CSE 127: Computer Security

WI24

Lecture 6 - Isolation

Slide Credit: Kirill Levchenko, Stefan Savage, Stephen Checkoway, Hovav Shacham, David Wagner, Deian Stefan, Dan Boneh, and Zakir Durumeric , Nadia Heninger, George Obiado, Earlence Fernandes, Imani Munyaka

Announcements

Assignment 2 Released
Start Now - very serious

OH on Calendar

Book Chapters and Topics

Topic	Book	Chapter
Security Mindset	Security Engineering	1
x86 assembly	Hacking…	
Buffer Overflow	Computer Security	10
ROP	Hacking	
Control Flow		
Isolation		

Today

- Return-oriented programming
- Control flow integrity
- Heap corruption
- **Isolation**

Today

- Understand basic principles for building secure systems
- Understand mechanisms used to build secure systems

Running untrusted code

We often need to run buggy or untrusted code.

Running untrusted code

We often need to run buggy or untrusted code.

- Desktop applications
- Mobile apps
- Untrusted user code
- Web sites, Javascript, browser extensions
- PDF viewers, email clients
- VMs on cloud computing infrastructure

Systems must be designed to be resilient in the face of vulnerabilities and malicious users.

Principles of secure system design

- Least privilege
- Privilege separation
- Complete mediation
- Fail safe/closed
- Defense in depth
- Keep it simple

Principle of Least Privilege

- Users should only have access to the data and resources needed to provide authorized tasks

Principle of Least Privilege

- Users should only have access to the data and resources needed to provide authorized tasks

  Examples:

-
  - Faculty can only change grades for classes they teach
  - Only employees with background checks have access to classified documents

Principle of privilege separation

Least privilege requires dividing a system into parts to which we can limit access

- Break system into compartments
- Ensure each compartment is isolated
- Ensure each compartment runs with least privilege
- Treat compartment interface as trust boundary

Example: Multi-user operating system

In this system:

- Users can execute programs/processes
- Processes can access resources

What's the threat model?

What are the assets?

What security properties do we want to preserve?


CLASSIFIED
CLASSIFIED
CONFIDENTIAL
TOP SECRET
TOP SECRET
TOP SECRET


ENTRY RESTRICTED
TO
MCCC AND DMCCC
ON DUTY


EMERGENCY EXIT
Push Until Alarm Sounds
Door Can Be Opened
In 15 Seconds

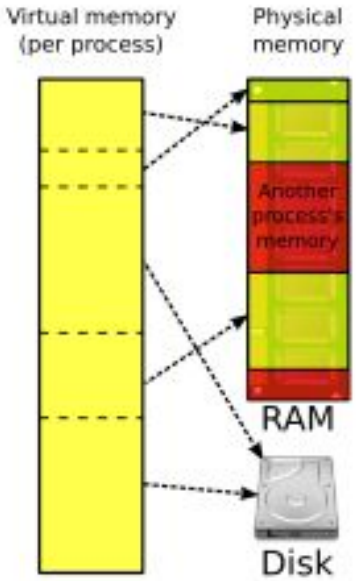
Multi-user OS security properties

- **Memory isolation**
  - Process should not be able to access another process's memory

- **Resource isolation**
  - Process should only be able to access certain resources

Process memory isolation

- How are individual processes memory-isolated from each other?
  - Each process gets its own virtual address space, managed by the operating system
- Memory addresses used by processes are virtual addresses (VAs) not physical addresses (PAs)
  - The CPU memory management unit (MMU) does the translation



Principle of complete mediation

- Every memory access goes through address translation
  - Load, store, instruction fetch
  - Virtual memory allows address space much larger than physical memory
  - Also means that operating system mediates all process memory accesses and enforces access control policy

Resource isolation in the Unix security model

In Unix, everything is a file: files, sockets, pipes, hardware devices…

- Permissions to access files are granted based on user IDs
  - Every user has a unique UID
- Access Operations: Read, Write, Execute
- Each file has an access control list (ACL)
  - Grants permissions to users based on UIDs and roles (owner, group, other)
  - root (UID 0) can access everything

Role-Based Access Control

In a general access control system we can specify permissions in a matrix:

	hw/	exams/	grades/	lectures/
cse127-instr	r/w	r/w	r/w	r/w
cse127-tas	r/w	read	-	r/w
cse127-students	read	-	-	read
cse-students	-	-	-	read

Capabilities vs. ACLs

ACL: System checks where subject is on list of users with access to the object.

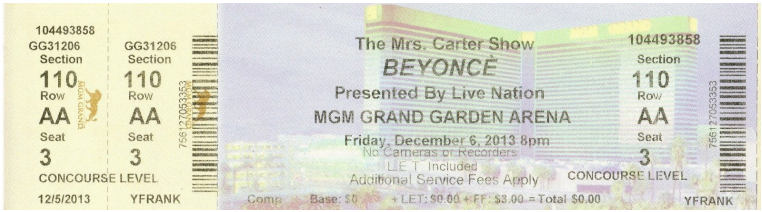
- Permissions stored by column of access control matrix

Capabilities: Subject presents an unforgeable ticket that grants access to an object. System doesn't care who subject is, just that they have access.

- Row of access control matrix

I haven't started PA 2.

I started PA 2.



Unix file permissions are a simplified ACL

```
nadiah@login:/cse/htdocs/classes/wi21/cse127-a$ ls -l
total 32
-rw-rw-r-- 1 nadiah cse127-a-wi 18660 Jan 14 00:34 index.html
drwxrwxr-x 2 nadiah cse127-a-wi 4096 Jan 13 08:42 pa
drwxrwxr-x 2 nadiah cse127-a-wi 4096 Jan 13 19:57 resources
drwxrwsr-x 3 nadiah cse127-a-wi 4096 Jan 14 00:34 slides
```

Permissions grouped by user owner, group owner, other

Operations: read, write, execute

directory

Unix file permissions are a simplified ACL

```
nadiah@login:/cse/htdocs/classes/wi21/cse127-a$ ls -l
total 32
-rw-rw-r-- 1 nadiah cse127-a-wi 18660 Jan 14 00:34 index.html
drwxrwxr-x 2 nadiah cse127-a-wi 4096 Jan 13 08:42 pa
drwxrwxr-x 2 nadiah cse127-a-wi 4096 Jan 13 19:57 resources
drwxrwsr-x 3 nadiah cse127-a-wi 4096 Jan 14 00:34 slides
```

Permissions grouped by user owner, group owner, other

Operations: read, write, execute

Unix file permissions are a simplified ACL

```
nadiah@login:/cse/htdocs/classes/wi21/cse127-a$ ls -l
total 32
-rw-rw-r-- 1 nadiah cse127-a-wi 18660 Jan 14 00:34 index.html
drwxrwxr-x 2 nadiah cse127-a-wi 4096 Jan 13 08:42 pa
drwxrwxr-x 2 nadiah cse127-a-wi 4096 Jan 13 19:57 resources
drwxrwsr-x 3 nadiah cse127-a-wi 4096 Jan 14 00:34 slides
```

Permissions grouped by user owner, group owner, other

Operations: read, write, execute

Process UIDs

Process permissions are determined by UID of user who runs it unless changed.

- Real user ID (RUID)
  - Used to determine which user started the process
  - Typically same as the user ID of parent process
- Effective user ID (EUID)
  - Determines the permissions for process
  - Can be different from RUID (e.g. because setuid bit on the file being executed
- Saved user ID (SUID)
  - EUID prior to change

setuid

- A program can have a setuid bit set in its permissions
- This impacts fork and exec
    - Typically inherit three IDs of parent
    - If setuid bit set: use UID of file owner as EUID

-rwsr-xr-x 1 root root 54256 Mar 26 2019 /usr/bin/passwd

setuid

- A program can have a setuid bit set in its permissions
- This impacts fork and exec
    - Typically inherit three IDs of parent
    - If setuid bit set: use UID of file owner as EUID

-rwsr-xr-x 1 root root 54256 Mar 26 2019 /usr/bin/passwd

- Temporary privilege elevation (normal user can suddenly have privilege of root)
- Program needs to be written defensively and lower privs as soon as possible Can be super dangerous: need to think about least privilege

setuid, setgid, and sticky bit

- There are three bits:
  - setuid: set EUID of process to ID of file owner
  - setgid: set effective group ID of process to GID of file

setuid, setgid, and sticky bit

- There are three bits:
  - setuid: set EUID of process to ID of file owner
  - setgid: set effective group ID of process to GID of file
- sticky bit
  - on: Only file owner, directory owner, and root can rename or remove file in the directory
  - off: If user has write permission on directory, can rename or remove files, even if not owner

```
drwxrwxrwt 10 root root 12288 Jan 18 20:55
```

Overview of Unix file security mechanism

- **Pro**: Simple and flexible
- **Con**:
  - Coarse-grained
  - Nearly all system operations require root access.
  - In practice, common to run many services as root. This violates principle of least privilege and increases attack surface.

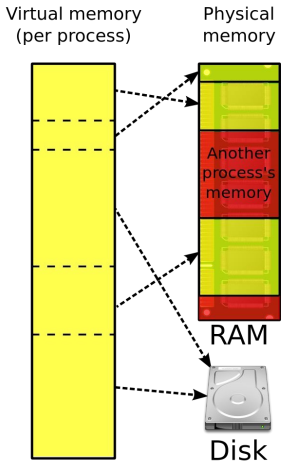
Virtual Memory

- 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

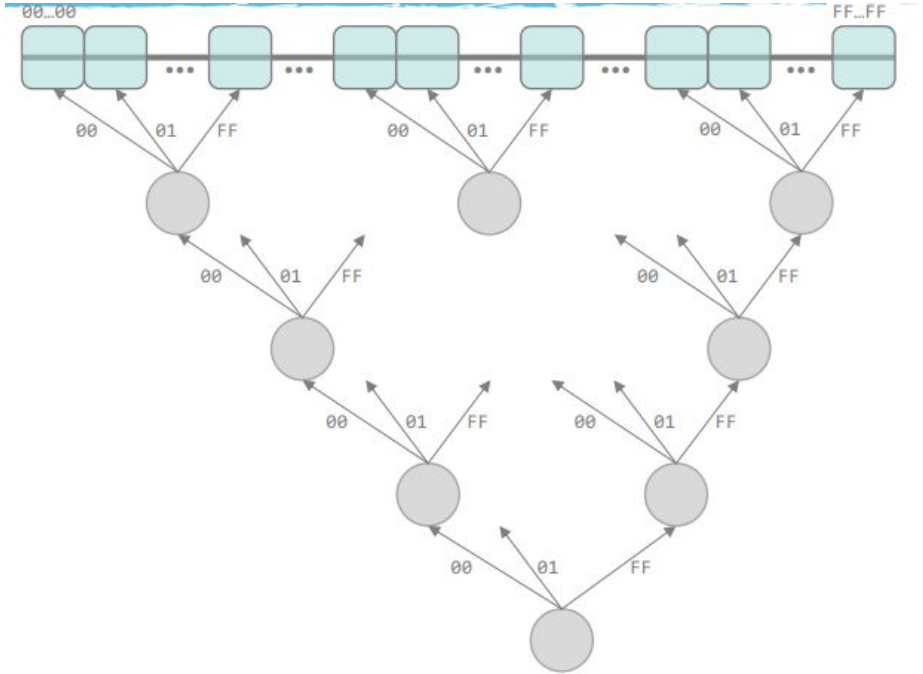
- Isolation
  - Provides (to a process, an operating system, a peripheral) a virtualized view of memory with limited visibility/access to the underlying memory space
  - i.e. you only get to even “name” the subset of the memory available to you
- Memory Access Polymorphism
  - Different access implementations 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 64 bit addresses?
  - 64 bits * 2^{64} (128 exabytes) to store any possible mapping
  - Hmm…

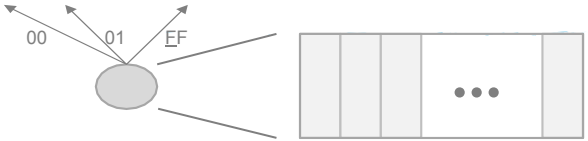
Making Address Translation work

- ***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



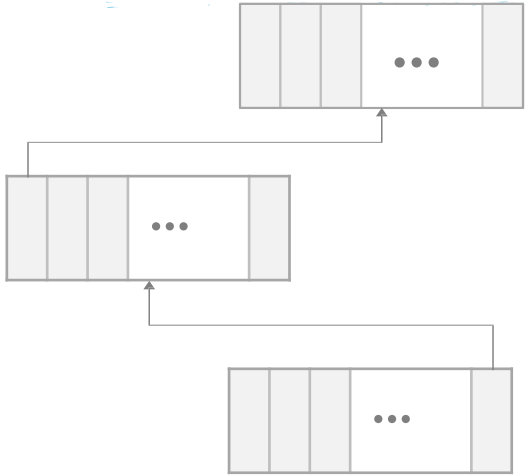
Page Tables

- 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



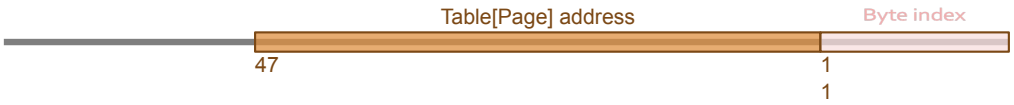
Page Tables

- 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
    - Table descriptor (internal node)
    - Page descriptor (leaf node)

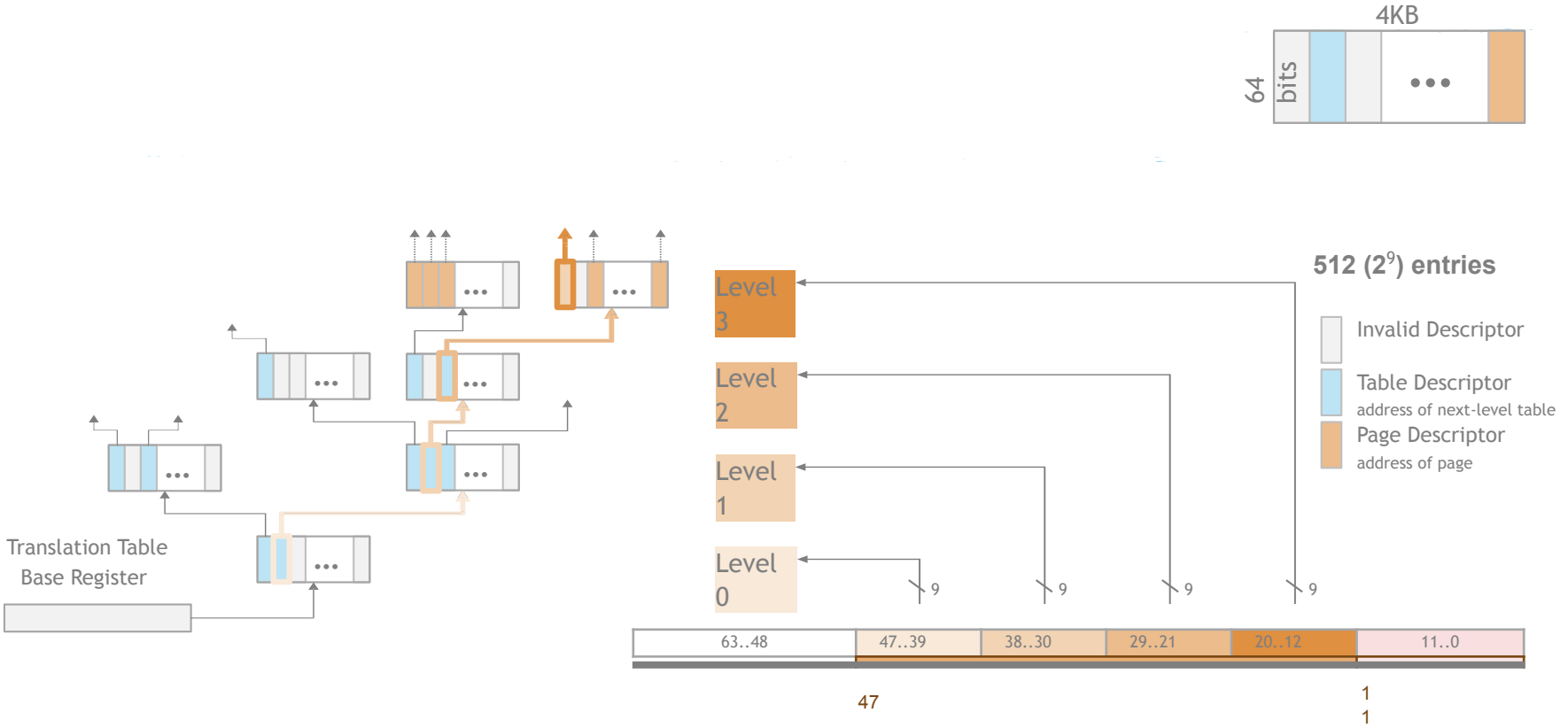


Page Tables

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



Page Table Walk



Address Translation

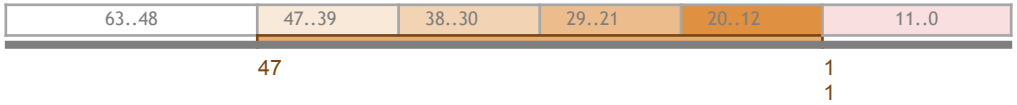
- Every memory access a process performs goes through address translation*
  - Load, store, instruction fetch
  - “complete mediation”
- That’s a very expensive operation to perform several times for each instruction
  - Multiple optimizations in the page table structure (not covered here)
  - Translation 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 the page isn't present in memory)
    - 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 and/or user programs (e.g., mprotect())
  - If a program attempts the wrong mode of access (e.g., fetch an instruction from an address on a page without execute mode set) the processor will generate a **fault**
  - Aside: where do you think they store the access mode information in the 64bit descriptor?



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 instruction
  - 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 translation 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 or that you can't write to a file for which you don't have write permissions)

Privilege Levels

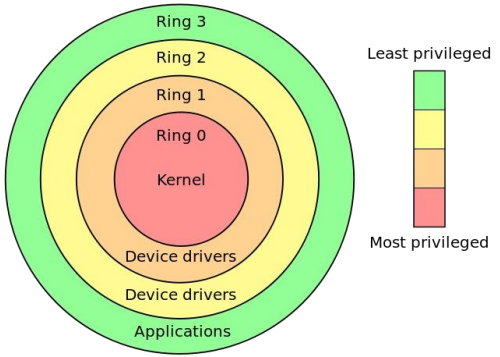
- Multiple privilege levels
  - Processor states
  - Typically, just two used by the operating system:
    - Privileged and Non-privileged
    - Kernel Mode and User Mode
    - Supervisor and Normal

    } Synonyms

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

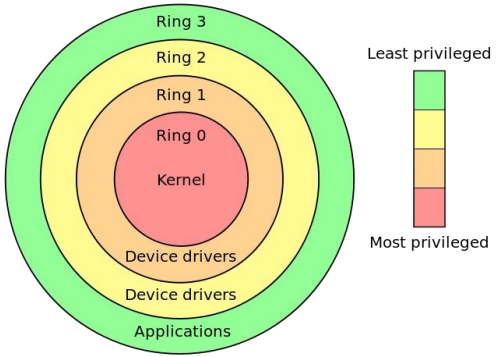
Kernel isolation

- Kernel is isolated from user processes
  - Separate page tables
  - Processor privilege levels ensure user space code cannot use privileged instructions
- Interface between userspace and kernel: system calls



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 Intel processors include **multiple** modes more privileged than ring 0



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

Example: Smartphone OS design

Does the threat model for a smartphone differ from a desktop?

What's the threat model?

What are the assets?

What security properties do we want to preserve?

Android process isolation

- Android uses Linux and sandboxing for isolation
- Each app runs under its own UID
- Apps can request permissions, which are basically capabilities
- Reference monitor checks permissions on intercomponent communications

Software fault isolation (SFI)

Placing untrusted components in their own address space provides isolation, but comes with overhead.

Software fault isolation wants to partition apps running in the same address space.

- Kernel modules should not corrupt kernel
- Native libraries should not corrupt

Software fault isolation (SFI)

Placing untrusted components in their own address space provides isolation, but comes with overhead.

Software fault isolation wants to partition apps running in the same address space.
- Kernel modules should not corrupt kernel
- Native libraries should not corrupt

JVM SFI approach: Partition process memory into segments
- Memory isolation: Instrument all loads and stores
- Control flow integrity: Ensure all control flow is restricted to CFG that instruments loads/stores
- Complete mediation: Disallow privileged instructions
- Syscall-like interface between isolated code

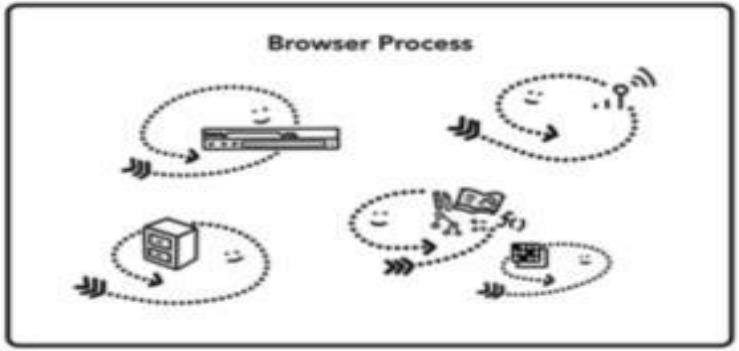
Example: Browser design

What's the threat model?

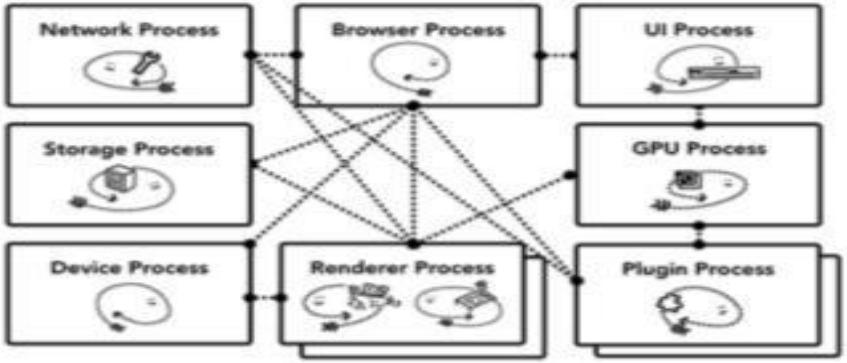
What are the assets?

What security properties do we want to preserve?

Chrome Security Architecture



Pre-2006



Modern

Modern Browswer Security Model

- Browser process
  - Handles the privileged parts of browser (network requests, address bar, bookmarks)
- Renderer process
  - Handles untrusted attacker content: JS engine, DOM, etc.
  - Communication restricted to remote procedure calls
- Many other processes (GPU, plugin, etc.)

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 what a **system call** is

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
- Switch between two usermode processes requires switching between the respective process' address spaces
  - This potentially requires flushing TLBs, etc – can be slow
  - Thankfully isn't that frequent (10ms or more between process switches)
- But system calls are very common, need to be fast and efficient

Process confinement: system call interposition

Observation: To damage a host system (e.g. make permanent changes), an app must make system calls

- To delete or overwrite files: unlink, open, write
- For network attacks: socket, bind, connect, send

Idea: Monitor app's system calls and block unauthorized calls

Key component: Reference monitor

- Mediates requests from applications
  - Enforces confinement
  - Implements a specified protection policy
- Must always be invoked
  - Every application must be mediated
- Tamperproof
  - Reference monitor cannot be killed, or if killed then monitored process is killed too
- Small enough to be analyzed and validated

System Call Interposition in Linux

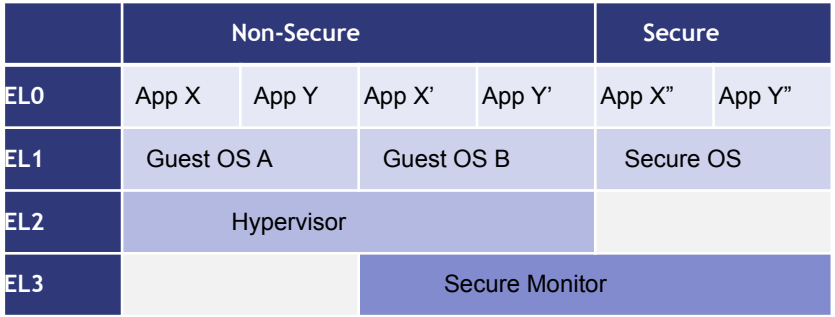
seccomp-bpf: Linux kernel facility used to filter process syscalls

- Syscall filter written in the BPF language
- Used in Chromium, Docker containers...

- Container: process-level isolation
- Container prevented from making syscalls filtered by seccomp-bpf

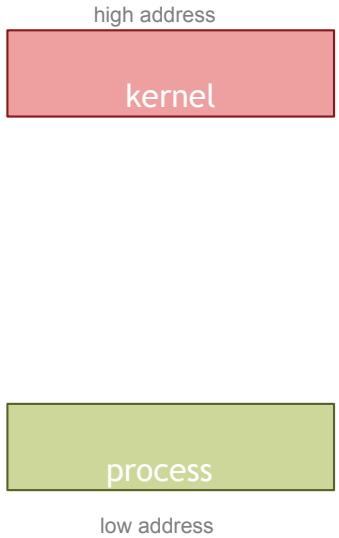
ARM Privilege Levels

- 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)



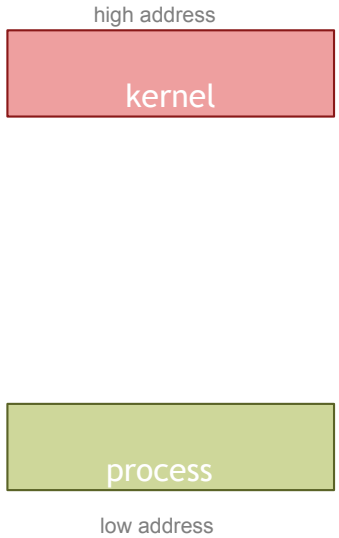
Kernel Mapping

- To make system calls fast, kernel's virtual memory space is mapped into *every process*, but made ***inaccessible*** when in user mode
  - This way, system calls are fast, just switch into Kernel mode and go; memory is all in the same place (i.e., pointers work)
- Thus, separate permission bits in page tables
  - Unprivileged (usermode): UR, UW, UX
  - Privileged (kernel): PR, PW, PX



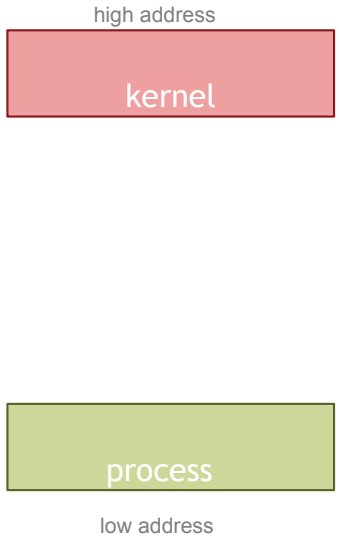
Kernel Mapping

- When a process makes a system call and transfers control to the kernel:
  - Kernel's memory space is already mapped
    - No change to the Translation Table Base Register
  - Calling process' memory space remains mapped and accessible
- On a process switch, the userland page table is swapped
  - Translation Table Base Register updated
  - But all processes share the same kernel mapping



Kernel Mapping

- What kind of access should kernel have to user mode 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

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)***
      - Remember there may be multiple threads/processors running at the same time
    - Nested pointers
      - e.g., pointers to data structures with pointers in them that the kernel will use

Example: NULL Dereference

- What happens here?

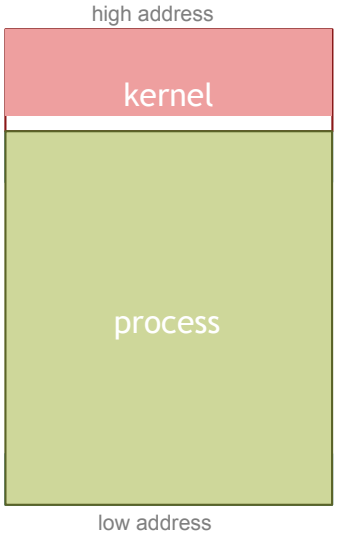
```
char *p = NULL;

*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 (common) 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)

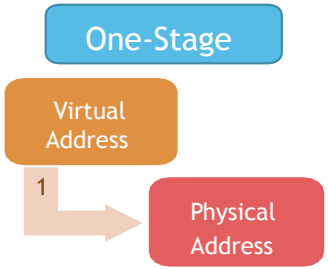
Virtual machines

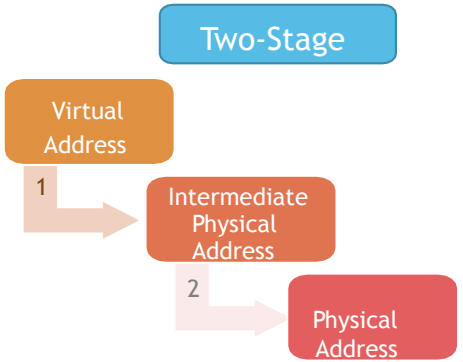
- So far we've discussed a situation with isolated user processes but sometimes we want to provide isolation between OSs
  - Why would we do that?

- 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

Virtual Machines

- Virtual machines allow a single piece of hardware to emulate multiple machines
- Useful for cloud computing and also for isolation
- Intel has hardware support for x86 virtualization: VMM support in hardware so that operating system can be run in ring 0 without requiring VMM intervention for syscalls

VMs and Isolation

VM Isolation for the cloud:

- VMs from different customers may run on the same machine
- Hypervisor tries to isolate VMs to minimize information leaks

VM Isolation for the end user:

- Qubes OS: A desktop OS where everything is a VM
- Every window frame UI identifies VM source

Hardware isolation: Secure enclaves

- Intel Software Guard eXtensons (SGX)
  - Runs trusted code in an *enclave*
  - Enclave memory encrypted and only decrypted in the CPU
  - Can’t be read even by malicious OS

- Why do we want to protect a program against a malicious OS?

Hardware isolation: Secure enclaves

- Intel Software Guard eXtensons (SGX)
  - Runs trusted code in an *enclave*
  - Enclave memory encrypted and only decrypted in the CPU
  - Can't be read even by malicious OS
- Why do we want to protect a program against a malicious OS?

  Example applications:
  - DRM (Digital Rights Management)
  - Secure remote computation
  - Protecting crypto keys or sensitive information

iOS Secure Boot

Apple devices use a secure enclave coprocessor as part of its boot chain.

Hardware-based root of trust: code and code-verifying keys baked into boot ROM (read-only memory).

Each step of the boot process verifies that the bootloader, kernel are signed by Apple.

What are the positives and negatives of this kind of design?

Physical isolation: Air gap

To ensure that a misbehaving app cannot harm the rest of the system, you could run it on physically isolated system.

What kinds of systems would you do this for?

What are the downsides?

Principles: Fail closed

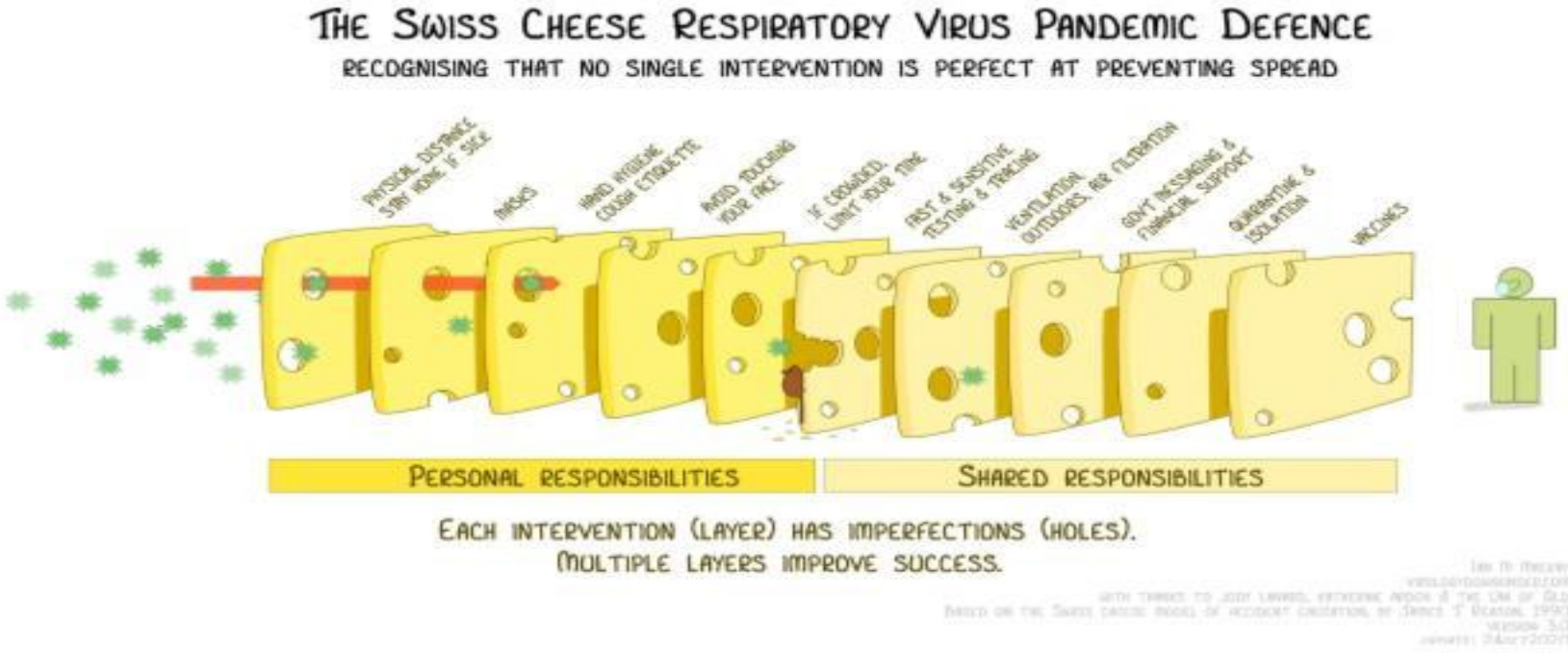
Fail Open systems allow access
Fail Closed systems that block access.

What's the problem with failing open?

Why might system designers choose to fail open?

Principles: Defense in depth

We do not expect any of our defenses to be perfect.



Principles: Keep it simple

We *have* to trust some components of our system.

In general keeping the Trusted Computing Base small and simple makes it easier to verify.

- In theory a hypervisor can be less complex than a full host operating system.
- A small OS kernel has less attack surface than one with many features.

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)

Software and hardware isolation techniques

- Memory isolation
- Resource isolation and access control
- System call interposition
- Sandboxing
- Containers
- Virtualization
- Secure enclaves
- Physical air gap

Lesson: Complete isolation is often inappropriate; applications need to communicate through regulated interfaces

Principles of secure system design

- Least privilege
- Privilege separation
- Complete mediation
- Fail safe/closed
- Defense in depth
- Keep it simple

Next time on CSE 127…