Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Address Space Layout Randomisation (ASLR)
Mod03_Lec15

Hello and welcome to this lecture in the course for secure systems engineering, so in the last two lectures we had actually looked at how one particular vulnerability in the stack can be exploited by the attacker, in the earlier lecture we had lecture we had looked at return to Lipsey attack where the attacker overflows a buffer present in the stack and replaces the return address with one specific function in the lipsey library, then the last lecture we had looked at a more advanced form of attack where the attacker is not restricted to functions in the lipsey but could create a payload which executes almost any arbitrary instructions and the way the attacker does this is by a concept of ROP programs or return oriented program.

So these attacks are some of the most advanced attacks on programs, in this particular lecture we will be looking at a concept known as ASLR or address space layout randomisation in order to prevent such ROP and return to lipsey attacks, so in order to understand the impact of ASLR, we would look at these attacks from the attackers prospective.

(Refer Slide Time: 1:41)

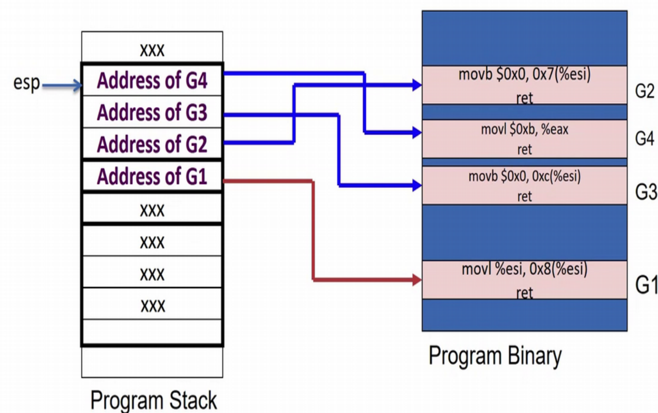
The Attacker's Plan

- Find the bug in the source code (for eg. Kernel) that can be exploited
  - Eyeballing
  - Noticing something in the patches
  - Following CVE
- Use that bug to insert malicious code to perform something nefarious
  - Such as getting root privileges in the kernel

Attacker depends upon knowing where these functions reside in memory. Assumes that many systems use the same address mapping. Therefore one exploit may spread easily.

65

ROP Attack



66

Address Space Randomization

- Address space layout randomization (ASLR) randomizes the address space layout of the process
- Each execution would have a different memory map, thus making it difficult for the attacker to run exploits
- Initiated by Linux PaX project in 2001
- Now a default in many operating systems

Memory layout across boots for a Windows box

67

So the attackers plan is as follows, the attacker let us say wants to create an exploit for a particular application, let us say the application here would be the operating system and say for example the application here for example could be the operating system kernel, so the first thing as we know the attacker should be doing is to find a vulnerability in the operating system kernel, the way he do to this is by looking at the source code or by noticing something in the patches that get applied to the operating system kernel and find out that there is a vulnerability in one of these patches or the patches actually fix and a specific vulnerability or the third way is by following the CVE.

So once he has found a particular vulnerability the source code, the next thing is to use this vulnerability to subvert the execution, the next part is to use this vulnerability to inject malicious code into that particular application, example is the application happens to be the

operating system kernel the malicious code for instance could obtain a shell which has root privileges.

Now as we know if the attacker obtains a shell with root privileges then he gets complete control of the entire system, now from the last three attacks we have studied in this course what we do understand is that the payloads that the attacker writes is highly dependent on the addresses for example in the ROP attack, the attacker would need to know the exact addresses where these gadgets are present in the lipsey library.

So for example over here in this case the attacker would need to know where the address of gadgets1, gadget2, gadget3 and gadget4 are present in the library, this assumption, the attacker assumption is that if he is able to find these gadget in a particular system, so the attackers assumption is that if the attackers assumption is that, if he is able to find these gadgets in a particular system then that attack would be successful.

Now the address space layout randomisation or the ASLR works so as to make it difficult for the attacker to find such gadgets, what the ASLR achieves is that it randomises the address space of an application, thereby making it difficult for the attacker to get specific addresses, so for example in this figure as we see over here which shows the memory layout for three different instances of the same application, so what we see over here is that the memory layout changes every time you run the application.

Let us take for example one particular library over here which is known as the ADVAP123, so this is present at a particular location during the first part of the application, during the second run the ADVAP123 is present at another location and in the third run, third location and so on, so since this location are changing in every run of the application, it would be difficult for the attacker to identify the exact location where the gadgets are going to be present, thereby preventing the ROP attacks from executing.

So the ASLR is not a new concept, it was initiated by the Linux PaX project in 2001 and since 2012 or 2013 it is becoming quite common and by added by default in the operating system, Windows and Linux operating systems now support ASLR by default.

(Refer Slide Time: 5:58)

ASLR in the Linux Kernel

- Locations of the base, libraries, heap, and stack can be randomized in a process' address space
- Built into the Linux kernel and controlled by /proc/sys/kernel/randomize_va_space
- randomize_va_space can take 3 values
  0 : disable ASLR
  1 : positions of stack, VDSO, shared memory regions are randomized
  the data segment is immediately after the executable code
  2 : (default setting) setting 1 as well as the data segment location is randomized

68

In the Linux operating systems ASLR would randomize the locations of libraries, the location of the heap, the stack and so on with each run of the application, so if you are current Linux system especially in Ubuntu systems, you can look at this particular file, you can cracked this file/proc/sys/kernel/randomize_va_space to determine whether your operating system has ASLR enable or not.

So this particular file would have 3 values 0, 1 or 2, 0 means that ASLR is disable in that particular system, while 2 has the highest level of randomization, where 2 is the highest level of randomization with a value of 1 what is achieved is that the position of the stack is randomized, similarly the positions of the VDSO, shared memory regions and so on are randomized with the value of 2 what it means is that it achieved all the randomization that is achieved with 1 as well as the data segment location are also randomized, so this 2 now is the default setting in most Linux systems.

(Refer Slide Time: 7:22)

ASLR in Action

```
chester@aahalya:~/tmp$ cat /proc/14621/maps
08048000-08049000 r-xp 00000000 00:15 81660111   /home/chester/tmp/a.out
08049000-0804a000 rw-p 00000000 00:15 81660111   /home/chester/tmp/a.out
b75da000-b75db000 rw-p 00000000 00:00 0
b75db000-b771b000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771b000-b771c000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771c000-b771e000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771e000-b771f000 rw-p 00142000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771f000-b7722000 rw-p 00000000 00:00 0
b7734000-b7736000 rw-p 00000000 00:00 0
b7736000-b7737000 r-xp 00000000 00:00 0          [vdso]
b7737000-b7752000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b7752000-b7753000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b7753000-b7754000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bf9aa000-bf9bf000 rw-p 00000000 00:00 0          [stack]
```

First Run

```
chester@aahalya:~/tmp$ cat /proc/14639/maps
08048000-08049000 r-xp 00000000 00:15 81660111   /home/chester/tmp/a.out
08049000-0804a000 rw-p 00000000 00:15 81660111   /home/chester/tmp/a.out
b75dd000-b75de000 rw-p 00000000 00:00 0
b75de000-b771e000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771e000-b771f000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771f000-b7721000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b7721000-b7722000 rw-p 00142000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b7722000-b7725000 rw-p 00000000 00:00 0
b7737000-b7739000 rw-p 00000000 00:00 0
b7739000-b773a000 r-xp 00000000 00:00 0          [vdso]
b773a000-b7755000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b7755000-b7756000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b7756000-b7757000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bfd2000-bfde7000 rw-p 00000000 00:00 0           [stack]
```

Another Run

69

ASLR in the Linux Kernel

- Permanent changes can be made by editing the /etc/sysctl.conf file

/etc/sysctl.conf, for example:
kernel.randomize_va_space = ***value***
sysctl -p

70

So let us look at ASLR in action, so what you can do is you could run a particular program find out that PID of that program and as we have done before looked at the memories space of that particular program by this particular command cat/proc the PID value of that particular process/maps and you would get the memory map, now if you run that same program again obviously you would get a different PID and if you look at the maps for that new process you would see that it is a change in the address map and this change is occurring due to ASLR.

Let us take for example the library, the lipsey library, so in the first run the lipsey library is present at this location b75d 000 while in the second run the same lipsey library is present at the b75de000, just we see that each run of the an application thus we see with each of the

application, the memory address we see each of the application the virtual address map for that application is changing.

Now since a virtual address map changes the locations of the gadget would change therefore the ROP space attack which the attacker has created will not work all the time, so for your linux procs you can actually change the value of the ASLR by editing this particular file /etc/sysctl.conf in this if you actually add this line like kernel.randomize_va_space and specify a value of 0, 1 or 2, the support for ASLR can be modified for your particular system.

(Refer Slide Time: 9:21)

Internals : Making code relocatable

- Load time relocatable
  - where the loader modifies a program executable so that all addresses are adjusted properly
  - Relocatable code
    - Slow load time since executable code needs to be modified.
    - Requires a writeable code segment, which could pose problems
- PIE : position independent executable
  - a.k.a PIC (position independent code)
  - code that executes properly irrespective of its absolute address
  - Used extensively in shared libraries
    - Easy to find a location where to load them without overlapping with other modules

71

So will now look at the internals of ASLR in order to understand this we will need to know how libraries are made relocatable essentially there are two ways to achieve this one is known as the load time relocatable and the other one is known as the PIE or PIC which stands for position Independent executable and position independent code respectively, in the load time relocatable the main functionality rest with the loader where when the library gets loaded, the loader would pass through the library and adjust each address properly, so that the library executes in the right expected manner.

In the PIC technique relative addressing is extensively used to achieve the same purpose, so will be looked at both the load time relocatable as well as the PIC executable in more details.

(Refer Slide Time: 10:23)

Load Time Relocatable

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

```
chester@aahalya:~/sse/aslr$ make lib_reloc
gcc -g -c mylib.c -o mylib.o
gcc -shared -o libmylib.so mylib.o
```

Refer https://chetrebeiro@bitbucket.org/casl/sse.git (directory src/relocgot) 72

So let us start with load time relocatable and let us say that we want to create a library like this, so this is the library we create it comprises of one global variable called mylib_int and it has to functions setmylib_int which are essentially takes a parameter X and sets our global variable to that particular value and the second function get mylib_int returns the current value that global data mylib_int.

So now you can compile this library by using GCC as shown over here, now by default at least in my compiler, yes in this compiler by default compiling it when this particular syntax will create a load time relocatable code, now this particular code can be also obtained from this link in the directory source/relocgot, to understand how low time relocatable code works, we disassemble this library using OBJ dunk and we just evaluate one of these functions which is the setmylib_int.

(Refer Slide Time: 11:30)

Load Time Relocatable

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

```
0000046c <set_mylib_int>:
 46c:   55                      push   %ebp
 46d:   89 e5                   mov    %esp,%ebp
 46f:   8b 45 08                mov    0x8(%ebp),%eax
 472:   a3 00 00 00 00          mov    %eax,0x0
 477:   5d                      pop    %ebp
 478:   c3                      ret
```

Relocatable table present in the executable that contains all references of mylib_int

```
chester@aahalya:~/sse/aslr$ readelf -r libmylib.so

Relocation section '.rel.dyn' at offset 0x304 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
000015ec  00000008 R_386_RELATIVE
00000473  00000a01 R_386_32          000015f8   mylib_int
0000047d  00000a01 R_386_32          000015f8   mylib_int
000015cc  00000106 R_386_GLOB_DAT    00000000   __gmon_start__
000015d0  00000206 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses
000015d4  00000306 R_386_GLOB_DAT    00000000   __cxa_finalize
```

Store binary value in the symbol memory location

Offset in memory where the fix needs to be made

74

So as we see over here these are the instructions for setmylib_int and the first two instructions creates the stack frame for that particular function, the third instruction this one here moves the content of X that is present in this location EBP plus 8 which essentially is the argument this particular function, so the argument X's move to the EAX register then the contents of X should be stored into the global mylib_int, in order to do this we have this mov instruction where EAX register is moved to 0X0.

So this 0X0 is something what the compiler has put in because this library is low time relocatable library, next we see what happens when we actually executes a program that uses this particular library, so what we do is that create a program link it to this particular library and then use GDP on that program, notice that the actual address of mylib_int is not filled in over here in fact it is just left is 0X0.

So what is expected is that when this library gets loaded, the loader would pass through each of this functions and wherever there is 0X0 it would replace that with the actual address of mylib_int, so in order to achieve this there is special section in the library which will permit the loader to determine which locations the object file need to be modified at the load time, so this particular section is known as the relocatable table, so we can obtained by using this command readelf-r-libmylib.so, that is the library that we have created.

So notice that we have several entries over here but two interested entries for us is the first and second, so each of these entries determines the exact location in the library code at the loader would need to modify, notice that a mylib_int is used in exactly two locations, one is

at this point over here in function setmylib_int and the second one is at this location getmylib_int, each of these locations have an entry in this particular table.

So notice that the locations which we need to modify is at an offset 473 and 47d in the object file, so you could see that the location 473 corresponds to this 0X0 so know that this is at an offset 472 that is A3 is at an offset of 472 and 473 corresponds to this particular 0s, another thing to note is that type, so each of these types is of 32-bit thus at a time when this particular library gets loaded, the loader would look into this table and passed through each row in this table, it will go through this location 473 which corresponds to this 0X0, it would determine that here there is at 32-bit value that is required and this value corresponds to the mylib_int and therefore it would obtain the correct address for mylib_int and replace the 0X0 with the actual address of mylib_int.

So in order to check whether the loader is actually doing its job what we can do is we can write a small program which is linked, which will link to this particular library that will compile that program and use GDP to debug that particular program as follows.

(Refer Slide Time: 15:52)

Load Time Relocatable

```
unsigned long mylib_int;

void set_mylib_int(unsigned long x)
{
        mylib_int = x;
}

unsigned long get_mylib_int()
{
        return mylib_int;
}
```

```
0000046c <set_mylib_int>:
 46c:   55                      push   %ebp
 46d:   89 e5                   mov    %esp,%ebp
 46f:   8b 45 08                mov    0x8(%ebp),%eax
 472:   a3 00 00 00 00          mov    %eax,0x0
```

The loader fills in the actual address of mylib_int at run time.

```
Breakpoint 1, main () at driver.c:9
9               set_mylib_int(100);
(gdb) disass set_mylib_int
Dump of assembler code for function set_mylib_int:
0xb7fde46c <set_mylib_int+0>:   push   %ebp
0xb7fde46d <set_mylib_int+1>:   mov    %esp,%ebp
0xb7fde46f <set_mylib_int+3>:   mov    0x8(%ebp),%eax
0xb7fde472 <set_mylib_int+6>:   mov    %eax,0xb7fdf5f8
0xb7fde477 <set_mylib_int+11>:  pop    %ebp
0xb7fde478 <set_mylib_int+12>:  ret
End of assembler dump.
000015d4  00000306 R_386_GLOB_DAT    00000000   __cxa_finalize
```

75

So what we do is we set a breakpoint in main and then when the breakpoint is hit, we do a disassemble setmylib_int in GDP, so notice the difference between this assembly code and this assembly code, so this assembly code is what the compiler gives out after compilation of the library, while this assembly code is what is obtained at runtime after the loader has inserted the library into the address space, note the same instructions over here you have push EBP mov stack pointer, base pointer and so on, the same instructions are present over here

but also notice that the 0X0 which is present in this instruction is replaced with this address B7FTF5F8 which is the actual address for mylib_int.

So, notice that the loader has achieved on his job, it has replace zero in this location with the actual address of mylib_int, similarly you could also check that the mylib_int in this get mylib_int is also replaced to point to the correct address of mylib_int.

(Refer Slide Time: 17:08)

Load Time Relocatable

Limitations

- Slow load time since executable code needs to be modified
- Requires a writeable code segment, which could pose problems.
- Since executable code of each program needs to be customized, it would prevent sharing of code sections

76

So the limitations for the load time relocatable technique is that it has extremely slow load time, since essentially we have the loader which passes through each and every location which needs to be modified and fills and replaces the zeros in that location with the actual address, secondly it requires a writable code segment, which could essentially pose problems, the third limitation of load time relocatable technique is that it prevents sharing of executable code, will not go into the details about this but in operating systems there is a process called copy on write.

Where typically library codes are all shared between all processes in the system, third limitation of the load time relocatable code is that each program code, should have its own customized copy of the library, so this is not what we want in practice, in practice typically to prevent publication all programs that are running in a machine would use the same copy of the shared library, for example Lipsey all programs that are run in the system would use the same copy of Lipsey, so as not to duplicate Lipsey in the ram, however with load time relocatable since each a program need a very customized version of the library, that for such kind of sharing will not be possible.

(Refer Slide Time: 18:39)

PIC Internals

- An additional level of indirection for all global data and function references
- Uses a lot of relative addressing schemes and a global offset table (GOT)
- For relative addressing,
  - data loads and stores should not be at absolute addresses but must be relative

Details about PIC and GOT taken from ...
http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

77

A lot of the disadvantages of the load time relocatable technique are removed by using PIC or programmable independent code technique, so with this particular technique libraries can be made relocatable in the virtual address space of the process, so essentially what is the done here is that a lot of relative addressing is used additionally instead of relying on the loader to actually change each and every address in the code, a special table known as global offset table or GOT table is used, so we will see how this PIC works and how the GOT table is used to resolve the actual address of a variable.

(Refer Slide Time: 19:21)

Global Offset Table (GOT)

- Table at a fixed (known) location in memory space and known to the linker
- Has the location of the absolute address of variables and functions

Without GOT

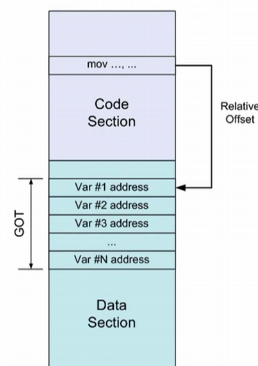
```
; Place the value of the variable in edx
mov edx, [ADDR_OF_VAR]
```

With GOT

```
; 1. Somehow get the address of the GOT into ebx
lea ebx, ADDR_OF_GOT

; 2. Suppose ADDR_OF_VAR is stored at offset 0x10
;    in the GOT. Then this will place ADDR_OF_VAR
;    into edx.
mov edx, DWORD PTR [ebx + 0x10]

; 3. Finally, access the variable and place its
;    value into edx.
mov edx, DWORD PTR [edx]
```



78

So let us say that we have an instruction like this where we want to move the address of a variable to this register EDX, now typically without GOT we would require to know the

actual address of this particular variable and therefore this particular instruction would result in non-relocatable code, if you have a global offset table however the same single instruction gets converted into three instructions as follows, first we load the address of the GOT table into this register called EDX then we load an offset in the table more precisely an offset of 16 bytes in the table into this register EDX.

Then we load an offset into the table more precisely an offset of 16 bytes into this register EDX, now at this location EDX plus 16 bytes what is present is the actual address of this particular variable, next what we do is load the contents of this EDX register into this particular register thus we see that instead of directly loading the address of the variable into the EDX register which is making the code non-relocatable instead we use the GOT table, the GOT table contains the actual addresses where the variables are present and therefore we load the content of the GOT table and access the actual variable directly using the contents of the EDX.

(Refer Slide Time: 21:16)

So we will take our library that we have created and see how the compiler generates code for using this GOT table, so the code that we will take is as before comprising of our two library functions setmylib_int and getmylib_int and more importantly at this particular time is this particular global variable mylib_int, so we have seen what happens when the load time relocatable code, now with PIC the code looks much different for setmylib_int, so what we see is that additionally there is a call to some function called I 686 get pc thunk and additionally there are certain other changes as well.

So let us look at more details, so first we see the call instruction which are essentially is a call to this particular function over here which stores the contents of the stack pointer into the ECX register and then returns, so what we see over here is the contents of these ECX register we have the address of this particular instruction, now you had 1180 to the contents of this ECX register, so this 1180 is the offset for the GOT table, so ECX +1180 is an offset to the got table.

Further we subtract 8 bytes 1 ECX which is an offset in the GOT table which contains the actual address of mylib_int, so this actual address of mylib_int is move into the EAX register, third we note that we are storing contents of the EDX register into the location pointed to by the EAX register, so note that the EAX register contains the pointer to the correct address of mylib_int, therefore the store instruction would store the contents of the EDX register to the correct location of mylib_int.

(Refer Slide Time: 23:12)

Advantage of the GOT

- With load time relocatable code, every variable reference would need to be changed
  - Requires writeable code segments
  - Huge overheads during load time
  - Code pages cannot be shared
- With GOT, the GOT table needs to be constructed just once during the execution
  - GOT is in the data segment, which is writeable
  - Data pages are not shared anyway
  - Drawback : runtime overheads due to multiple loads

CR

81

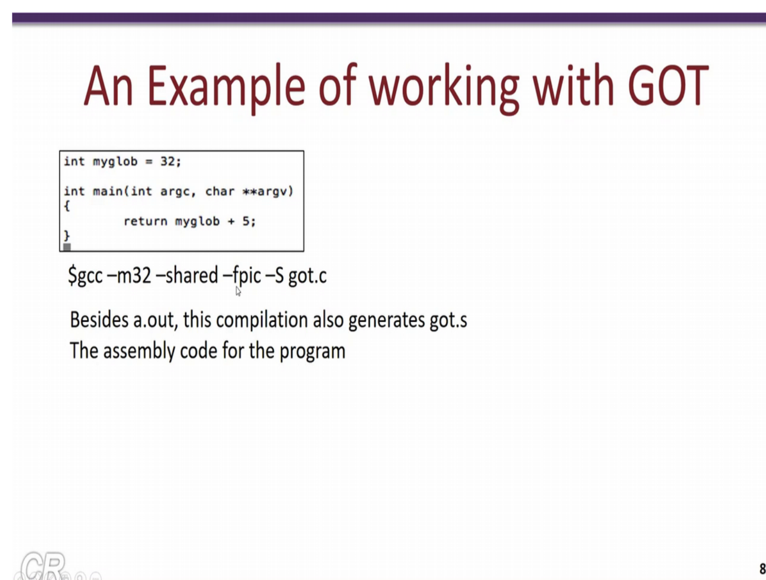
So the advantages of using PIC technique that is with using the GOT is that you have reduced the load time considerably, unlike the previous case where the loader goes on changes each and every location where the global data is used, here we are using a single GOT table which stores the correct address for the global data, thus the overheads by loading the library is reduced considerably, further most if you are not changing achievable code, which we not required that the codes segments are writable.

This is quite unlike the load time relocatable technique value actually required the code segment to be writable, further since we do not require customized libraries for each program,

we can actually share the libraries between various programs in the system, all of these advantages are achieved because of the GOT table, the GOT table is present in the data segment and as we know the data segment is writable, therefore add load time all that the loader needs to do is go and fill in the GOT table.

The drawback of this particular scheme is that now the runtime overheads have increased, instead of directly going and accessing a particular variable, now we need to find out the actual variable from the GOT table and then make an indirect access to that particular variable, thus resulting in increased runtime.

(Refer Slide Time: 24:39)



Let us look at an example of working with GOT, let us take this small example where we have a my global data initialize to 32 and in the program we increment with the global data by 5 and return that value, now in order that the compiler generates a GOT table, we need to specify additional option at compile time which is minus shared minus fpic.

(Refer Slide Time: 25:02)

```
        .file   "got.c"
.globl myglob
        .data
        .align 4
        .type   myglob, @object
        .size   myglob, 4
myglob:
        .long   32
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    __i686.get_pc_thunk.cx
        addl    $_GLOBAL_OFFSET_TABLE_, %ecx
        movl    myglob@GOT(%ecx), %eax
        movl    (%eax), %eax
        addl    $5, %eax
        popl    %ebp
        ret
        .size   main, .-main
        .ident  "GCC: (Debian 4.4.5-8) 4.4.5"
        .section        .text.__i686.get_pc_thunk.cx,"axG",@progbits,__i686.get_
pc_thunk.cx,comdat
.globl __i686.get_pc_thunk.cx
        .hidden __i686.get_pc_thunk.cx
        .type   __i686.get_pc_thunk.cx, @function
__i686.get_pc_thunk.cx:
        movl    (%esp), %ecx
        ret
        .section        .note.GNU-stack,"",@progbits
```

Data section

Text section

The macro for the GOT is known by the linker. %ecx will now contain the offset to GOT

Load the absolute address of myglob from the GOT into %eax

Fills %ecx with the eip of the next instruction.
Why do we need this indirect way of doing this?
In this case what will %ecx contain?

83


```
        .file   "got.c"
.globl myglob
        .data
        .align 4
        .type   myglob, @object
        .size   myglob, 4
myglob:
        .long   32
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        call    __i686.get_pc_thunk.cx
        addl    $_GLOBAL_OFFSET_TABLE_, %ecx
        movl    myglob@GOT(%ecx), %eax
        movl    (%eax), %eax
        addl    $5, %eax
        popl    %ebp
        ret
        .size   main, .-main
        .ident  "GCC: (Debian 4.4.5-8) 4.4.5"
        .section        .text.__i686.get_pc_thunk.cx,"axG",@progbits,__i686.get_
pc_thunk.cx,comdat
.globl __i686.get_pc_thunk.cx
        .hidden __i686.get_pc_thunk.cx
        .type   __i686.get_pc_thunk.cx, @function
__i686.get_pc_thunk.cx:
        movl    (%esp), %ecx
        ret
        .section        .note.GNU-stack,"",@progbits
```

Data section

Text section

The macro for the GOT is known by the linker. %ecx will now contain the offset to GOT

83

The assembly code for this particular program look something like this it has got a data section over here which is my global and it has got the codes so went over here which is under this my text, so note that the compiler has created these additional functions getpc_thunk.cx which are essentially loads the address of the next instruction into the ECX, more details we see that this particular instruction adds the offset of the GOT table to the ECX register.

Then we load the absolute address of myglob global variable from the GOT table into the EAX register and finally we can indirect load of the myglob global data to the EAX register, so after these for instructions we are finally been able to load the contents of the EAX register, the disassembly of this particular code look something like this, so we see that there

is a data segment comprising of this global data myglob and the codes segments starts from this particular section, which is denoted as the text section, as we see over here these are the important instructions which we have to analyse, so first instruction we see is that there is a call to this get_pc_thunk.

What this call does is that it jumps to this particular function over here, those the contents of the stack pointer into this ECX register, thus the ECX register contains the address of this particular instruction, the addl instruction, in this instruction what we do is add the offset of the GOT table, the global offset table to this ECX register, so now the ECX register as the contents of the GOT table, now we take a offset in the GOT table and that to ECX and load the contents into EAX register.

So now EAX register as the correct address of myglob, the global address, now we load the contents of that global data into EAX register, thus at the end of this instruction the EAX register has the contents of myglob which is 32, we added 5 to this contents and return this particular data.

(Refer Slide Time: 27:32)

More

```
chester@aahalya:~/tmp$ readelf -S a.out
There are 27 section headers, starting at offset 0x69c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE
  [ 2] .hash             H
  [ 3] .gnu.hash         G
  [ 4] .dynsym           D
  [ 5] .dynstr           S
  [ 6] .gnu.version      V
  [ 7] .gnu.version_r    V
  [ 8] .rel.dyn          R
  [ 9] .rel.plt          R
  [10] .init             P
  [11] .plt              P
  [12] .text             PROGBITS        00000370 000370 000118 00  AX  0   0 16
  [13] .fini             PROGBITS        00000488 000488 00001c 00  AX  0   0  4
  [14] .eh_frame         PROGBITS        000004a4 0004a4 000004 00   A  0   0  4
  [15] .ctors            PROGBITS        000014a8 0004a8 000008 00  WA  0   0  4
  [16] .dtors            PROGBITS        000014b0 0004b0 000008 00  WA  0   0  4
  [17] .jcr              PROGBITS        000014b8 0004b8 000004 00  WA  0   0  4
  [18] .dynamic          DYNAMIC         000014bc 0004bc 0000c8 08  WA  5   0  4
  [19] .got              PROGBITS        00001584 000584 000010     WA  0   0  4
  [20] .got.plt          PROGBITS        00001594 000594 000014 04  WA  0   0  4
```

```
chester@aahalya:~/tmp$ readelf -r ./a.out

Relocation section '.rel.dyn' at offset 0x2d8 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
000015a8  00000008 R_386_RELATIVE
00001584  00000106 R_386_GLOB_DAT    00000000   __gmon_start__
00001588  00000206 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses
0000158c  00000406 R_386_GLOB_DAT    000015ac   myglob
00001590  00000306 R_386_GLOB_DAT    00000000   __cxa_finalize
```

offset of myglob in GOT

GOT it!

84

If we use readelf to determine the various sections of the eroded file that we have just compiled, we see that the 19th section as the GOT table, so this is at an offset of 584 bytes and has an address of 1584 from the start, further we can then look at the relocatable data and see the offset of this global data myglob in the GOT table.

(Refer Slide Time: 27:59)

Deep Within the Kernel
(randomizing the data section)

loading the executable

```
1  static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs *regs)
2  {
3          struct file *interpreter = NULL; /* to shut gcc up */
4          unsigned long load_addr = 0, load_bias = 0;
5     ...
6  #ifdef arch_randomize_brk
7          if ((current->flags & PF_RANDOMIZE) && (randomize_va_space > 1))
8                  current->mm->brk = current->mm->start_brk =
9                          arch_randomize_brk(current->mm);
10 #endif
11    ...
12 out_free_ph:
13         kfree(elf_phdata);
14         goto out;
15
```

Check if randomize_va_space is > 1 (it can be 1 or 2)

```
1  unsigned long arch_randomize_brk(struct mm_struct *mm)
2  {
3          unsigned long range_end = mm->brk + 0x02000000;
4          return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
5  }
```

Compute the end of the data segment (m->brk + 0x20)

```
10 unsigned long
11 randomize_range(unsigned long start, unsigned long end, unsigned long len)
12 {
13         unsigned long range = end - len - start;
14
15         if (end <= start + len)
16                 return 0;
17         return PAGE_ALIGN(get_random_int() % range + start);
18 }
```

Finally Randomize

85

So now that we know how a library can be made relocatable either using the PIC technique or by low time relocatable, we will now see how ASLR works, so in order to understand this we have look at snippets of the operating system, essentially what is happening is that when the operating system invokes this function load elf binary, so this particular function is invoked when you want to either load binary data to a process or a shared library gets loaded into a particular process.

So in the operating system you would have a function like this and importantly for us there is a if condition is and here where we check whether this randomize_va_space is greater than one, so we collect that in the Linux systems the randomize_va_space would take 3 values 0, 1 or 2, if the value is either 1 or 2 it would mean that ASLR is enable on that particular system, so if ASLR is enable on the system we invoke this part particular function arch_randomize_break which essentially is present here.

So what this function does is invoke this randomize range which returns some particular random address, so this random address is then return here and at this random address is where the library is loaded, thus we see how the operating system uses some random location in the process address space in order to load the library, further on the time of loading the loader would either fill the GOT table or go specifically to those particular locations and fill addresses in those global data.

So in addition to the data even the function should be made relocatable, in the next lecture we will see how functions are made relocatable. Thank you.