CSE 127 Computer Security

Stefan Savage, Spring 2020, Lecture 7

System Security I:
Isolation: Inter-Process, User/Kernel, VMs

Logistics

- A couple quick announcements
  - PA2 extension (same time, but next Tuesday)
  - PA3 will still be assigned this Thursday

So… where we left things

- We know lots of ways to **corrupt control flow**
  - Stack overflow, heap overflow, pointer subterfuge, double free, format strings, etc…
- Once you corrupt control flow, attacker can **run code of their choice**
  - Either directly or using ROP

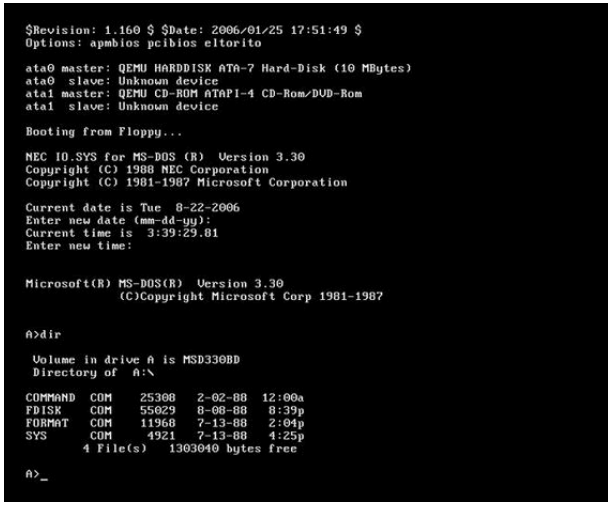
- So… is that it? Is your entire computer a charred piece of rubble sacrificed in the war on memory?
- Not necessarily
  What happens when you execute this?

```
char *p = NULL;

*p = 20;
```

Dues the whole system crash on a NULL pointer reference?

- MS-DOS/IBM DOS
  (circa the early 1990s)
  NULL ref will crash the whole system
  - Why? System provides no protection or isolation
  - No memory protection
    - All memory available to access
    - In fact, the interrupt vector table is stored at address o
  - No protected kernel
    - All operations available to programs



- Modern operating systems
  - No. At least minimal investment in secure design to provide protect processes from each other and kernel from processes

A step back: key secure design principals

- **Least privilege**
  - Only provide as much privilege to a program as is *needed* to do its job
- **Privilege separation**
  - Divide system into different pieces, each with separate privileges, requiring multiple different privileges to access sensitive data/code (AND vs OR)
- **Complete mediation**
  - Check **every** access that crosses a trust boundary against security policy
- **Defense in depth**
  - Use more than one security mechanism (belt and suspenders)
- **Simple designs are preferred**

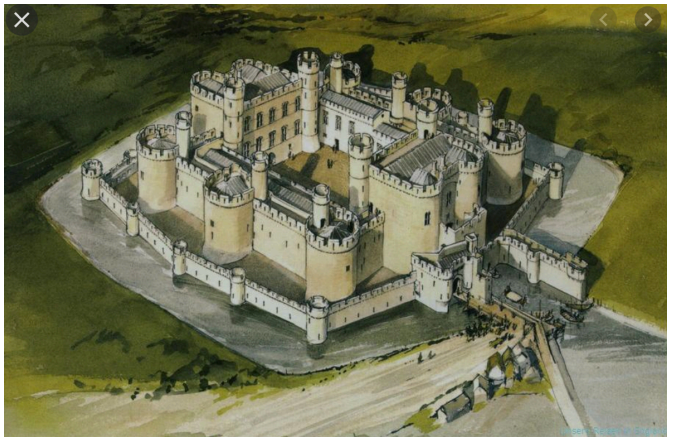
Security principles via metaphor

least privilege

privilege separation

Security principles via metaphor

Complete mediation



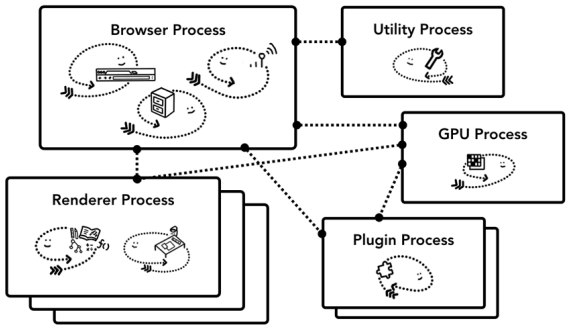
Defense in depth

So what does this mean for us?

- Every interface in our system is a *potential trust boundary*
  - **Processor-defined interfaces**: Memory reference, privileged instructions
  - **Software-defined interfaces**: system calls, file accesses, network messages, etc.
  - Lots of other levels of granularity
    - Between parts of a chip, between browser tabs, between users of a service, etc.
- We need to:
  - Separate functionality appropriately (least privilege & privilege separation)
  - Check access across trust boundaries (complete mediation)
  - Have safe ways to increase and decrease privilege where needed

Example: Modern browsers

- Browser process
  - Handles the privileged parts of browser (e.g., network requests, address bar, bookmarks, etc.)
- Renderer process
  - Handles untrusted, attacker content: JS engine, etc.
  - Communication restricted to RPC to browser/GPU process



- Many other processes (GPU, plugin, etc)

https://developers.google.com/web/updates/2018/09/inside-browser-part1

Returning to operating systems: How does this work in Linux/Unix?

- **Process abstraction**
  - Each user can have one or more processes
  - Processes have UIDs (User IDs) that indicate what they're allow to access
- **Process isolation**
  - Keep processes from touching each other's memory or state directly
- **User/Kernel privilege separation**
  - Limit privileged operations to operating system kernel
  - Check requests from user against security policy
  - Protect operating system kernel from user processes

Brief interlude: User permissions in UNIX

- Permissions in UNIX granted according to UID
  - A process may access files, network sockets, ….
- Each process has a User ID (UID)
  - Special user root (aka superuser) has UID o , can access any file
- Each file has an Access Control List (ACL)
  - Grants permissions to users according to UIDs and roles (owner, group, other)
  - Everything is a file!

- But how can the passwd program work?
  - Needs to write /etc/passwd file with new passwd
  - But normal users can't be allowed to write it

Brief interlude: User permissions in UNIX

- Really two UIDs
  - Real user ID (RUID)
    - Typically, the same as the user ID of the parent process
    - Used to determine which user started the process
  - Effective user ID (EUID)
    - Determines the current permissions for the process
    - Can *be temporarily different* from RUID

- `setuid` progams
  - A program can have a `setuid` bit set in its permissions
  - If so, the caller's EUID is set to the UID of the file
    - Temporary privilege elevation (normal user can suddenly have privilege of root)
    - Program needs to be written defensively and lower privs as soon as possible

Process Isolation

- Process boundary is a trust boundary
  - Any inter-process interface is part of the attack surface
- How are individual processes isolated from each other?
  - Which mechanism(s) ensures that one process cannot affect another, without proper authorization?
  - In particular, why can't one rogue process write over the memory of another process? (e.g., overwrite the return address on its stack)

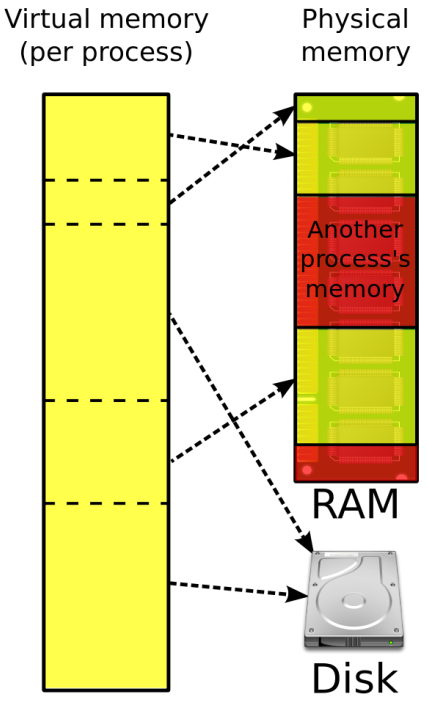
Virtual Memory

- Each process gets its own ***virtual address space***, managed by the operating system
  - A "personalized" view of the entire addressable memory space
  - As if this is the only process on the system
- Primary security mechanism for isolating processes from each other

https://c1.staticflickr.com/2/1603/26666393696_48c102e12f_b.jpg

Virtual Memory

- Memory addresses used by processes are ***virtual addresses***
- Virtual addresses are mapped by the operating system into ***physical addresses***, corresponding to actual storage locations
- ***Address translation*** is the mechanism for mapping virtual to physical addresses



https://en.wikipedia.org/wiki/Virtual_memory#/media/File:Virtual_memory.svg

Address Translation properties

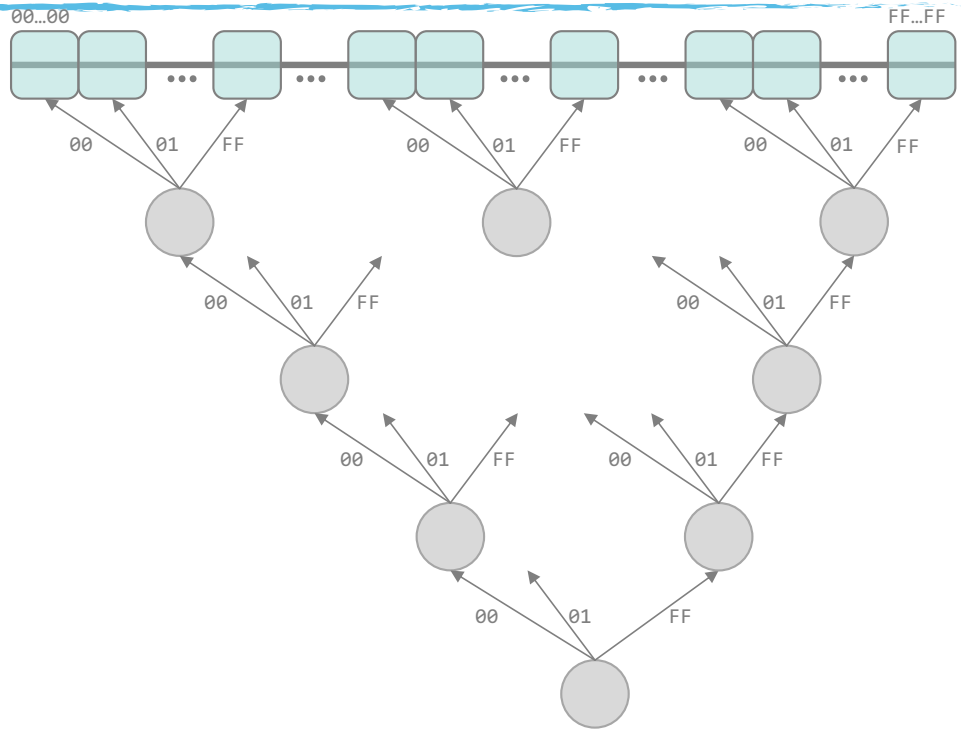
- Isolation
  - Provides (to a process, an operating system, a peripheral) a virtualized view of memory with limited visibility/access to the underlying memory space
  - i.e. you only get to even "name" the subset of the memory available to you
- Memory Access Polymorphism
  - Different access implementations for different memory regions/types
    - Rules for speculative access, out of order access, caching, backing store, etc.
    - Notably, access controls (i.e., read, write, execute, etc)

Making Address Translation work

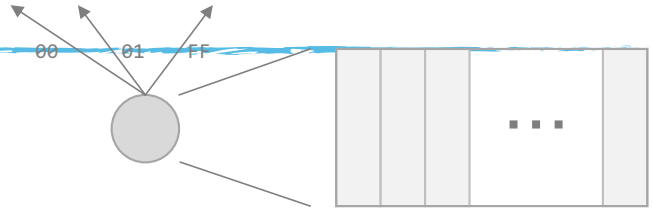
- Using 64-bit ARM architecture as an example…
- How to practically map arbitrary 64bit addresses?
  - 64 bits * 2^{64} (128 exabytes) to store any possible mapping
  - Hmm…

Making Address Translation work

- ***Page***
  - Basic unit of mapping granularity
  - Usually 4KB (or multiple thereof)
    - 2^{12}
  - Still 52 bits * 2^{52} (208 petabytes) to store any possible page mapping
- Multi-level ***Page Table***
  - Sparse tree of page mappings
  - Use virtual address as path through tree
  - Leaf node stores corresponding physical address
  - Each process gets its own tree
  - Root kept in a dedicated register: Translation Table Base Register



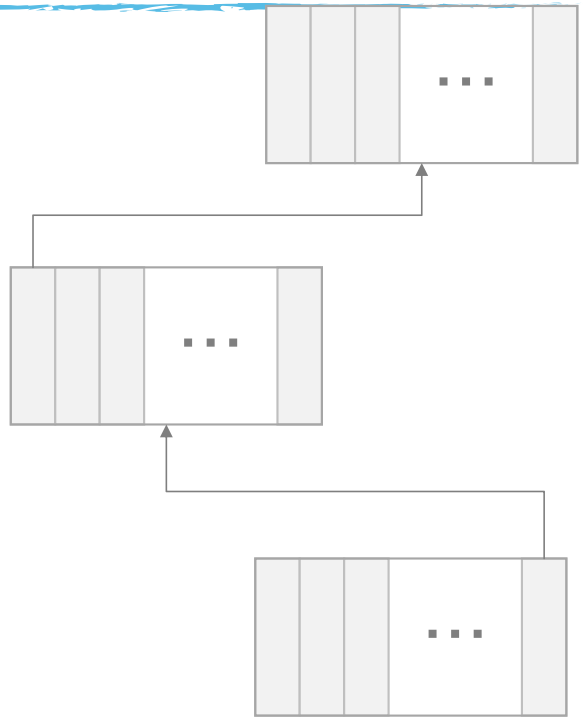
Page Tables



- Data structures used to store address mapping
  - Nodes of the tree
- Each table/node is:
  - Array of translation descriptors
  - Same size as a memory page

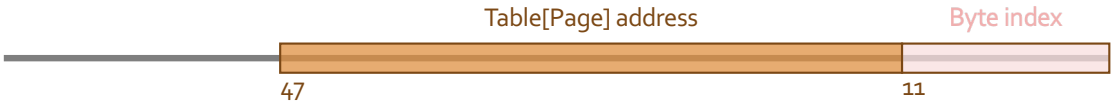
Page Tables

- Data structures used to store address mapping
  - Nodes of the tree
- Each table/node is:
  - Array of translation descriptors
  - Same size as a memory page
- Organized into a tree
  - Iteratively resolve n bits of address at a time
  - Each descriptor is either
    - Page descriptor (leaf node)
    - Table descriptor (internal node)

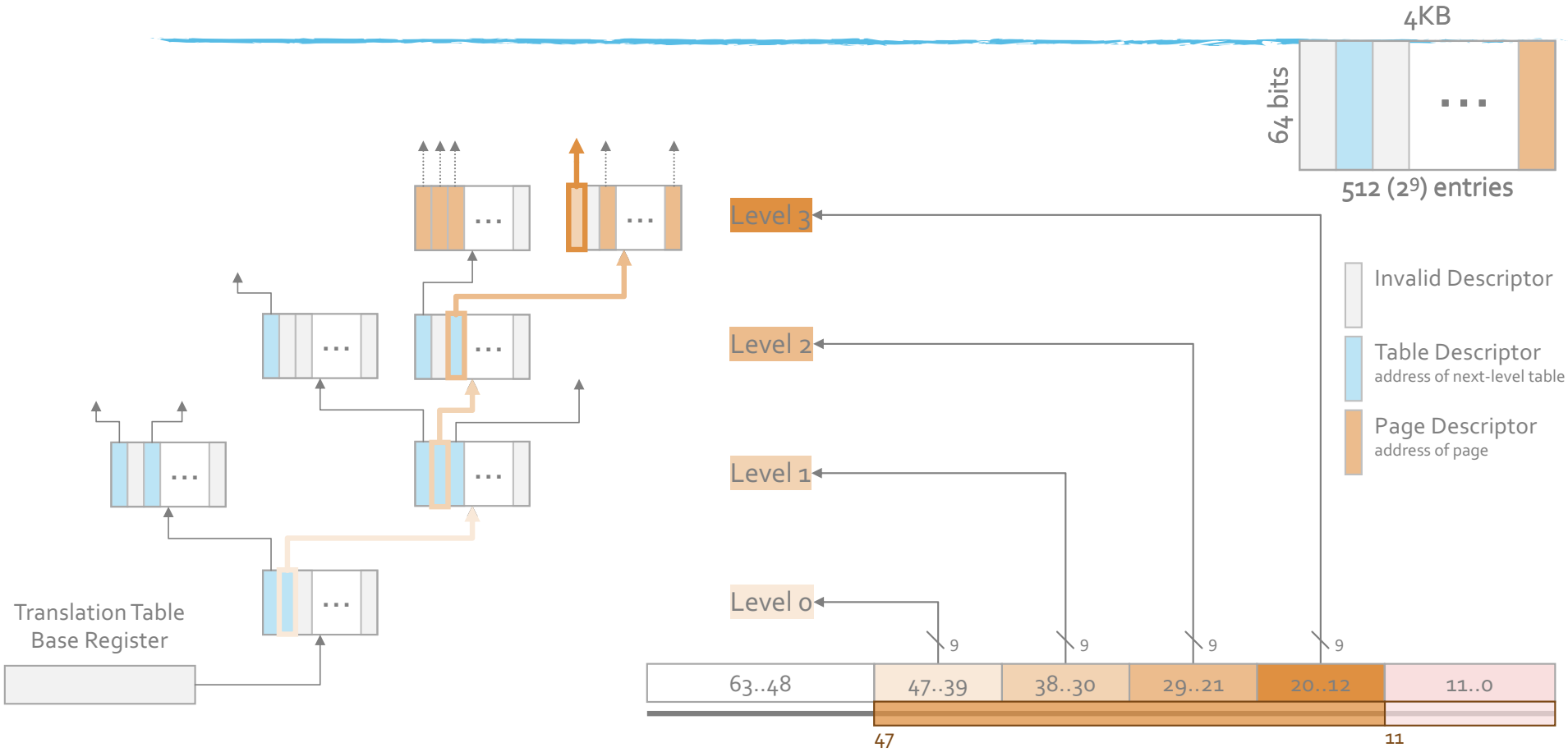


Page Tables

- In reality, the full 64bit address space is not used.
  - Working assumption: 48bit addresses



Page Table Walk



Address Translation

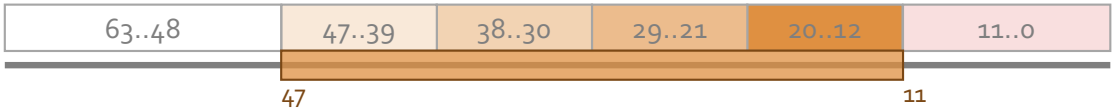
- Every memory access a process performs goes through address translation*
  - Load, store, instruction fetch
  - "complete mediation"
- That's a very expensive operation to perform several times for each instruction
  - Multiple optimizations in the page table structure (not covered here)
  - Translation Lookaside Buffer (TLB)
- *Assuming the system supports virtual memory. May not be available on low-end embedded systems or microcontrollers
  - **May not apply to every cache access. Architecture-dependent.

Translation Lookaside Buffer (TLB)

- Small cache of recently translated page addresses
  - Before translating a referenced address, the processor checks the TLB
    - Typically done in parallel with memory cache lookup
  - Identifies:
    - Physical page corresponding to virtual page (or that page isn't present)
    - If page mapping allows the *mode of access* (access control)

Access Control

- Not everything within a processes' virtual address space is equally accessible
- Page descriptors contain additional access control information
  - Read, Write, eXecute permissions
    - This is how we get DEP/W^X on the stack/heap
  - Set by the operating system
  - If a program attempts the wrong mode of access (e.g., an execute to an address on a page without the execute mode set) the processor will generate a fault
  - Aside: where do they store that information?



Ok, but what about the OS?

- Good question! We've protected Processes from touching each other's memory (unless we want them to) but those protections are provided by the OS
- What is the attack surface of the OS?
  - Memory accesses
  - Privileged instructions
  - System calls and faults
  - Device accesses (e.g., Direct Memory Access from GPU/NIC/Disk Controller/etc)
- Need a combination of hardware and software protection
  - Hardware for interfaces at the granularity of instructions (e.g., setting the page table base register)
  - Software for interfaces at the granularity of system abstractions (e.g., ensuring that the read() system call can't access memory not available to process)

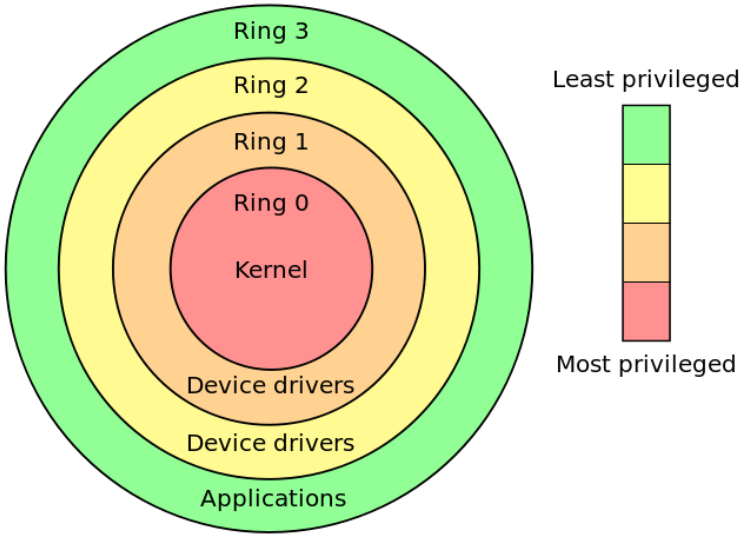
Privilege Levels

- Multiple privilege levels
  - Processor states
  - Typically, just two used by the operating system:
    - Privileged and Non-privileged
    - Kernel Mode and User Mode
    - Supervisor and Normal

    } Synonyms
- Processor operates at some privilege level
  - Protected system register holds value of *current privilege level*
- Sensitive system operations require certain *minimum privilege level*

Intel Privilege Levels

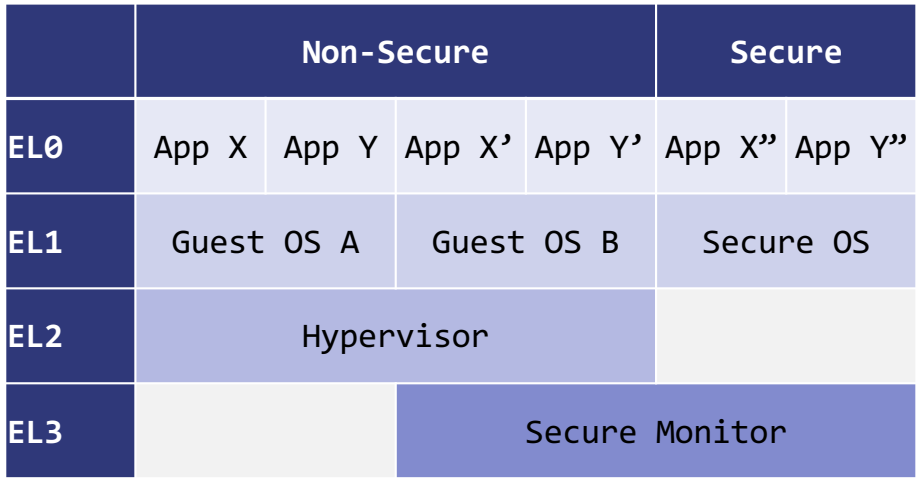
- 4 ***rings*** (2 used by OS)
  - `Ring 0` most privileged
  - Ring 3 is least (user programs)
  - "Nothing will ever be more privileged than the kernel"
    - This proves not to be true over time… subsequent processors include multiple modes more privileged than ring 0



https://en.wikipedia.org/wiki/Protection_ring

ARM Privilege Levels

- 2 ***worlds***
  - Secure and Non-Secure
- 4 ***exception levels*** (2 used by OS)
  - EL0 least privileged
  - "Nothing will ever be less trusted than a user mode application"
    - This proves not to be true (user-level sandboxing of code within a browser)



Privilege Levels

- Boundary between privilege levels is a trust boundary
  - Any cross-privilege interface is part of the attack surface
- Dedicated mechanisms for safely changing privilege level are needed
  - Anyone can drop privileges, elevating is harder

Privilege Levels

- To enter more privileged state the process:
  - Prepares arguments, including id of the desired entry point and
  - Executes a special instruction that initiates the transfer
- Each privilege level defines a set of entry points for less privileged callers
  - Also, specific registers for passing arguments
    - Typically pointers to more data in less privileged memory
  - These are the only valid entry points when calling from less privileged state
    - The higher-privileged callee is in control of what code is executed
- Details vary by architectures, but core concept is consistent
- This is how you get system calls

System Calls

- User-mode process may need frequent assistance from kernel
  - I/O operations (files, network, devices, etc.)
  - System information (time, environment, etc.)
  - Process control (fork, signals, mutex, etc.)
- Kernel has its own page table (for its code and data)
  - Also maintains the page tables for all other processes
- Normally, switching between a user-mode process and the kernel would require switching between the respective process' address space and the kernel's
  - This potentially requires flushing TLBs, etc -- slow
- But system calls are common, need to be fast and efficient

Kernel Mapping

- To make system calls fast, kernel's virtual memory space is mapped into *every process*, but made inaccessible in usermode
  - This way, system calls are fast, just switch into Kernel mode and go; memory is all in the same place

- Separate permission bits in page tables
  - Unprivileged (usermode): UR, UW, UX
  - Privileged (kernel): PR, PW, PX

Kernel Mapping

- How does the processor know which page table to use when referencing a virtual address?
- Not based on the current privilege level
  - Both memory spaces are simultaneously mapped
- Based on the value of the address
  - High addresses are translated using kernel's page table, low addresses using usermode page table
- Access is controlled by permissions specified within the page table

Kernel Mapping

- When a process makes a system call and transfers control to the kernel:
  - Kernel's memory space is already mapped
  - Calling process' memory space remains mapped and accessible
- On a process switch, the userland page table is swapped
  - Translation Table Base Register updated
- All processes share the same kernel mapping

Kernel Mapping

- What kind of access should kernel have to usermode memory?

Kernel Security

- Threat model:
  - Confidentiality and integrity of kernel memory and control flow must be protected from compromise by usermode processes
  - All usermode processes are untrusted and potentially malicious
- Operating model:
  - Usermode processes make frequent calls into the kernel, with data passing back and forth
    - Example: network packets, file contents, etc.

Kernel Security

- Kernel must be careful to keep track of whether it is operating on kernel data or usermode data
  - Avoid becoming a ***confused deputy***, i.e. being manipulated into abusing its privileges (called an **elevation of privilege** or **privilege escalation** attack)
- A usermode process may trick the kernel into writing attacker-controlled data into kernel memory or leaking kernel memory to the attacker

Simple example

- **read()** system call
  - **ssize_t read(int** *fd***, void ****buf***, size_t** *count***);**
  - Reads `count` bytes from the file specified by file descriptor `fd`, and write it into the buffer at address `buf`
- What could happen if the attacker calls read() with `buf`="the address of a sensitive data structure in the kernel"?

Kernel Security

- Separate mechanisms for operating on usermode and kernel data
- Software:
  - `copy_to_user()` and `copy_from_user()`
    - Safely copy data between user and kernel buffers
  - Special care needs to be taken with:
    - ***Time Of Check vs Time Of Use (TOCTOU)***
    - Nested pointers
      - e.g., pointers to data structures with pointers in them that the kernel will use
    - Uninitialized variables
      - Structure padding

Example: NULL Dereference

- What happens here?

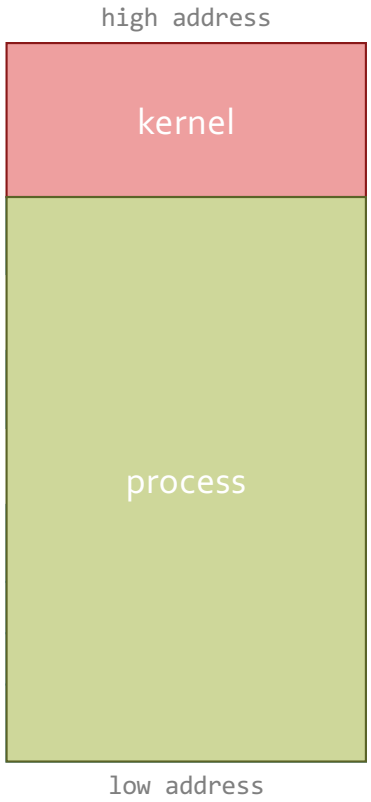
```
char *p = NULL;


*p = 20;
```

- Dereferencing NULL pointers can lead to a crash (Denial of Service).
- However, there is more to it…
  - After all, there is an actual address `0`. What's there? What happens when we access it? Why do we crash?

NULL Dereference & Return-to-User

- Assume attacker is a userland process trying to attack the kernel
  - ***Elevation of privilege***
- What if that process mapped page `0`?
- What happens if this process manages to trigger a NULL pointer dereference in the kernel?
  - Instead of crashing, the kernel will use attacker-controlled data on page `0`.
- This is known as a ***Return-to-User*** attack.



NULL Dereference & Return-to-User

- Aside: compilers are too smart for our own good
  - Disappearing NULL checks

```
static unsigned int tun_chr_poll(
        struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;
    //...

    if (sock_writeable(sk)
        || (!test_and_set_bit(SOCK_ASYNC_NOSPACE,
                              &sk->sk_socket->flags)
        && sock_writeable(sk)))
                mask |= POLLOUT | POLLWRNORM;
```

https://lwn.net/Articles/342330/

NULL Dereference & Return-to-User

- One countermeasure:
  - Prevent unprivileged allocation of page `0` (or any page up to some minimum).
    - Reserves low addresses as "guard" pages. Attempts to access trigger memory access violation.
    - Present on most modern operating systems.
    - But how big a region? Why does it matter?

Kernel Security

- Separate mechanisms for operating on usermode and kernel data
- Hardware:
  - Special load and store instructions that operate as if in usermode, even when processor is executing in kernel mode.
    - To prevent inadvertently overwriting sensitive data
  - ARM
    - Privileged Access Never (PAN) processor state that prevents kernel mode access to usermode data
      - To prevent inadvertently leaking sensitive data
    - Privileged eXecute Never (PXN) page permission
      - To mark usermode pages non-executable in kernel mode
  - Intel (equivalent)
    - Supervisor Mode Access Protection (SMAP), SM Execution Protection (SMEP)

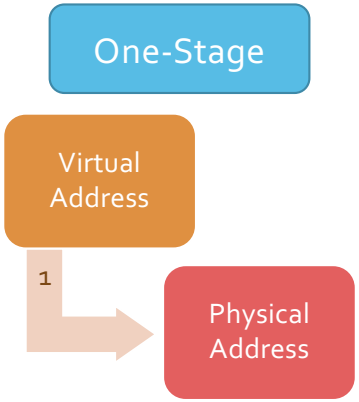
Virtual machines

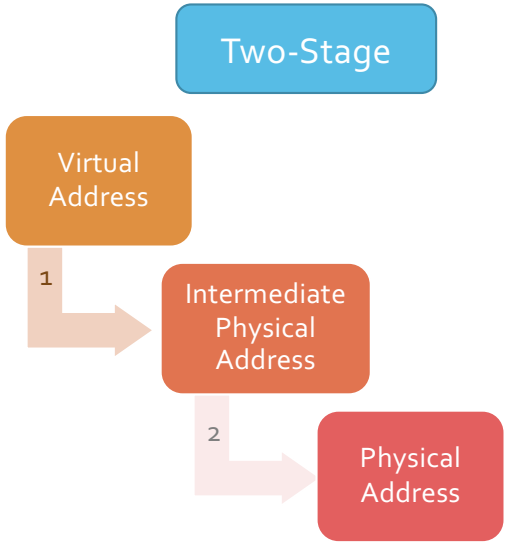
- So far we've discussed a situation with isolated user processes but sometimes we want to provide isolation between OSs
  - Why would we do that?

- The hardware running the OS is virtualized – a virtual machine (VM)
  - Each OS is oblivious to this happening (mostly) and still provides isolation between processes the way it used to
  - **Hypervisor** implements VM environment and provides isolation between VMs
    - Thing of hypervisor as the OS for the Oses
    - Each OS thinks it is running on a physical machine, just like each process thinks they have all the memory

Virtualization

- Multiple stages of address translation to support virtualization
  - Nested page tables
  - Modern hardware has special support for this





Lots of details

- Vary between versions of processor, hypervisor & operating system
  - Lots of optimizations

- Also, whole other range of hardware protections now for "enclaves"
  - E.g., ARM TrustZone, Intel SGX, iPhone SEP (separate core)
  - Protected *physical* memory can only be accessed by code in enclave (even hypervisor can't see it)
  - Guards against compromised operating system

Summary

- Process isolation
  - Hardware support (MMU)
  - Provides separate address spaces to different processes
  - Control modes of access to memory (i.e., R,W,X)
- User/Kernel Privilege separation
  - Processor privilege modes used to limit access to sensitive instructions/memory
  - Careful checking of syscall interface from user processes
  - Map kernel into all process address spaces to make system calls fast
    - Next class we'll talk about why we can't do that anymore
- Virtual machines
  - Same idea, but add another level of isolation (hypervisor -> OS -> process)

Next class

- Side channels
  - How we bypass isolation without violation control flow integrity