defined in test code so there is no problem and there is no overflow that occurs and test code completes successfully. On the other hand if we increase the size of A from say 64 to 128 let us see what would happen.

(Refer Slide Time: 05:19)



So we do this as follows, run this and store the output in this file E2 (sorry) dot slash fine payload and store the output in E2 therefore E2 now contains 128 A's as follows. Now if you run test code and we would have this buffer over here as we have seen in the previous video and test code would segmentation fault and would have a fault. Now by changing the length of this particular string we would be able to identify the exact length of the input which causes this problem. If I change this to say 104 run the fine payload, store the length of E2 of 104 byte is in E2 and run it you see that it works so 104 is not going to solve our problem.

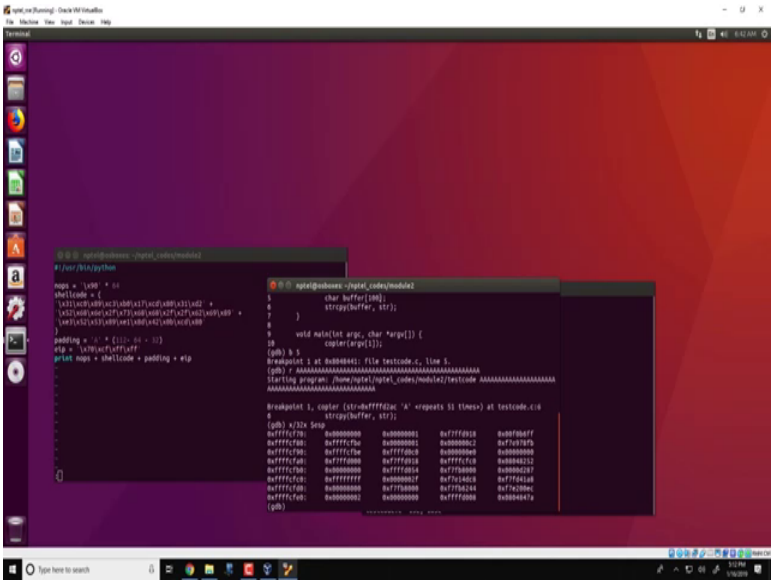
On the other hand if I make it 108 and the exactly the same thing we see that the done gets printed as well as after done there is a segmentation fault this occurs precisely because with a length of exactly 108 the frame pointer which is stored on the stack is overwritten but not the written address therefore after the function gets completes its execution which we will see as in

the code therefore after the copier completes its execution we note that the frame pointer present in the stack has been overwritten but not the written address.

Therefore the return can come back to the main and printf done would get executed. However since the frame pointer has been modified the main will not be able to (execute) exit (())(07:37) and that is why we obtain a segmentation fault after the printf completes execution. Now if on the other hand we change padding from 108 to 112 which means that we have four extra bytes of padding, what we can expect is that it is not just the frame pointer that gets (())(8:01) but also the adjacent memory which essentially is the return address would also get modified.

So in such a case what we can expect is that the copier function will not be able to return back to main. So when we run this program again with payload of 112 instead of a 108 we would see that the done statement is not executed this is because the return to main has been modified and the copier is trying to return to some invalid argument at a location 0x41414141 which is essentially the (loca) A's that you are inserted in the return address location and which has illegal instructions.

(Refer Slide Time: 09:05)



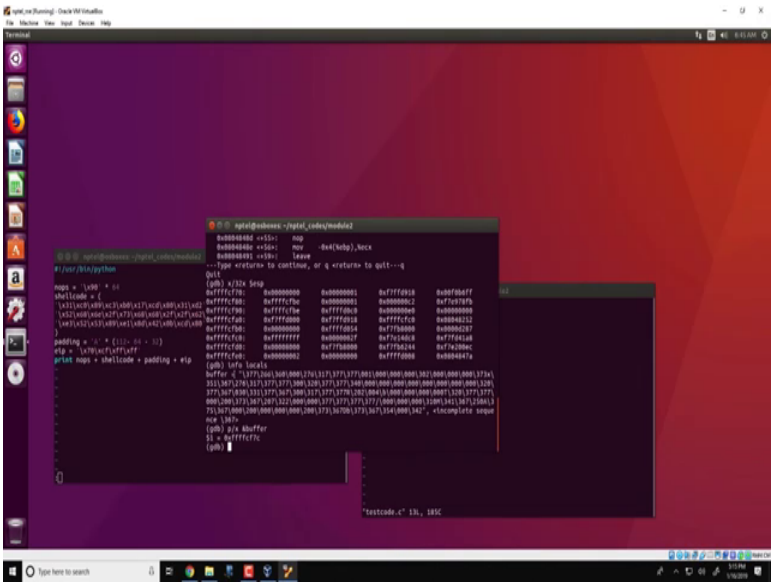
So the next thing we will actually look at is to force up specific payload to execute, so we would look at this particular file call next underscore payload which looks something like this. Soit comprises of four parts one is move ups then you have the shell code then you have some padding and finally you have an EIP. So what we do want to do in with the shell code is to fill

the buffer with all of this there should be 64 move ups then a shell code present over here which is of 32 bytes and then we have some padding which is of size 112 minus 64 minus 32 of bytes and our finally a value of FFFF CF70 so what exactly is this value, is what we will actually identify.

What we expect should happen during the buffer overflow is that after this 100 bytes gets is filled the frame pointer which is stored in the stack will get overwritten, next the written address to main would also get overwritten. So what we want is that this written address should now point to this shell code. So we would require to know where exactly in memory the shell code is present and in order to notice we would need to use GDB. So will open an another terminal and run GDB as follows. Test code will list and we put a breakpoint in line number 5 ok and run it some input say A's ok and at this particular point we look at the contents of the stack.

So that as we have seen before would contain X32X dollar ESP which is as follows.

(Refer Slide Time: 11:39)

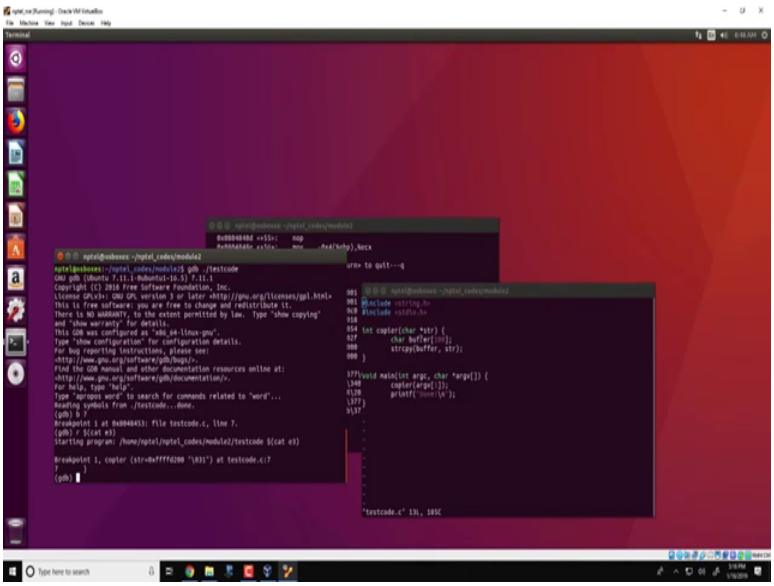


Now the (())(11:38) the return address after copier is 0804847A and we want to know where this written address is present on the stack and we see that this written address is present over here this is at the location FFFF CFEC and further more we want the address of buffer so that is obtained from the location in 4 locals and P slash x (())(12:27) and we note that FFFF CF7C is where the buffer is present so that is that means the buffer is actually present somewhere here.

So what we do is ideally we would like to fill our payloads starting from this location so starting from this location we would want our payload that is the shell code defined over here to be present. However to get a better alignment what we do is we add some NoOps initially so the NoOps is present by this opcode is given by this opcode 90 and whenever the processor sees this opcode of 90 it is just going to skip the instruction to nothing in that instruction we fill in NoOps starting at in the begging of the buffer and then at some offset in this case 64 bytes we fill in the shell code.

Now we need to jump back somewhere in the NoOps at a decent at a reasonable alignment so that the shell code executes. So lot of this is obtained by trail and error and what we found that is if we specify the location FFFF CF70 it would indicate it would actually work. So we will see what happens when we give such an input. So first of all we create the shell code that we (we create a shell code that) we want to execute as follows.

(Refer Slide Time: 14:14)

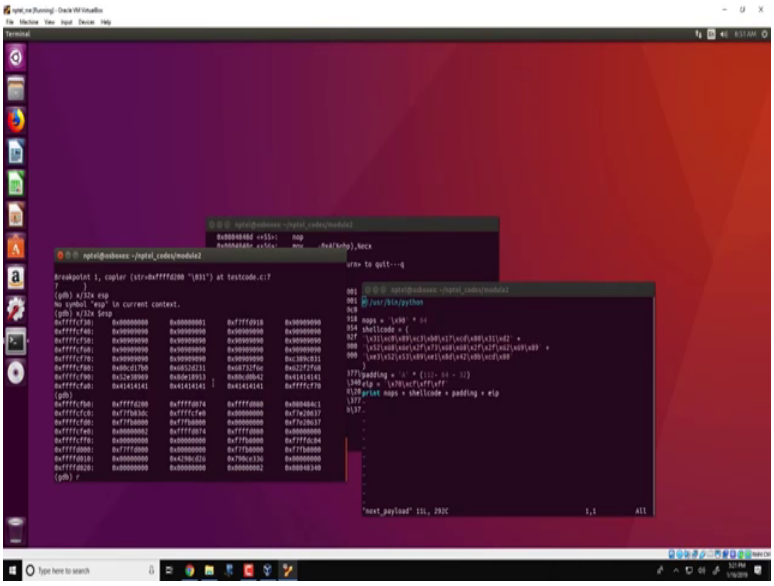


So we say next payload and store it in this file called E3 and then we run GDB with test code break at line number 7 which comprises which is end of string copy and then run giving the newly formed payload as or the input.

So this is done by at E3 so which would take the input from the file E3 and run with that particular input. So even this let us see what has happened is that the copier has executed further string copy has executed and since the string which we have given is much larger than 100 bytes

therefore buffer has overwritten and the string has been strategically created in such a way that the written address present on the stack has been replaced with a specific return address which forces the payload present in the buffer to execute. So let us see this more in detail.

(Refer Slide Time: 15:34)



So we will print out the stack as before X32X ESP dollar ESP we would look at the payload that we have created and we see at this particular point the buffer actually starts is supposed to be present we see that there are 64 NoOps comprising of the byte 90. So if you actually count it would be like staring from 90 over here to this particular thing there would be 64 bytes of NoOps then we would see that the payload is actually present starting from this location. Note that the alignment is based on little Indian.

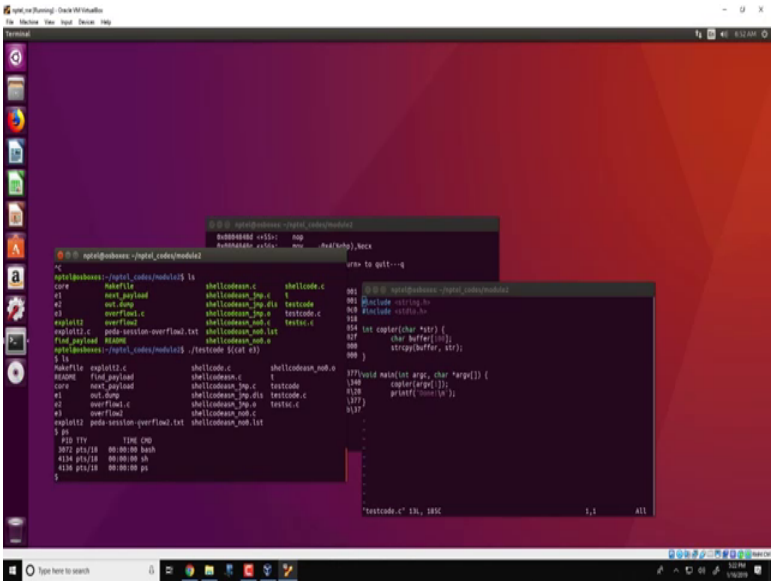
So therefore when we specify C3C089C3 what we actually see here is little Indian notation for the same. So the next 32 bytes which will comprise of this shell code so what the shell code actually does is that it encodes the machine operations which actually creates a shell we will not go into details about how the shell code is created right now. So you could look at the previous video or there are various tools online which would create this shell code (())(17:03). So if you actually look at this we see that this are the first four bytes of the shell code in the little Indian notation next four bytes and so on.

Now the shell code ends over here with 420BCD80 which is here 420BCD80 and then we have a lot of padding which is present infact we have 112 minus 64 minus 32 bytes of padding that we

see is the adding with (())(17:35) which is actually present from here to here so there are 1, 2, 3 and 4 so that ends up in 64 bytes of padding which is 112 minus 64 minus 32 and finally we see that this location which was suppose to have the return address so this location is overwritten with the value FFFF CF70, so what this means is that when this copier function completes its execution it is going to pick the memory contents from this location and put this into the EIP register.

So corresponding to FFFF CF70 is this memory over here, so what is going to happen is that the instruction pointer would point to this location and it would pick up each byte from here this is 90 90 90 90 and start executing this NoOps so all of this NoOps gets (execute) executed and then you actual shell code which is staring at this location would then get executed. Eventually by the end of this 32 bytes of operation that gets executed your shell would have been created the shell gets created ideally by system called to the operating system using an interrupt 80H which tells the OS that a new process ahs to be created. So this is what is actually happening over here so let us actually run this code and see that the shell is executed,

(Refer Slide Time: 19:24)

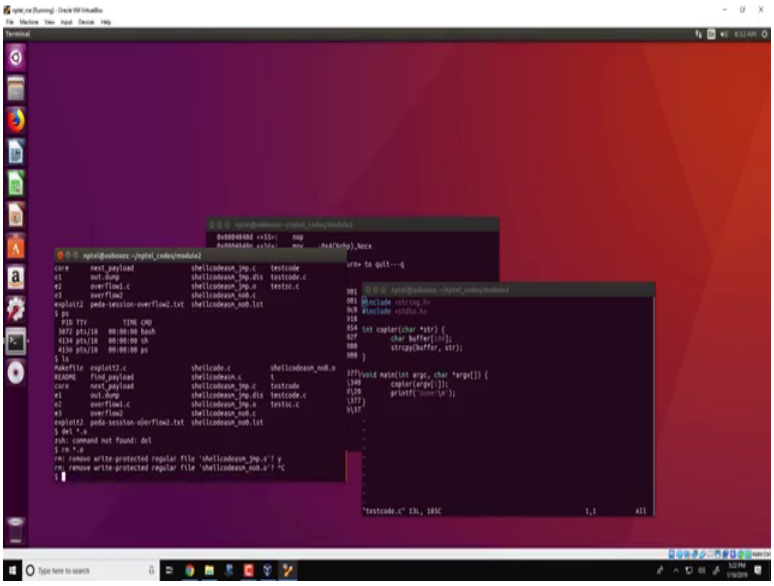


so when you run it so we can un it outside the GDB and we can run it as follows so we can say dot slash test code at E3 dot slash test code dollar at E3 and what we see is that a shell gets created. So this shell is the SH shell so we can do all the shell commands you can also look at PS that is the processes that are executing in this shell and we see that infact there are three

processes, PS is the process that is the command that we have given and the parent for this process is the SH process and the original batch shell which was created with this terminal is (()) (20:54).

So in this way we have seen that we have injected this particular code called test code dot C with a payload we force the buffer to overflow and the payload which was present in the buffer to execute.

(Refer Slide Time: 21:06)



So many of the malware actually use this particular technique they obtain a payload and once an attacker is able to obtain such a payload he can do anything in the system like example delete all the object files like this and so on, thank you.