Information Security-5-Secure System Engineering

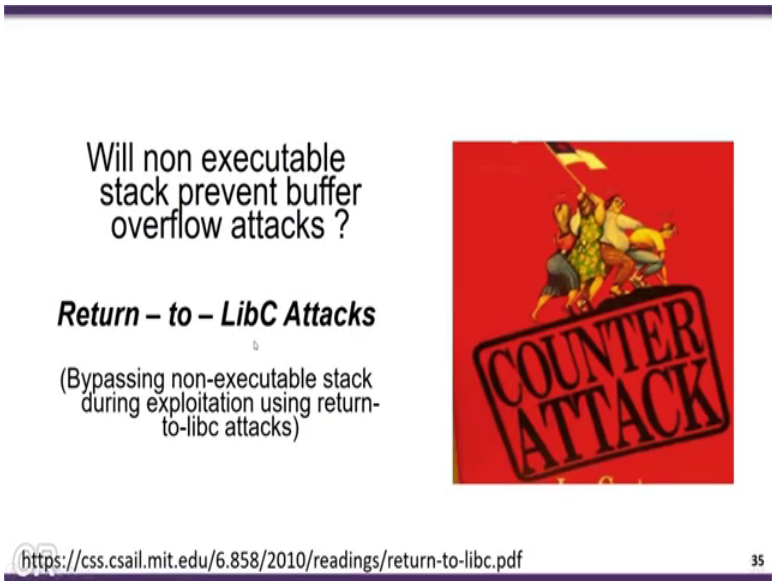
Professor Chester Rebeiro
Indian Institute of Technology, Madras
W2_2
Return to LibC Attacks

Hello and welcome to this lecture in the course for Secure System Engineering, in the previous two lectures what we have seen was how an attacker could inject code in the stack of another process for exploiting a buffer overflow vulnerability and then force that particular code to execute. Later on we see two different mechanisms for this particular vulnerability. The first was using canaries where buffer overflows were detected, the second is using something known as the NX bit which is supported by the processors.

As we have seen in the previous lecture the NX bit would disable executing from that stack. However security has always been a cat and mouse game the attackers would find a way to create an exploit and then the defenders would then find a way to prevent that particular exploit from running on their system. Now the attackers again would find a way to bypass this defense mechanisms and still get their attack to run. So in this way there is a constant cycle between the attackers and defenders and as a result the attacks are getting more and more powerful and thereby better defend strategies are required to prevent these attacks.

In this particular lecture what we will look at is a new form of attack known as the Return to LibC Attack, so this particular attack also exploits a buffer overflow vulnerability in the stack however unlike the previous exploit that we have seen where code is executed from the stack in this particular attack we do not execute any code form the stack and in this way the attack would run even with the NX bit defense that is present for that particular stack. So let us see how this attack proceeds.

(Refer Slide Time: 02:28)


COUNTER
ATTACK

Will non executable stack prevent buffer overflow attacks ?

Return – to – LibC Attacks

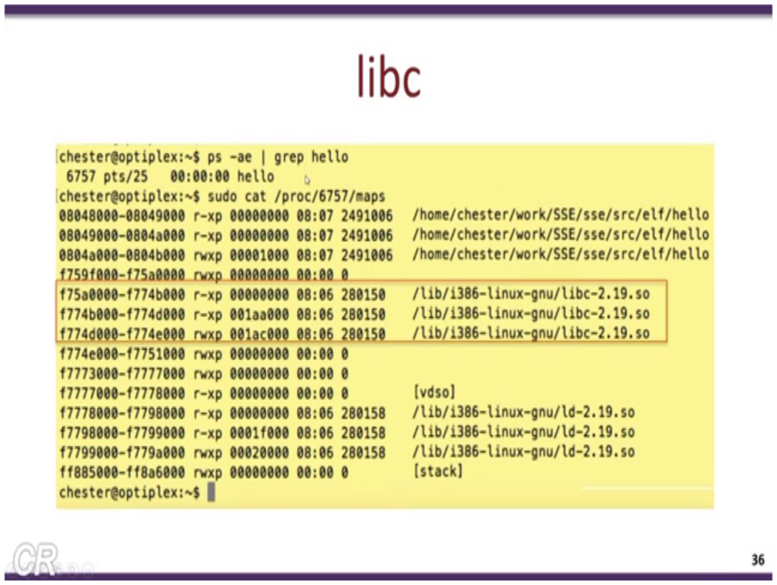
(Bypassing non-executable stack during exploitation using return-to-libc attacks)

https://css.csail.mit.edu/6.858/2010/readings/return-to-libc.pdf

35

Let us start of by understanding what LibC is, so LibC is a dynamically linked library that gets attached to almost every C program that is written. Now the LibC contains implementations of all the standard rather a large number of the standard C function cause that we are typically used to for example string copy, print f, scan f, f open, f close string cat and so on are all present in LibC so there are easily over thousand functions that are implemented in LibC.

(Refer Slide Time: 03:08)



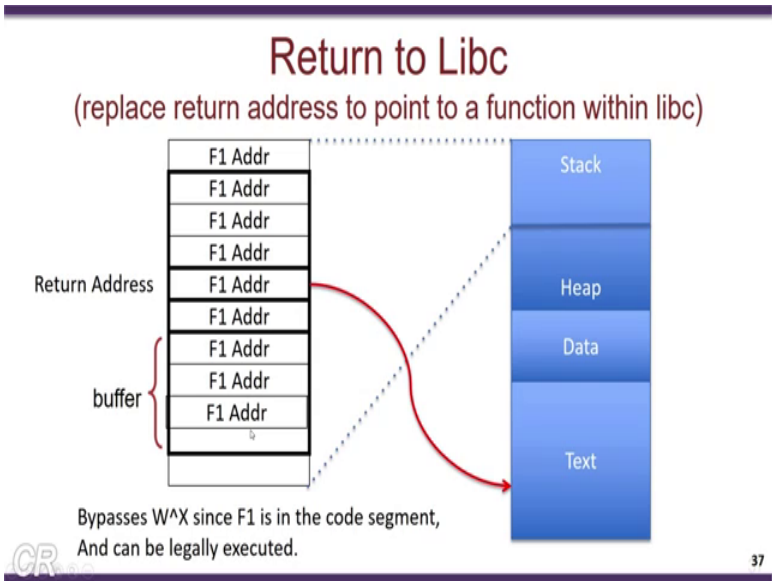
libc

```
chester@optiplex:~$ ps -ae | grep hello
 6757 pts/25   00:00:00 hello
chester@optiplex:~$ sudo cat /proc/6757/maps
08048000-08049000 r-xp 00000000 08:07 2491006   /home/chester/work/SSE/sse/src/elf/hello
08049000-0804a000 r-xp 00000000 08:07 2491006   /home/chester/work/SSE/sse/src/elf/hello
0804a000-0804b000 rwxp 00001000 08:07 2491006   /home/chester/work/SSE/sse/src/elf/hello
f759f000-f75a0000 rwxp 00000000 00:00 0
f75a0000-f774b000 r-xp 00000000 08:06 280150    /lib/i386-linux-gnu/libc-2.19.so
f774b000-f774d000 r-xp 001aa000 08:06 280150    /lib/i386-linux-gnu/libc-2.19.so
f774d000-f774e000 rwxp 001ac000 08:06 280150    /lib/i386-linux-gnu/libc-2.19.so
f774e000-f7751000 rwxp 00000000 00:00 0
f7773000-f7777000 rwxp 00000000 00:00 0
f7777000-f7778000 r-xp 00000000 00:00 0         [vdso]
f7778000-f7798000 r-xp 00000000 08:06 280158    /lib/i386-linux-gnu/ld-2.19.so
f7798000-f7799000 r-xp 0001f000 08:06 280158    /lib/i386-linux-gnu/ld-2.19.so
f7799000-f779a000 rwxp 00020000 08:06 280158    /lib/i386-linux-gnu/ld-2.19.so
ff885000-ff8a6000 rwxp 00000000 00:00 0         [stack]
chester@optiplex:~$
```

36

Even the simple hello world program that we have written a few lectures back which essentially just prints hello world by invoking the print f statement. Even this program has the LibC present in its virtual address space, example if you look at this particular slide which displays the virtual memory map for the hello world program we see that there are three entries corresponding to LibC. So these three entries correspond to the entire LibC. Now LibC has as I mentioned over a thousand different functions and all of this functions are present in this virtual address map and can be access by any of this addresses that are present over here. So given that we know that what LibC is let us see how a return to LibC attack actually works.

(Refer Slide Time: 04:06)



Return to Libc

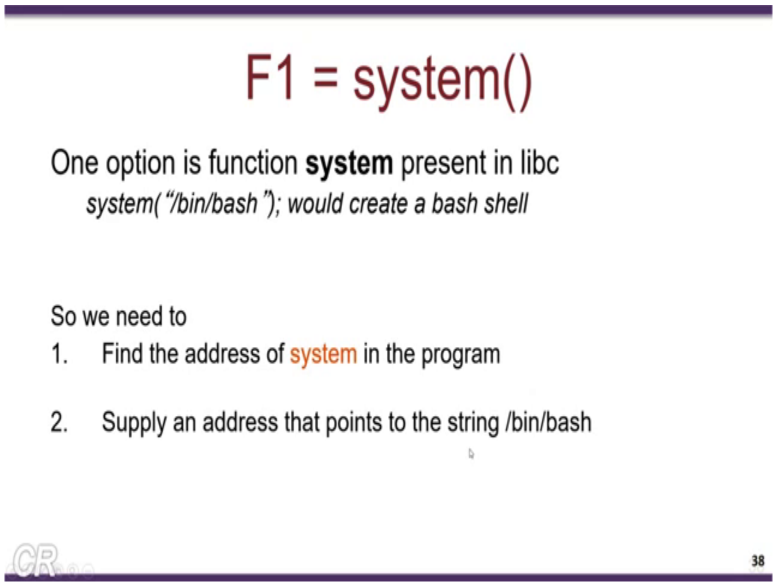
(replace return address to point to a function within libc)

Bypasses W^X since F1 is in the code segment,
And can be legally executed.

So just like the previous attack where the attacker overflowed a buffer and made the return address point to this starting location of that particular buffer on the stack. In a similar way the return to LibC attack would overflow the buffer and then replace the return address with some other arbitrary address however this arbitrary address is now pointing to some particular function in a LibC. So as we know LibC is part of the whole segment of the particular process and therefore it can execute.

In other words, the NX bit is not set for the functions in LibC. So what happens over here is assuming that this function F1 is a valid function in LibC we are overflowing the buffer until the point where the return address rather the valid return address which is present on the stack is replaced with the address for F1 which in fact is present in the code segment of the particular process. So in this way we are able to subvert execution and in spite of the NX bit set we are able to execute some arbitrary function present in the LibC. So the next question is what should be this function F1?

(Refer Slide Time: 05:39)



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 program
2. Supply an address that points to the string /bin/bash

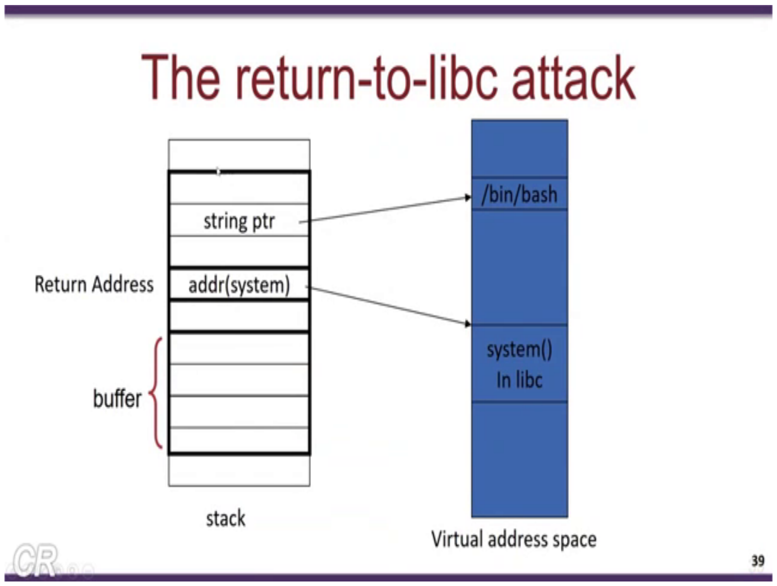
38

So there are multiple options that are possible so one option that we will actually look at today is known as the system function. So one function that we will look at for F1 is the function system. Now system is a function which is present in LibC so it takes string as input rather it takes a pointer to a string as input and this string contains an executable for example over here we have system and we are passing the string slash bin slash bash so the result of this particular function called would be that a bash shell gets created.

Now let us say that we want to build a payload which as before creates a shell this would mean that we would require to subvert execution by a function like this like system and pass the slash bin slash bash as the parameter to system. So this mean we require two things we need to specify the address for system as the F1 address and overflow the buffer using that particular address of system. Secondly somehow we should be able to pass a pointer to this particular string slash bin slash bash so that system executes and it is able to obtain an argument which is slash bin slash bash.

So to summarize what we need to mount a return to LibC attack is as follows, we need to find the address of the system function in your LibC second we need to overflow the a buffer with this particular address so that the written address located on the stack is replace by the address of system. Secondly we need to pass the argument to system which essentially is a pointer to the string slash bin slash bash. So let us see how this can take place.

(Refer Slide Time: 07:43)

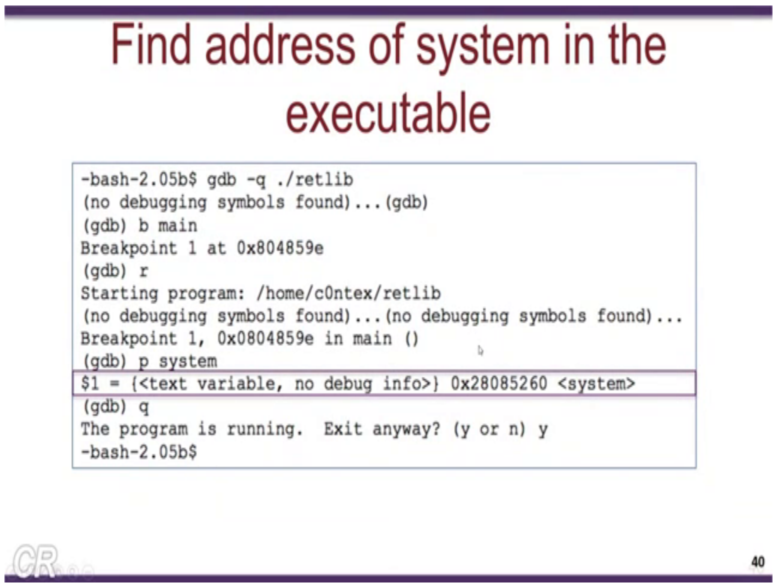


The return-to-libc attack

So what we need essentially is a the stack layout which looks something like this. At the location where the return address is stored we store the address of the system. Secondly at an offset of 4 bytes higher in the stack we have a character pointer which points to some location in the virtual address space by the string slash bin slash bash is present. Now what happens here is that when the function completes it execution and returns the written address is taken from the stack in this case the return address is the system, execution is into the system function present in LibC and the argument for this particular function is this character pointer pointing to slash bin slash bash.

So this function system in LibC would essentially execute the executable slash bin slash bash. So the next thing that we need to do determine is the address of system and what exactly is this particular pointer. So let us start with how we find out that in the entire virtual address space is this particular function system present.

(Refer Slide Time: 09:03)



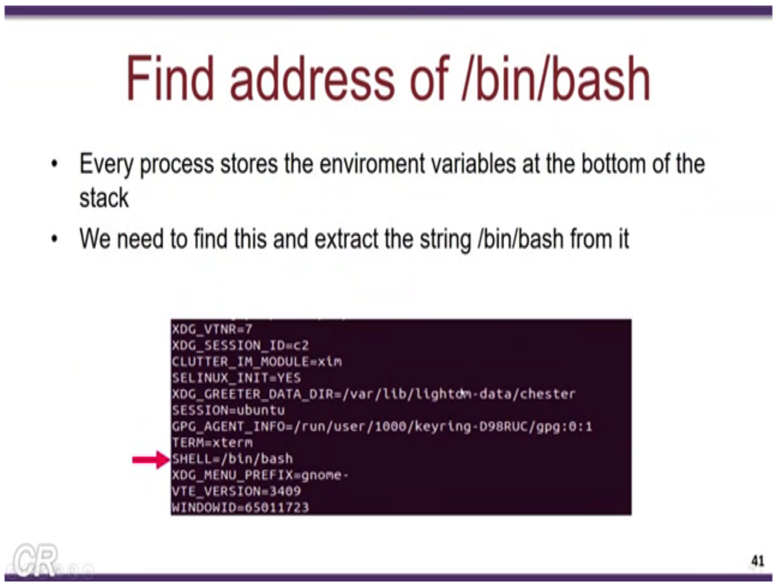
Find address of system in the executable

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x28085260 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

40

So this we can do as follows we can use something like GDB or any other tool like OBJDUMP so on and run this GDB with this option and the executable and we set a break point at main, run the program and when the break point is hit we say P system which means print system and as we see over here we get this value of this particular thing which is 28085260 which essentially is the address of the location where system function is present.

(Refer Slide Time: 09:42)



Find address of /bin/bash

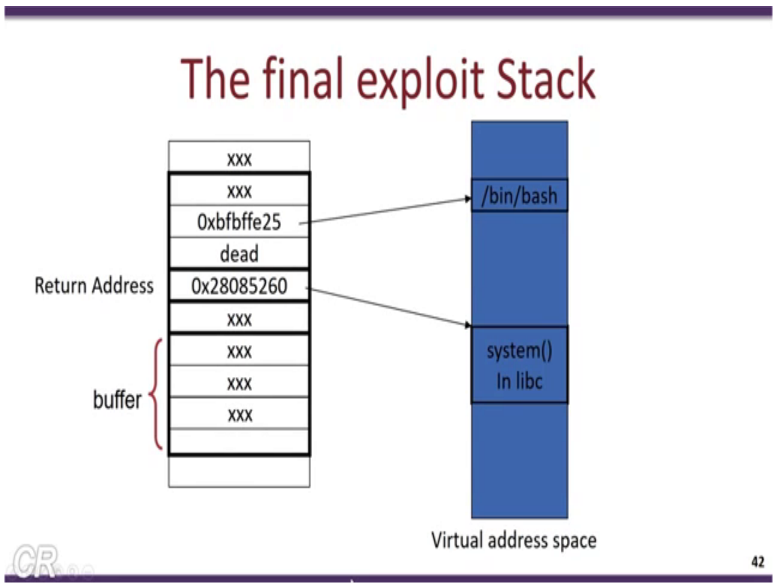
- Every process stores the enviroment variables at the bottom of the stack
- We need to find this and extract the string /bin/bash from it

```
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
SELINUX_INIT=YES
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/chester
SESSION=ubuntu
GPG_AGENT_INFO=/run/user/1000/keyring-D98RUC/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_MENU_PREFIX=gnome-
VTE_VERSION=3409
WINDOWID=65011723
```

41

In a similar way we can find somewhere in the entire executable where the string slash bin slash bash is present. So in this particular example over here we have considered the environment variables that is present in the process. So as we see over here there is one particular line in this environment variables which actually has the string slash bin slash bash. Now what we need to find out is by using a something like GDB we determine the exact address where slash bash slash bash is present. Now that we have identified where the address of system is present and also we have found a string in the executable which contains slash bin slash bash and we found the address of that particular string, we are ready to build our exploit.

(Refer Slide Time: 10:34)

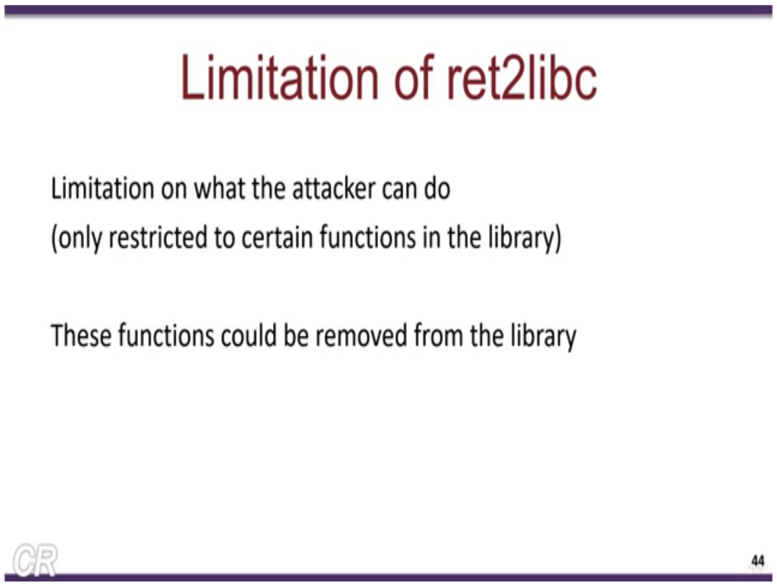


So what we do is as follows, we overflow the buffer and at the location where the return address was present we replace it with this particular address 0x28085260 which essentially is the address for the system function present in LibC and at an offset of 4 bytes on the stack we have the address bff (bffe25) which essentially is the pointer to the slash bin slash bash string but we have found out in the environment. So when this particular program runs we would have the bash slash bin slash bash getting executed.

Now in order that we terminate this particular program properly what we can additionally do is add a function exit over here which essentially has the address 281130d0, so therefore what happens now is the system function executes it takes slash bin slash bash pointer as argument and after that function completes this particular function which points to the exit function would get executed. So in this way we are able to subvert execution to the system function present in

LibC we are able to run a payload which in this case is the slash bin slash bash shell and then also exit from this particular program.

(Refer Slide Time: 12:16)



Limitation of ret2libc

Limitation on what the attacker can do
(only restricted to certain functions in the library)

These functions could be removed from the library

One major advantage of the return to LibC attack is that it can work with a non-executable stack, however there is a major limitation of the return to LibC attack. The limitation is that the attacker is constraint by the functions that the LibC offers. For example if the LibC does not have a system function then the attack which we have discussed in this particular lecture will not function thus there is a limitation to what the attacker can do. In the next lecture we will look at another attack known as the Return Oriented Programming or the ROP attack where the attacker is not restricted to the functions that LibC supports but rather can create any arbitrary (play) payloads, thank you.