Co-funded by the
Erasmus+ Programme
of the European Union

ISSES

ISSES – Information Security Services
Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant
N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

HARDWARE SECURITY

COMPUTER SECURITY

Lecture 2

ISSES 2017-2020
Novi Sad, Serbia

Information Security Services Education in Serbia (ISSES)

2.7 HARDWARE SECURITY

Hardware Security?

- The last lecture was focused to the idea of security *of* hardware: whether someone has physical access to it.
- This lecture is more on how the hardware and its architecture may provide security.
- A lot of this information will only make perfect sense once we've reached lectures 5, 6, and 7 and have thoroughly discussed operating systems and how they interface with hardware.
- The story of hardware security will then naturally follow into a discussion of the various types of security-specific hardware including TC, TEEs, and the most secure of all, crypto-modules.

A note on technology

- As may be assumed, a discussion of this nature is heavily bound up with the specific platform in use.
- It seemed simplest to, for the purposes of this lecture, to focus on one specific platform.
- In our case it will be x86 and x86_64 with a bias towards Intel.
- Why? Because this is something most of use on a daily basis and it has had some fairly severe security vulnerabilities in the past.

The problem

- The only way things happen on modern systems is if the CPU executes some instructions (this is not *strictly* speaking true, as we'll learn during this lecture, but it's close). This means that whatever the system does, there's an instruction that does it.
- Coding for a system does, in the end, mean telling the CPU which commands to execute in which order.
- So let's say a piece of code we've written executes the assembly commands for writing a new master boot record (MBR) on a hard drive. It's only about twenty instructions or so, setting up some registers and asking for interrupt 0x13.
- Should the CPU execute it or not? If it is us installing a new bootloader, yes. If it is a virus it shouldn't. How can it decide?

Modes

- Of course, the answer to this is that, sometimes, *the CPU cannot possibly tell.*
- The x86 architecture labors under the need to be backwards compatible.
- This means that even the most modern i9 or Threadripper has to be able to act *exactly* like an Intel 8086 CPU down to running the same code.
- One way this is provided for (since the original 8086 is *nothing* like a modern CPU) is by allowing each CPU to be in different modes of operation.

Real Mode

- All x86 CPUs start in what's called Real Mode.
- Real Mode is… peculiar, by modern standards.
- It can't address more than 1MB of memory for a start, this as you may imagine is quite inconvenient.
- Real Mode also has *no protection of any sort from anything.*
- Any code can access anything else, modify anything it likes, execute any instruction, and do anything.
- On the plus side, access to hardware is very easy and only requires the use of BIOS routines.
- Operating Systems used to start work like this and a lot of very old PC user software was built to run in this no-limits mode.

Real Mode

- Real Mode's memory limitations are a historical curiosity
- Early PC CPUs only had 20 physical lines for memory addresses, and 1MB was considered to be quite a large amount at the time (the first PCs shipped with as little as 16K and could not fit more than 64K on the motherboard until higher-capacity memory chips were available)
- However, something like Real Mode was how computers simply worked until someone thought to make a system support multiple interactive users.
- Suddenly, the system needed two distinct ways of working: one which allowed the kernel to do whatever it wished, and another which protected the other users on the system from a malicious or simply clumsy user.

Protected Mode

- Protected mode is what replaced Real Mode and what computers spend a vast proportion of their time in.
- Strictly speaking, Protected Mode isn't used anymore: Long Mode is, Long Mode being a 64-bit extension of protected mode. However, the Long Mode does most of what Protected mode does (with some differences we will touch upon later) and it even emulates fully old-school Protected Mode when required so we won't bother ourselves with the difference overmuch.
- Protected mode introduces two things that are relevant to us: *segmented memory translation* and *privilege rings.*

A brief reminder regarding memory translation

- Your previous education should have prepared you for this, but in case you have forgotten a reminder.
- What you write into code (or what your compilers, assemblers, and linkers, variously, write into code) are *logical addresses.* These must be translated into *physical* addresses, i.e. the actual bit of a memory chip with the data in it.
- Memory modules don't have any sort of protection. Whatever addresses they get they change/return as requested.
- Therefore, the way to protect memory has to be in the conversion between logical addresses and physical addresses also known as *memory address translation.*

Memory Translation Rules

CPU Mode	Translation ruleset
Real with A20 disabled	linear = ((logicalSegment << 4) + logicalOffset) % (2 << 20) physical = linear
Real with A20 enabled	linear = ((logicalSegment << 4) + logicalOffset) physical = linear
Protected	linear = DT[segmentIndex] + logicalOffset physical = linear
Protected with paging	linear = DT[segmentIndex] + logicalOffset physical = PT[(linear & 0xFFFFF000) >> 20] + linear + 0x00000FFF

Note: 32-bit only, some considerable simplifications employed in order for the example to be more illustrative. If you wish to know more on the subject the official Intel Manuals are best.

Where does the segment come from?

- The preceding rules use segment or the segment index.
- Where do these come from?
- The processor has special registers for these, depending on what the memory is used for: CS for memory used for code, SS for memory used for the stack, and DS for memory used for data.
- These registers contain just addresses (the logicalSegment in the formulae of the past slide) in Real Mode, but in other modes contain something called a *segment selector.* A segment selector contains the index and a few other things besides which will be mentioned later on.

Implications for security

- Segmentation and paging are used for extending the memory space, introducing virtual addresses, enabling swapping and all manner of fascinating things.
- These things are not the subject of this course, however.
- What matters to us is that this enables memory to be divided *at a hardware level* into well-defined areas which we can annotate with security details.

Interactions between segmentation and paging

- The chief reason segmentation and paging exist are historical. Segmentation came first (80286) and then paging (80386). The ARM architecture, for instance, doesn't have segmentation in the same way as x86 does relying exclusively on paging.
- Modern x86 does much the same and modern operating systems actually only really define a handful of segments.
- x86_64 doesn't really *have* segments as such, though it is adept at emulating them.

Linux 5.6.4 32-bit segment layout

```
/*
 * The layout of the per-CPU GDT under Linux:
 *
 *   0 - null
 *   1 - reserved
 *   2 - reserved
 *   3 - reserved
 *
 *   4 - unused
 *   5 - unused
 *
 *  ------- start of TLS (Thread-Local Storage) segments:
 *
 *   6 - TLS segment #1                 [ glibc's TLS segment ]
 *   7 - TLS segment #2                 [ Wine's %fs Win32 segment ]
 *   8 - TLS segment #3
 *   9 - reserved
 *  10 - reserved
 *  11 - reserved
 *
 *  ------- start of kernel segments:
 *
 *  12 - kernel code segment
 *  13 - kernel data segment
 *  14 - default user CS
 *  15 - default user DS
 *  16 - TSS
 *  17 - LDT
 *  18 - PNPBIOS support (16->32 gate)
 *  19 - PNPBIOS support
 *  20 - PNPBIOS support
 *  21 - PNPBIOS support
 *  22 - PNPBIOS support
 *  23 - APM BIOS support
 *  24 - APM BIOS support
 *  25 - APM BIOS support
 *
 *  26 - ESPFIX small SS
 *  27 - per-cpu                       [ offset to per-cpu data area ]
 *  28 - stack_canary-20               [ for stack protector ]
 *  29 - unused
 *  30 - unused
 *  31 - TSS for double fault handler
 */
```

- From /arch/x86/asm/segment.h
- Comment helps describe the global descriptor table (GDT)
- The local descriptor table is, nowadays, never used.
- Not all of these are actually segments, since the table where segments are stored contains other sorts of things (some of which we'll mention more later)
- The ones we care about are 12, 13, 14, and 15. These are the segments for kernel code, for kernel data, for user code and user data.
- They are defined to universally range from 0 to 0xfffff, in other words, to cover all available space.
- Intel calls this 'flat mode.'

Linux 5.6.4 64-bit segment layout

```
#define GDT_ENTRY_KERNEL32_CS           1
#define GDT_ENTRY_KERNEL_CS             2
#define GDT_ENTRY_KERNEL_DS             3

#define GDT_ENTRY_DEFAULT_USER32_CS     4
#define GDT_ENTRY_DEFAULT_USER_DS       5
#define GDT_ENTRY_DEFAULT_USER_CS       6

/* Needs two entries */
#define GDT_ENTRY_TSS                   8
/* Needs two entries */
#define GDT_ENTRY_LDT                   10

#define GDT_ENTRY_TLS_MIN               12
#define GDT_ENTRY_TLS_MAX               14

#define GDT_ENTRY_CPUNODE               15

/*
 * Number of entries in the GDT table:
 */
#define GDT_ENTRIES                     16
```

Security details

- Given that the segments cover all addressable space, the question arises: why separate kernel ones and user ones?
- Because of those very security details.
- The segments defined differ not in their dimensions but their flags:

```
[GDT_ENTRY_KERNEL_CS]         = GDT_ENTRY_INIT(0xc09a, 0, 0xfffff),
[GDT_ENTRY_KERNEL_DS]         = GDT_ENTRY_INIT(0xc092, 0, 0xfffff),
[GDT_ENTRY_DEFAULT_USER_CS]   = GDT_ENTRY_INIT(0xc0fa, 0, 0xfffff),
[GDT_ENTRY_DEFAULT_USER_DS]   = GDT_ENTRY_INIT(0xc0f2, 0, 0xfffff),
```

Note. Flags are indicated in red. Code from Linux 5.6.4 kernel in /arch/x86/kernel/cpu/common.c.

Flags

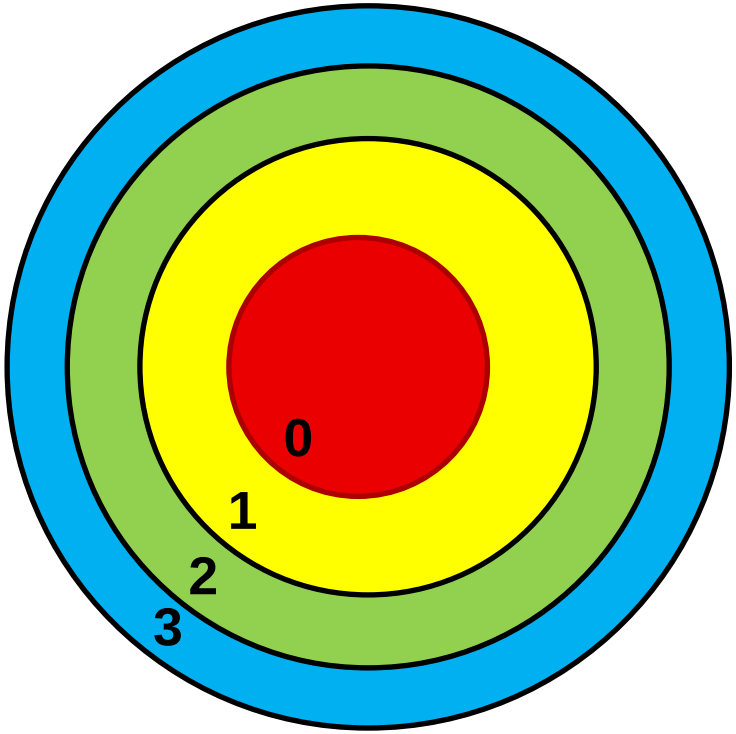
```
KERNEL CS: 1100 0000 1001 1010
USER   CS: 1100 0000 1111 1010

KERNEL DS: 1100 0000 1001 0010
USER   DS: 1100 0000 1111 0010
```

Note. The entries marked in blue are the difference in type, between code segments and data segments. The entries marked in red are the security detail difference.

The ring model

- The security detail in question is the security level also known as the *ring.*
- Rings as a term, and the model itself, first appeared as part of MULTICS an incredibly influential early multi-user operating system.
- MULTICS had a much more elaborate system of eight rings, but the central idea is still used.
- In the ring model of security each successive security level has all the access the previous one did, plus some extra. In that sense they can be imagined as forming rings.
- This system is used in non-technical contexts too. Consider the gradation between CONFIDENTIAL, SECRET, and TOP SECRET classifications for documents.

The ring model



- Ring 0, all access
- Ring 1
- Ring 2
- Ring 3, minimal access

Privilege levels in x86

- The x86 architecture defines 4 privilege levels conventionally numbered from 0 (most privilege) to 3 (least).
- These are also commonly referred to as 'Ring 0', 'Ring 1', 'Ring 2', 'Ring 3'.
- Ring 0 is treated a bit specially and is commonly referred to as 'privileged.'
- The CPU is always in *some* privilege level and this is detectable using the CS register. The CS register, as stated before, is the segment selector for code and in protected mode it's expected to contain the index (in bits 15 to 3) the flag that decides whether to use the GDT (0) or LDT (1) in bit 2, and in bits 1 and 0 the current privilege level.

Reading the CS register

```
#include <iostream>

using namespace std;

int main() {
    volatile short out = 0;
    asm volatile (
                    "movw %%cs, %0\n\t"
                    : "=r" (out)
                    :);
    cout << hex << out << endl;
    return 0;
}
```

CS output

0x33 = 0000 0000 0011 0011

CPL = 11_2 = 3 (Ring 3)

TI = 0_2 = 0 (Uses GDT)

INDEX = 0 0000 0000 0110_2 = 6

```
#define GDT_ENTRY_DEFAULT_USER32_CS     4
#define GDT_ENTRY_DEFAULT_USER_DS       5
#define GDT_ENTRY_DEFAULT_USER_CS       6
```

This can be used to determine that the code shown was executed as a user process on a 64-bit version of recent Linux.

Information Security Services Education in Serbia (ISSES)

2.8 An introduction to CPU security features

What x86 privilege levels protect

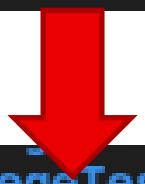
- What those privilege levels protect are:
  - Instructions
  - Registers
  - I/O ports (with some additional permission levels being considered here)
  - Memory

Instruction protection

- Certain instructions in the x86/x86_64 instruction set are set so that they belong to a specific privilege level.
- The CPL information stored in the CS register is used to compare if the instruction can even be executed or not.
- Despite there being a number of rings defined in the architecture, in practice the important one is 0.
- Protected instructions are, in practice, restricted to privileged, i.e. ring 0 mode.
- Commands executed in the wrong mode trigger a general protection fault.

Violating protection

```cpp
#include <iostream>

using namespace std;

int main() {
    asm volatile (
                    "hlt\n\t"
                    :
                    :);
    return 0;
}

```



```
veljko@sunchaser:~/tmp/privilegeTester$ g++ -o halt halt.cpp
veljko@sunchaser:~/tmp/privilegeTester$ ./halt
Segmentation fault (core dumped)
```

Instruction protection

Instruction	Purpose
LGDT	Loads values into the GDT register. We'll shortly see why this is a high-security operation.
LLDT	Loads values into the LDT register.
LTR	Loads task register.
LIDT	Loads IDT register containing the location of the interrupt handler table.
LMSW	Loads values into the machine status word. Would allow, among other things, switching into Real mode. Dangerous.
CLTS	Clear task-switched flag in CR0. Interferes with machine-aided context switching.
INVD	Flushes, without writeback, all internal CPU caches.

Note. Source of this information is the Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A, System Programming Guide, Part 1, Section 5.9.

Instruction protection

Instruction	Purpose
WBINVD	Flushes all internal CPU caches while writing them back to the main memory.
INVLPG	Invalidates the contents of the specified entry in the Translation Lookaside Buffer, a cache which memorizes translations from logical to physical addresses.
HLT	Halts processor.
RDMSR	Reads model-specific registers, registers that only exist on certain CPUs which control additional non-standard features.
WRMSR	Writes model-specific registers.
RDPMC	Reads performance-monitoring counters which are hardware-implemented profiling aids. Very useful, but a potential side-channel attack (see later).
RDTSC	Reads time-stamp counter. Technically, this is only forbidden if the 3rd bit of the CR4 control register is set to 1.

Register Protection

- Some instructions are perfectly okay most of the time, but can become forbidden for non-privileged levels based on the registers they try to manipulate.
- This makes sense as all the sensitive internal state that makes the system work properly is held in various registers.
- Therefore, these must be protected. Sometimes they can't be accessed at all, and the only way to modify them is through special instructions: in these cases, the instructions may be themselves restricted, as we've seen.
- However, sometimes the registers are directly accessible.
- In these cases instructions may be forbidden depending on what the operands are so that movq %rax, %rbx is perfectly okay, but movq %cr0, %rax causes a general protection fault.

A selection of protected registers

Register	Purpose
FLAGS	Readable, but cannot be changed. Modifying it directly (by popping a value from the stack using the POPF family of instructions) is only permissible at ring 0. Modifying parts of it through functions is possible, see IO protection later.
CR0	Control register. Crucially important, contains such niceties as whether protected mode is active or not.
CR2	Control register. Used to resume execution in case of page fault. Hijacking it allows hijacking the execution path of the code.
CR3	Control register. Used to store the root of the paging directory which, on modern systems, is crucial for all security.

I/O protection

- To understand I/O protection it is first necessary to understand how the I/O mechanism on modern CPUs works (see also Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1, Chapter 19).
- There's two ways x86 and derived architectures can access external devices for I/O
  - Separate I/O address space
  - Memory-mapped I/O
- In the case of the separate I/O address space specialized instructions are used.
- In the case of memory-mapped I/O, normal mov instructions and so on can be used: part of the memory address space is simply mapped onto various I/O ports.

I/O address space

- The I/O address space is completely separate and unrelated to main memory.
- It consists of 65 536 individually addressable 8-bit I/O ports which may be doubled and redoubled to form 8-bit, 16-bit, or 32-bit ports, provided alignment is followed.
- The I/O address space has the benefit of being guaranteed to complete before any further instructions are executed.
- Access is performed through IN, OUT, INS, and OUTS instructions which, as stated, may take bytes, words, or double words as needed.

Memory-mapped I/O

- In memory mapped I/O a certain portion of the memory is mapped on a hardware level to I/O ports.
- These can then be read, and written to, and manipulated with any instruction whatsoever which can also access memory.
- Typically the EPROM is mapped to the very topmost area of the physical address space with I/O ports underneath, and then RAM below that.
- It's vital to prevent caching of these addresses for obvious reasons, this is done using the Memory Type Range Registers (MTRRs) which allow the definition of areas in the physical memory space with forbidden caching (using the UC type)

Protection of the I/O address space

- Protection of the I/O address space is done through the concept of IO Privilege.
- This means that, strictly speaking, x86 doesn't quite support the ring structure as it is possible to have things in ring 3 have more privileges than other things in ring 3 due to the differing I/O level.
- The first element of I/O protection is the IOPL field in the EFLAGS register of the processor.
- Then, I/O-sensitive instructions (as this class is called) can only be performed if the CPL (the last two bits of the CS register) is less than or equal to the IOPL field in the EFLAGS register (bits 13 and 12).

I/O sensitive instructions

Instruction	Purpose
IN	Read singly from an I/O address.
INS	Read multiply from an I/O address.
OUT	Write singly to an I/O address.
OUTS	Write multiply to an I/O address.
CLI	Clear interrupt-enable flag.
STI	Set interrupt-enable flag.

Note. The above means that it is possible, in POSIX systems to engineer an unlikely scenario in which a userspace program *stops interrupts.* This requires root privileges, however, and invariably leads to the system crashing. More on this in lectures dealing explicitly with operating system security.

The Task State Segment

- Before we tackle the second element to I/O protection, it is necessary to mention what a task state segment is.
- The TSS is an entry in the segment table that defines not a traditional memory segment but the descriptor of a hardware-controlled task. The TSS is a mechanism that allows for hardware-controlled task-switching.
- Hardware-controlled task switching is not really supported by x86_64 but the TSS still exists and is used for some still-handy contents. Among these is the I/O Permission Bit Map.

I/O Permission Bit map

- The I/O Permission bit map is a variable-sized region in the memory addressed by the I/O map base field in the TSS entry which points to its first byte. It is terminated by a byte containing all ones.
- Each bit of the I/O permission bit map corresponds to one I/O port address.
- If the bit is cleared it signifies permission, if it is set, it signifies that access is forbidden.
- This test is, naturally, only performed if the CPL is greater than the IOPL.

Protecting memory-mapped I/O

- Memory-mapped I/O is not protected using blocked instructions or blocked I/O port addresses.
- Instead MMIO security is managed using the same mechanism used to manage the security of any memory address: one built into the segmentation and paging mechanisms that are a part of x86 and x86_64 architectures.

Protecting memory

- This will be the subject of further discussion in several other lectures.
- Memory is protected through two subsystems: segmentation and paging.
- Segmentation-level protection overrides paging-level protection.
- Thus if one wishes to use pages to provide more granular permissions the *segment* must be less strict and the page more strict, not vice-versa.
- Despite the fact that x86_64 largely uses a flat memory model and does not need segments to access memory, segments are still retained for certain purposes, hence their continuing relevance.

Segmentation protection

- Privilege checks are done whenever segment translation takes place, i.e. whenever code tries to access data, stack info, or code in some segment other than itself.
- In C such pointers/jumps are called 'far,' i.e. a far jump is a jump to a new segment and a far pointer is a pointer to data outside the current segment or, rather, a pointer which takes into account both a segment selector and an offset.
- Privilege checking during segmentation takes into accounts three different values: RPL, CPL, and DPL.

CPL, DPL, and RPL

- **CPL** is the current privilege level and it represents, in segment protection computations the privilege level of the *calling* code. It is always discoverable, as has been said, through the last two bits of the code segment selector, i.e. register CS.
- **DPL** is the descriptor privilege level. Each segment entry in the GDT or LDT has a privilege level field specifying the level protection of said code/data. This is the security detail that started our exploration of CPU-level security features.
- **RPL** is the requested privilege level. It allows the CPL to be weakened to that level. It's located in the segment *selector* and is used to signify to the system that a more-privileged level is executing an instruction for the benefit of a less-privileged level.

Some more details on RPL

- What's the point of RPL?
- Consider a less privileged piece of code (ring 3) asking a more privileged piece of code (ring 0) to do some I/O for it.
  - Later we'll learn that this is what a syscall is.
- This means that using a mechanism we're yet to discuss, ring 3 code somehow calls a specially designated ring 0 code to do something for it.
- Some of those operations are meant to be at level 0, sure, but some *aren't.*
- Without RPL we'd be vulnerable to an attack that'd look something like the following.

What RPL protects us against

```
write(myFile, pMyData, n);
//I've written data I've chosen into myfile. This is permitted.
read(myFile, pSystemMemory, n);
//I've asked that contents of myfile be read into a buffer I specify. I've provided memory that's not mine but belongs to the system. Normally I can't touch it, but because read is executed by system (ring0) code, this is permissible.
```

Segment protection logic

```
if(max(CPL, RPL)  <= DPL){
    loadSegment();
}else{
    generalProtectionFault();
}
```

Note. Since lower-valued privilege levels are more privileged than higher-valued privilege levels the 'max' function in essence selects the least-privileged among CPL and RPL.

Page-level protection

- Segments are no longer as used as they once were.
- While they still exist (and still divide kernel and user code in modern systems) the chief mechanism of protection is the *page.*
- Page protection somewhat collapses the granularity of protection level. In practice no OS used privilege levels 1 and 2 much, using largely only 0 (kernel code) and 3 (user code).
- We can see in the quoted Linux code that Linux defines segments in precisely this way today.
- Because of this, the level of protection on pages is lesser: a page is either privileged or not or, in the terminology applied: a user page (0) or a supervisory page (1).

Page-level protection

- Pages also include read/write protection as an addition to domain (i.e. user and supervisor).
- This means that pages may be designated as read-only.
- This is a security and reliability measure as it allows pages that are meant to contain code to be configured in such a way that writing to them always fails.
- This helps fight viruses
- It also prevents the use of self-modifying code which, while it can have legitimate uses, is also incredibly bad for system reliability.

Information Security Services Education in Serbia (ISSES)

2.9 WHERE TO FROM HERE?

Further hardware features

- The preceding has only been an introduction to hardware security features of only one architecture, albeit a highly complex one.
- We will continue talking about hardware security throughout the course as various types of security are only really possible if the hardware helps make them a reality.
- Topics of particular interest are:
  - Trusted computing & security-only hardware.
  - Interaction between privilege levels.
  - Memory protection beyond privilege levels.
  - Advanced CPU security

Trusted computing

- Trusted computing is a very general term, but in lecture 3 we'll use it to cover all sorts of hardware designed specifically and exclusively to provide security.
- This includes Trusted Execution Environments, and Cryptographic Modules
- It also includes talk of privilege levels below 0—modern computers have an entire extra computer embedded whose sole purpose is to provide security services.

Interaction between privilege levels.

- We've described in great details how to *segregate* code and data based on privilege level.
- But for a computer to be a unified whole, there must be some way for code to communicate between privilege levels.
- Lectures 4 and 6 will cover this as a part of introduction to operating system security introducing the mechanisms of interrupt handling, system calls, and call gates.

Beyond privilege levels.

- Security is not only about protecting the system from user code, it's also about protecting various bits of user code from one another.
- Lectures 6 and 8 will cover this in detail discussing how the concept of virtual address spaces may be used to effectively isolate processes in modern operating systems.

Advanced CPU security

- The CPU can also be used to provide advanced security, and the operating system may be relied upon to exploit these features in order to harden the system to various forms of attack.
- We'll discuss this as a part of lecture 10.