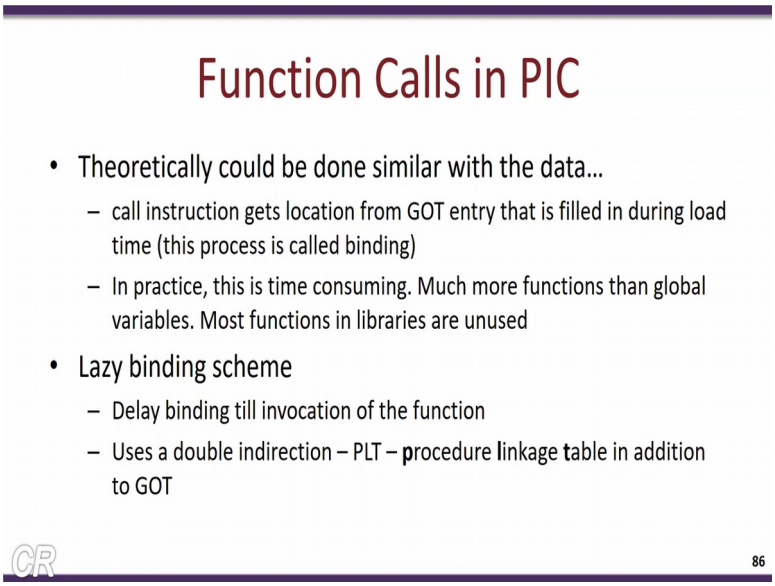
**Information Security 5 Secure System Engineering**
**Professor Chester Rebeiro**
**Indian Institute of Technology Madras**
**Address Space Layout Randomisation (ASLR) (Part 2)**
**Mod03_Lec16**

Hello and welcome to this lecture in the course secure system engineering, in the previous lecture we had actually looked at ASLR and we seen how ASLR is actually dependent on relocatable libraries and we also looked at two ways to achieve relocatable data, so we looked at load time relocatable and the use of GOT tables to achieve load time relocation of global data, in this particular lecture we will look at functions, essentially if we have functions present in the library how do we make these functions relocatable.

(Refer Slide Time: 0:57)




Functions can be made relocatable in a very similar way as how data was made relocatable, so for example every time there is call to a function, we could get the actual address for that particular function from the GOT table with functions we can do precisely what we have done with data, we could use a GOT table and store the actual addresses for the functions in this GOT table, wherever there is a call to a particular function we look up the GOT table obtained the actual address for that particular function and then make a branch to that particular location.

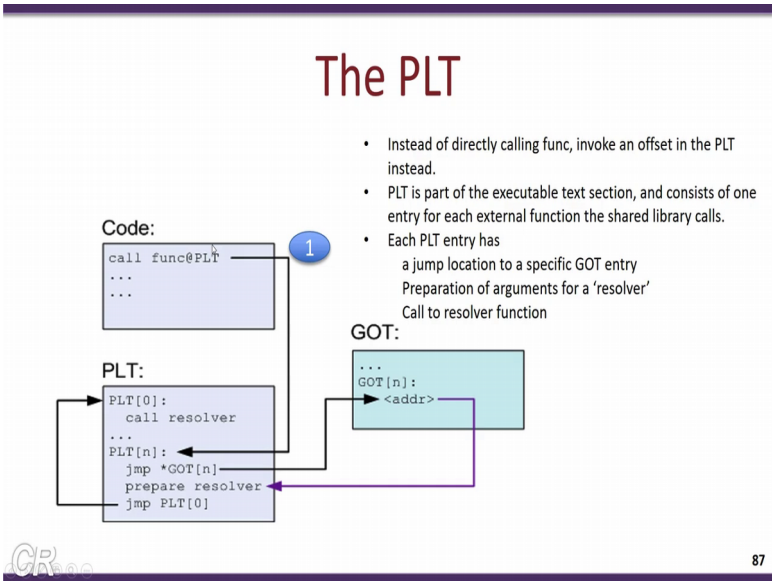
This way we can achieve relocatable functions that is only at a time of loading the particular library into a process address space only then the GOT entry for that function will be filled in, therefore only then will the actual addresses of the function be known however this is not

what is done in practice, in practice having a GOT table just like how we have done with data is quite time consuming, the reason being that there are far more number of functions in a library then the number of global data.

Furthermore most of the functions in the library are unused, in a specific library for example Lipsey out of the hundreds of functions that are present you may at most use 2 or 3 functions, so it does not make sense that at loading time we try to resolve all of these hundreds of functions, therefore what is done in practice is a scheme call lazy binding unless a function is actually used only then will the address of that function will resolved.

So in other words lazy binding essentially delays binding for a function until the function is invoked and in order to achieve this we use a double indirection technique by making use of another table known as the PLT table or procedure linkage table, so this table is used in addition with a GOT table which is already present.

(Refer Slide Time: 3:17)

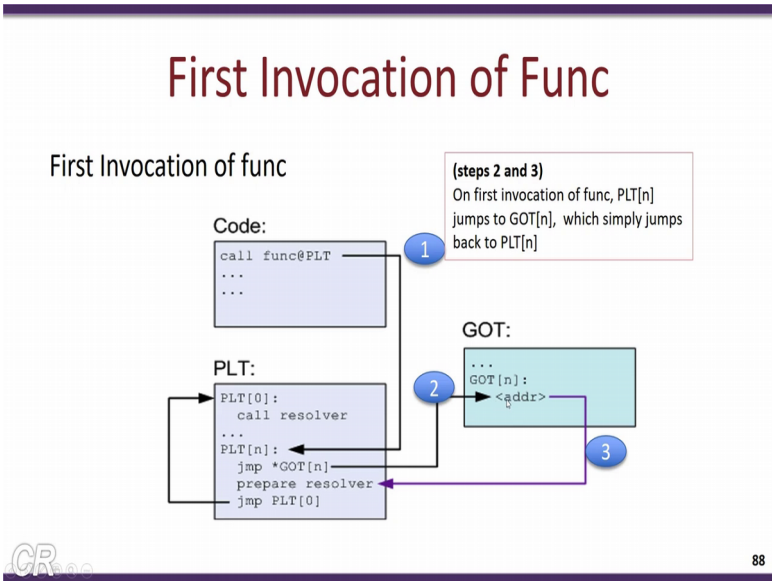



So let us see how function calls are resolved in practice, so let us say we have a function present in a library and let us take for example say print F and what happens is that when you compile your code is so the actual call to print F is replaced with a call to a function call print F at PLT, so it would looks something like this where the actual call to function punk is replaced with a call to func at PLT.

Now PLT is a table which looks something like this is function present in the library has an entry in the PLT, so for an example func at PLT has an entry in the PLT table which is this, the entry for a function in the PLT comprises of these three statements, first there is an

indirect jump then there is a prepare resolver and finally the jump to PLT 0, so let us say how this function at PLT actually works.

(Refer Slide Time: 4:18)

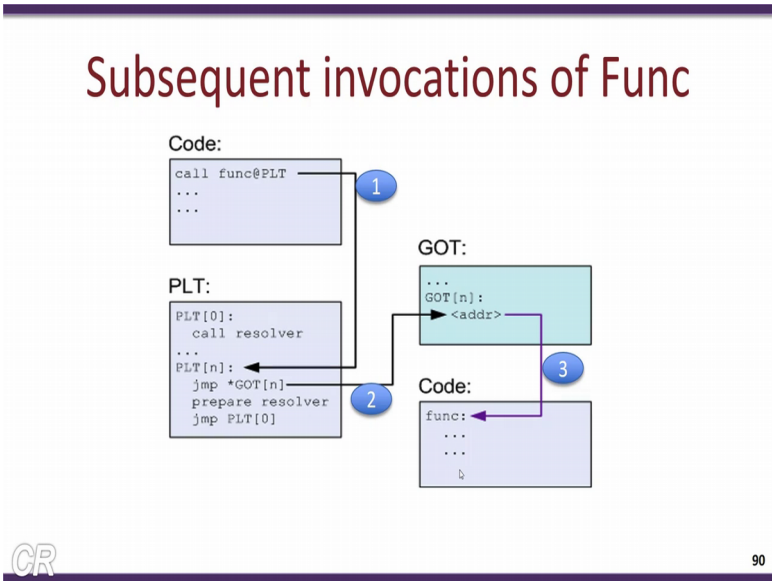



The first invocation of func works as follows, so as I have mentioned the compiler would replace the call to func with the call to func at PLT, so this would mean that these instructions would get executed, the first function is an indirect jumped based on an address in the GOT, so this particular GOT entry corresponds to the func, initially after loading is address corresponds to the second instruction in the PLT which is the prepare resolver, so essentially what is going to happen here is when you make an indirect jump, so you jump to a location specified this particular address, so in another words initially the jump is just to the next line.

So therefore you jump to the next line, run this insertion call prepare resolver and then make a jump to PLT 0 which calls the resolver, so what the resolver does is that it determines the actual address for this function func and fills that address in the GOT entry, thus at the end of the call to the resolver, the entry in the GOT contains the actual address of func, after the call to the resolver the actual functions get invoked.

Thus we see that the first invocation of the func invokes func at PLT which in turn just makes a dummy branch to this, prepares the resolver and calls the resolver, the resolver identifies the actual address for the func function, fills that address in the GOT entry over here and then invokes the function.

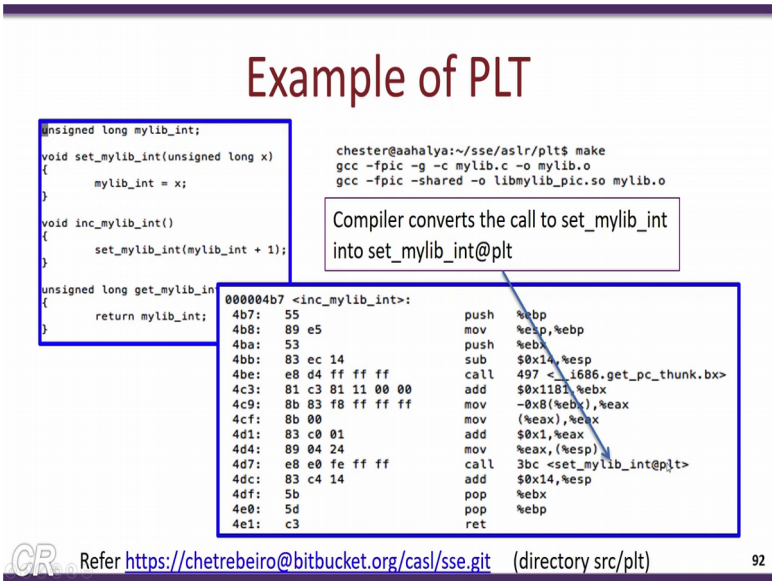
(Refer Slide Time: 6:06)




So now let us look at what happens on subsequent invocations to func, so let us say we are making the 2nd, 3rd, 4th or 5th invocation to func and as we know the compiler has already replace the direct call to func to a call to func at PLT, so therefore the execution was supposed to come into this PLT, the first instruction over here which is the indirect branch based on the GOT entry, so now what done over here is that you to the first execution we have change the contents of address to point to the actual address of func.

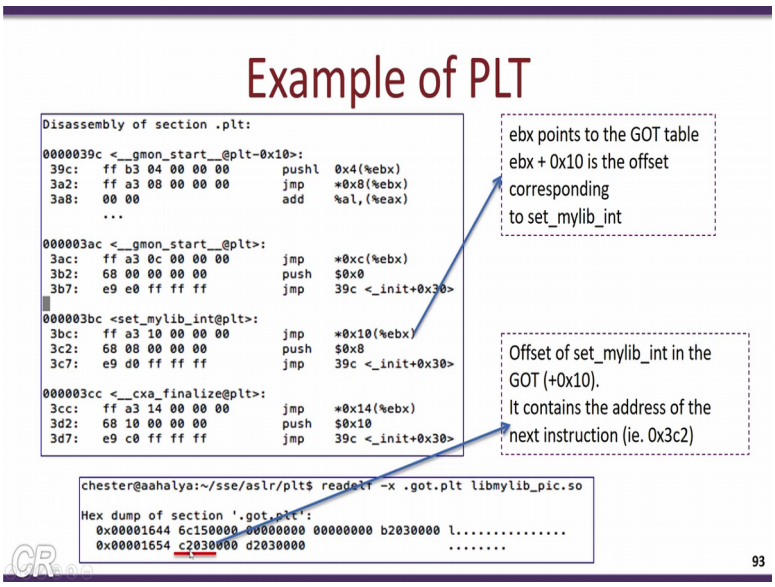
Therefore, the result of this jump instruction is that it is going to take the address present in the GOT entry and jump to that address, therefore the actual function would then get invoked, in this way for all subsequent invocations the resolver not invoked but rather the jump would directly go into this particular code, thus we see for the first invocation of jump there is considerable amounts of overhead because the resolver gets invoked which has to resolve the actual address function and fill in the GOT table with that particular address, all subsequent invocations of func would just have an additional jump is required to jump to the correct address of func.

(Refer Slide Time: 7:41)




So let us take an example of PLT, let us start with this particular library that we have written, so this library has a three functions, setmylib_int, increment mylib_int and getmylib_int, so we compile this particular library using the F, minus F pic file, a flag and thus create this library libmylib_pic.co, so when we do the object jump for this particular function say increment mylib_int, what you see the call to setmylib_int is replaced with a call to setmylib_int at PLT, so note that the compiler has automatically change a call, a function invocation to this, to a function, the function invocation at PLT.

(Refer Slide Time: 8:39)



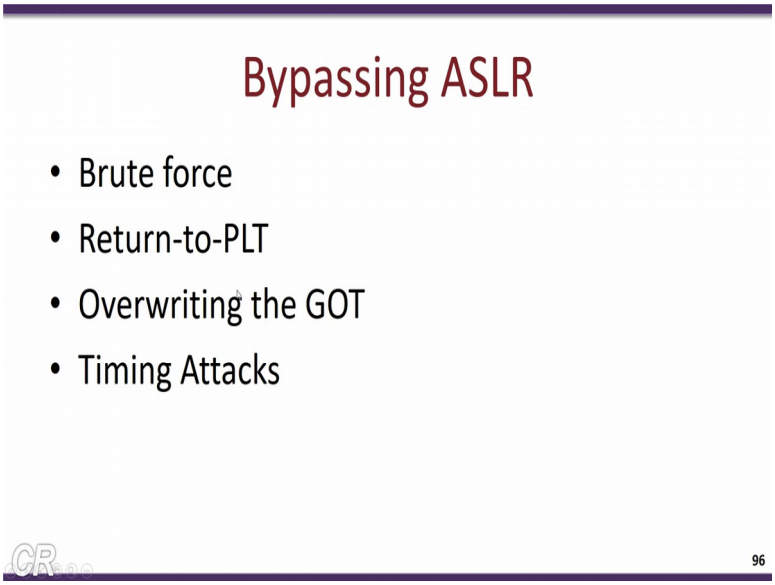

So let us dig a bit more deeper into the contents of setmylib_int, so if you do a disassembly of this we see that setmylib_int present at the location 3BC that is going to be in the PLT, so it

would have these three instructions, so it has an indirect jump as we said, so the indirect jump is based on an address at an offset of 16 bytes in the GOT table, so this particular offset corresponds to a function setmylib_int, so then there is a push to create the arguments for the resolver and then there is a jump to the resolver.

So note that, we can also look at the contents of the GOT table at the time of compilation, so we can do this by using the command readelf – x.got.plt libmylib_pic.so, so the output of this particular command would actually be the PLT table, the got.plttable the output of this command is the GOT table for this particular function, note that the contents at an offset of 16 bytes is c203 which in little Indian notations stands for 0x3c2, so this is initially points to the next instruction in the PLT table.

So thus what is going to happen is that the indirect jump is going to look into the GOT table at an offset of 16 bytes and jump to the location specified at this offset which in this case is 3c2, thus in the first invocation of setmylib_int we are going to jump to the next instruction over here which is push 0x8 and then we are going to execute this instruction which is the call to the resolver, so the resolver is going to execute and it is going to change this particular address to the correct address of setmylib_int, now all subsequent invocations of setmylib_int would come here and directly jump to the correct address of setmylib_int which is specified in the GOT. PLT table.

(Refer Slide Time: 11:00)




So thus we have seen how ASLR works, so ASLR requires modifications to the kernel, to ensure that libraries are loaded at random locations, further on it requires relocatable libraries

which are located, which are made relocatable either at load time or by using PIC technique, both data as well as functions are made relocatable this way, so in the recent years attackers have been able to bypass ASLR as well, so in spite of ASLR being present in the system, attackers have been able to bypass the ASLR and create, exploits for vulnerabilities present in the system and therefore the attackers have been able to run payloads in spite of the ASLR enabled in the systems.

There are various techniques by which ASLR can be bypassed, four of them are actually shown over here, one way of bypassing ASLR is if the attacker determines either by brute force or by special attacks known as timing attacks, where the attacker finds out where in the entire virtual address space of the process is the library actually loaded, now if the attacker finds out where the library is loaded then it the attacker could adjust the gadgets and the offsets within this particular library and still be able to run the exploit, other attacks have also been created known as the return to PLT and also another one known as overwriting the GOT.

So all of these attacks are quite recent, so we will not go into details about them in this particular course but for those of you who are interested there are a lot of online resources about how to create such attacks, thank you.