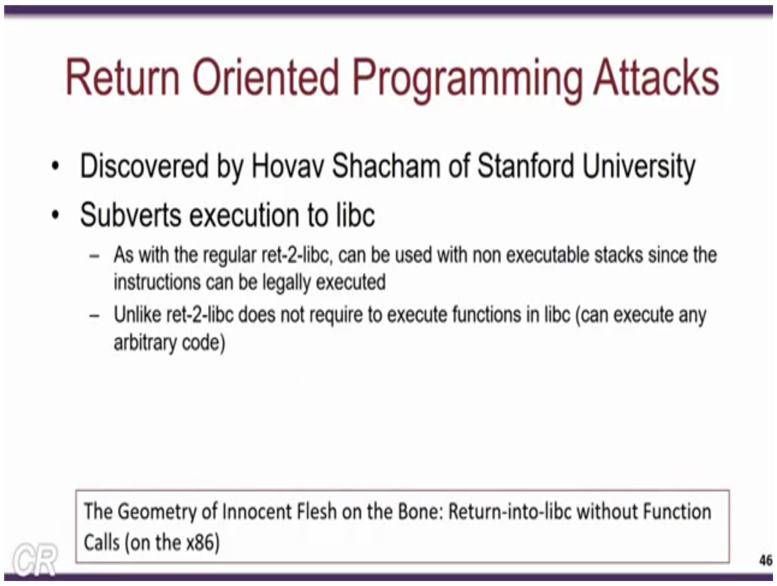
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Return Oriented Programming (ROP)

Hello and welcome to this lecture in the course for secure systems engineering. In the previous lecture we had looked at this attack call return to libc which had the advantage that it could work with a non-executable stack. However, as we seen in the previous lecture the major limitation of the return to libc attack is the fact that the attacker is constrained with what the libc offers.

So we had seen that the attacker could only invoke all the functions that are present in libc, we had seen for example if the system function present in libc was removed then the attack that we have built in the previous lecture will not execute anymore.

(Refer Slide Time: 1:16)



Return Oriented Programming Attacks

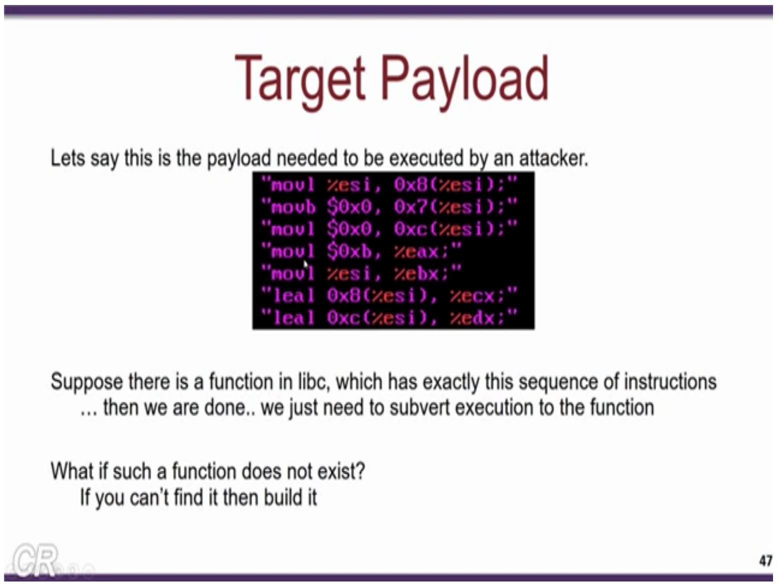
- Discovered by Hovav Shacham of Stanford University
- Subverts execution to libc
  - As with the regular ret-2-libc, can be used with non executable stacks since the instructions can be legally executed
  - Unlike ret-2-libc does not require to execute functions in libc (can execute any arbitrary code)

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

46

So let us start this particular lecture with what is the ROP attack. So the ROP attack or return oriented programming attack was discovered by Hovav Shacham in Stanford University. So just like the return to libc attack it subverts execution of a program by overwriting the return address present in the stack with some address present in libc. However, unlike the return to libc attack the ROP attack is not restricted to execute functions that are present in libc, rather it can execute almost any arbitrary code.

(Refer Slide Time: 2:00)



Target Payload

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

```
"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;"
```

Suppose there is a function in libc, which has exactly this sequence of instructions
... then we are done.. we just need to subvert execution to the function

What if such a function does not exist?
If you can't find it then build it

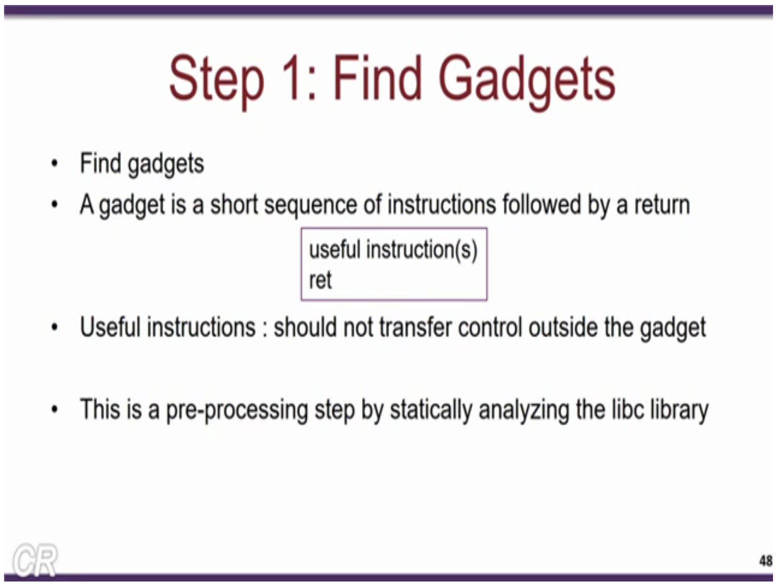
47

So let us take a small example. Let us say that the payload that the attacker wants to execute is as follows. It comprises of 7 different instructions. So now the first thing the attacker would do in order to execute these instructions in the payload is to find some function in or the libc which has all of these 7 instructions in this order. However, quite likely he will not find such a sequence of instructions present completely in one function. Therefore, it the return to libc attack would not work.

However, in the ROP attack we do things a little bit differently instead of relying on a specific function in the libc library which has all of these instructions in this particular sequence we build a function that executes all of these instructions in this particular sequence. Now suppose there is a function in the libc which has exactly this sequence of instructions then we are done the attacker would just need to subvert execution to that particular function and his payload can be executed.

However, most likely there would not be a function in libc which has exactly this sequence of instructions. Therefore, what we do in the ROP attack is something different. What we do is that if we cannot find specific function which has this entire sequence of instructions, we try to build it ourselves. So let us look at how we go about building a sequence of instructions that creates this particular target payload that the attacker would want to execute.

(Refer Slide Time: 4:00)


Step 1: Find Gadgets

- Find gadgets
- A gadget is a short sequence of instructions followed by a return

```
useful instruction(s)
ret
```

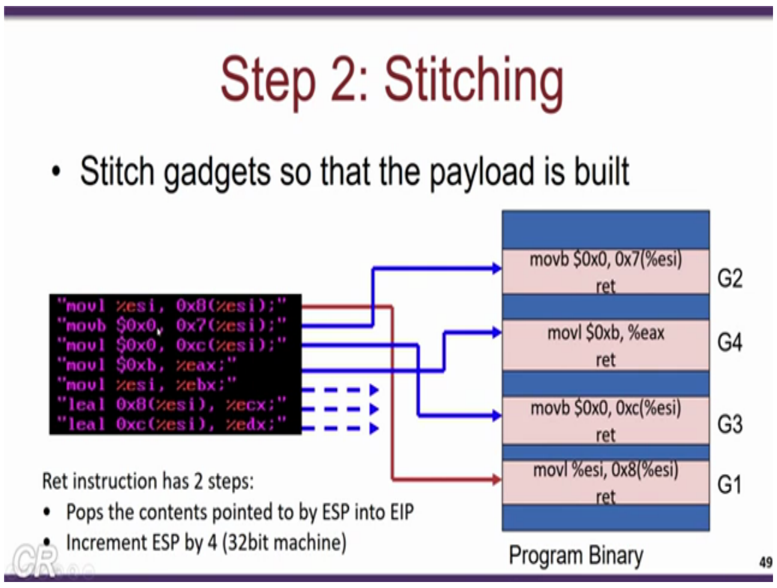
- Useful instructions : should not transfer control outside the gadget
- This is a pre-processing step by statically analyzing the libc library

48

The first step is finding something known as gadgets. Now a gadget is a short sequence of instructions which is followed by a return. So a typical gadget would look something like this, it would have one or more useful instructions and then it would have a return.

Now one restriction for these useful instructions is that it should not transfer control outside the gadget. So how do we find such gadgets? So we would do a pre-processing by statistically analysing the libc library and identify all the possible gadgets that are present in the library.

(Refer Slide Time: 4:36)


Step 2: Stitching

- Stitch gadgets so that the payload is built

```
"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;"
```

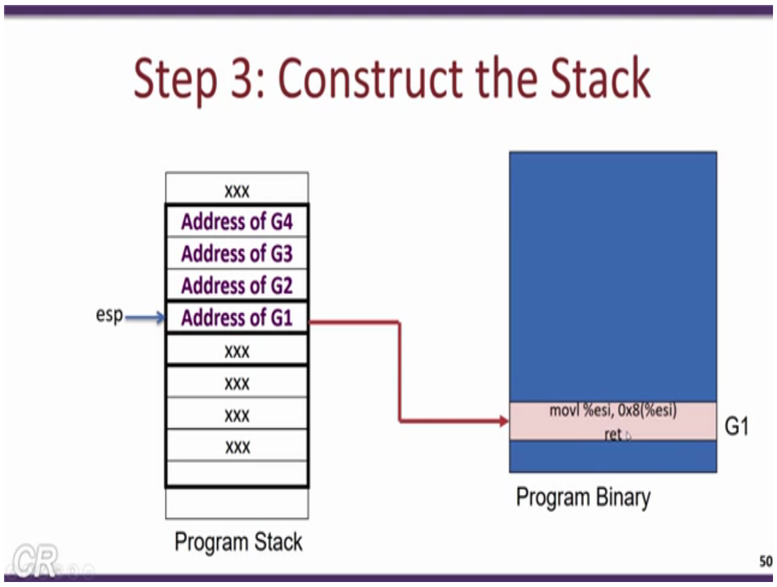
Ret instruction has 2 steps:
- Pops the contents pointed to by ESP into EIP
- Increment ESP by 4 (32bit machine)

49

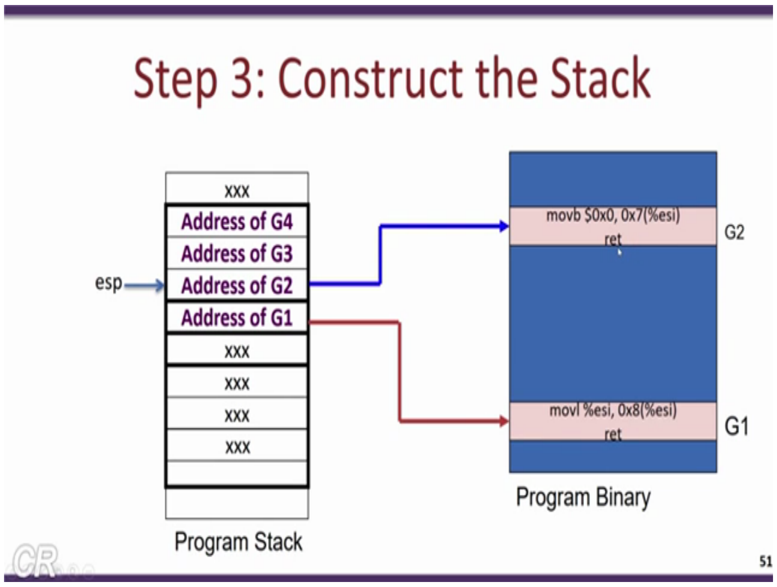
The second step is to stitch the various gadgets together. Now let us take our target payload as we have done before. The second step is that we want to stitch these useful gadgets together. So let us take the same target payload that we wanted to execute which contains 7 some arbitrary instructions and we assume that we have been able to find gadgets for each of these instructions.

So let us say that this is the program binary in particular this is the libc library, what we have identified is 7 different gadgets which will do exactly this functionality, each gadget for example here gadget G1 has this instruction followed by a return. Similarly there is a gadget G2 which has this instruction followed by a return and so on and in this way we identify gadgets in the libc file which has this functionality.

(Refer Slide Time: 5:50)



Step 3: Construct the Stack



Step 3: Construct the Stack

So the next step is we need to stitch these gadgets together so that to achieve this functionality and the way we do that is as follows so we go back to our stack and we build a stack as follows in the location where the return address should be present we put the address of the gadget 1, 4 bytes above that we put the address of gadget 2, then gadget 3 and gadget 4. What happens during execution is as follows, when the function that is getting executed returns at that particular time the stack pointer is pointing to this location where the original return address should be present.

However, since we have replaced that original return address with the address of gadget 1, this address is taken and loaded into the instruction pointer. Therefore, what we would achieve is that these two instructions are executed. At the time when the function that is executing is going to return, the stack pointer is pointing to this particular location where the return address should have been present.

Now what we have done is we have replaced that return address with the address of gadget 1. Now when the return instruction is executed there are two things that simultaneously occur. First the contents pointed to by the stack pointer that is the address of gadget 1 gets loaded into the program counter. Second thing the stack pointer increments by a value of 4. Therefore, the stack pointer is pointing to the address gadget 2.

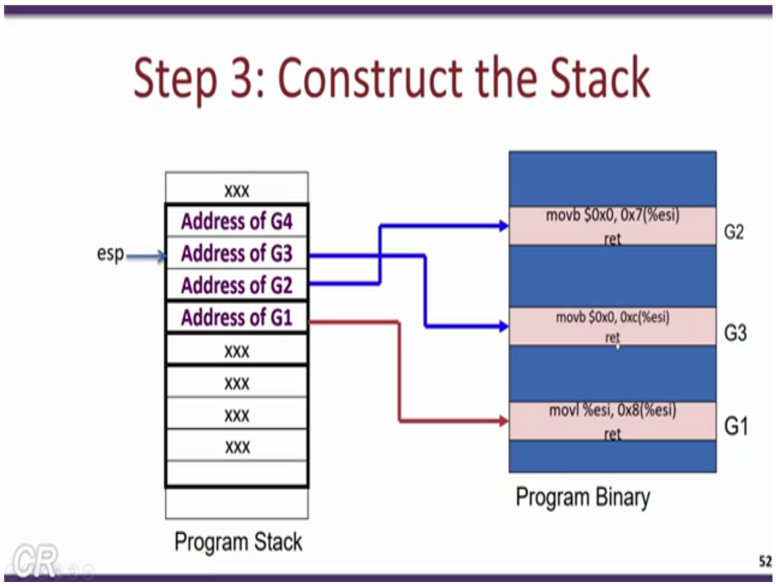
Now since the program counter is counting to the address of gadget 1 which is this location these two instructions will execute that is movl esi to 8 plus esi and then there is a return. In order to get these gadgets to work together we need to stitch these gadgets. In order to do that we overflow the buffer and arrange the stack in the following way that is at the location where the return address should be present, we put the address of G1, followed by the address of G2, followed by the address of G3 and finally the address of G4.

Now when the function that we are executing is about to return we have the stack pointer which is pointing to the address of gadget 1. Now when this function actually executes the return instruction there are two things that occur simultaneously. First the address of G1 gets taken from this particular location which is pointed to by the stack pointer and loaded in the program counter.

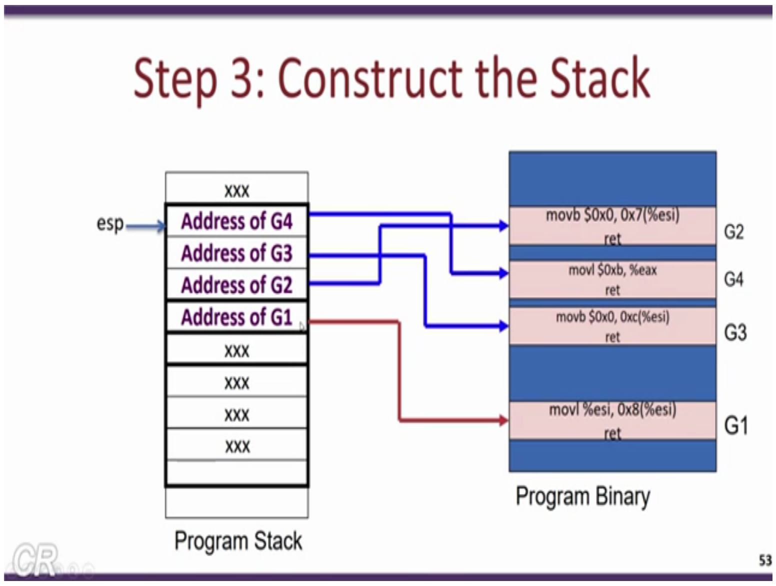
The second thing which happens atomically is that the stack pointer increments by 4 and points to the address of G2. Now since the program counter has the address of G1, the instructions present in gadget 1 will execute that is namely this mov instruction and then the return, just before the return instruction in gadget 1 is going to get executed we have the stack pointer which is pointing to the address of gadget 2.

So when the return instruction gets executed what happens as before is that the contents of the stack pointer that is in this case the address of G2 is loaded into the program counter atomically the stack pointer is incremented by 4 and points to the address of G3. Now since the program counter is pointing to the address of G2 we have gadget 2 getting executed which is the mov byte instruction followed by the return.

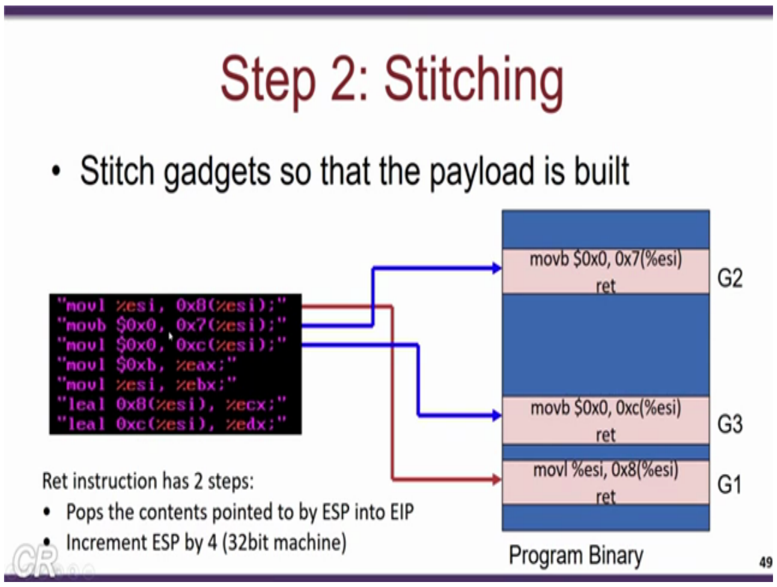
(Refer Slide Time: 9:54)



Step 3: Construct the Stack



Step 3: Construct the Stack



Step 2: Stitching

- Stitch gadgets so that the payload is built

```
"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;"
```

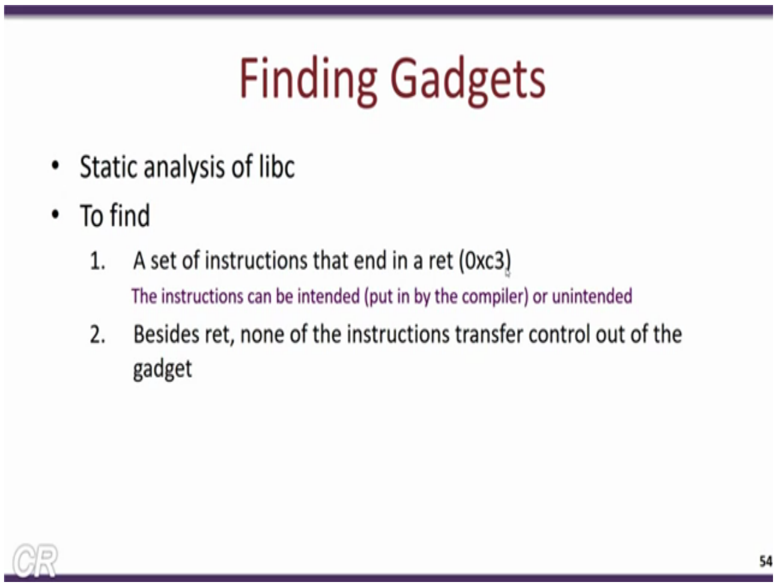
Ret instruction has 2 steps:

- Pops the contents pointed to by ESP into EIP
- Increment ESP by 4 (32bit machine)

Now as we have seen in the previous gadget when the return executes we have the address of gadget 3 taken into the program counter and therefore the processor would execute gadget 3 instructions comprising of the mov byte 0 to the offset of esi plus c. Similarly when there is the return instruction in gadget 3 it would result in the gadget 4 getting executed. In this way by appropriately placing address of gadgets on the stack we are able to execute the different gadgets and therefore we are able to achieve the sequence of instructions that a target payload had to achieve.

So note that unlike the return to libc attack we are not relying on specific functions in the libc library to mount our attack and to subvert execution. But rather we are depending on small snippets of codes which we call gadgets and we are creating these the stack contents in such a way that these various gadgets execute and are able to give the same effect as having a sequence of the target payload instructions getting executed.

(Refer Slide Time: 11:12)



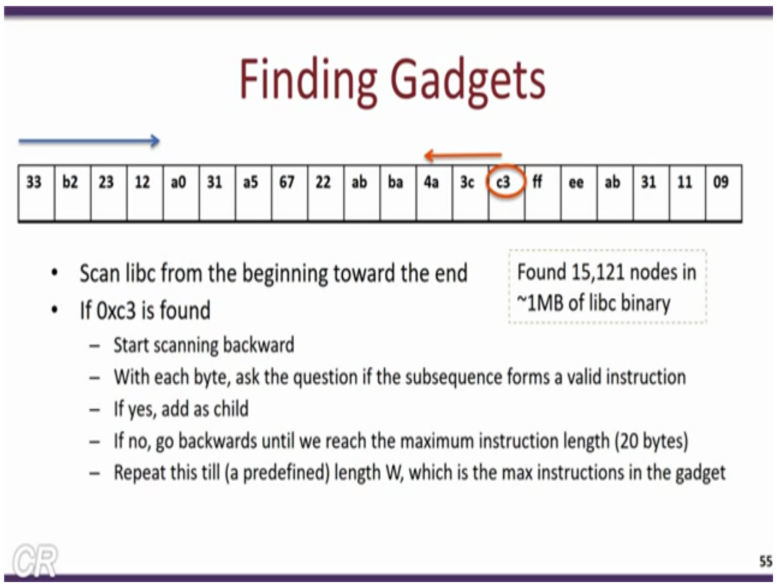
Finding Gadgets

- Static analysis of libc
- To find
  1. A set of instructions that end in a ret (0xc3)
     The instructions can be intended (put in by the compiler) or unintended
  2. Besides ret, none of the instructions transfer control out of the gadget

54

So the next thing we will actually look at is how do we find such gadgets in our library? The way to go about it is by static analysis of the libc library. So we need to find all the gadgets that the libc library contains. As we know a gadget is sequence of useful instructions which is ending with the return instruction. So the opt code for this return instruction is these numbers 0xc3. So what we need to do is to identify gadgets which are ending with the opt code 0xc3.

(Refer Slide Time: 11:46)



Finding Gadgets

33	b2	23	12	a0	31	a5	67	22	ab	ba	4a	3c	c3	ff	ee	ab	31	11	09

- Scan libc from the beginning toward the end
- If 0xc3 is found
  - Start scanning backward
  - With each byte, ask the question if the subsequence forms a valid instruction
  - If yes, add as child
  - If no, go backwards until we reach the maximum instruction length (20 bytes)
  - Repeat this till (a predefined) length W, which is the max instructions in the gadget

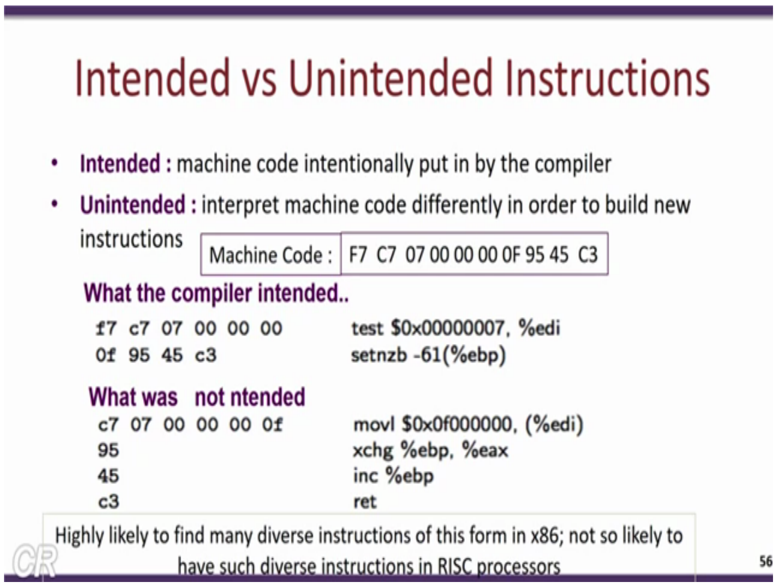
Found 15,121 nodes in ~1MB of libc binary

55

The way we find gadgets is by statically analysing the libc library. So we open the library in binary mode and start to scan the library byte by byte until we get c3 instructions. So for example over here the first c3 that we obtain over here is this, now we assume that this c3 corresponds to a return instruction and once we have found this c3 instruction we try to build gadgets out of it.

So in order to do this we look at the previous byte values and try to form useful instructions. So if you are able to form a useful instruction we create a tree with c3 as the root and that useful instructions as (childs) children of that root. So we keep going backwards until a predefined length W is achieved, which is the maximum number of instructions in the gadget in a typical libc file which is about 1 megabyte in size, we find over 15000 different gadgets. Now whenever we want to build a payload we need to select appropriate gadgets from these 15000 so that our target payload execution is achieved.

(Refer Slide Time: 13:06)



Intended vs Unintended Instructions

- **Intended :** machine code intentionally put in by the compiler
- **Unintended :** interpret machine code differently in order to build new instructions

Machine Code : F7 C7 07 00 00 00 0F 95 45 C3

What the compiler intended..

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

What was not ntended

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

Highly likely to find many diverse instructions of this form in x86; not so likely to have such diverse instructions in RISC processors

56

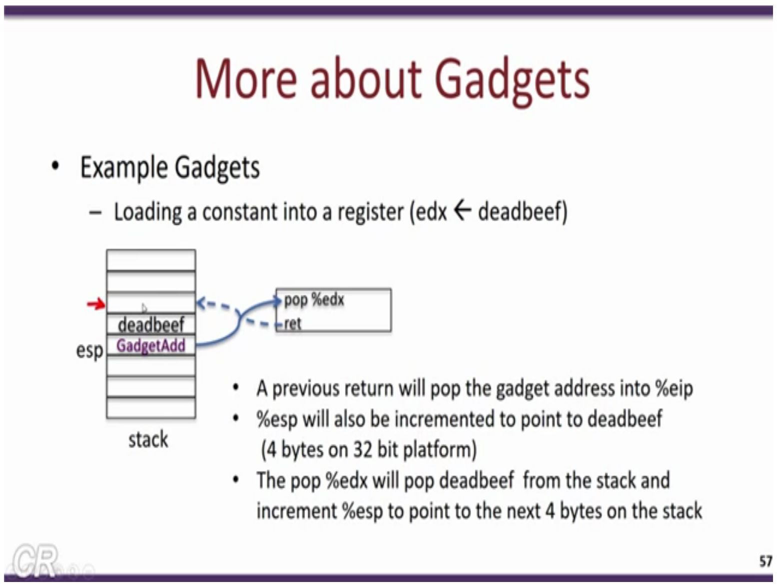
Now there are two ways gadgets can be found one is known as intended and the other is known as unintended. So let us look at this by taking the help of this particular example. Let us say while a scanning the libc file we found a sequence of byte values as follows. So depending on how we interpret these byte values we can get different instructions. So for example suppose we start interpreting these byte values starting from F7 then we can find a useful instruction F7, C7, 07, 00, 00, 00 which gets interpreted by the processor as test 7 edi, the remaining part 0f, 95, 45, c3 gets interpreted by to an other instruction setnzb.

However, we see that this is may be what the compiler had intended to do and therefore we call this as intended instructions. So note that these instructions atleast these two instructions do not form a gadget. Now if we just offset our analysis by 1 byte and state that instead of starting from F7 we start with C7 then we get a different sequence of instructions. Now let us say we start interpreting from C7 then the sequence of byte C7, 07, 00, 00, 00, 0F gets interpreted to this movl instruction, while the bytes 95, 45 and c3 gets interpreted as exchange increment and return instructions.

Thus we find that this is a valid gadget, although these instructions were not intended by the compiler, we are able to synthesize these instructions and hence this gadget by changing the offset from where we analyse the instructions. So in this way we are able to get a diverse set of different type of gadgets in fact for X86 platforms the number of gadgets that can be found is extremely high because of the large number of instructions that are supported.

On the other hand for RISC processor such as ARM of RISC5 (RISC-V) the number of gadgets that we find are much more lesser thus sys based processors such as the X86 are more prone to ROP attacks then RISC processor such as ARM or open RISC or RISC5 (RISC-V).

(Refer Slide Time: 15:56)



More about Gadgets

- Example Gadgets
  - Loading a constant into a register (edx ← deadbeef)

deadbeef
GadgetAdd
esp
stack
pop %edx
ret

- A previous return will pop the gadget address into %eip
- %esp will also be incremented to point to deadbeef (4 bytes on 32 bit platform)
- The pop %edx will pop deadbeef from the stack and increment %esp to point to the next 4 bytes on the stack

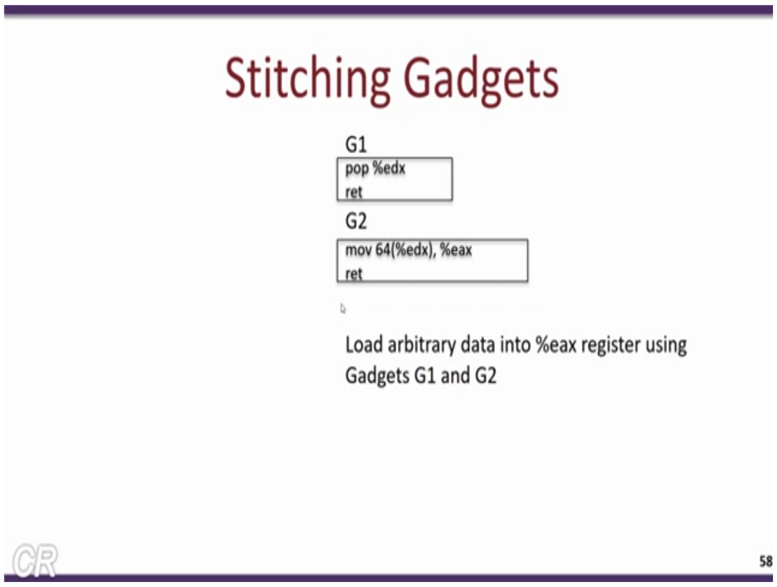
57

So let us take a few examples of gadgets, so let us start with the simple one where we want to (prove) move some data into the register edx. So in this particular example let us say that we want to move deadbeef into the register edx, the gadget which we will use for this purpose contains two instructions pop edx followed by a return. Now let us say that the stack pointer is pointing to a location in the stack which contains this particular gadget address.

Now when the function or the gadget being executed executes the return instruction as we have said there are two things that would happen first the contents of the stack pointed to by the stack pointer which in this case is gadget address, this contents would get moved into the program counter, at the same time the stack pointer would be incremented by 4 and would point to this location comprising of deadbeef.

Now since the program counter is pointing to this particular gadget we would have the pop instruction getting executed. Now the pop instruction would take the current contents of the stack pointer which is deadbeef and load these contents into the edx register simultaneously the stack pointer is incremented by 4 bytes. Therefore, at the time of return the stack pointer is pointing to 4 bytes above the deadbeef location.

(Refer Slide Time: 17:35)

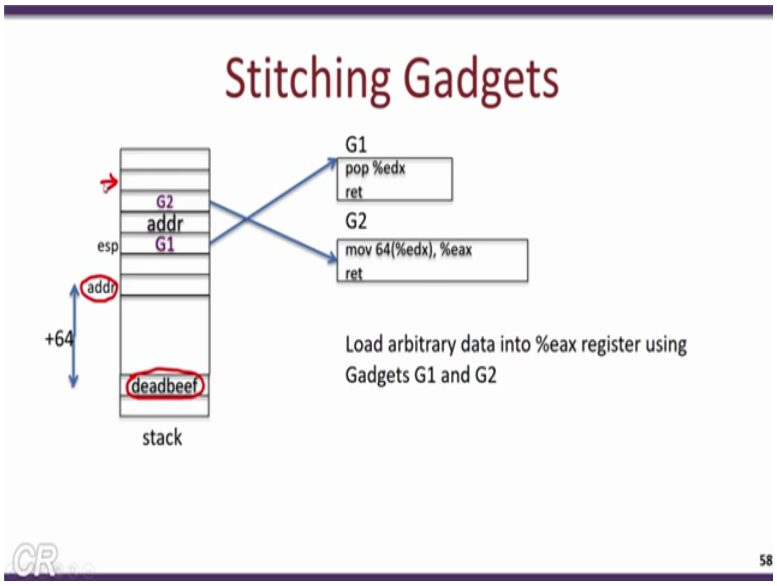


Stitching Gadgets

G1
```
pop %edx
ret
```
G2
```
mov 64(%edx), %eax
ret
```

Load arbitrary data into %eax register using Gadgets G1 and G2

58



Stitching Gadgets

G1
```
pop %edx
ret
```
G2
```
mov 64(%edx), %eax
ret
```

Load arbitrary data into %eax register using Gadgets G1 and G2

58

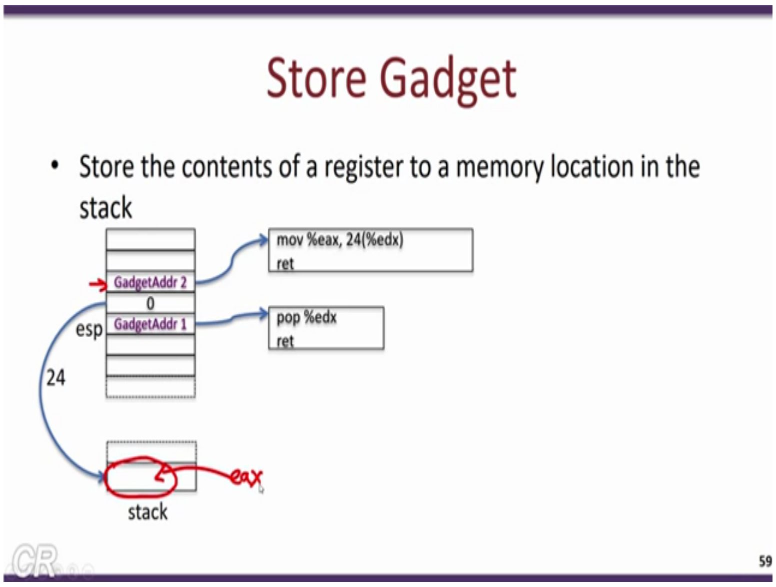
Now let us take another example where we want to load arbitrary data into the eax register using two gadgets G1 and G2. The gadget G1 comprises as before of the pop edx followed by return while the gadget G2 comprises of this instruction where the contents of the memory location pointed to by the contents of edx plus 64 bytes is moved into the eax register. So let us see how we can stitch these two gadgets together so that we can get arbitrary data into the eax register so the stack would look something like this.

Now let us take another example where we want to load some arbitrary data into the eax register. Now after parsing and searching through the entire libc file we found two gadgets which would help us achieve this, the first gadget G1 is what we have seen before and essentially has this instruction pop followed by a return, the second gadget has a mov instruction followed by a return the mov instruction moves the contents of edx register plus 64 bytes into the eax register.

In order to achieve this where we want to load arbitrary data into the eax register, we need to create our stacks which would look something like this way. So what we do is that at a particular address plus 64 bytes we put the necessary data which we want to move into the eax register, then we arrange gadgets in this particular form, the stack pointer is pointing to gadget 1 initially, then we have this address over here and then we have gadget 2.

So let us see how these two gadgets work together to achieve our task. So first when gadget 1 executes the stack pointer is incremented to point to this and the contents of edx register would be address, after pop gets executed the stack pointer is pointing to G2. When the gadget G2 executes, it takes the contents of edx register which essentially is address add 64 bytes to this and the contents of this location which is deadbeef is then loaded into the eax register, at the time of return the stack pointer is pointing as follows over here.

(Refer Slide Time: 20:15)



Store Gadget

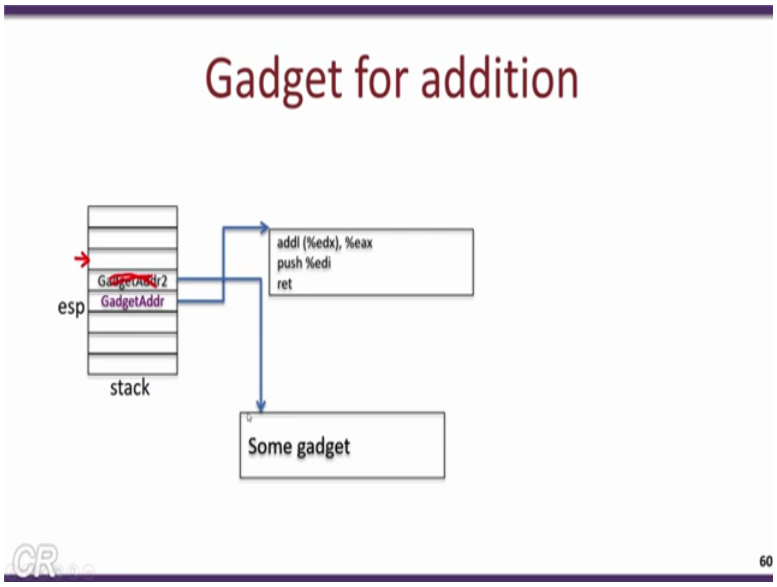
- Store the contents of a register to a memory location in the stack

Now let us take another example of storing data present in the eax register into some location on the stack, again after parsing through the entire libc we found two gadgets which will be useful for us. The first gadget is similar as what we have seen before comprising of the pop and return instruction, while the second gadget comprises of a mov instruction where the contents of the eax register is moved into the location edx plus 24.

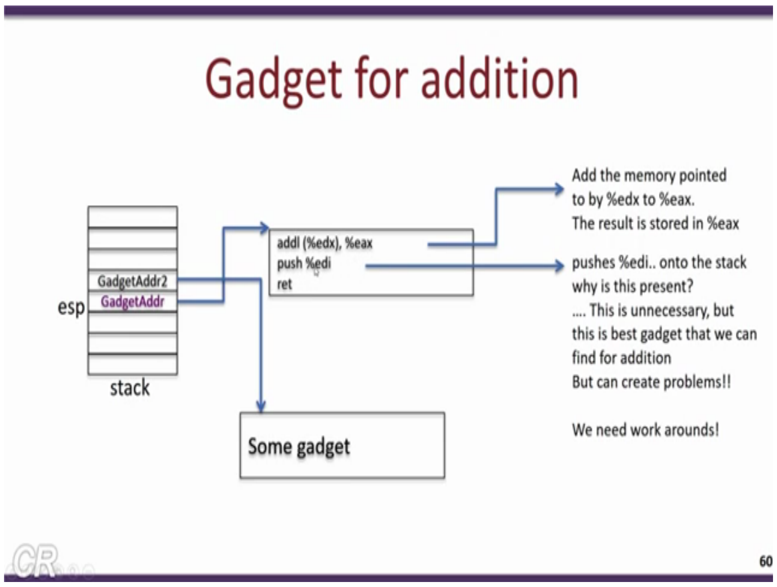
In order to achieve this particular store we arrange the stack as follows, so we have gadget address 1 followed by a 0, then gadget address 2. Now let us see how these two gadgets work together, so when the stack pointer is pointing to gadget address 1 it would result in this gadget getting executed and at this particular time the stack pointer is pointing to this location. Thus, edx is loaded with a value of 0 and also during the pop a stack pointer is incremented to point to gadget address 2.

Now when gadget 2 gets executed, the contents of edx which is 0 plus 24 bytes which happens to be over here when the gadget 2 gets executed the contents of the eax register is stored in the location pointed to by edx plus 24 bytes. Now edx has a value of 0 due to this gadget 1 getting executed therefore the contents of eax would be stored in this particular location over here as follows.

(Refer Slide Time: 22:05)



Gadget for addition

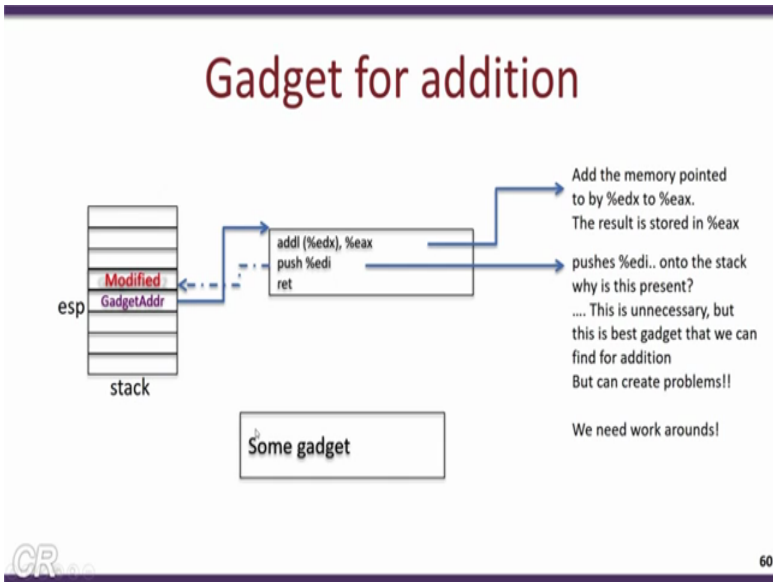


Gadget for addition

Add the memory pointed to by %edx to %eax.
The result is stored in %eax

pushes %edi.. onto the stack
why is this present?
.... This is unnecessary, but this is best gadget that we can find for addition
But can create problems!!

We need work arounds!



Gadget for addition

Add the memory pointed to by %edx to %eax.
The result is stored in %eax

pushes %edi.. onto the stack
why is this present?
.... This is unnecessary, but this is best gadget that we can find for addition
But can create problems!!

We need work arounds!

Now let us take another example of a gadget which does addition, after parsing the entire libc file the best gadget that we had found is as one showed over here. So what this a gadget does is what we want to do, so it takes the contents of the memory location pointed to by the edx register and adds that contents into the eax register. However, this gadget also has an additional push edi instruction also present, why does this push instruction present?

Essentially after parsing through the libc file this is the only gadget which we had and unfortunately for us it has complicated things for us because of this additional push instruction. Now this push instruction would make things more difficult for us and let us see how? Now suppose we have arrange our stack like this, we have gadget address which is pointing to this add thing and then we have a gadget address 2 which is pointing to some other gadget.

Now when this particular gadget executes we have the stack pointer pointing here as we want we have the contents of the memory location pointing to by the edx register to be added into the eax register. Unfortunately the push instruction now modifies the contents of this gadget address 2 in the stack because as we know when we have a push instruction it does two things it writes the contents of the register on the stack corresponding to the stack pointer and also increments the stack pointer by 4 bytes. So now we see that while the (add) addition gadget has worked properly the gadget 2 present in the stack will not execute because it has got corrupted.

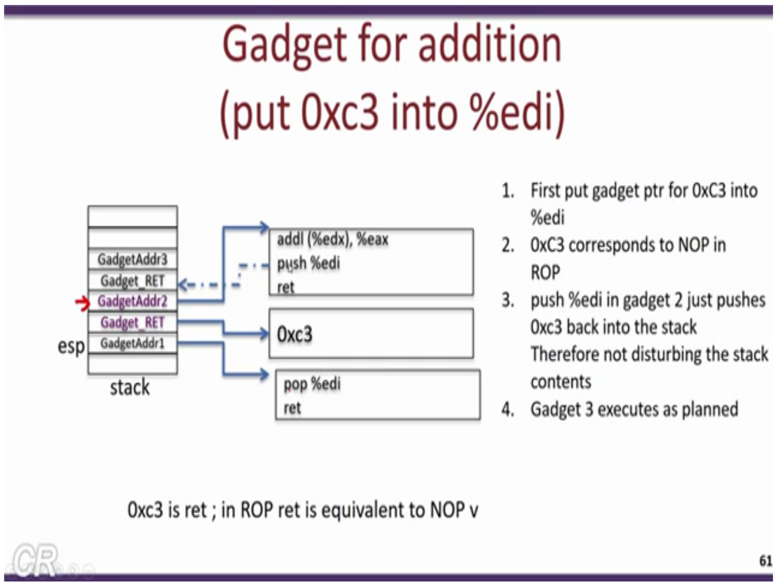
So let us take another example for a gadget for addition so after parsing through the entire libc library the only gadget that we have found or rather the best gadget that we have found which actually does a task looks something like this, it has an add instruction the contents of the memory pointed to by the edx register gets added to the eax register and the result is stored in the eax register.

Unfortunately for us this particular gadget also has another instruction which is the push edi register. So why is this instruction present? Of course this instruction is not what we want, unfortunately we do not find a better gadget to do our task and therefore we should live with this and try to eliminate the problems that it can cause. To understand let us first see what problems that push percentage edi can cause to our entire scheme.

So let us say we arrange our stack as follows, we have gadget address 1 over here, followed by gadget address 2 which points to some other gadget, when gadget 1 executes it does the addition as we want and stores the contents in the eax register, but the push edi is going to modify the contents of the stack. As we know the push instruction would store the contents of the edi registers into the stack pointed to by the stack pointer.

Therefore, our stack gets corrupted and we will not be able to execute the second gadget. Therefore, we need a way to actually work around this particular problem.

(Refer Slide Time: 25:38)



Gadget for addition (put 0xc3 into %edi)

1. First put gadget ptr for 0xC3 into %edi
2. 0xC3 corresponds to NOP in ROP
3. push %edi in gadget 2 just pushes 0xc3 back into the stack Therefore not disturbing the stack contents
4. Gadget 3 executes as planned

0xc3 is ret ; in ROP ret is equivalent to NOP v

61

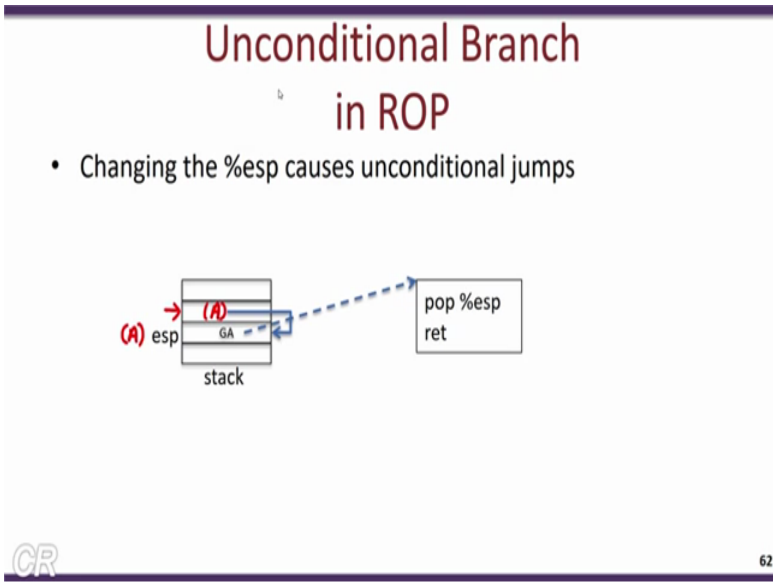
So one way to do this is by introducing another gadget known as the gadget return which essentially points to a gadget containing just 0xc3. So essentially as we know 0xc3 corresponds to the return instruction. Now in the ROP world (0) the return instruction is simpler to a no opt instruction, it does nothing but simply just increments the stack pointer. So let us see how this particular scheme works.

So we have the stack pointer pointing to gadget 1 initially and therefore we have this gadget which we have used which essentially pops the contents of the stack into the edi register. So the stack is at this particular time pointing to this location and therefore the address of this particular gadget return is loaded into the edi register and at the same time as we know the pop would increment the stack pointer to be pointing to this location.

Now at this location we have gadget address 2 which does the addition as we want and as the memory pointed to by edx to the eax register now the push instruction would push the contents of the edi into the stack. Now the contents of the edi what we have done due to the previous gadget contains this gadget return, this gadget return essentially is pointing to this gadget comprising of just a return, simply increments the stack pointer to point to gadget address 3.

So in this way we see that we have been able to work around our issue where there is this additional push present in the add gadget that we have found.

(Refer Slide Time: 27:26)



Unconditional Branch in ROP

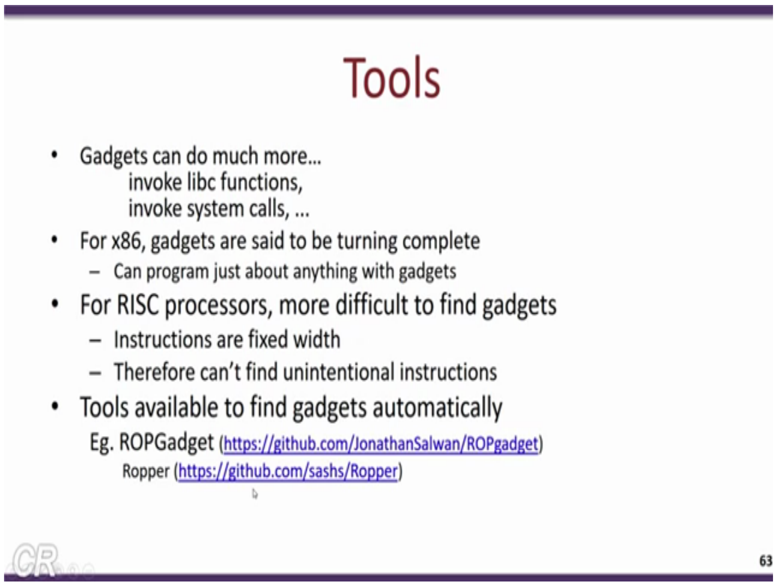
- Changing the %esp causes unconditional jumps

Now let us take another example where we demonstrate how unconditional branching can be done in ROP. So what do we mean by unconditional branching in ROP is that we want to change stack pointer from its sequential increments so that gadgets can be executed in some arbitrary order, we will demonstrate this by showing how the same gadget can be executed over and over again in a loop.

So for this we take this particular example of a gadget comprising of a pop instruction which is a pop to the stack pointer itself followed by a return. Now as we know when this particular gadget is executing we have the stack pointer present here, further we assume that we have this location specified as A and the same location A is present in this memory location. So what this pop instruction does is that it pops the contents of the location pointed to by the stack pointer in this case A and stores these particular contents in the stack pointer.

Thus, the stack pointer is then reset to this A location (reset to this A location) and re executes this particular gadget. So in this way we see that the stack pointer continuously keeps moving between these two locations in an infinite loop. So in this way we show how we can use unconditional branching to create loops in our ROP programs. In a similar way we could also have conditional branching as well implemented in ROP although the techniques are much more involved.

(Refer Slide Time: 29:08)



Tools

- Gadgets can do much more...
  invoke libc functions,
  invoke system calls, ...
- For x86, gadgets are said to be turning complete
  - Can program just about anything with gadgets
- For RISC processors, more difficult to find gadgets
  - Instructions are fixed width
  - Therefore can't find unintentional instructions
- Tools available to find gadgets automatically
  Eg. ROPGadget (https://github.com/JonathanSalwan/ROPgadget)
  Ropper (https://github.com/sashs/Ropper)

63

So these were some of the examples of gadgets that we have created, in reality there are a lot of tools available on the internet which you could use to build ROP gadgets. So some quite famous tools which we have used is this ROP gadget and Ropper so both are available at these locations. So we have personally verified a check this ROP gadget by Jonathan Salwan and it works on any object file provided is an ELF object, it would output all the gadgets that can be created from that particular library.

So these were some of the examples of different gadgets that we have created, in reality ROP programs can be quite complex you can achieve quite a few different program functionality in fact for X86 systems the ROP programs are said to be turning complete and can implement just about any program. In RISC type of processors like ARM implementing ROP based programs is much more difficult.

On the internet there are several tools which would help you in building these ROP programs. For example the tool ROP gadget and Ropper present in these websites can be used to analyse object files and print all the ROP gadgets present in these object files. So in the github repo for this particular course we would be adding some ROP examples and demonstrate how ROP gadgets can be built. For example to write small programs such as like factorizing a number of finding the factorial of a number, thank you.