CS 587: Computer Systems Security Systems

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

October 3, 2011

Part I

Trusted Computer Base and System Layers

Trusted Computer Base (TCB)

- The Trusted Computer Base (TCB) is that part of the computer system which, if it fails can impact security.

Trusted Computer Base (TCB)

- The Trusted Computer Base (TCB) is that part of the computer system which, if it fails can impact security.
- TCB issues:
    - The larger the TCB is, the more difficult it is to make it secure.

Trusted Computer Base (TCB)

- The Trusted Computer Base (TCB) is that part of the computer system which, if it fails can impact security.
- TCB issues:
  - The larger the TCB is, the more difficult it is to make it secure.
  - The TCB should be as simple as possible:

Trusted Computer Base (TCB)

- The Trusted Computer Base (TCB) is that part of the computer system which, if it fails can impact security.
- TCB issues:
  - The larger the TCB is, the more difficult it is to make it secure.
  - The TCB should be as simple as possible:
    - Minimize misuse

Trusted Computer Base (TCB)

- The Trusted Computer Base (TCB) is that part of the computer system which, if it fails can impact security.
- TCB issues:
  - The larger the TCB is, the more difficult it is to make it secure.
  - The TCB should be as simple as possible:
    - Minimize misuse
    - Enable better verification

What must the trusted computing base contain

- It must contain the OS
- It must contain any critical applications
    - It is the applications that actual determine what is to be written (integrity)
    - Availability cannot be provided without regard to applications who perform the critical tasks
- Security is not a monolithic property
- Security is not a property
- In any event, the goal is really to limit loss, not prevent all attacks

Layered systems

- Systems built in layers

Layered systems

- Systems built in layers
- Higher levels depend on lower levels, but lower levels do not depend on higher levels.

Layered systems

- Systems built in layers
- Higher levels depend on lower levels, but lower levels do not depend on higher levels.
- Hence, if component C depends upon C' for its security and C' is insecure, then C cannot be secure.

Layered systems

- Systems built in layers
- Higher levels depend on lower levels, but lower levels do not depend on higher levels.
- Hence, if component C depends upon C' for its security and C' is insecure, then C cannot be secure.
- Since a component almost always depends upon its lower levels for security, the TCB usually includes all lower levels.

Layered systems

- Systems built in layers
- Higher levels depend on lower levels, but lower levels do not depend on higher levels.
- Hence, if component C depends upon C' for its security and C' is insecure, then C cannot be secure.
- Since a component almost always depends upon its lower levels for security, the TCB usually includes all lower levels.
- The lower the level that protections can be added the smaller the TCB.

Layered systems

- Systems built in layers
- Higher levels depend on lower levels, but lower levels do not depend on higher levels.
- Hence, if component C depends upon C' for its security and C' is insecure, then C cannot be secure.
- Since a component almost always depends upon its lower levels for security, the TCB usually includes all lower levels.
- The lower the level that protections can be added the smaller the TCB.
- The smaller the TCB, the easier it is to validate.

Layered protection

- Protecting the system depends on the layering of the system

Layered protection

- Protecting the system depends on the layering of the system
- Layers from low to high:

Layered protection

- Protecting the system depends on the layering of the system
- Layers from low to high:
    - Hardware

Layered protection

- Protecting the system depends on the layering of the system
- Layers from low to high:
    - Hardware
    - Architecture

Layered protection

- Protecting the system depends on the layering of the system
- Layers from low to high:
  - Hardware
  - Architecture
  - BIOS

Layered protection

- Protecting the system depends on the layering of the system
- Layers from low to high:
  - Hardware
  - Architecture
  - BIOS
  - Operating System

Layered protection

- Protecting the system depends on the layering of the system
- Layers from low to high:
    - Hardware
    - Architecture
    - BIOS
    - Operating System
    - Application

Layered protection

- Protecting the system depends on the layering of the system
- Layers from low to high:
    - Hardware
    - Architecture
    - BIOS
    - Operating System
    - Application
- Attacks can come at any of these layers.

Hardware attacks

Hardware attacks can be viewed from the security they deny:

Hardware attacks

Hardware attacks can be viewed from the security they deny:

Confidentiality electromagnetic waves

Hardware attacks

Hardware attacks can be viewed from the security they deny:

Confidentiality	electromagnetic waves
Integrity	radiation

Hardware attacks

Hardware attacks can be viewed from the security they deny:

Confidentiality	electromagnetic waves
Integrity	radiation
Denial of Service	power failure

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:
  - **Trap instruction** necessary to invoke OS system calls

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:

Trap instruction	necessary to invoke OS system calls
Interrupts	asynchronous hardware events

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:

Trap instruction	necessary to invoke OS system calls
Interrupts	asynchronous hardware events
Memory hierarchy	caches and TLBs

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:

Trap instruction	necessary to invoke OS system calls
Interrupts	asynchronous hardware events
Memory hierarchy	caches and TLBs

- Processor errata typically has to do with some combinations of unusual events.

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:

Trap instruction	necessary to invoke OS system calls
Interrupts	asynchronous hardware events
Memory hierarchy	caches and TLBs

- Processor errata typically has to do with some combinations of unusual events.
- Architectures tend not to have systemic problems

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:

Trap instruction	necessary to invoke OS system calls
Interrupts	asynchronous hardware events
Memory hierarchy	caches and TLBs

- Processor errata typically has to do with some combinations of unusual events.
- Architectures tend not to have systemic problems
  At least I don't think they do.

Architecture attacks

- Architecture attacks are based on flaws in the computer architecture design or implementation. These typically have something to do with:

Trap instruction	necessary to invoke OS system calls
Interrupts	asynchronous hardware events
Memory hierarchy	caches and TLBs

- Processor errata typically has to do with some combinations of unusual events.
- Architectures tend not to have systemic problems

  At least I don't think they do.

  Who would do that anyway?

BIOS attacks

The BIOS

- Contains the boot loader

BIOS attacks

The BIOS

- Contains the boot loader
- Boot loader loads the OS kernel

BIOS attacks

The BIOS

- Contains the boot loader
- Boot loader loads the OS kernel
- What happens if it loads the wrong OS kernel or modifies the correct one?

BIOS attacks

The BIOS

- Contains the boot loader
- Boot loader loads the OS kernel
- What happens if it loads the wrong OS kernel or modifies the correct one?
- BIOS need not be used once system boots, but probably is for ACPI, . . .

Part II

Operating System

Operating system

The operating system consists of:

kernel through which all services are provided to processes and

Operating system

The operating system consists of:

kernel	through which all services are provided to processes and
system processes	which perform services not included in the kernel.

Operating System Kernel

The kernel is the program that:

- executes privileged instructions and
- implements the process abstraction.
- traditionally the kernel tends to be fairly difficult to attack.
    - Because bugs in the kernel can destabilize the system (causing crashes) the kernel is very conservatively maintained.
    - Kernel code is extensively read and reviewed by very skilled people.
- but this assumes that the kernel is not of enormous complexity and goes through an appropriate assurance process
- today, kernels are neither conservatively maintained or carefully read, they change at too high a rate.
- Kernels such as Linux/Window are over 10 million lines of code

Kernel overview

Why kernels are important to security:

- All operations of a process which effect the outside world (files, networks, user interface, or other processes) are mediated by the kernel.

Kernel overview

Why kernels are important to security:

- All operations of a process which effect the outside world (files, networks, user interface, or other processes) are mediated by the kernel.
- The kernel must protect (isolate) processes from each other (to implement the process abstraction) and hence must have a protection mechanism.

Kernel overview

Why kernels are important to security:

- All operations of a process which effect the outside world (files, networks, user interface, or other processes) are mediated by the kernel.
- The kernel must protect (isolate) processes from each other (to implement the process abstraction) and hence must have a protection mechanism.
- The kernel enables the uniform control of protection since it applies to every process.

Kernel overview

Why kernels are important to security:

- All operations of a process which effect the outside world (files, networks, user interface, or other processes) are mediated by the kernel.
- The kernel must protect (isolate) processes from each other (to implement the process abstraction) and hence must have a protection mechanism.
- The kernel enables the uniform control of protection since it applies to every process.
- All kernels provide protections beyond what is needed for process abstraction.

OS vs. Interpreters

- An interpreter like the JVM, reads each instruction (byte code), checks its legality, and then executes it.
- An interpreter is software which checks instructions
- In an OS, instructions run on the hardware.
- But in applications, privileged instructions are intercepted by hardware
- Thus the computer can run user code safely at full speed
- While isolating that code from harming others
- And perform a safe transition to OS kernel code.

Process abstraction

To support the process abstraction, the architecture must implement:

memory protection	so that one process cannot access another's memory.

Process abstraction

To support the process abstraction, the architecture must implement:

memory protection	so that one process cannot access another's memory.
time interrupts	so that a process does not hog the CPU.

Process abstraction

To support the process abstraction, the architecture must implement:

memory protection	so that one process cannot access another's memory.
time interrupts	so that a process does not hog the CPU.
privileged instructions	so that processes not interfere with each other.

Process abstraction

To support the process abstraction, the architecture must implement:

memory protection	so that one process cannot access another's memory.
time interrupts	so that a process does not hog the CPU.
privileged instructions	so that processes not interfere with each other.
trap instructions	a controlled means of entering into privilege mode (and the kernel).

Process abstraction

To support the process abstraction, the architecture must implement:

memory protection	so that one process cannot access another's memory.
time interrupts	so that a process does not hog the CPU.
privileged instructions	so that processes not interfere with each other.
trap instructions	a controlled means of entering into privilege mode (and the kernel).

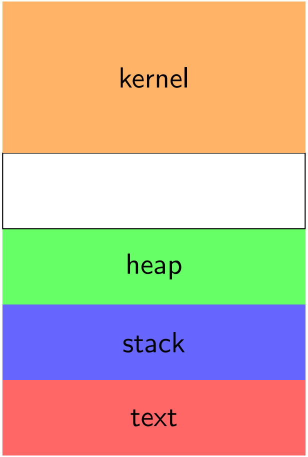
The kernel alone deals with these privileged instructions while processes operate using only unprivileged instructions.

Privileged instructions

Privileged instructions protect that which would break the process abstraction:

interrupts	prevents a process from seizing control of the processor
I/O devices	shared resources
virtual memory	isolates processes from each other and from the kernel

Memory layout (Virtual memory)



- Virtual memory is divided into kernel and user space
- User space contains a single process with components
  - heap: contains dynamically allocated storage
  - stack: contains local variables and procedure linkage and
  - text: contains program code plus constants
- In privileged mode can read all memory
- In unprivileged mode can read only user space memory

Kernel example: read system call

- POSIX system call: `read(fd, b, s)`
  - fd is a file descriptor (an integer identifier for a file-like device)
  - buffer is a pointer into a character array
  - size is the number of bytes to be read into the process
- It is a request to the OS kernel to read s bytes into buffer b of a file identified by fd.
- The OS kernel may either do it or refuse to do it (returning an error)

User space invoking of read system call

- The process pushes the parameters on the stack (three values)
- the trap is invoked with the system call number (corresponding to the read system call)
- when execution returns to the process the result of the system call is returned

Kernel processing of read system call

- Control enters the kernel at a fixed location
- The system call prologue is executed
- The system call number is used to lookup the system call address
- The system call, `syscallRead` is invoked
    - It checks that the arguments are well formed
    - It checks that the process has permissions to do the read
    - It performs the read
- The results are returned to user space
- A return from interrupt instruction turns off privilege space

Vulnerabilities

- Read passes a pointer into the kernel. Kernel must check that the pointer is in user space.
- If it's not, user space program could cause part of the Kernel to be overwritten
- Must ensure that every byte of the buffer is in user space
- Must ensure that the process is authorized to read the value

Kernel structure

Kernel is a combination of:

Machine dependent components	privilege instructions, layout of hardware structures (eg. page tables), and performance critical code.

Kernel structure

Kernel is a combination of:

Machine dependent components	privilege instructions, layout of hardware structures (eg. page tables), and performance critical code.
Machine independent components	The vast majority of code. There is an abstract machine assumed by the machine independent components implemented with the architecture plus machine dependent components.

Kernel structure

Kernel is a combination of:

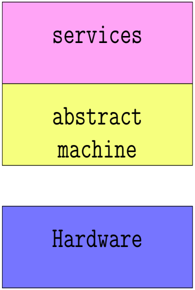
Machine dependent components	privilege instructions, layout of hardware structures (eg. page tables), and performance critical code.
Machine independent components	The vast majority of code. There is an abstract machine assumed by the machine independent components implemented with the architecture plus machine dependent components.

The porting to a new architecture then involves the writing of a new machine dependent component.

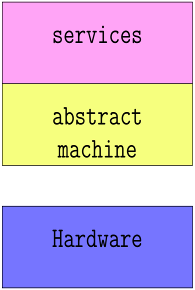
Device drivers

- Most of the code in an OS (over 2/3rds) is for device drivers.
- These device drivers are not architecture independent in that the same device controller can be used by different architectures.
- These are a disproportionate source of bugs (since they are often designed by device manufacturers or even third parties)
- The devices themselves often behave erratically
- They are hard to test, because access to appropriate hardware is required
- e.g., Microsoft said 27% of blue-screen-of-death due to NVidia drivers

Kernel layering

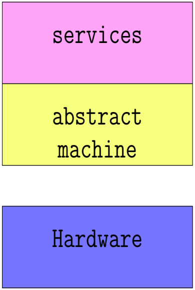


Kernel layering



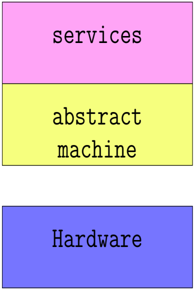
Services	Networking, filesystem, IPC, sub-page memory allocation.

Kernel layering



Services	Networking, filesystem, IPC, sub-page memory allocation.
Abstract machine	paging, system calls/interrupts, synchronization, device drivers.

Kernel layering



Services	Networking, filesystem, IPC, sub-page memory allocation.
Abstract machine	paging, system calls/interrupts, synchronization, device drivers.
Hardware	there is a great latitude to the architecture.

The role of the compiler

- Almost all of the the kernel is written in C

The role of the compiler

- Almost all of the the kernel is written in C
- There is about 8,600 lines of assembler to support i386 (most in math emulator), and there is some in-line assembly code.

The role of the compiler

- Almost all of the the kernel is written in C
- There is about 8,600 lines of assembler to support i386 (most in math emulator), and there is some in-line assembly code.
- The compiler cannot emit privilege instructions since this is outside its model. Privilege instructions are coded in assembly (.S files) or with in-line assembly code (using asm directives).

The role of the compiler

- Almost all of the the kernel is written in C
- There is about 8,600 lines of assembler to support i386 (most in math emulator), and there is some in-line assembly code.
- The compiler cannot emit privilege instructions since this is outside its model. Privilege instructions are coded in assembly (.S files) or with in-line assembly code (using asm directives).
- Also synchronization (eg. Test-and-Set) and trap must be done with assembler.

The role of the compiler

- Almost all of the the kernel is written in C
- There is about 8,600 lines of assembler to support i386 (most in math emulator), and there is some in-line assembly code.
- The compiler cannot emit privilege instructions since this is outside its model. Privilege instructions are coded in assembly (.S files) or with in-line assembly code (using asm directives).
- Also synchronization (eg. Test-and-Set) and trap must be done with assembler.
- Linux runs on multiple different architectures, these must support at least the abstract machine in terms of the process level abstractions and protections.

The role of the compiler

- Almost all of the the kernel is written in C
- There is about 8,600 lines of assembler to support i386 (most in math emulator), and there is some in-line assembly code.
- The compiler cannot emit privilege instructions since this is outside its model. Privilege instructions are coded in assembly (.S files) or with in-line assembly code (using asm directives).
- Also synchronization (eg. Test-and-Set) and trap must be done with assembler.
- Linux runs on multiple different architectures, these must support at least the abstract machine in terms of the process level abstractions and protections.
- C is problematic for writing secure code (many bugs possible)

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members
- Processes invoke monitor

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members
- Processes invoke monitor
- At most one process can be actively executing in the monitor

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members
- Processes invoke monitor
- At most one process can be actively executing in the monitor
- Other processes in the monitor are sleeping—waiting for an event

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members
- Processes invoke monitor
- At most one process can be actively executing in the monitor
- Other processes in the monitor are sleeping—waiting for an event
- Non-preemptive scheduling

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members
- Processes invoke monitor
- At most one process can be actively executing in the monitor
- Other processes in the monitor are sleeping—waiting for an event
- Non-preemptive scheduling
- Note that the kernel is inherently concurrent

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members
- Processes invoke monitor
- At most one process can be actively executing in the monitor
- Other processes in the monitor are sleeping—waiting for an event
- Non-preemptive scheduling
- Note that the kernel is inherently concurrent
- This is another source for bugs

Logical model of the kernel

Semantically, the kernel is a monitor:

- Class-like definition with only methods as public members
- Processes invoke monitor
- At most one process can be actively executing in the monitor
- Other processes in the monitor are sleeping—waiting for an event
- Non-preemptive scheduling
- Note that the kernel is inherently concurrent
- This is another source for bugs

(This is the model of the original Unix system).

The process-kernel interface

The process and kernel share same address space.

The process-kernel interface

The process and kernel share same address space.

- The kernel does not trust the process

The process-kernel interface

The process and kernel share same address space.

- The kernel does not trust the process
  The process needs some safe way of invoking the kernel

The process-kernel interface

The process and kernel share same address space.

- The kernel does not trust the process
  The process needs some safe way of invoking the kernel
- The process must trust the kernel

The process-kernel interface

The process and kernel share same address space.

- The kernel does not trust the process
  The process needs some safe way of invoking the kernel
- The process must trust the kernel
  The kernel can access the processes address space

The process-kernel interface

The process and kernel share same address space.

- The kernel does not trust the process
  The process needs some safe way of invoking the kernel
- The process must trust the kernel
  The kernel can access the processes address space
- The availability of a process depends on the OS

The process-kernel interface

The process and kernel share same address space.

- The kernel does not trust the process
  The process needs some safe way of invoking the kernel
- The process must trust the kernel
  The kernel can access the processes address space
- The availability of a process depends on the OS
  Kernel can prevent availability to process but cannot provide availability

Process-kernel communication

- The kernel acts as a server, it is always ready to accept communication from the process (synchronous mechanism is sufficient)

Process-kernel communication

- The kernel acts as a server, it is always ready to accept communication from the process (synchronous mechanism is sufficient)
- The process acts as a client, it may not be ready to accept communications from the kernel (need asynchronous mechanism)

Process-kernel communication

- The kernel acts as a server, it is always ready to accept communication from the process (synchronous mechanism is sufficient)
- The process acts as a client, it may not be ready to accept communications from the kernel (need asynchronous mechanism)

Mechanisms for kernel-process communication

System calls	procedure call-like mechanism to enter the kernel
Signals	asynchronous notification to processes from kernel
Proc filesystem	filesystem representation of various system state
Netlink	Kernel communication for specialized communication

Kernel interface has widened over time

- More information available to user space: affecting confidentiality
- More ways of changing state from user space: affecting integrity
- Things should be going in the opposite direction
- Narrowing interfaces
- Thus improving confidentiality and integrity

User space access issues

When copying structures to user space, care must be taken to ensure that the kernel does not leak information to the process:

1. The padding areas of structures must be zeroed or
2. the whole structure must be zeroed before copying over members.

Process structures

In Unix, the process credentials (as well as file descriptors and other resources) are:

- inherited from the parent.
- changed by system calls

Hence the initial login process for a user sets the UID on whose behalf the process executes and then spawns other processes with the UID inherited from parent.

Authorization

- Need to limit what processes can do
- And thus narrow ability to perform attacks
- Need sophisticated authorization to implement various trust models
- But more sophisticated authorization results in higher complexity
- How do you build authorization which is both usable and provides sufficient protections?

OS problems

- Failure to check input parameters (they come from user space and therefore are untrusted)
- Failure to initialize values copied to user space (confidentiality)
- Loadable modules
- Race conditions
- Device drivers
- Complexity
- Authorization limitations

Part III

Virtual Machines (VMs)

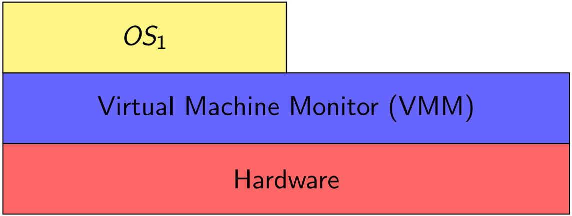
Virtual Machines (VMs)

- There is one more system layer, an optional one, that needs to be talked about because it is increasingly important
- A virtual machine is a software implementation of a machine.
- the most interesting machine is a computer, containing processor and I/O devices
- In the ideal case, the VM is indistinguishable from the hardware
- An OS runs within a VM
- A VM Monitor (VMM) implements one or more VMs
- There are two types of VMMs

Bare metal	a VMM that runs directly on the hardware
Hosted	a VMM that runs on top of an OS

Bare metal VMM

- Bare metal VMM implementing 2 VMs
- VMM is also called a Hypervisor



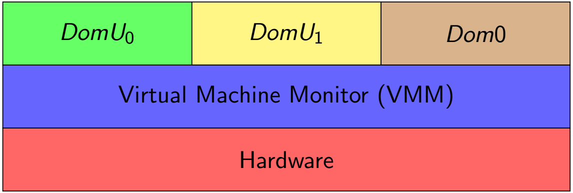
Bare Metal VMM

- The OS runs in unprivileged mode, VMM in privileged mode.
- In a fully virtualized system, the hardware intercepts privileged instructions/interrupts.
- And transfers control to a Virtual Machine Monitor (VMM)
- The virtual machine simulates what the hardware would do
- Safely multiplexing the operations from different OSs
- An alternative is to use paravitualization
- Which uses VMM hypercalls to request privileged operations
- Examples: Xen, Vmware VMX

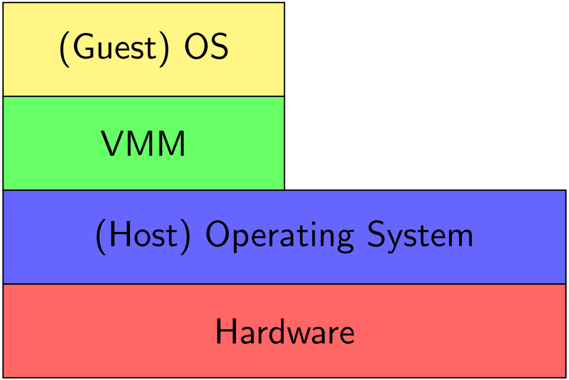
Bare Metal VMs–device drivers

- Bare metal VMs have the device driver problem.
- How to support all those devices?
- Use an OS which supports many devices—Linux.
- Now we have two operating systems,
  - Dom0: the privileged OS with device drivers
  - DomU: a guest OS which uses only virtual devices
- This enables the VMM to be very small, about 100K lines of code
- But the DomU's depend upon Dom0

Xen



Hosted VMs



Hosted VMs

- Hosted VMs are implemented on top of an OS
- Cannot rely on the architecture to intercept privilege instruction
- So it could use software, but that is slow
- To speed things up, use binary rewriting to translate instruction streams on the fly
- Binary rewriting reads a sequence of instructions and replaces them with an equivalent sequence (in this case, without privileged instructions)
- Translation occurs once, gets reuse many times
- Examples: VMware

Intel architecture

- The original Intel architecture could not support full virtualization
- So VMware used binary rewriting
- Which was pioneered by an earlier company, Transmeta which built an Intel compatible architecture
- Xen use paraviturализation
- Then Intel and AMD introduced self virtualizing extensions to x86
- And now this is widely used to support OSs such as Windows which are proprietary and hence cannot be ported to paravirtualizing VMs

Security Implications of VMs

- Hosted VMs are vulnerable to the OSs they run on top of
- It does not help to run a super secure OS on top of a vulnerable OS
- Bare metal OSs are vulnerable to their VMMs
- But the VMMs are relatively small and easy to secure
- The DomU OSs are also vulnerable to the Dom0 OSs
- But with care we can make these only sensitive to device drivers
- But if DomU encrypts I/O, device drivers only effect availability.

Part IV

The programming toolchain

Application programming dependencies

- Application programs depend not only on the OS but on the user space software
- These effect the correctness of application programs and thus impact every aspect of their security
- The primary effects are due to
  - programming language and thus the ability to express correct programs
  - user space software which produces binary executable
- Note that after an executable is produced, the OS is the entity with which the process interacts.

Programming language effect

- Programming language semantics have an important role on vulnerabilities
- Type safety prevents buffer overflow
- Automatic garbage collection prevents double free, use after free, and other insidious memory errors
- Threads enable memory race conditions

Application program tool chain

What happens when you compile a C program?

1. The compiler runs your program through the C preprocessor including system and application headers.
2. The program is converted to an intermediary form and syntax and semantic checks are made.
3. The program is optimized
4. Assembly language is produced
5. Assembly language is converted into binary code (e.g., ELF format)
6. The binary code is linked with system and application libraries
7. An executable is produced
8. At run time, dynamic runtime libraries are loaded with the executable, and executed.

Incorporating attack code in applications

- Include files contain code
- Only what is needed to complete linking is pulled in from library
- Thus an attacker's `printf` can replace library `printf`
- Compilation/Assembly can insert malware or vulnerabilities
- Dynamic libraries mean that library code can be substituted after compilation

Other dependencies

- The environment variable LD_LIBRARY_PATH specifies where libraries may be found.
- PATH specifies where executables can be found if pathname not fully specified
- Setting environment variables is a non-privileged operation

Trusting trust

- Ken Thompson, the inventor of Unix, gave a paper entitled On Trusting Trust for his Turing Award lecture.
- Back in the early days, a tape of Unix was ordered by the National Security Agency.
- Thompson ponder how he could put in a trap door
- He encoded a username/password in the login program
- But that could be easily removed
- So he put code in the compiler which would
  - Detect if it was compiling the login program
  - Detect if the trap door was removed from the login program
  - And if so reinsert the trap door

Trusting trust (cont’d)

- Now the trap door could be removed from the login program
- But it was still visible in the compiler
- Thompson wrote code to detect if the compiler was being compiled and whether the trap door code was removed from the compiler; if so reinsert the trap door code.
- The compiler binary was created
- The code was removed from the compiler.
- Now no source code evidences any back door.
- It is done all at the binary level.

Trusting trust conclusions

- Transitory code can be used to compromise systems
- The lower the level of transitory code, the easier it is to hide
- In Thompson's case, binary instead of source code
- But it is possible to hide it even lower, in the BIOS
- Or the hardware
- Where it would be very difficult to find.

Part V

Application-level security services

Application security dependencies

- Inherently, Integrity and Availability depend on applications
- But what about security services?
- e.g., authentication, authorization, encryption
- Where should these be located?

Application-level security services

If it cannot be handled at the operating system level, then it must be handled at the application level. The disadvantages are:

Application-level security services

If it cannot be handled at the operating system level, then it must be handled at the application level. The disadvantages are:

1. Increases the size of the TCB,

Application-level security services

If it cannot be handled at the operating system level, then it must be handled at the application level. The disadvantages are:

1. Increases the size of the TCB,
2. Each application must be individually configured,

Application-level security services

If it cannot be handled at the operating system level, then it must be handled at the application level. The disadvantages are:

1. Increases the size of the TCB,
2. Each application must be individually configured,
3. Individually secure application may together be insecure (composition),

Application-level security services

If it cannot be handled at the operating system level, then it must be handled at the application level. The disadvantages are:

1. Increases the size of the TCB,
2. Each application must be individually configured,
3. Individually secure application may together be insecure (composition),
4. Bugs in the application may cause protections to be bypassed,

Application-level security services

If it cannot be handled at the operating system level, then it must be handled at the application level. The disadvantages are:

1. Increases the size of the TCB,
2. Each application must be individually configured,
3. Individually secure application may together be insecure (composition),
4. Bugs in the application may cause protections to be bypassed,
5. Applications may be insufficiently protected, and

Application-level security services

If it cannot be handled at the operating system level, then it must be handled at the application level. The disadvantages are:

1. Increases the size of the TCB,
2. Each application must be individually configured,
3. Individually secure application may together be insecure (composition),
4. Bugs in the application may cause protections to be bypassed,
5. Applications may be insufficiently protected, and
6. Not possible to analyze the protection configuration.

Application level (cont'd)

- It is not feasible to study protections unless they are abstracted away from their implementations.
- Application level protections make that more difficult to do.
- Although integrity depends on the correctness of the executable, decoupling of correctness and security should be maximized.

Part VI

OS principles and ratings

OS protection principles

Several principles were espoused by Salzer and Schroeder '75 and are still valuable today:

Least privilege	provide the minimum privilege required to perform a function.
Economy of mechanism	The protection system design should be small, simple, and straightforward.
Open design	security should not depend on ignorance of attackers.
Complete mediation	Every access must be checked
Permission based	The default is to deny access.
Separation of privilege	Use multiple mechanisms to protect important items, including separation of duties.
Least common mechanism	Share as little as possible
Ease of use	So that the mechanism is not avoided.

Additional OS features

Trusted Path	Ensure that user is entering information (such as passwords) only to the appropriate program.

Additional OS features

Trusted Path	Ensure that user is entering information (such as passwords) only to the appropriate program.
Object reuse	ensure reused objects don't contain leftover info.

Additional OS features

Trusted Path	Ensure that user is entering information (such as passwords) only to the appropriate program.
Object reuse	ensure reused objects don't contain leftover info.
Auditing	after the fact "forensics":

Accountability and audit log	have a record of what users did.
Reduce the size	audit logs can be very large, so there needs to be an effective way of searching it
Intrusion detection	find in real time suspicious events so that they can be examined.

Advantages of kernel-level protections

But what are the advantages of kernel level protections?

Layered Design: Segregates application level correctness from kernel level protections.

- Kernel level protections are small and general purpose and hence likely to be extensively verified.
- Failures of application correctness does not effect kernel protection. (This is not the case w/application level protection).
- It is possible to answer the question: What happens if the application is incorrect but the kernel protections are correctly implemented?

Advantages of kernel-level protections

But what are the advantages of kernel level protections?

Layered Design: Segregates application level correctness from kernel level protections.

- Kernel level protections are small and general purpose and hence likely to be extensively verified.
- Failures of application correctness does not effect kernel protection. (This is not the case w/application level protection).
- It is possible to answer the question: What happens if the application is incorrect but the kernel protections are correctly implemented?

Better abstractions: since kernel-based protections must be general purpose they lead to thinking about better abstractions.

Control of security policy: by externalizing protection, the organization owning the system controls the security rather than application developer/packager.

Control of security policy: by externalizing protection, the organization owning the system controls the security rather than application developer/packager.

Determining sensitive programs: The protection configuration enables identification of the most sensitive programs,

Control of security policy: by externalizing protection, the organization owning the system controls the security rather than application developer/packager.

Determining sensitive programs: The protection configuration enables identification of the most sensitive programs,

Least privilege: Provides application with the minimum privilege to do their function (this is a safety concern).

Control of security policy: by externalizing protection, the organization owning the system controls the security rather than application developer/packager.

Determining sensitive programs: The protection configuration enables identification of the most sensitive programs,

Least privilege: Provides application with the minimum privilege to do their function (this is a safety concern).

System level protections: Kernel protections apply to all applications on a system—they cannot be bypassed.

Control of security policy: by externalizing protection, the organization owning the system controls the security rather than application developer/packager.

Determining sensitive programs: The protection configuration enables identification of the most sensitive programs,

Least privilege: Provides application with the minimum privilege to do their function (this is a safety concern).

System level protections: Kernel protections apply to all applications on a system—they cannot be bypassed.

Analysis: automatically can determine properties arising from system protection.

Control of security policy: by externalizing protection, the organization owning the system controls the security rather than application developer/packager.

Determining sensitive programs: The protection configuration enables identification of the most sensitive programs,

Least privilege: Provides application with the minimum privilege to do their function (this is a safety concern).

System level protections: Kernel protections apply to all applications on a system—they cannot be bypassed.

Analysis: automatically can determine properties arising from system protection.

Simplified application code: since it need not have protection code.

Conclusions

- Systems are layered for security
- Upper layers depend on lower layers and are therefore vulnerable to them
- And the layers all the way to the top (applications) are necessary for some security property
- Each layer can be attacked, often by breaking the abstractions that their designers relied upon.
- And that provides a large number of paths to attack.