

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

Week – 08
Lecture – 38
Preventing Buffer Overflow Attacks

Hello. In the previous video, we had seen how attacker essentially an unauthorized user of the system could cause his own code to execute in the system by exploiting what was known as a buffer overflow bug. So, we had seen at least we had created such a buffer overflow bug and shown how the attacker could create his exploit code and run that exploit code which created a shell in the system.

So, this particular example was an application based example in the user space, but very similar kind of exploits can be written in the operating system. So, what we will be seeing in this video are techniques of how this buffer overflow vulnerability is overcome, and also how the attack has progressed over the years, and evolved over the years to make more powerful attacks in order to overcome these protections.

(Refer Slide Time: 01:32)

Non-executable stack

- Mark the stack pages as non-executable.

```
// without exec
char shell[] =
"/bin/sh;cd /tmp;cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 32 | xargs -0 bash &&";

char large_string[1024];

void main() {
  char buffer[64];
  int i;
  long *long_ptr = (long *) large_string;

  for(i=0; i < 32; ++i) *(long_ptr + i) = (i) * 0x4 + 32;
  long_ptr[i] = (i) * buffer;

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

strcpy(buffer, large_string);
```

```
bash$ gcc overflow1.c
bash$ ./a.out
Segmentation Fault
```

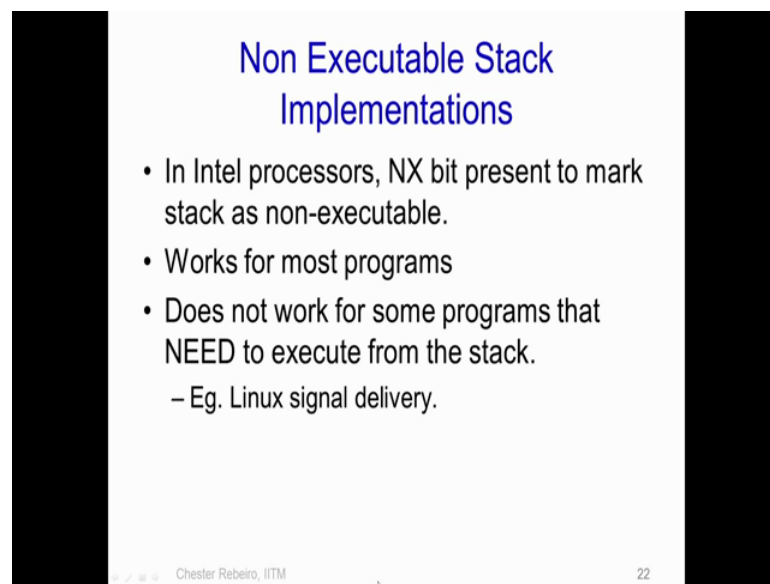
Chester Rebeiro, IITM 21

The first and the most obvious way to prevent the buffer overflow attack which occurs in

the stack are by making the stack pages non-executable. What we seen is that the attacker would force the CPU to execute an exploit code which is also present in the stack. So, for instance over here, this buffer which is defined on the stack also contain the exploit code and the string copy was executed in such a way that after string copy completed its execution, it would cause the exploit code that is the shell code which is present on the stack to be executed.

So, one obvious way to prevent this attack is to make the stack pages non-executable, so and this is what is done in systems that are used these days. So, if you would actually run this particular program on your Intel systems, and instead of getting the exploit code to execute, you would get a segmentation fault. This would be caused because the program is trying to execute some instructions onto the stack.

(Refer Slide Time: 02:58)



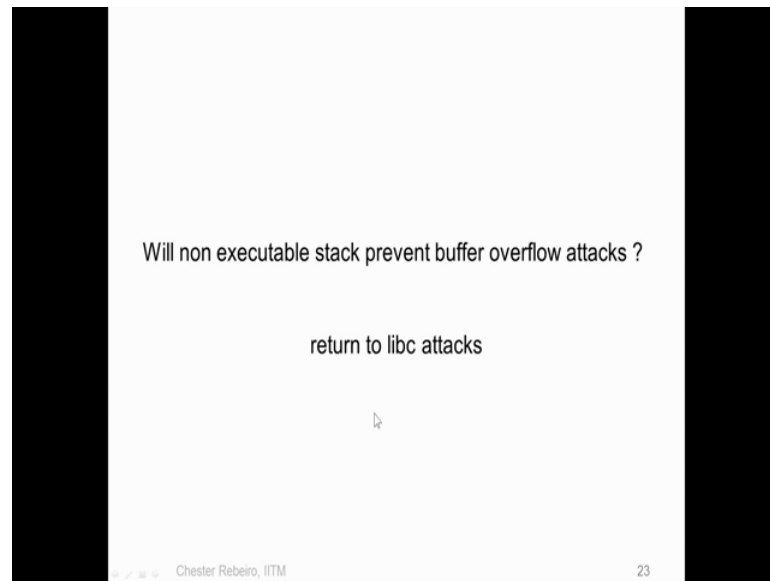
Non Executable Stack Implementations

- In Intel processors, NX bit present to mark stack as non-executable.
- Works for most programs
- Does not work for some programs that **NEED** to execute from the stack.
 - Eg. Linux signal delivery.

Chester Rebeiro, IITM 22

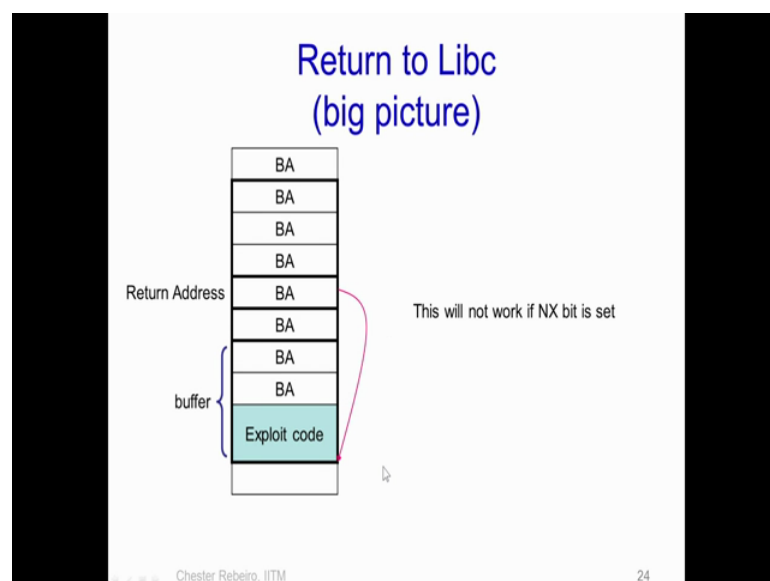
In Intel machines, an NX bit is present in the page tables to mark the stack as non-executable. While this work for most of the programs that is in most programs it is a added benefit, but the problem is some programs even though they need not be malicious require to execute from the stack. These programs need to execute from the stack in order to function properly. Therefore, setting the NX bit is not always very beneficial for all programs.

(Refer Slide Time: 03:37)



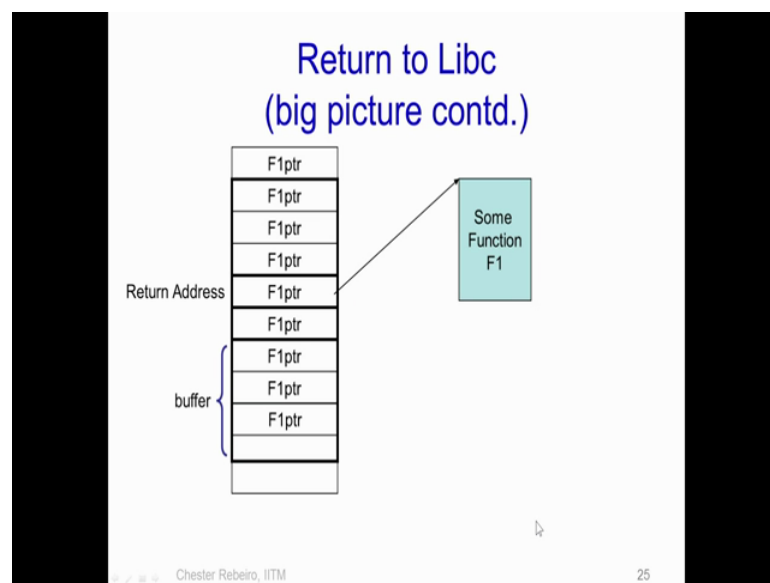
The next thing, we are going to ask ourselves is that if we make the stack as non-executable, will it completely prevent buffer overflow attacks. And in fact, it does not. Over the years buffer overflow attacks have evolved to something known as the return to libc attacks, which could be used even on systems, which have a non-executable stack.

(Refer Slide Time: 04:13)



Let us see in very brief how a return to libc attack works. So, essentially what we have done, when we try to overflow the buffer in the stack is that we had the exploit code present onto the stack, and we had replaced the valid return address with the address of the buffer. Now this did not work for us in modern day systems, because the stack was set to non-executable. Let us look at how return to libc works in spite of having the non-executable stack that is in spite of having the NX bit set.

(Refer Slide Time: 04:56)

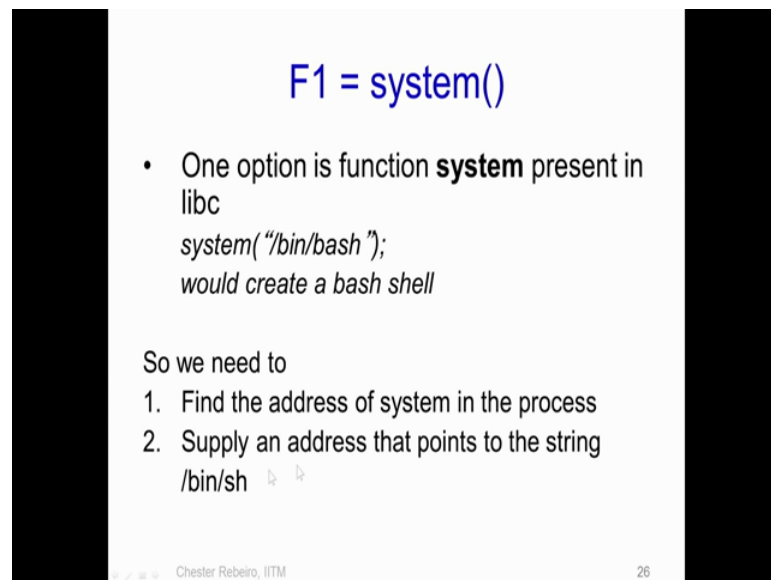


So, what the return to lib c does is that instead of forcing the return address to branch to a location within the stack, it forces the return address to go to some other location containing some other function. Let us say this function is F 1. So, what is filled onto the stack through a buffer overflow is a pointer to F 1, therefore when the function completes executing the return address taken from here is a pointer to F 1, and therefore, the CPU is forced to execute this function F 1.

Now, the next question is - what is this function F 1? So, one thing is certain that this function cannot be the attacker's own exploit code. So, it has to be some valid function which is already present in the code segment and which has the permission to execute. So, what would this function F 1 be, that is point number 1; and point number 2 is how will an attacker use a normal function which is present in the program to do something

malicious such as to run and exploit code.

(Refer Slide Time: 06:23)



F1 = system()

- One option is function **system** present in libc
`system("/bin/bash");`
would create a bash shell

So we need to

1. Find the address of system in the process
2. Supply an address that points to the string /bin/sh

Chester Rebeiro, IITM 26

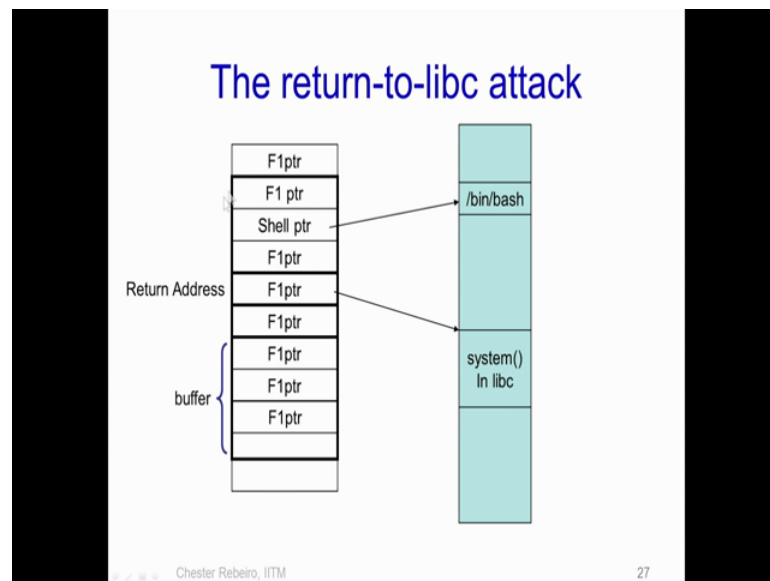
There are various ways in which the function can be implemented, but what we will see today is the function F 1 implemented using this particular function called system. Now, system is a function which is present in the library called libc, and what it does is that it takes a character pointer to a string which would take an executable. The string, which is passed to system takes an executable name and when executed it would essentially execute this particular program.

So, in this particular example, what system would do is that it would execute the bash shell there by creating a bash shell. Now libc is a library, which is used by most programs. So, even a normal hello world program that you write would quite likely use the libc library. So, what this means is that in your processor's address space, there is the function system. So, what the attacker would need to do now is to just identify in your processor's address space where the system function resides. Next he needs to somehow pass an argument to this in order to run a program.

So, suppose let us say if we are continuing the example, what we seen in the previous video where the attacker creates an exploit which executes a shell. So, in this case also, if

the attacker wants to execute the same shell, then in addition to finding the systems functions address in the memory space, the attacker would somehow needs to pass this particular string slash bin slash sh as a parameter to the system. So, this means that the attacker would need to find an address that points to the string slash bin slash sh.

(Refer Slide Time: 08:32)

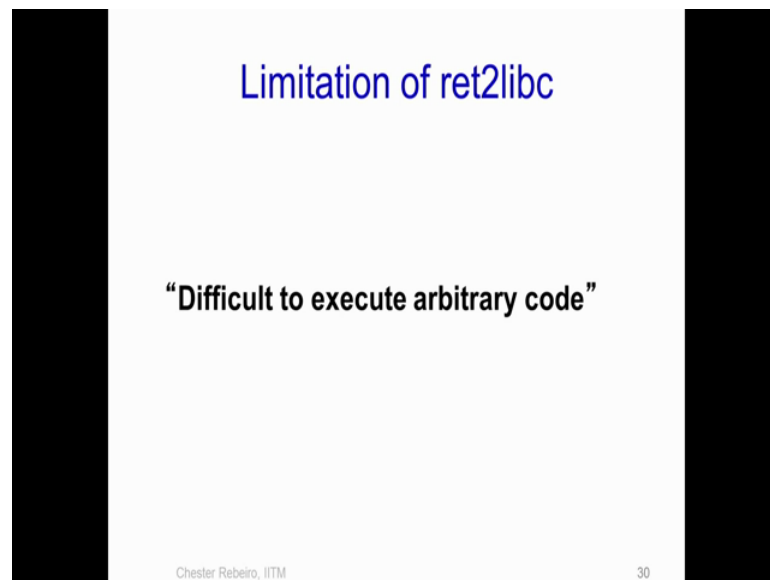


So, what is typically done is have a stack frame something like this, where the return address or rather the valid return address is replaced by F 1 pointer which is a pointer to the system function which is present in libc. And if your program uses the dynamically linked library libc, then such an address will be valid. Second, what the attacker needs to do is to pass an argument to the system function, which essentially has an executable name.

So, essentially somewhere in your address space, the attacker needs to find this particular string being present that is slash bin slash bash or slash bin slash sh, and fill in the stack with the pointer to this particular string. So, essentially, it requires two things; one is the return address in the stack to be modified with the address of the system call which is present in libc, and also the parameter passed to libc should present the stack on the stack. So, in this case it is a shell pointer, which points to this particular string slash bin slash bash.

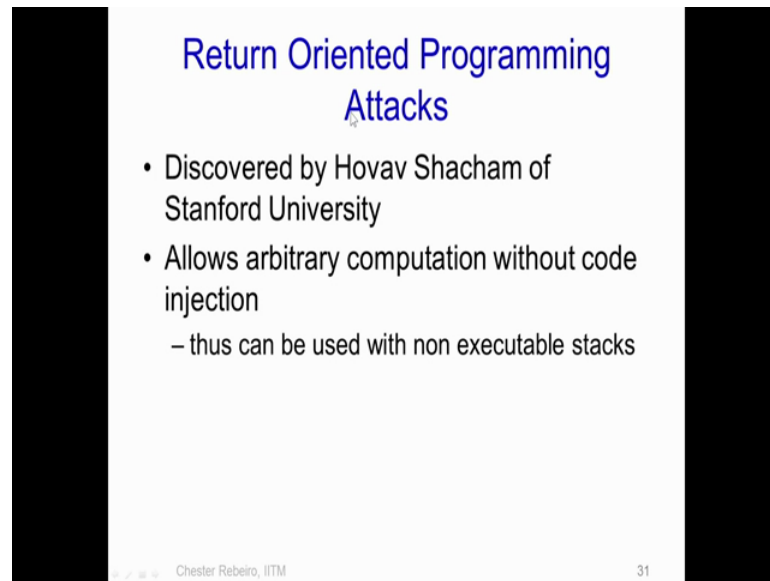
So, once this is done, when the function returns instead of returning to the valid address, it is going to cause the CPU to execute this particular system. Now during this process also the pointer to this particular shell slash bin slash bash would be considered in the system call and it would result in a shell being formed. So, from the shell, the attacker could then spawn various other things and run his own programs. So, in this way, the return to libc attack works in spite of having a stack which is non-executable. So, note that we are not executing any of these instructions; we are just reading and writing to the stack while the real execution occurs in the code segment itself by the function system.

(Refer Slide Time: 10:51)



Now, the limitation of the return to lib c is that it is extremely difficult to execute some arbitrary code. So, we have seen one example of how the attacker could execute a shell, but the amount that an attacker could do with the return to libc type of attack is very limited. And therefore, the attacks over the period of time have evolved to something stronger.

(Refer Slide Time: 12:24)



The slide features a white background with a black border on the left and right sides. The title 'Return Oriented Programming Attacks' is centered at the top in a blue font. Below the title, there is a bulleted list of two main points. The first point states that the attack was discovered by Hovav Shacham at Stanford University. The second point states that it allows for arbitrary computation without code injection, and includes a sub-point that it can be used with non-executable stacks. At the bottom of the slide, there is a small footer containing navigation icons, the name 'Chester Rebeiro, IITM', and the page number '31'.

Return Oriented Programming Attacks

- Discovered by Hovav Shacham of Stanford University
- Allows arbitrary computation without code injection
 - thus can be used with non executable stacks

Chester Rebeiro, IITM 31

And more recently, there is something known as the return oriented programming attacks. So, this is one of the most powerful attacks that are known which utilize buffer overflows. So, this is also applicable for systems which have non-executable stack. The return oriented program or also for short known as the ROP attack was discovered by Hovav Shacham in the Stanford University. And it allows any arbitrary computation or any arbitrary code to be injected into the program in spite of having a non-executable stack. Let us see with a very small example how this thing works.

(Refer Slide Time: 12:13)

Gadgets (1)

Lets say this is the payload needed to be executed by an attacker.

```
"nopl %esi, 0x8(%esi);"  
"nopl $0x0, 0x7(%esi);"  
"nopl $0x0, 0xc(%esi);"  
"nopl $0xb, %eax;"  
"nopl %esi, %ebx;"  
"leal 0x8(%esi), %ecx;"  
"leal 0xc(%esi), %edx;"
```

Chester Rebeiro, IITM 32

Let us say that the code that the attacker wants to execute is given by these set of assembly lines. So, essentially if the attacker manages to execute these assembly codes, then his job is done, he will be able to run whatever exploit he wants. Now what the ROP attacker is used something known as the gadget.

Now gadgets essentially mean splitting this particular code or the payload as it is called into small components, which are known as gadgets. So, it has been shown that a variety of different pay loads could be executed just by using gadgets, and therefore we could have a variety of different exploit codes that have been used by the attacker. Let us see what a gadget is.

(Refer Slide Time: 13:12)

Gadgets (2)

- Scan the entire binary for code snippets of the form

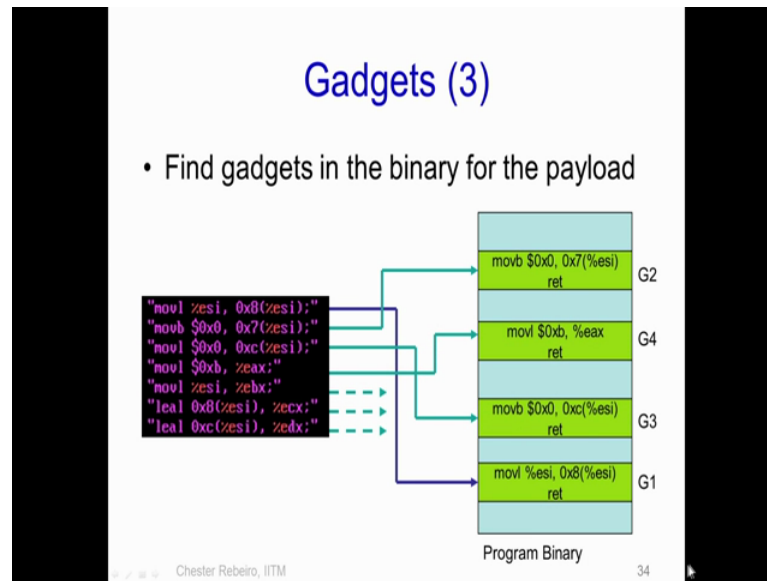
useful instruction(s)
ret

- This is called a gadget

Chester Rebeiro, IITM 33

Essentially a gadget is some useful instruction, and when I say useful instruction over here it means that it is one of these instructions in the payload that it needs to be executed as part of the exploit followed by a 'ret' - a return. So, ret as you know in the Intel instruction set it means the return from the particular function. So, this very simple thing is - what is the gadget? Now, what the attacker needs to do is corresponding to the payload that he wants to execute, he needs to scan the entire binary code of the executable in order to find such useful gadgets.

(Refer Slide Time: 14:00)



So, for instance, if we take this particular payload, the attacker may find that somewhere in the program's binary that is among all the instructions that are present there is this particular gadget present which does almost the same thing what is required that is this particular instruction in the payload is `mov esi, 0x8 esi`.

And this exact same thing is present over here. Similarly, what the attacker would do is he would scan the other paths of the program binary in order to find more such gadgets. The second line in the payload is present in the gadget two; the third line is present here and so on. So, he essentially what he would do is force this payload to execute by using gadgets. So, what he is going to overflow the stack with is a chain of gadgets, so essentially the stack is going to contain G 1, then G 2, G 3 and G 4.

These gadgets are or the addresses to these gadgets are organized in such a way that when the first valid return address is met instead of finding the valid return address the address of gadget 1 is found and as a result this instruction corresponding to gadget 1 is executed. And then there is a return and the stack is arranged in such a way that gadget 2 will then execute then gadget 3, gadget 4 and so on. So, in total although the instructions are not in contiguous locations what the attacker has managed to do is he has managed to execute all these instructions. So, in this way, he is able to execute his payload and it has

been shown that a large set of such gadgets are feasible in programs.

(Refer Slide Time: 16:22)

Other Precautions for buffer overflows

- Use a programming language that automatically check array bounds
 - Example java
- Use securer libraries. For example C11 annex K, gets_s, strcpy_s, strncpy_s, etc. (_s is for secure)

Chester Rebeiro, IITM 35

So, other precaution for buffer overflows is to use a programming language such as java which automatically checks array bounds. So, this will ensure that no array is accessed out of its limits. Another way is to use more secure libraries for example, the C 11 specification annex K, specifies these secure libraries to be used. So, for example, the get s underscore s, string copy underscore s, string n copy underscore s, so all these have the same functionality as we as the standard functions that we use, but these functions are secure and would prevent buffer overflows.

(Refer Slide Time: 17:14)

Canaries

- Known (pseudo random) values placed on stack to monitor buffer overflows.
- A change in the value of the canary indicates a buffer overflow.
- Implemented in gcc by default.
- Evaded if canary is known

```
function:  
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
leave  
ret
```

Insert a canary here

check if the canary value has got modified

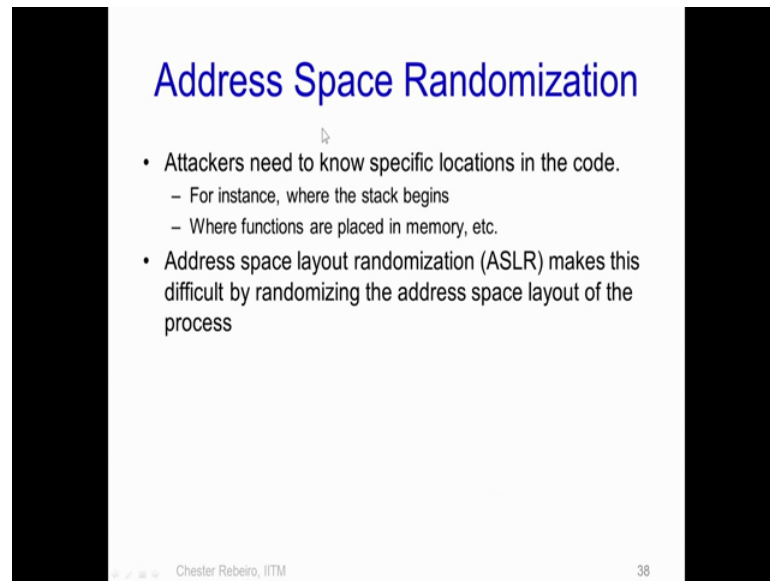
Stack (top to bottom):	
	stored data
3	
2	
1	
	ret addr
	sfpr (%ebp)
	Insert canary here
	buffer1
	buffer2

Chester Rebeiro, IITM 36

Another popular use to prevent buffer overflows is by the use of canaries. So, a canary is a known pseudo random value which is placed onto the stack in order to detect buffer overflows. What is done is that at the start of the function a canary is inserted onto the stack push some canary value on to the stack as shown over here. So, in addition to the parameters, the return address the frame pointer, we have now a canary also. And just before returning or leaving the function, the canary is checked to find out whether it is modified.

Now if a buffer overflow occurred then as we know the buffer overflow would modify the addresses in the stack, and as a result the canary value would be modified, and therefore, we would be able to detect a change in the canary value if a buffer overflow occurred. And therefore, we will be able to perhaps stop the program. So, in these days in recent versions of the gcc compiler such canaries are implemented by default. So, this being said the entire use of canary is evaded if the canary value is known that is if the attacker manages to know what the canary value is used then he could just change or set this value of canary in such a way that this canary is not changed at all and therefore, its use is limited.

(Refer Slide Time: 18:56)



The slide features a title 'Address Space Randomization' in blue text at the top. Below the title is a bulleted list with two main points. The first point is 'Attackers need to know specific locations in the code.', which has two sub-points: 'For instance, where the stack begins' and 'Where functions are placed in memory, etc.'. The second point is 'Address space layout randomization (ASLR) makes this difficult by randomizing the address space layout of the process'. At the bottom of the slide, there is a footer that reads 'Chester Rebeiro, IITM' on the left and the number '38' on the right.

- Attackers need to know specific locations in the code.
 - For instance, where the stack begins
 - Where functions are placed in memory, etc.
- Address space layout randomization (ASLR) makes this difficult by randomizing the address space layout of the process

Chester Rebeiro, IITM 38

Another way to prevent this particular attack something known as the address space randomization or ASLR, in this particular counter measure technique it uses the fact that the attackers need to know specific locations in the code. In the return to lib c attack for instance, the attacker needed to know where the function F 1 or in our example where the function system was located in the program space. N

ow if we had ASLR enabled then the layout of the address space is randomized therefore, the attacker would find it difficult to determine where exactly the function is present that needs to be exploited. In other words, the attacker would find it difficult to find out where the function system is present in the address space, thus it would make the attack much more difficult

Thank you.