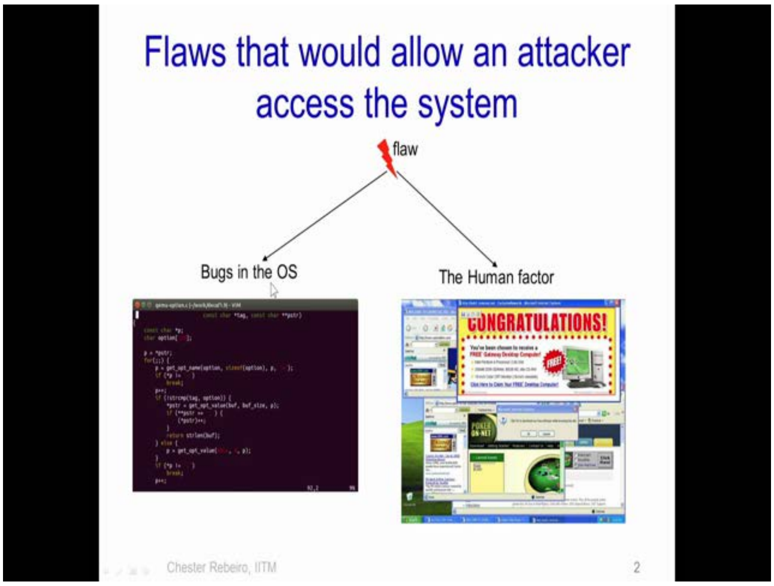
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 08
Lecture – 37
Operating System Security (Buffer Overflows)

Hello. In this video, we will talk about Buffer Overflows. Essentially, Buffer Overflows is the vulnerability in the system and it is not just restricted to the operating system, but it could be pertaining to any application that runs in the system. Now buffer overflows is vulnerability and that allows malicious applications to enter into the systems, even though they do not have a valid access essentially it would allow unauthorized access into the system.

Let us look at Buffer Overflows in this particular lecture.

(Refer Slide Time: 01:02)

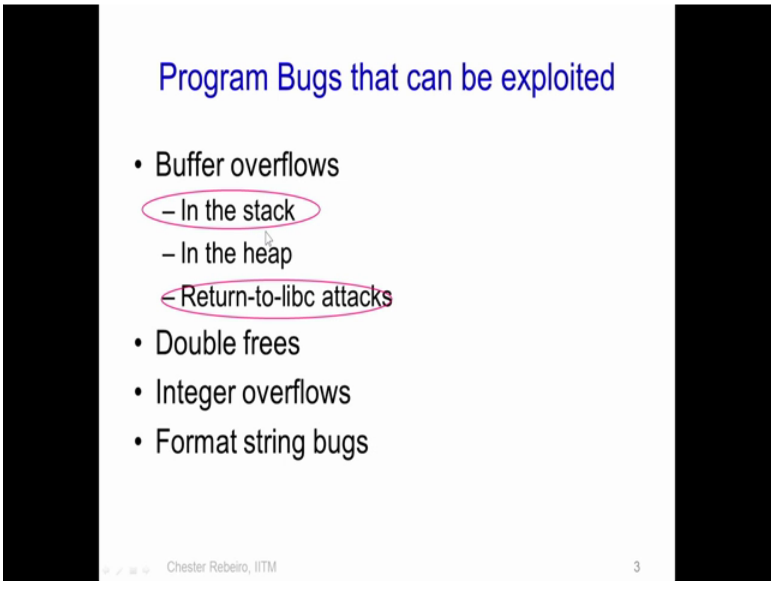


So, when we look at how an unauthorized user or an unauthorized attacker could gain access into the system. So, we see that it is just by flaws present. There are two types of flaws that a system can have. One it could have bugs present in the application or the operating system in this particular case. Or it could have flaws due to the human factor when we for instance browse the internet where we see many such web pages opening and prompting us to click on particular things which would take us to may be a malicious website and a result of that would cause malicious applications to be downloaded into the system.

Another one, which is more pertaining to the operating system, is when there are bugs in the operating system code. Now, modern day operating systems especially the ones that we typically use on a desktop and servers are extremely large pieces of code. For instance the current Linux kernel has over ten million lines of code and all these codes are obviously, written by programmers and will have numerous bugs. So, these bugs are not very easy to detect; however, if an attacker decides to look he could find such a bug and he could then exploit this bug in the operating system to gain access into the OS.

And as you know that once the attacker gains access into the OS, he will be able to do various things like he will be able to execute various components of the operating system code, he could control all the resources present in the system, he could also control which users execute in the system and so on. Thus, an unauthorized access through a bug in the operating system is a very critical aspect.

(Refer Slide Time: 03:20)



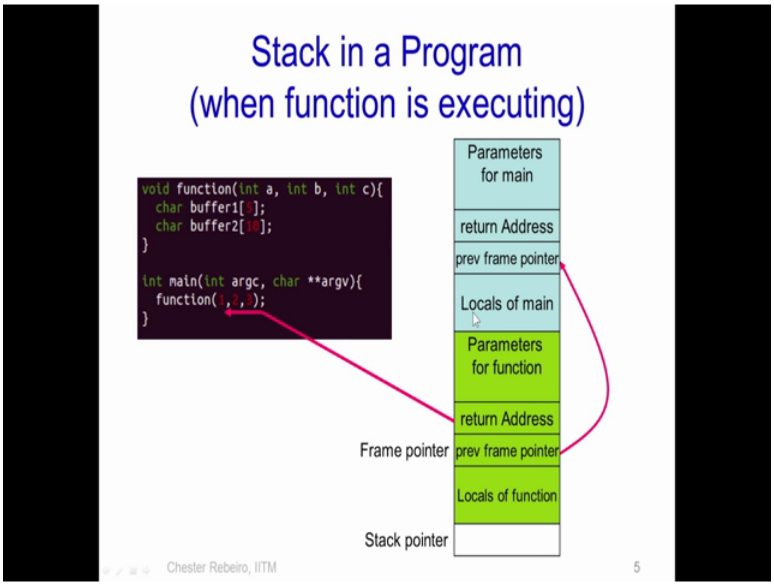
There are a number of bugs that an attacker can exploit in order to gain unauthorized access into the operating system. So, here is the list of some of them. So, they could be buffer overflows in the stack of the program or in the OS in the heap, they may be something known as Return-to-libc attacks. There are double frees, essentially this occurs when a single memory location which is dynamically allocated through something like a malloc gets freed more than once, there are integer overflow bugs, and there are format string bugs.

There are essentially numerous different ways that bugs can be exploited by an attacker to enter into the system. So, what we will be seeing today are the bugs in the stack and something known as a Return-to-libc attack which essentially is a variant of the buffer overflow attack in the stack.

(Refer Slide Time: 04:22)

So, in order to understand how the buffer overflows work in the stack, we first need to know how a stack is managed. Let us see how the user stack of a process is managed.

(Refer Slide Time: 04:38)



Let us say we take this very simple example which has two functions the main; and in this main function, we invoke another function with parameters 1, 2 and 3. And this function just allocates two buffers - buffer 1 of 5 bytes and buffer 2 of 10 bytes. So, as we know when we execute this program in the system, the operating system creates a process comprising of various things like the instruction area containing the text or the various instructions of this particular program the data section the heap as well as the stack.

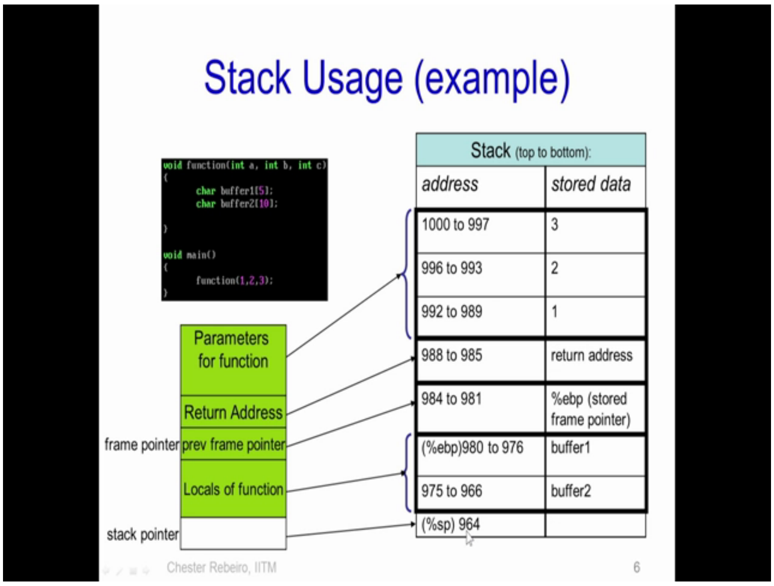
The stack in particular is used for passing parameters from one function to another and it is also used to store local variables. Let us see how the stack is used in this particular example. Let us say that this is the stack and this stack corresponds to when the main function is executing. Now when main wants to invoke this function over here that is function 1, 2 and 3 it begins to push something on to the stack. So, what is pushed onto the stack you will see now?

So, first the 3 parameters 1, 2 and 3 which are passed from main two function are pushed on to the stack that is parameters for function 1, 2 and 3 would be pushed onto the stack, then the return address is pushed on to the stack. So, this return address will point to the instruction that follows this function invocation. As we know in order to invoke function in an x 86 space processor, the instruction that is used is the call. The return address will point to the next instruction following the call.

So, after the return address something known as the previous frame pointer is push onto the stack, this frame pointer points to the frame corresponding to the main function. So, this is the frame which is used to when the function is executed, while this frame is used when main is executed. Now after the previous frame pointer is used the local variables, which are defined in function, are then allocated. In this case, we have two character arrays which are allocated 1 is of size 5 and the other is of size 10 bytes.

Besides all of this, we have 2 CPU registers which are used to manage this stack pointer 1 is the frame pointer which is typically the register bp in Intel x86, and the other one is the stack pointer or s p in the x86 nomenclature. Now the frame pointer points to the current functions framed. So, it actually points to this particular thing corresponding to the frame for function. Now after this function completes its execution and returns, this previous frame pointer is loaded into the register bp, therefore the frame pointer will then point over here that is the frame corresponding to the function main. Now the stack pointer on the other hand points to the bottom of the stack.

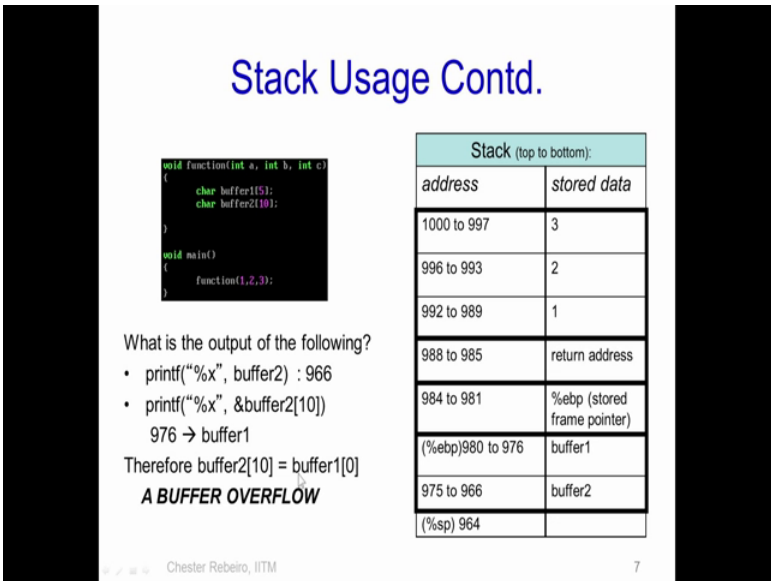
(Refer Slide Time: 08:37)



Now, let us look at this in more detailed. Let us say that this is the stack and this is the address for the various stack locations and this is the data stored in that particular address. Let us assume that the top of the stack is 1000, and it decrements downwards. So, this was the stack corresponding to the function when it is invoked. So, we first see that there are the parameters that are passed to the function are pushed onto the stack this is the parameters 3, 2 and 1 which are pushed onto the stack. So, we note that each of these parameters since they have defined as integer in this function is given 4 bytes. The integer a, which is passed to function would start at the address location 997; and from there be 4 bytes 997, 998, 999 and 1000. Similarly, the second and third parameters also take 4 bytes.

The return address for this function at essentially the point at which the function has to return is also given 4 bytes. While the base pointer since it is a 32 system is also given 4 bytes, then we have the buffer 1 which is allocated as a local of the function, which is given 5 bytes 976 to 980; and then buffer 2 is allocated 10 bytes. So, these two arrays are the locals of the function the base pointer points to this particular location and the stack pointer points over here to the address number 964.

(Refer Slide Time: 10:42)

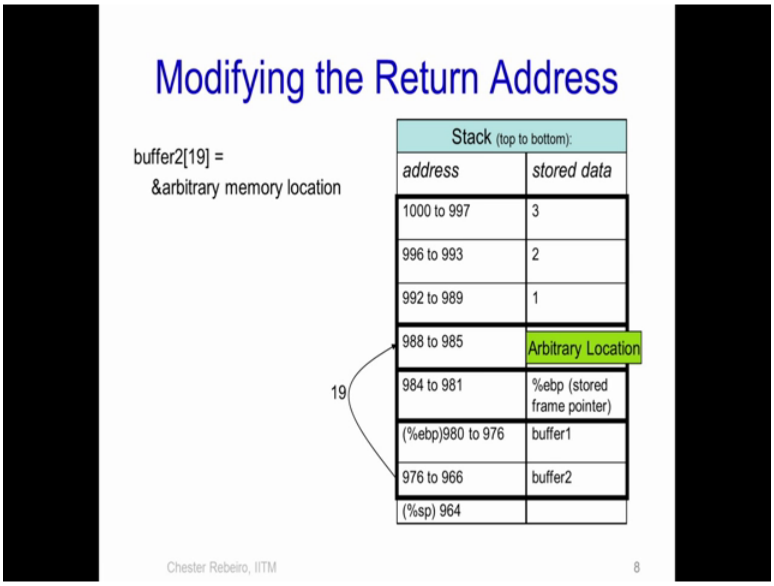


Now, let us look at some very simple aspects. Let us say what would happen if we print this particular line, so printf percentage x buffer 2. So, as we know buffer 2 corresponds to the address of this particular array, this particulars printf statement would print the address of buffer 2. So, if we look up the stack we see that the start address of buffer 2 is 966; therefore, this printf will print 966.

Now what happens if we do something like this printf ampersand buffer 2 of 10? So, we know that buffer 2 is of 10 bytes and will have indexes from 0 to 9; now buffer 2 of 10 is 966 plus 10 which is 976. So, what is going to be printed over here is - 976. Now it so happens that 976, is outside the region of buffer 2, in fact, 976 is in buffer 1. Therefore, what we are getting now is that we are printing an address which is outside buffer 2, and this is what is known as a buffer overflow.

Essentially, we have defined a buffer of 10 bytes, but we are accessing data which is outside the buffer 2 area. So, we are accessing the 10th, 11th, 12th and so on byte. So, this is known as a buffer overflow. Now, what we will see next is how this buffer overflow can be exploited by an attacker, and how an attacker could then force a system to execute his own code.

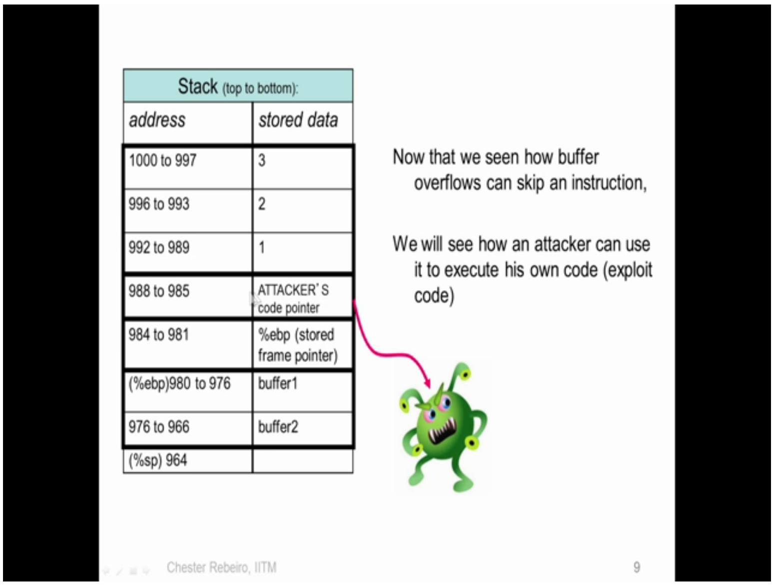
(Refer Slide Time: 12:39)



Now one important thing from the attackers' perspective is the return address. If the attacker could somehow fill this buffer 2 in such a way that he would cause a buffer overflow, and modify this particular return address then let us see what would happen. Let us say buffer he makes this particular statement. So, buffer 2 of 19 is some arbitrary memory location. So, what the attacker is doing he is that he is forcing this buffer 2 to overflow and he is of overflowing it in such a way that the return address which was stored onto the stack is replaced with his own filled location.

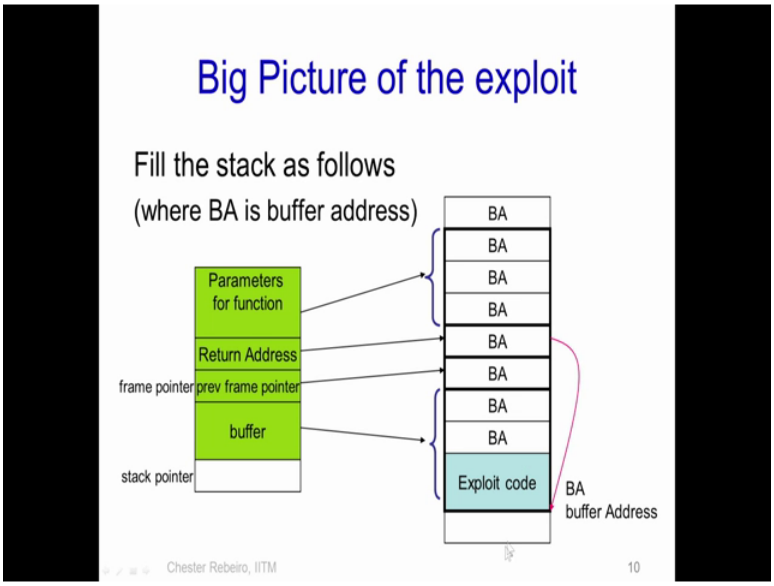
After the function completes executing, it would look into this location and instead of getting the valid return address it would get this arbitrary location, and then it would go to this arbitrary location and start to execute code. So, what we would see is that instead of returning to the main function as would be expected in the normal program, since the attacker has changed this return address to some arbitrary location the processor would then cause this instructions corresponding to this arbitrary location to be fetched and execute it. So, now, it looks quite obvious what the attacker could do in order to create an attack.

(Refer Slide Time: 14:19)



Essentially what the attacker is going to do is that he is going to change this return address the valid return address present in the stack with a pointer to an attack code. Therefore, when the function returns instead of taking the standard return address, it would pick the attackers code pointer; and as a result, it would then cause the attackers code to begin to execute. So, now, we will see how the attacker could change the return address with it is own code pointer.

(Refer Slide Time: 14:56)

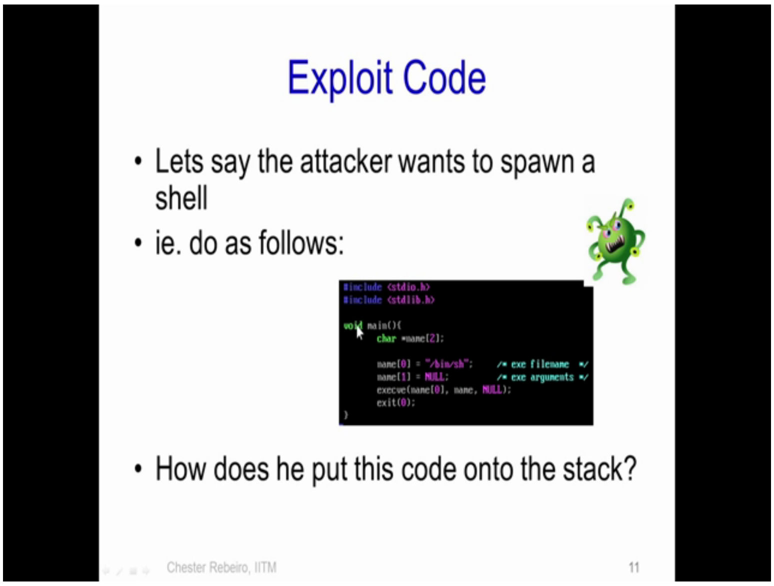


In order to do this, what we assume is that the attacker has access to this particular buffer. So, this means that the attacker will be able to fill this particular buffer as required. For instance, this buffer could be say passed through a system call, for instance, if the operating system would require that a character string be passed through the system called by the user, and thus the attacker will be able to fill that character string and pass it to the kernel. So, what the attacker is going to do is to create a very specific string that it has the exploit code, and it is also able to force the operating system or for that matter any application to execute this exploit code. So, how it is going to do it is as follows.

So, in the buffer, the attacker will do two things; first he will put the exploit code that is the code which the attacker wants to execute in the lower most region of the buffer, and then begin to overflow the buffer, essentially what he is going to put is this address location BA. Now B here is the address of this exploit code. So, it is assumed or it if a smart attacker will be able to determine what the address location is of buffer, and he will overflow the buffer with BA. So, he keeps overflowing the buffer with BA, and when this happens at one particular case the returned address present in the stack is changed from the valid return address to B A.

Thus, when the function returns what the CPU is going to see is the address BA, in the return address location thus it is going to take this address BA and start executing code from that. So, since BA corresponds to the address where the exploit code is present, the CPU would then begin to execute this exploit code. And in this way, the attacker could force the CPU or the processor to execute the exploit code.

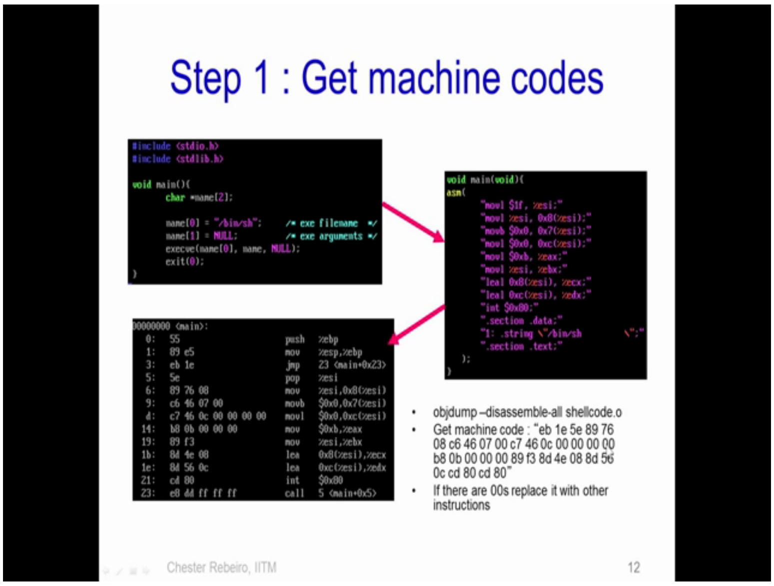
(Refer Slide Time: 17:33)



So, now we will see how a one particular attack code is created, and how an attacker can force an application or an operating system to execute that exploit code. Let us take a very simple example of the exploit code, which is shown over here. Essentially this exploit code which we call as shell code does nothing but only executes a particular shell, the shell is specified by slash bin slash sh.

And this particular function, xxcv is invoked in order to execute this shell the parameters are name 0, which comprises of the executable name, and there is name 1 which is essentially a null terminated string. So, we will see how this particular code can be forced to be executed by an unauthorized attacker. The first question that needs to be asked is how does this attacker manage to put this code onto the stack.

(Refer Slide Time: 18:48)



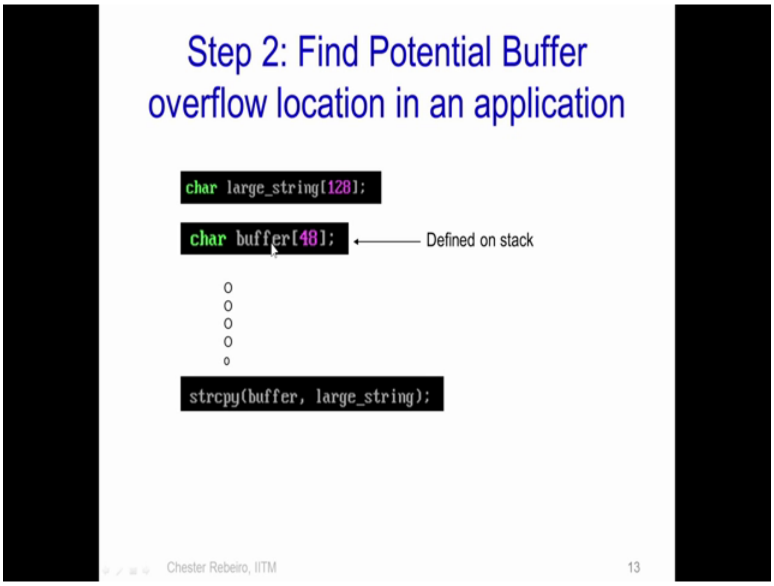
The first step in doing so is that the attacker needs to obtain the binary data corresponding to this particular program. In order to do this, what the attacker does is that he will re write this program in assembly code. So, this assembly code as we see here does exactly the same thing as done by this program.

The next thing what the attacker would do is to compile this particular assembly code and get what is known as the object dump. The object dump is obtained by running this particular command thing. So, he first compiles this particular code to get what is known as the shell code dot o which is the object file, and then he will run this particular command which is of objdump disassemble all shellcode dot o to get this particular file.

Now, what is important for us over here is this particular column or the second column. The numbers what you see over here the hexadecimal numbers are in fact, the machine code for this program. The numbers like eb, 1e, 5e, 89, 26, 08, and so on correspond to the machine code of this particular program. In other words, if the attacker manages to put this machine code onto the stack and is able to force execution to this particular machine code, then the attacker would be able to execute the shell as required.

The machine code is shown over here and one thing which is required for this particular attack is to replace all the zeros present in this machine code with some other instructions, so that you do not have any zeros present over here.

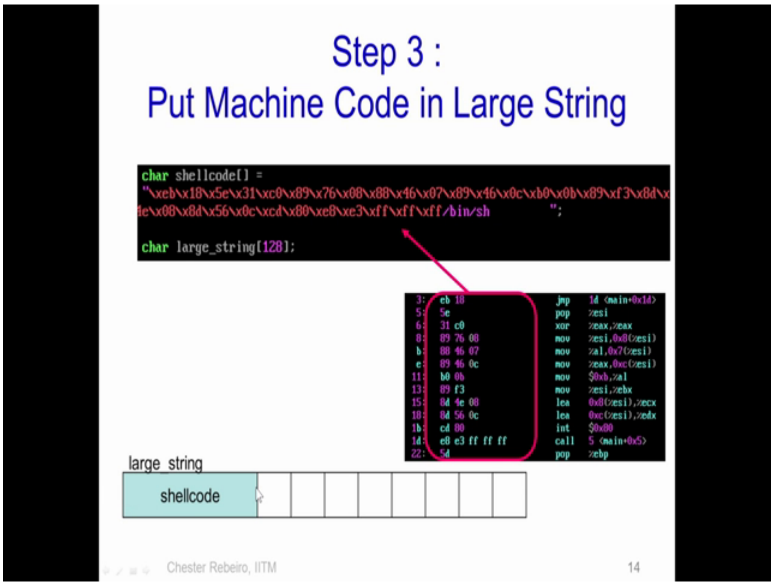
(Refer Slide Time: 20:48)



The next thing is to scan the entire application in order to find one location which can be exploited for a buffer overflow. So, essentially the requirements for a buffer overflow is that the attacker finds in the application code a command such as this a string copy buffer comma large string where buffer is a small array and it is defined locally in the stack while the large string is a much larger array. So, as we know the way this particular function strcpy works is that the large string gets copied to buffer and this copying will continue byte by byte until there is a slash 0, which is found in large string; in which case the strcpy will complete executing and will return.

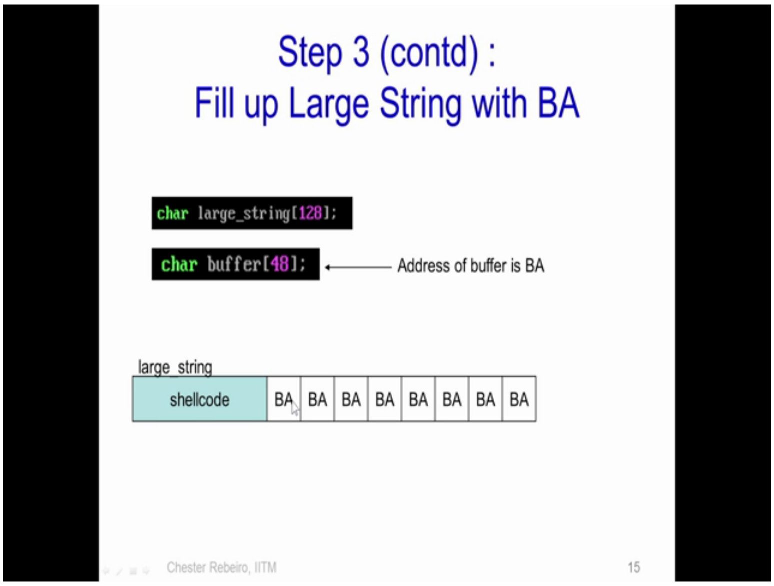
Let us assume that the attacker has found such a case where we have the buffer a large string and a string copy, and the buffer is a small array defined onto the stack, and how does the attacker make use of this.

(Refer Slide Time: 22:07)



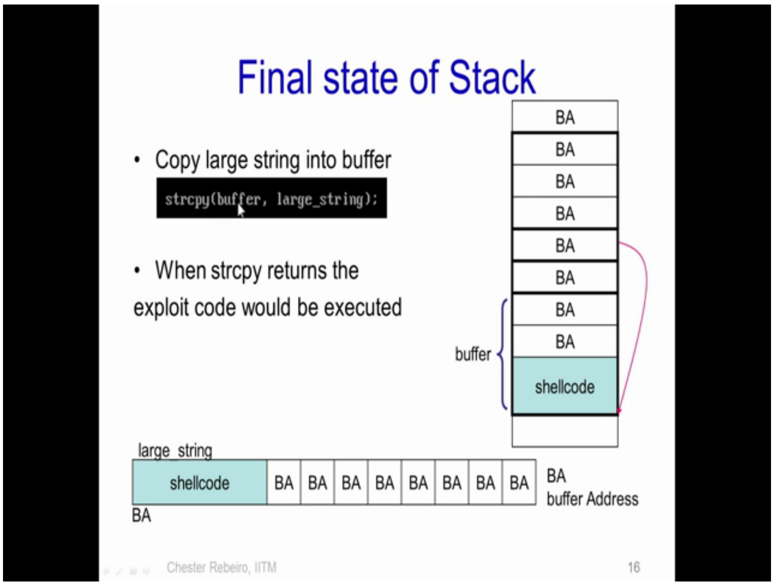
So, what the attacker would then do is create something known as the shell code array which essentially is the code that he wants to execute. So, he creates the shell code array comprising of all the assembly up codes or machine codes which it wants to execute and he places this code or the shell code in the first part of the large string. So, if you look at this the large string array, which is a very large string of 128 bytes in this case, gets the shell code in the first part.

(Refer Slide Time: 22:46)



Then he computes what the address of the buffer should be, and fills the remaining part of the large string with the buffer address.

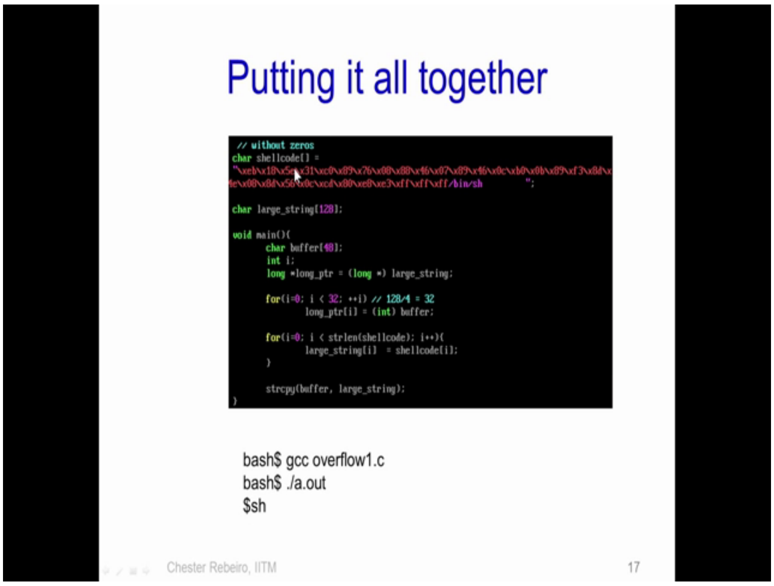
(Refer Slide Time: 23:00)



Now he needs to force this string copy to execute with this buffer and the large string which he has just created as shown over here. Now as result of this string copy being executed there is a stack frame created for the string copy and as we know that the string copy will continue to copy bytes from the large string to the buffer until it finds the slash 0. So, in such a way what would happen onto the stack is that the large string gets copied.

First, the shell code gets copied and then the buffer address keeps getting overflowed on to the buffer and keeps going on until a slash 0 is found. So, when the string copy executes what we have seen is that instead of getting the valid return address it now gets what is known as the buffer address. So, essentially we know that the buffer address points to this particular location, and therefore, the CPU would be forced to returned to this location pointed to by the buffer address and execute the shell code as a result the attacker would be able to execute the shell code which in this particular example was the exploit.

(Refer Slide Time: 24:22)



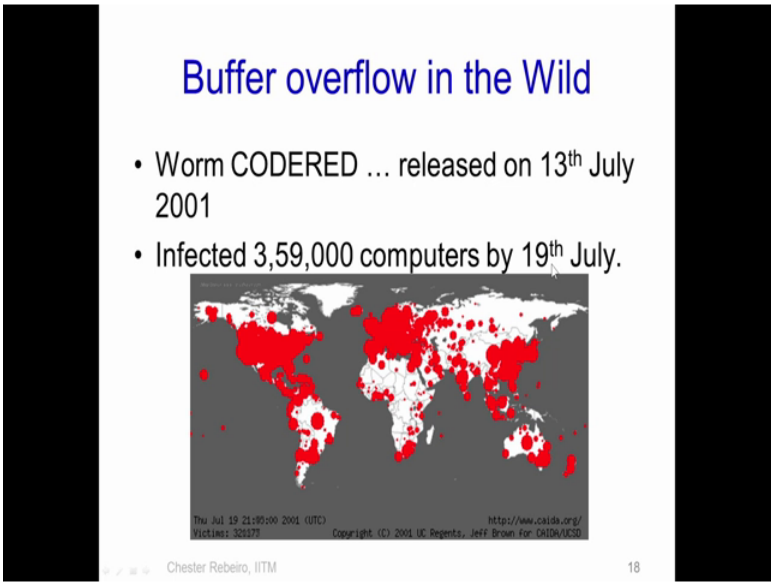
Let us look at the entire thing all together. We have the shell code which is the code which the attacker wants to execute. Over here, we have just defining it as a global array, but in reality this could be entered through various things like a scanf or it could also come in through the network card, essentially we passed a packet with particular format containing the exploit and various other different ways of passing in the shell code.

Next, let us assume that somewhere in the application there is this particular code, we have the large string and we have short string, which is buffer, which is a local array a locally defined array and therefore, gets created onto the stack.

So, what we first do is somehow manage to fill the long string or the large string with the address of buffer. So, if you recollect, this is the BA paths which are present, then we will copy the shell code on to the large string. So, we have created this large string in the format that we require. In the first part of this large string is that the shell code, and then it is followed by the buffer return address. And then if there is a function like string copy which copies large string into buffer it will result in a buffer overflow to occur and instead of the function returning to this particular point soon after string copy.

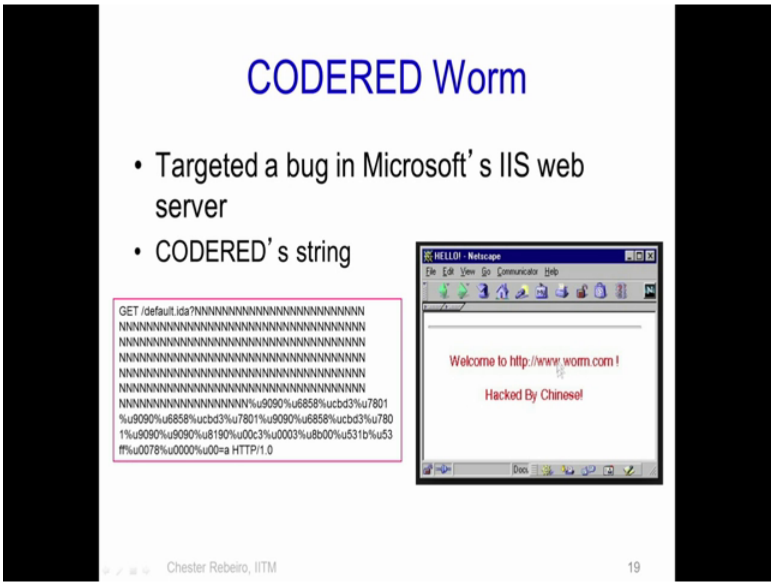
On the other hand, execute this particular shell code and cause this shell specified by this command to be executed. So, if we actually see this if we execute gcc overflow dot c or this is called overflow 1 dot c and run dot slash a dot out. Instead of just doing this string copy and exiting, this particular program created a new shell due to the exploit code that is executing.

(Refer Slide Time: 26:44)



So, buffer overflows are an extremely well known bug and extremely exploited by various different malware and viruses over the last decade or actually more than a decade. And one of the first viruses that actually use the buffer overflow was the worm called CODERED which was released on July 13th 2001. So, this created a massive chaos all over the world, and the red spots actually show how the virus spread across the world in about or rather in less than a day or a few hours. So, we see that this particular virus which used the buffer overflow infected roughly 359000 computers by July 19th 2001.

(Refer Slide Time: 27:36)



Essentially the targeted application by this worm or this particular worm was the Microsoft IIS web server and the string which was executed was as shown over here. So, this string was actually the exploit code which was executed and what it resulted was something like this being displayed in the web browser.

Thank you.