Reference Monitors

Gang Tan
Penn State University
Spring 2019

CMPSC 447, Software Security
Defense Mechanisms for Software Security

- Static enforcement of security properties
  - Analyze the code before it is run (e.g., during compile time)
  - Static analysis

- Dynamic enforcement
  - Analyze the code when it is running
    - E.g., stopping the program to prevent dangerous operations
  - AKA, reference monitors
Agenda

- Generation discussion of reference monitors
  - Safety properties
- Useful reference monitors in practice
  - OS-level reference monitors
  - Software-based fault isolation
  - ...

Reference Monitors

* Some slides adapted from the lecture notes by Greg Morrisett
Reference Monitor

- Observe the execution of a program and halt the program if it’s going to violate the security policy.
Common Examples of RM

- Operating system monitors users applications
  - Monitor system calls by user apps
  - Kernel vs user mode
  - Hardware based
- Software-based: Interpreters, language virtual machines, software-based fault isolation
- Firewalls
- ...
- Claim: majority of today’s security enforcement mechanisms are instances of reference monitors
Requirements for a Monitor

- Must have (reliable) access to information about what the program is about to do
  - e.g., what syscall is it about to execute?
- Must have the ability to “stop” the program
  - can’t stop a program running on another machine that you don’t own
  - stopping isn’t necessary; transitioning to a “good” state may be sufficient
- Must protect the monitor’s state and code from tampering
  - key reason why a kernel’s data structures and code aren’t accessible by user code
- In practice, must have low overhead
What Policies Can be Enforced?

- Some liberal assumptions:
  - Monitor can have infinite state
  - Monitor can have access to entire history of computation
  - But monitor can’t guess the future – the decision to determine whether to halt a program must be computable

- Under these assumptions:
  - There is a nice class of policies that reference monitors can enforce: **safety properties**
  - There are desirable policies that no reference monitor can enforce precisely
Analysis of the Power and Limitations of Execution Monitoring

“Enforceable Security Policies” by Fred Schneider
Execution Traces

- System behavior $\sigma$: a finite or infinite execution trace of system events
  - $\sigma = e_0 e_1 e_2 e_3 \ldots e_i \ldots$, where $e_i$ is a system event
- Example: a trace of memory operations (reads and writes)
  - Events: read(addr); write(addr, v)
- Example: a trace of system calls
  - System-call events: open(...); read(...); close(...); gettimeofday(...); fork(...); ...
- Example: a system for access control
  - Oper($p, o, r$): Principal $p$ invoked an operation involving object $o$ and requiring right $r$ to that object
  - AddP($p$, $p'$): Principal $p$ invoked an operation to create a principal named $p'$
A system modeled as a set of execution traces (its behaviors)

- $S = \{\sigma_1, \sigma_2, ..., \sigma_k, ...\}$
- Each trace corresponds to the execution for a possible input

For example

- Sets of traces of reads and writes
- Sets of traces of system calls
Definition of Security Policy

- A security policy $P(S)$: a logical predicate on sets of execution traces.
  - A target system $S$ satisfies security policy $P$ if and only if $P(S)$ holds.
- For example
  - A program cannot write to addresses outside of [0, 1000]
    - $P(S) = \forall \sigma \in S. \forall e_i \in \sigma. e_i = \text{write}(\text{addr}, v) \rightarrow \text{addr} \in [0, 1000]$ 
  - A program cannot send a network packet after reading from a local file
    - $P(S) = \forall \sigma \in S. \forall e_i \in \sigma. e_i = \text{fileRead}(\ldots) \rightarrow \forall k > i. e_k \neq \text{networkSend}(\ldots)$
Constraint on Monitors: Property

- Can a reference monitor see more than one trace at a time?
  - A reference monitor only sees one execution trace of a program
- So we can only enforce policies $P$ s.t.:
  - $\mathbf{(1)} \quad P(S) = \forall \sigma \in S. P(\sigma)$
  - where $P$ is a predicate on individual traces
- A security policy is a **property** if its predicate specifies whether an individual trace is legal
  - The membership is determined solely by the trace and not by the other traces
What is a Non-Property?

- A policy that may depend on multiple execution traces
- Information flow polices
  - Sensitive information should not flow to unauthorized person explicitly or implicitly
  - Example: a system protected by passwords
    - Suppose the password checking time correlates closely to the length of the prefix that matches the true password
      - Timing channel
    - To rule this out, a policy should say: no matter what the input is, the password checking time should be the same in all traces
      - Not a property
More on Implicit Information Flow

- Suppose \( x \) is a secret boolean variable whose value should not be observable by an adversary
  
  ```
  if (x=0) y=100; else y=1000;
  printf("y=%d", y); // y is observable by the adversary
  ```

- By observing \( y \), an adversary can infer the value of \( x \)!

- A policy to rule the above out cannot just constrain one execution trace
More Constraints on Monitors

Shouldn’t be able to “see” the future.

- Assumption: must make decisions in finite time.
- Suppose $P(\sigma)$ is false, then it must be rejected at some finite time $i$; that is, $P(\sigma[..i])$ is false

\[(2) \quad \forall \sigma. \quad \neg P(\sigma) \rightarrow (\exists i. \quad \neg P(\sigma[..i]))\]

Once a trace has been rejected by the monitor, then any further events from the system cannot make the monitor to revoke that decision

\[(3) \quad \forall \sigma. \quad \neg P(\sigma) \rightarrow (\forall \sigma'. \quad \neg P(\sigma\sigma'))\]
Reference Monitors Enforce Safety Properties

A predicate $P$ on sets of sequences s.t.

1. $P(S) = \forall \sigma \in S. P(\sigma)$
2. $\forall \sigma. \neg P(\sigma) \rightarrow (\exists i. \neg P(\sigma[..i]))$
3. $\forall \sigma. \neg P(\sigma) \rightarrow (\forall \sigma'. \neg P(\sigma\sigma'))$

is a **safety property**: “no bad thing will happen.”

Conclusion: a reference monitor can’t enforce a policy $P$ unless it’s a safety property.
Safety and Liveness Properties
[Alpern & Schneider 85,87]

- **Safety**: Some “bad thing” doesn’t happen.
  - Proscribes traces that contain some “bad” prefix
  - Example: the program won’t read memory outside of range [0,1000]

- **Liveness**: Some “good thing” does happen
  - Example: program will terminate
  - Example: program will eventually release the lock

- **Theorem**: Every security property can be decomposed into a safety property and a liveness property
Classification of Policies

“Enforceable Security Policies” [Schneider 00]
Policies Enforceable by Reference Monitors

- Reference monitor **can** enforce any safety property
  - Intuitively, the monitor can inspect the history of computation and prevent bad things from happening
- Reference monitor **cannot** enforce liveness properties
  - The monitor cannot predict the future of computation
- Reference monitor **cannot** enforce non-properties
  - The monitor cannot inspect multiple traces simultaneously
Safety has its benefits:

- They compose: if P and Q are safety properties, then P & Q is a safety property (just the intersection of allowed traces.)
- Safety properties can approximate liveness by setting limits. e.g., we can determine that a program terminates within k steps.
- We can also approximate many other security policies (e.g., info. flow) by simply choosing a stronger safety property.
Security Automata for Reference Monitors

- Non-deterministic State Automata
  - Possibly with an infinite number of states
  - Note: some infinite-state automata can be reformulated by other forms of automata (e.g., push-down automata)

![Automata Diagram]

Fig. 1. No `Send` after `FileRead`
Practical Issues

In theory, a monitor could:
- examine the entire history and the entire machine state to decide whether or not to allow a transition.
- perform an arbitrarily long computation to decide whether or not to allow a transition.

In practice, most systems:
- keep a small piece of state to track history
- only look at labels on the transitions
- have a small set of labels
- perform simple tests

Otherwise, the overheads would be overwhelming.
- so policies are practically limited by the vocabulary of labels, the complexity of the tests, and the state maintained by the monitor
OS Policies and Hardware-Based Reference Monitors
Operating Systems circa ‘75

Simple Model: system is a collection of running processes and files.
- processes perform actions on behalf of a user.
  - open, read, write files
  - read, write, execute memory, etc.
- files have access control lists dictating which users can read/write/execute/etc. the file.

(Some) High-Level Policy Goals:
- Integrity: one user’s processes shouldn’t be able to corrupt the code, data, or files of another user.
- Availability: processes should eventually gain access to resources such as the CPU or disk.
- Confidentiality? Access control?
What Can go Wrong?

- read/write/execute or change ACL of a file for which process doesn’t have proper access.
  - check file access against ACL
- process writes into memory of another process
  - isolate memory of each process (& the OS!)
- process pretends it is the OS and execute its code
  - maintain process ID and keep certain operations privileged --- need some way to transition.
- process never gives up the CPU
  - force process to yield in some finite time
- process uses up all the memory or disk
  - enforce quotas
Key Mechanisms in Hardware

- Memory isolation using per-process page tables and Translation Lookaside Buffer (TLB)
  - provides an inexpensive check for each memory access.
  - maps virtual address to physical address
    - small, fully associative cache (8-10 entries)
    - cache miss triggers a trap
    - granularity of map is a page (4-8KB)

- Distinct user and supervisor modes
  - certain operations (e.g., reload TLB, device access) require supervisor bit is set

- Invalid operations cause a trap
  - set supervisor bit and transfer control to OS routine

- Timer triggers a trap for preemption
Hardware-Based Protection

- Based on virtual memory
  - Protect one process from reading/writing to other processes’ memory locations
  - Memory safety (fault isolation) at the process level
Virtual Memory

- Each process assumes a virtual address space
- A page table translates virtual pages to physical pages
  - Dedicated hardware for acceleration: TLB

<table>
<thead>
<tr>
<th>virtual page</th>
<th>physical page</th>
<th>permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xABCDO0000</td>
<td>rw</td>
</tr>
<tr>
<td>1</td>
<td>0xA07000000</td>
<td>r</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Happens on Load/Store?

1. Hardware extracts virtual page # from the virtual address
2. Hardware looks up virtual page # from the page table to get physical page # and permissions
3. If permissions allow, then hardware performs access to the physical address

*changing a page table is a privileged instruction
*the page table will not map virtual addresses to memory locations that the process is not supposed to access, such as other processes’ memory, and where the page table itself resides
Hardware Privilege Modes

- Generally two: kernel mode & user mode (x86 has more)
- At any time, CPU is in some mode
- Dangerous (“privileged”) instructions usable only in the kernel mode
  - E.g., direct access to any physical memory location, or privilege-mode change
  - Violations trap to a fixed address in the kernel
Moving Between Privilege Modes

- **Downgrade privilege**
  - Usually have a simple instruction for this

- **Upgrade privilege**
  - Usually have a special “system call” instruction (like function call), but
    - Go to a fixed address or through a fixed jumptable
    - Change into privileged (kernel) mode
Steps in a System Call

**User Process**
- calls `f=fopen("foo")`
  - library executes “break”
- calls `fread(f,n,&buf)`
  - library executes “break”

**Kernel**
- saves context, flushes TLB, etc.
- checks UID against ACL, sets up IO buffers & file context, pushes file ptr to context on user’s stack, etc.
- restores context, clears supervisor bit
- saves context, flushes TLB, etc.
- checks f is a valid file ptr, does disk access into local buffer, copies results into user’s buffer, etc.
- restores context, clears supervisor bit
Pros and Cons of Hardware-Based Protection

- **Pros**
  - Low overhead; built into the hardware
  - Transparent for applications

- **Cons**
  - Inter-process communication is cumbersome and slow
    - Signals, remote procedural calls (RPC), pipes, sockets, shared memory, ...
    - Involves context switch (changes to a new page table)
  - Granularity of protection is per-process
    - Doesn’t protect a buffer of size, say, 100 bytes
    - Buffer overflow attacks apply