Reference Monitors
CSM27 Computer Security

Dr Hans Georg Schaathun

University of Surrey

Autumn 2007
Outline

1. The session
2. The reference monitor
   - The reference monitor
3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation
4. Other issues
   - Memory protection
   - OS kernel issues
5. Conclusion
Session objectives

- See the advantages of enforcing security at the lowest layers
- Understand the concepts of reference monitors and trusted computing based
- Be aware of some of the security mechanisms used in the lowest layers.
Outline

1. The session
2. The reference monitor
   - The reference monitor
3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation
4. Other issues
   - Memory protection
   - OS kernel issues
5. Conclusion
The components

**Reference Monitor**  Guard at the gates.  Abstract machine mediating all access by subjects to objects.

**Security Kernel**  Implementation of the reference monitor.

**Trusted Computing Base (TCB)**  Complex system including security kernel and other protection mechanisms needed across a computer system.
Placing the Reference Monitor

- **RM in OS kernel**
  - System-wide policy enforcement
  - Little flexibility for needs of individual applications

- **RM in Application Software**
  - Adapted to very special needs.
  - Must be managed separately.

- **Application Software in RM**
  - create a sandbox for the application
  - e.g. Java Virtual Machine (JVM)
History or future

- **Execution Monitors** look only at the past
  - Blocks any attempt on an illegal operation.
  - Cannot predict future operations

- Compilers consider the future
  - Static type-checking bars possible violations in future execution

- Always,
  - The reference monitor checks every access request
  - fits into the system such that it cannot be bypassed
History or future

- **Execution Monitors** look only at the past
  - Blocks any attempt on an illegal operation.
  - Cannot predict future operations

- **Compilers consider the future**
  - Static type-checking bars possible violations in future execution

- Always,
  - The reference monitor checks every access request
  - Fits into the system such that it cannot be bypassed
History or future

- **Execution Monitors** look only at the past
  - Blocks any attempt on an illegal operation.
  - Cannot predict future operations

- **Compilers consider the future**
  - Static type-checking bars possible violations in future execution

- **Always,**
  - The reference monitor checks every access request
  - Fits into the system such that it cannot be bypassed
System Integrity

- Competing requirements
  - Users must be able to use (invoke) the OS.
  - Users must not be able to misuse the OS.

- Most importantly
  - The users must not be able to modify the operating system.
  - If a user could change the reference monitor,
    - he could bypass it too...
  - Users changing the OS, could change or bypass security features.
System Integrity

- Competing requirements
  - Users must be able to use (invoke) the OS.
  - Users must not be able to misuse the OS.

- Most importantly
  - The users must not be able to modify the operating system.

- If a user could change the reference monitor,
  - he could bypass it too...

- Users changing the OS, could change or bypass security features.
System Integrity

- Competing requirements
  - Users must be able to use (invoke) the OS.
  - Users must not be able to misuse the OS.
- Most importantly
  - The users must not be able to modify the operating system.
- If a user could change the reference monitor,
  - he could bypass it too...
- Users changing the OS, could change or bypass security features.
System Integrity

- Competing requirements
  - Users must be able to use (invoke) the OS.
  - Users must not be able to misuse the OS.

- Most importantly
  - The users must not be able to modify the operating system.

- If a user could change the reference monitor,
  - he could bypass it too...

- Users changing the OS, could change or bypass security features.
System Integrity

- Competing requirements
  - Users must be able to use (invoke) the OS.
  - Users must not be able to misuse the OS.
- Most importantly
  - The users must not be able to modify the operating system.
- If a user could change the reference monitor,
  - he could bypass it too...
- Users changing the OS, could change or bypass security features.
Outline

1. The session
2. The reference monitor
   - The reference monitor
3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation
4. Other issues
   - Memory protection
   - OS kernel issues
5. Conclusion
Outline

1. The session

2. The reference monitor
   - The reference monitor

3. Security at the Bottom
   - The lowest layers
     - The CPU
     - Controlled Invocation

4. Other issues
   - Memory protection
   - OS kernel issues

5. Conclusion
The Layers

- Applications
- Services
- Operating system
- OS kernel
- Hardware
The Layers

- **Hardware**
- **Operating System**
  - **OS Kernel**
- **Services**
- **Applications**
Rationale

- **Recall Design Principle #5**
  - How do you prevent circumventing the security through a lower layer?
    - The lowest layer needs protection.
  - How do you avoid a complexity preventing thorough analysis?
    - The core of the system is hopefully simple enough.
  - How do you minimise performance overhead?
    - In the kernel, you have access to primitive operations.
  - All of these suggest implementation in hardware and OS kernel.
Rationale

- Recall Design Principle #5
  - How do you prevent circumventing the security through a lower layer?
    - The lowest layer needs protection.
  - How do you avoid a complexity preventing thorough analysis?
    - The core of the system is hopefully simple enough.
  - How do you minimise performance overhead?
    - In the kernel, you have access to primitive operations.

- All of these suggest implementation in hardware and OS kernel.
Rationale

- Recall Design Principle #5
  - How do you prevent circumventing the security through a lower layer?
    - The lowest layer needs protection.
  - How do you avoid a complexity preventing thorough analysis?
    - The core of the system is hopefully simple enough.
  - How do you minimise performance overhead?
    - In the kernel, you have access to primitive operations.

- All of these suggest implementation in hardware and OS kernel.
Rationale

- Recall Design Principle #5
  - How do you prevent circumventing the security through a lower layer?
  - The lowest layer needs protection.

- How do you avoid a complexity preventing thorough analysis?
  - The core of the system is hopefully simple enough.

- How do you minimise performance overhead?
  - In the kernel, you have access to primitive operations.

- All of these suggest implementation in hardware and OS kernel.
Rationale

- Recall Design Principle #5
  - How do you prevent circumventing the security through a lower layer?
    - The lowest layer needs protection.
  - How do you avoid a complexity preventing thorough analysis?
    - The core of the system is hopefully simple enough.
  - How do you minimise performance overhead?
    - In the kernel, you have access to primitive operations.

- All of these suggest implementation in hardware and OS kernel.
Security at the Bottom

The lowest layers

Hardware

- CPU
  - ALU
  - Registers
- Bus
- Memory
- I/O
  - monitor
  - keyboard
  - network
Outline

1. The session
2. The reference monitor
   - The reference monitor
3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation
4. Other issues
   - Memory protection
   - OS kernel issues
5. Conclusion
Central Processing Unit

- The CPU is the brain and the hub of the computer
- ALU: Arithmetic Logic Unit
  - Does the work
  - Executes code (processes)
- Registers: temporary storage (workspace for ALU)
  - General purpose registers
  - Dedicated registers
    - Program counter (next instruction for execution)
    - Stack pointer (top of system stack)
    - Status register
CPU Protection Rings (80x86)
Processes share the CPU

- **A process** is a program in execution
  - corresponds to a subject in formal terms
  - associated with a user (principal)
- Different processes time-share the CPU
- Processes need protection from
  - other principals (security)
  - other processes of the same principal (reliability)
    - limits consequences of programming error
- Processes need protection of
  - Confidentiality
  - Integrity
Processes share the CPU

- A process is a program in execution
  - corresponds to a subject in formal terms
  - associated with a user (principal)

- Different processes time-share the CPU

- Processes need protection from
  - other principals (security)
  - other processes of the same principal (reliability)
    - limits consequences of programming error

- Processes need protection of
  - Confidentiality
  - Integrity
Processes share the CPU

- A process is a program in execution
  - corresponds to a subject in formal terms
  - associated with a user (principal)
- Different processes time-share the CPU
- Processes need protection from
  - other principals (security)
  - other processes of the same principal (reliability)
    - limits consequences of programming error
- Processes need protection of
  - Confidentiality
  - Integrity
Processes share the CPU

- A process is a program in execution
  - corresponds to a subject in formal terms
  - associated with a user (principal)

- Different processes time-share the CPU

- Processes need protection from
  - other principals (security)
  - other processes of the same principal (reliability)
    - limits consequences of programming error

- Processes need protection of
  - Confidentiality
  - Integrity
A **process** is a program in execution
- corresponds to a **subject** in formal terms
- associated with a **user** (principal)

Different processes time-share the CPU

Processes need protection from
- other principals (security)
- other processes of the same principal (reliability)
  - limits consequences of programming error

Processes need protection of
- Confidentiality
- Integrity
Processes share the CPU

- A process is a program in execution
  - corresponds to a subject in formal terms
  - associated with a user (principal)

- Different processes time-share the CPU

- Processes need protection from
  - other principals (security)
  - other processes of the same principal (reliability)
    - limits consequences of programming error

- Processes need protection of
  - Confidentiality
  - Integrity
The state of a process

- A process consists of
  - Executable code
  - Data
  - Execution context, e.g. CPU registers

- Fundamental unit of control.
- Each process with its own address space.
- Communication between processes very limited, e.g. in Unix:
  - Files (including pipes) have to be explicitly opened by both processes.
  - About 30 distinct signals can be sent via `kill(2)`
    - Data cannot be sent with the signal.
Threads

- The logical separation provides a basis for security
  - but context switches become expensive.
- Threads avoids the expensive context switching
- A thread are separate lines of executions within a process.
  - Share address space
  - No security barriers
  - Fast switching between threads
Threads

- The logical separation provides a basis for security but context switches become expensive.
- Threads avoids the expensive context switching

- A thread are separate lines of executions within a process.
  - Share address space
  - No security barriers
  - Fast switching between threads
Threads

- The logical separation provides a basis for security
  - but context switches become expensive.
- Threads avoids the expensive context switching
- A **thread** are separate lines of executions within a process.
  - Share address space
  - No security barriers
  - Fast switching between threads
Outline

1. The session
2. The reference monitor
   - The reference monitor
3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation
4. Other issues
   - Memory protection
   - OS kernel issues
5. Conclusion
A user process needs an operation with root privileges
- e.g. writing to memory
- the CPU does not recognise individual users and their individual privileges

What happens?
System call:
- CPU has to change protection ring.
- The OS performs the task on behalf of the user.

What can go wrong?
A user process needs an operation with root privileges
  - e.g. writing to memory
  - the CPU does not recognise individual users and their individual privileges

What happens?
System call:
  - CPU has to change protection ring.
  - The OS performs the task on behalf of the user.

What can go wrong?
A user process needs an operation with root privileges
- e.g. writing to memory
- the CPU does not recognise individual users and their individual privileges

What happens?
- System call:
  - CPU has to change protection ring.
  - The OS performs the task on behalf of the user.

What can go wrong?
Controlled Invocation

A user process needs an operation with root privileges
- e.g. writing to memory
- the CPU does not recognise individual users and their individual privileges

What happens?

System call:
- CPU has to change protection ring.
- The OS performs the task on behalf of the user.

What can go wrong?
Controlled Invocation

- A user process needs an operation with root privileges
  - e.g. writing to memory
  - the CPU does not recognise individual users and their individual privileges

- What happens?
  - System call:
    - CPU has to change protection ring.
    - The OS performs the task on behalf of the user.

- What can go wrong?
Atomic functions

- An atomic function executes as a unit
- A user cannot
  - execute selected parts of the function
  - insert execution steps in the middle of the function
- Calls with increased privileges should be restricted to
  - well-defined, atomic functions
- What happens if the atomic function is interrupted?
A bad scenario

- A user process (ring 3) has made a system call (ring 0).
- The CPU moves to ring 0 at the start of the call.
- ... moves to ring 3 at the end of the call.
- The call is interrupted (e.g. Ctrl-C) in the middle,
  - the protection ring is not reset
  - control returns to the user process, now running in ring 0.
- Secure error-handling is essential
Requirements for system calls

- A system call has to return to a safe state
  - regardless of circumstances

- What is a ‘safe state’?
  - Returning to correct protection ring
    - even on errors and interrupts
  - Leave all other units (memory, disk) in consistent state
  - Ideally, either
    - Fail, and make no changes; or
    - Complete, fully and correctly
The confused deputy problem

- Outer-ring procedure calls inner-ring function
  - e.g. to write to memory
  - inner-ring function can write to any memory address
  - How do we prevent abuse?

- Unix has *real* and *effective* user ID for every process
  - Real uid: uid of calling process (user)
  - Effective uid: uid of the controlled function

- The privileged process uses effective uid for access control.
Descriptors in 80x86

Descriptor Table

Selector

INDEX

RPL

15 3 2 1 0

descriptor  DPL
Descriptors in 80x86

Descriptor Table

Object

Descriptor

DPL

Memory segments et c.

Selector

INDEX

RPL

15

3

2

1

0
Descriptors in 80x86

Descriptor Table

Selector

15 3 2 1 0

INDEX RPL

descriptor DPL

=?
The system has to be able to handle exceptions in every layer, CPU, Kernel, process, et c. known as interrupts, traps, exceptions. caused by user request, software bugs, hardware failure.

On an exception, control moved to an exception handler.

An exception occur in an exception handler.

What happens on an exception in an atomic function?
Interrupts

The system has to be able to handle exceptions
  in every layer, CPU, Kernel, process, et c.
  known as *interrupts, traps, exceptions.*
  caused by *user request, software bugs, hardware failure*

On an exception, control moved to an exception handler.

An exception occur in an exception handler.

What happens on an exception in an atomic function?
Interrupts

- The system has to be able to handle exceptions in every layer, CPU, Kernel, process, et c.
- known as *interrupts, traps, exceptions.*
- caused by *user request, software bugs, hardware failure*

- On an exception, control moved to an exception handler.
- An exception occur in an exception handler.
- What happens on an exception in an atomic function?
Interrupts

The system has to be able to handle exceptions in every layer, CPU, Kernel, process, et c. known as *interrupts, traps, exceptions*. caused by *user request, software bugs, hardware failure*

On an exception, control moved to an exception handler. An exception occur in an exception handler.

What happens on an exception in an atomic function?
Traps at the CPU

- Trap \#n
- interrupt vector
- interrupt vector table
- memory
- interrupt handler
Traps at the CPU

- Trap \( \neq n \)
- Interrupt vector table
- Virus
- Interrupt handler
Signals to Unix processes

- Signals in Unix are similar to CPU traps
- Interrupts the normal flow of the program
- Control is transferred to a signal handler
- Signals caused by
  - Ctrl-C (Interrupt) and Ctrl-Z (stop/suspend)
  - Memory violations (terminates process by default)
  - Kill (terminates, cannot be overridden)
  - User-defined
- Signals are ignored or delayed within system calls.
  - Why is this?
Signals to Unix processes

- Signals in Unix are similar to CPU traps
- Interrupts the normal flow of the program
- Control is transferred to a signal handler
- Signals caused by
  - Ctrl-C (Interrupt) and Ctrl-Z (stop/suspend)
  - Memory violations (terminates process by default)
  - Kill (terminates, cannot be overridden)
  - User-defined

- Signals are ignored or delayed within system calls.
  - Why is this?
Signals to Unix processes

- Signals in Unix are similar to CPU traps
- Interrupts the normal flow of the program
- Control is transferred to a signal handler
- Signals caused by
  - Ctrl-C (Interrupt) and Ctrl-Z (stop/suspend)
  - Memory violations (terminates process by default)
  - Kill (terminates, cannot be overridden)
  - User-defined
- Signals are ignored or delayed within system calls.
  - Why is this?
Outline

1. The session
2. The reference monitor
   • The reference monitor
3. Security at the Bottom
   • The lowest layers
   • The CPU
   • Controlled Invocation
4. Other issues
   • Memory protection
   • OS kernel issues
5. Conclusion
Outline

1. The session
2. The reference monitor
   - The reference monitor
3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation
4. Other issues
   - Memory protection
   - OS kernel issues
5. Conclusion
Secure Addressing

- OS prevents processes from addressing memory without permission
- Instance of input checking
  - Cf. SQL injection
- Two approaches
  - Absolute addresses – OS checks against bounds
  - Relative addresses – OS translates to absolute addresses
    - Masking (or sandboxing)
    - Offset ((upper) bound must still be checked)
Secure Addressing

- OS prevents processes from addressing memory without permission
- Instance of input checking
  - Cf. SQL injection
- Two approaches
  - Absolute addresses – OS checks against bounds
  - Relative addresses – OS translates to absolute addresses
    - Masking (or sandboxing)
    - Offset ((upper) bound must still be checked)
Relative addresses

- Masking (or sandboxing)

\[
\underbrace{b'_1 b'_2 b'_3 \ldots b'_m} \underbrace{b_1 b_2 b_3 \ldots b_n}
\]

Segment address Relative address

- Offset ((upper) bound must still be checked)

\[
B' + B
\]

Segment address + Relative address
Function Codes
Motorola 68000

- Function codes signal CPU status to address interpreter

<table>
<thead>
<tr>
<th>FC2</th>
<th>FC1</th>
<th>FC0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(undefined, reserved)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>user data</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>user program</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(undefined, reserved)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(undefined, reserved)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>supervisor data</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>supervisor program</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>interrupt acknowledge</td>
</tr>
</tbody>
</table>
Tagged architectures

- Most compilers do type checking
  - could we enforce type checking at the CPU as well?
  - tagging the data with its type

- Tagged architectures are rare
  - More theoretical considerations than applications

- Security benefit?
  - Limits creative use of memory
  - ... fewer avenues to bypass security
  - Extra check on consistency
  - ... blocks some software bugs.
Tagged architectures

- Most compilers do type checking
  - could we enforce type checking at the CPU as well?
  - tagging the data with its type
- Tagged architectures are rare
  - More theoretical considerations than applications
- Security benefit?
  - Limits creative use of memory
  - ... fewer avenues to bypass security
  - Extra check on consistency
  - ... blocks some software bugs.
Outline

1. The session

2. The reference monitor
   - The reference monitor

3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation

4. Other issues
   - Memory protection
   - OS kernel issues

5. Conclusion
I/O Security Issues

- Which process reads the I/O?
- Can a subject intercept the password you type on the keyboard?
- Windows provides a secure attention sequence
  - Ctrl-Alt-Del invokes the login program.
  - Ctrl-Alt-Del cannot be redefined.
  - Only way to know which process you communicate with.
Outline

1. The session
2. The reference monitor
   - The reference monitor
3. Security at the Bottom
   - The lowest layers
   - The CPU
   - Controlled Invocation
4. Other issues
   - Memory protection
   - OS kernel issues
5. Conclusion
Summary

- Security at lower levels ensure consistency
- For operations with super-user privileges
  - atomic functions properly scrutinised
  - proper error-handling
  - unsafe exit must be prevented
- System divided in small units carefully scrutinised
- Tagging memory allows additional security checks in CPU
Microprocessors on smart cards used to have their entire card operating system in ROM. Currently, there are moves towards microprocessors where part of the operating system can be downloaded into EEPROM. What are the advantages and disadvantages of keeping the operating system in ROM? What are the security implications of moving parts of the operating system into EEPROM?