Lecture 20, May 15, 2026

ECS 235B, Foundations of Computer and Information Security

Spring Quarter 2026

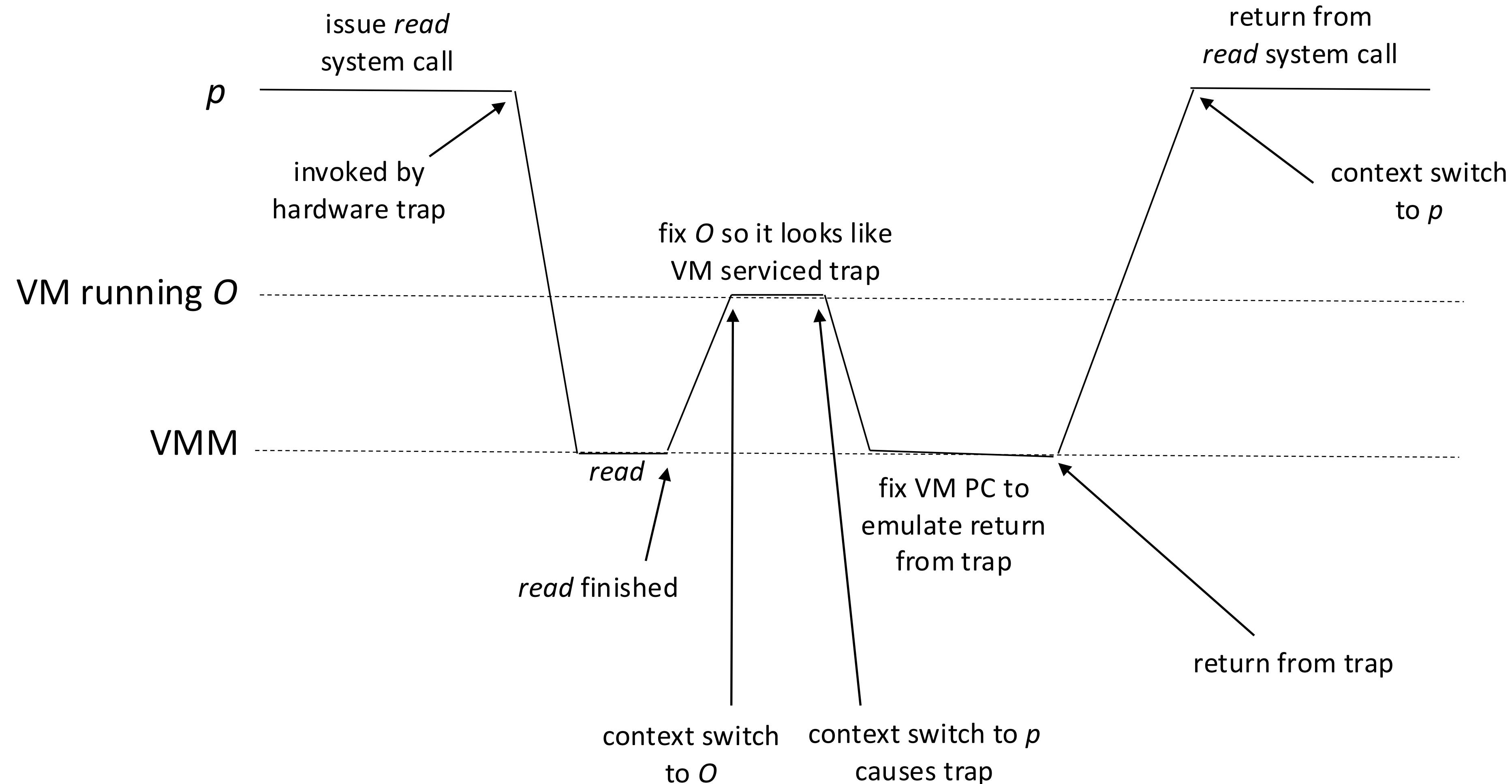
Virtual Machine Privileged Instructions

1. VMM runs VM with operating system *O*, which is running process *p*
    - *p* tries to read, so privileged operation traps to hardware
2. VMM invoked, determines trap occurred in VM
    - VMM updates state of VM to make it look like hardware invoked *o* directly, so *O* tries to read, causing trap
3. VMM does read
    - Updates VM to make it seem like *O* did read
    - Transfers control to *O*

Privileged Instructions

4. *O* tries to switch context to *p*, causing another trap
5. VMM updates VM running *O* to make it appear *O* did context switch successfully
   - Transfers control to *O*, which (as *O* apparently did a context switch to *p*) returns control to *p*

Privileged Instructions



Privilege and VMs

- *Sensitive instruction* discloses or alters state of processor privilege
- *Sensitive data structure* contains information about state of processor privilege

When Can VM Be Run?

- Can virtualize an architecture when:
    1. All sensitive instructions cause traps when executed by processes at lower levels of privilege
    2. All references to sensitive data structures cause traps when executed by processes at lower levels of privilege

Example: VAX System

- 4 levels of privilege (user, supervisor, executive, kernel)
  - CHMK changes privilege to kernel level
    - Sensitive instruction
    - Causes trap *except* when executed in kernel mode; meets rule 1
  - Page tables have copy of PSL, containing privilege level
    - Sensitive data structure
    - If user level processes prevented from altering page tables, trying to do so will cause a trap; meets rule 2

Multiple Levels of Privilege

- Hardware supports *n* levels of privilege
  - So each VM must appear to do this also
- But only VMM can run at highest level
  - So $n - 1$ levels available to each VM
- VMs must virtualize levels of privilege
  - Technique called *ring compression*

Virtualize Privilege Levels

- VAX/VMM must emulate 4 levels of privilege
  - Cannot allow any VM to enter kernel mode, and thereby bypass VMM
  - But VAX/VMS requires all four levels!
- Virtualize executive, kernel privilege levels
  - Conceptually, map both to physical executive level
  - Add VM bit to PSL; if set, current process is on VM
  - VMPSL register records PSL of running VM
  - All sensitive instructions obtain info from VMPSL or trap to VMM, which emulates instruction

Virtualization Mode

- Intel VT-i adds PSR.vm bit to process status register
  - When running guest OS, bit set; else bit cleared
  - When set, privileged instructions cause virtualization fault
    - Bit automatically cleared so VMM can service it
- Intel VT-x adds 2 modes, root and non-root operation
  - When running guest OS, in VMX non-root operation
  - Privileged instructions cause transition to VMX root mode
    - Then VMM carries out privileged instruction

Access by Class

- Divide users into different classes
  - Control access to system by limiting access of each class
- Example: IBM VM/370 associates various commands with users
  - Each command associated with *user privilege classes*
    - Class G (“general user”) can start VM
    - Class A (“primary system operator”) can control system accounting, availability of VMs, etc.
    - Class “Any” can access, relinquish access, to VM

Physical Resources and VMs

- VMM distributes these among VMs as appropriate
- Example: minidisks
  - System to run 10 VMs using one disk
  - Split disk into 10 minidisks
  - VMM handles mapping from (virtual) minidisk address to physical disk address

Example

- VM's OS tries to write to a disk
  - Privileged I/O instruction causes trap to VMM
  - VMM translates address in I/O instruction to address in physical disk
  - VMM checks that physical address in area of disk allocated to the VM making request
    - If not, request fails; error returned to VM
  - VMM services request, returns control to VM

Paging and VM

- Paging on ordinary machines is at highest privilege level
- Paging on VM is at highest virtual level
  - Handled like any other disk I/O
- Two problems:
  - On some machines, some pages available only from highest privilege level, but VM runs at next-to-highest level
  - Performance

First Problem

- VM must change protection level of pages available only from highest privilege level to appropriate level
- Example:
  - On VAX/VMS, kernel mode needed for some pages
  - But VM runs at executive mode, so must ensure only virtual kernel level processes can read those pages
  - In practice, VMS system allows executive mode processes to elevate to kernel mode; no security issue
  - But … executive mode processes on non-VM system cannot read pages, so loss of reliability

Second Problem

- VMM paging is transparent to VMs
- VMs paging: VMM handles it as above
  - If lots of VM paging, this may cause significant performance degradation
- Example: IBM VM/370
  - OS/MFT, OS/MVT access disk storage
    - If jobs depend on timings, delays caused by VMM may affect results
  - MVS does that and pages, too
    - Jobs depending on timings could fail under VM/370 that would succeed if run under MVS directly

VMM as Security Kernel

- VMM deals with subjects (the VMs)
  - Knows nothing about the processes within the VM
- VMM applies security checks to subjects
  - By transitivity, these controls apply to processes on VMs
- Thus, satisfies rule of transitive confinement

Example 1: KVM/370

- KVM/370 is security-enhanced version of VM/370 VMM
  - Goal: prevent communications between VMs of different security classes
  - Like VM/370, provides VMs with minidisks, sharing some portions of those disks
  - Unlike VM/370, mediates access to shared areas to limit communication in accordance with security policy

Example 2: VAX/VMM

- Can run either VMS or Ultrix
- 4 privilege levels for VM system
  - VM user, VM supervisor, VM executive, VM kernel (both physical executive)
- VMM runs in physical kernel mode
  - Only it can access certain resources
- VMM subjects: users and VMs

Example 2

- VMM has flat file system for itself
  - Rest of disk partitioned among VMs
  - VMs can use any file system structure
    - Each VM has its own set of file systems
  - Subjects, objects have security, integrity classes
    - Called *access classes*
  - VMM has sophisticated auditing mechanism

Example 3: Xen Hypervisor

- Xen 3.0 hypervisor on Intel virtualization technology
- Two modes, VMX root and nonroot operation
- Hardware-based VMs (HVMs) are fully virtualized domains, support unmodified guest operating systems and run in non-root operation mode
  - Xen hypervisor runs in VMX root mode
- 8 levels of privilege
  - 4 in VMX root operation mode
  - 4 in VMX root operation mode
  - No need to virtualize one of the privilege levels!

Xen and Privileged Instructions

- Guest operating system executes privileged instruction
  - But this can only be done as a VMX root operation
- Control transfers to Xen hypervisor (called *VM exit*)
- Hypervisor determines whether to execute instruction
- After, it updates HVM appropriately and returns control to guest operating system (called *VM entry*)

Problem

- Physical resources shared
  - System CPU, disks, etc.
- May share logical resources
  - Depends on how system is implemented
- Allows covert channels

Container

- Unlike VM, all containers on a system share same kernel, execute instructions natively (no emulation)
- Each container contains libraries, applications needed to execute the program(s) contained in it
- Isolates contents from other containers

Example: Docker

- Widely used in Linux systems
- Container with all libraries, programs, other data for contained software
- Runs as a daemon that launches containers, monitors them, controls levels of isolation using Linux kernel features
  - Containers have own namespace, file system, reduced set of capabilities
  - Control network access; each container can have this set as appropriate, and each assigned its own IP address
  - *root* user of container differs from that of system

Alternate Approach

- VMs present a full system (hardware and operating system)
  - But process in the VM may be able to optimize use of system resources better than the VM
  - Example: VM operating system assumes disk drive, but it’s really SSD
- Proposed: a kernel with only 2 functions:
  - Use hardware protections to prevent processes from accessing another’s memory, or overwriting it
  - Manage access to shared physical resources
  - Everything else is done at user level

Library Operating System

- A library, or set of libraries, that provide operating system functionality at the user level
  - Goal is to minimize overhead of context switching and provide processes with as much flexibility as possible
- Example: V++ Cache Kernel
  - Cache kernel tracks OS objects such as address spaces, and handles process co-ordination (like scheduling) -- runs in privileged mode
  - Application kernel manages process resources such as paging, when on page fault it loads new page mapping descriptor into Cache Kernel – runs in user mode

Example: Drawbridge

- Library OS developed for Windows 7
  - Supports standard Windows applications (Excel, IIS), gives access to features like DirectX
- Security monitor provides application binary interface (ABI), virtualizing system resources
  - Processes use library OS to access ABI; all interactions with operating system go through that interface
  - ABI has calls to manage virtual memory, processes and threads, etc.
- Library OS provides application services like frameworks, graphics engines

Example: Drawbridge (con’t)

- Kernel dependencies handled using Windows NT emulator at lowest level of library OS
  - Effect: all server dependencies, Windows subsystems moved into user space
- Human-computer interactions use emulated device drivers tunneling input, output between desktop and security monitor
- Provides process isolation
  - Experiment: run malware that deleted all registry keys
    - Under Drawbridge, only the process with the malware was affected
    - Without Drawbridge, all processes affected
  - Experiment: try attack vectors causing Internet Explorer to escape its normal protected mode (so writing to disk was unconstrainted, for example)
    - Drawbridge kept Internet Explorer properly confined

Sandboxes

- An environment in which actions are restricted in accordance with security policy
  - Limit execution environment as needed
    - Program not modified
    - Libraries, kernel modified to restrict actions
  - Modify program to check, restrict actions
    - Like dynamic debuggers, profilers

Examples Limiting Environment

- Java virtual machine
  - Security manager limits access of downloaded programs as policy dictates
- Sidewinder firewall
  - Type enforcement limits access
  - Policy fixed in kernel by vendor
- Domain Type Enforcement
  - Enforcement mechanism for DTEL
  - Kernel enforces sandbox defined by system administrator

Modifying Programs

- Add breakpoints or special instructions to source, binary code
    - On trap or execution of special instructions, analyze state of process
- Variant: *software fault isolation*
    - Add instructions checking memory accesses, other security issues
    - Any attempt to violate policy causes trap

Example: Janus

- Implements sandbox in which system calls checked
  - *Framework* does runtime checking
  - *Modules* determine which accesses allowed
- Configuration file
  - Instructs loading of modules
  - Also lists constraints

Configuration File

```
# basic module
basic

# define subprocess environment variables
putenv IFS=”\t\n “ PATH=/sbin:/bin:/usr/bin TZ=PST8PDT

# deny access to everything except files under /usr
path deny read,write *
path allow read,write /usr/*
# allow subprocess to read files in library directories
# needed for dynamic loading
path allow read /lib/* /usr/lib/* /usr/local/lib/*
# needed so child can execute programs
path allow read,exec /sbin/* /bin/* /usr/bin/*
```

How It Works

- Framework builds list of relevant system calls
  - Then marks each with allowed, disallowed actions
- When monitored system call executed
  - Framework checks arguments, validates that call is allowed for those arguments
    - If not, returns failure
    - Otherwise, give control back to child, so normal system call proceeds

Use

- Reading MIME Mail: fear is user sets mail reader to display attachment using Postscript engine
  - Has mechanism to execute system-level commands
  - Embed a file deletion command in attachment …
- Janus configured to disallow execution of any subcommands by Postscript engine
  - Above attempt fails

Example: Capsicum

- Framework developed to sandbox an application
- *Capability* provides fine-grained rights for accessing, manipulating underlying file
- To enter sandbox *(capability mode*), process issues *cap_enter*
- Given file descriptor, create capability with *cap_new*
  - Mask of rights indicates what rights are to be set; if capability exists, mask must be subset of rights in that capability
- At user level, library provides interface to start sandboxed process and delegate rights to it
  - All nondelegated file descriptors closed
  - Address space flushed
  - Socket returned to creator to enable it to communicate with new process

Example: Capsicum (con’t)

- Global namespaces not available
  - So system calls that depend on that (like *open*(2)) don’t work
    - Need to use a modified *open* that takes file descriptor for containing directory
  - Other system calls modified appropriately
    - System calls creating memory objects can create anonymous ones, not named ones (as those names are in global namespace)
- Subprocesses cannot escalate privileges
  - But a privileged process can enter capability mode
- All restrictions applied in kernel, not at system call interface

Program Confinement and TCB

- Confinement mechanisms part of trusted computing bases
  - On failure, less protection than security officers, users believe
  - “False sense of security”
- Must ensure confinement mechanism correctly implements desired security policy

Program Modification

- Source, binary code transformed to implement confinement constraints
- Can be done in several ways:
  - Code rewriter, used before compiling to alter source code
  - Compiler, transforming code as it compiles it
  - Binary code rewriter, used on the executable
  - Linking loader, used to transform linkages between program and library functions, system calls to validate interactions

Rewriting

- Software fault isolation: put untrusted modules in special virtual segments
  - Code modified so control flow remains in that segment when module invoked
  - All memory accesses in segment are to data in that segment

Implementation

- Each virtual segment has a unique *segment identifier* in upper part of virtual address
    - *Unsafe instruction* is one that accesses an address that cannot be verified to be in module’s segment
- Segment matching: analyze program, identify all unsafe instructions and wrap them so they are checked at run time
    - If check shows address not in module, trap it
- Alternative: set upper bits of any virtual address to segment identifier
    - Illegal memory accesses handled in usual way

Implementation (con’t)

- Threat: untrusted module issues system call to close file that trust3ed modules rely on
  - Causes program crash or other undesirable actions
- Trusted arbitration code places in its own segment
  - This accepts RPC requests from other modules, validates them, and translates them into system calls
  - Results returned via RPC
- Untrusted modules rewritten so system calls done vis the arbitration code (ie, using RPC to that module)

Rewriting

- Can put security-sensitive parts into separate trusted process
  - Application rewritten so untrusted parts invoke trusted parts via IPC
  - Both trusted, untrusted parts must be started to run application
- Example: Nizza architecture
  - Untrusted process executed on VM
  - AppCore, a trusted process, executed in trusted computing environment
    - Analyze application to identify security-sensitive components
    - Place these components into a standalone process (AppCore). May need to be altered to conform to security policy
    - Transform rest of process to use AppCore to execute security-sensitive components

Compiling

- Compiler implements a security policy so resulting executable provides desired isolation
  - Example: type-safe languages, in which compiler verifies use of types is consistent
- Certifying compiler includes proof that program satisfies specified security properties
  - Proof can be validated before execution

Transforming Compiler

- CCured imposes type safety on C programs by adding semantics to constructs that can produce undefined results
  - Safe pointer of type *t* points to the address of an object of type *t*, or 0 (NULL pointer)
  - Sequence pointer points into memory area of objects of type *t*; so check is that it is a pointer of type *t*, points to object of type *t* in that memory area
  - Dynamic pointer can point to untyped areas of memory, or memory of arbitrary type (this is tagged with type of values currently in that area)
- Type inference algorithm used to construct CCured program honoring type rules

Certifying Compiler

- Touchstone works on type-safe subset of C
  - All array references are checked to ensure they are in bounds
- Compiler translates program into assembly
- VCGen generates verification conditions
  - Works on per-function basis using symbolic execution
    - Type specifications declare types of arguments (preconditions) and return values (postconditions)
  - Builds a predicate based on machine instructions
  - On a return instruction, emits a predicate that includes check on instantiation of preconditions, predicate built from assembly language, and a check on postconditions
  - Predicate can be proved iff program satisfies postcondition and registers preserved on entry are not changed
- Theorem prover verifies proof

Loading

- Like sandboxing, but framework embedded in libraries and not a separate process
- When called, a constrained library applies security policy rules to determine whether it should take desired action
- Example: Aurasium for Android apps
  - Goal: prevent exfiltration of sensitive data or misuse of resources
  - Adds code to monitor all interactions with phone's resources; these can be considerably more granular than default permissions set at installation

Aurasium

- Goal: prevent exfiltration of sensitive data or misuse of resources on Android phone by apps
  - Adds code to monitor all interactions with phone's resources; these can be considerably more granular than default permissions set at installation
- First part: tool that inserts code to enforce policies when app calls on phone resources, such as SMS messages
- Second part: use modified Android standard C libraries that determine whether app's requested system call should be blocked
- App signatures verified before Aurasium transforms app; then Aurasium signs app
  - Issue is that when Aurasium transforms app, original signature no longer valid

Covert Channels

- Shared resources as communication paths
- *Covert storage channel* uses attribute of shared resource
  - Disk space, message size, etc.
- *Covert timing channel* uses temporal or ordering relationship among accesses to shared resource
  - Regulating CPU usage, order of reads on disk