Reference monitors

Suman Jana

*Original slides from Vitaly Shmatikov
Reference Monitor

• Observes execution of the program/process
  – At what level? Possibilities: hardware, OS, network

• Halts or confines execution if the program is about to violate the security policy
  – What’s a “security policy”?
  – Which system events are relevant to the policy?
    • Instructions, memory accesses, system calls, network packets...

• Cannot be circumvented by the monitored process
Enforceable Security Policies

• Reference monitors can only enforce safety policies [Schneider ‘98]
  – Execution of a process is a sequence of states
  – Safety policy is a predicate on a prefix of the sequence
    • Policy must depend only on the past of a particular execution; once it becomes false, it’s always false

• Not policies that require knowledge of the future
  – “If this server accepts a SYN packet, it will eventually send a response”

• Not policies that deal with all possible executions
  – “This program should never reveal a secret”
Reference Monitor Implementation

- Policies can depend on application semantics
- Enforcement doesn’t require context switches in the kernel
- Lower performance overhead

Integrate reference monitor into program code during compilation or via binary rewriting
What Makes a Process Safe?

• **Memory safety:** all memory accesses are “correct”
  – Respect array bounds, don’t stomp on another process’s memory, don’t execute data as if it were code

• **Control-flow safety:** all control transfers are envisioned by the original program
  – No arbitrary jumps, no calls to library routines that the original program did not call

• **Type safety:** all function calls and operations have arguments of correct type
OS as a Reference Monitor

• Collection of running processes and files
  – Processes are associated with users
  – Files have access control lists (ACLs) saying which users can read/write/execute them

• OS enforces a variety of safety policies
  – File accesses are checked against file’s ACL
  – Process cannot write into memory of another process
  – Some operations require superuser privileges
    • But may need to switch back and forth (e.g., setuid in Unix)
  – Enforce CPU sharing, disk quotas, etc.

• Same policy for all processes of the same user
Hardware Mechanisms: TLB

• TLB: Translation Lookaside Buffer
  – Maps virtual to physical addresses
  – Located next to the cache
  – Only supervisor process can manipulate TLB
    • But if OS is compromised, malicious code can abuse TLB to make itself invisible in virtual memory (Shadow Walker)

• TLB miss raises a page fault exception
  – Control is transferred to OS (in supervisor mode)
  – OS brings the missing page to the memory

• This is an expensive context switch
Steps in a System Call

[Morrisett]

**User Process**
- Calls `f=fopen("foo")`
- Library executes “break”

**Kernel**
- Saves context, flushes TLB, etc.
- Checks UID against ACL, sets up IO buffers & file context, pushes ptr to context on user’s stack, etc.
- Restores context, clears supervisor bit

**User Process (cont.)**
- Calls `fread(f,n,&buf)`
- Library executes “break”

**Kernel (cont.)**
- Saves context, flushes TLB, etc.
- Checks `f` is a valid file context, does disk access into local buffer, copies results into user’s buffer, etc.
- Restores context, clears supervisor bit
Midterm grades

Secure Software Development Midterm Scores

mean: 65.16
std dev: 14.68
median: 61
max: 96
min: 38
25%: 55.75
75%: 77.75
Modern Hardware Meets Security

• Modern hardware: large number of registers, big memory pages

• **Isolation** ⇒ each process should live in its own hardware address space

• ... but the performance cost of inter-process communication is increasing
  – Context switches are very expensive
  – Trapping into OS kernel requires flushing TLB and cache, computing jump destination, copying memory

• Conflict: *isolation vs. cheap communication*
Software Fault Isolation (SFI)

[Wahbe et al. SOSP ‘93]

• Processes live in the same hardware address space; software reference monitor isolates them
  – Each process is assigned a logical “fault domain”
  – Check all memory references and jumps to ensure they don’t leave process’s domain

• Tradeoff: checking vs. communication
  – Pay the cost of executing checks for each memory write and control transfer to save the cost of context switching when trapping into the kernel
Fault Domains

• Process’s code and data in one memory segment
  – Identified by a unique pattern of upper bits
  – Code is separate from data (heap, stack, etc.)
  – Think of a fault domain as a “sandbox”

• Binary modified so that it cannot escape domain
  – Addresses are masked so that all memory writes are to addresses within the segment
    • Coarse-grained memory safety (vs. array bounds checking)
  – Code is inserted before each jump to ensure that the destination is within the segment

• Does this help much against buffer overflows?
Verifying Jumps and Stores

• If target address can be determined statically, mask it with the segment’s upper bits
  – Crash, but won’t stomp on another process’s memory
• If address unknown until runtime, insert checking code before the instruction
• Ensure that code can’t jump around the checks
  – Target address held in a dedicated register
  – Its value is changed only by inserted code, atomically, and only with a value from the data segment
Simple SFI Example

- Fault domain = from 0x1200 to 0x12FF
- Original code: `write x`
- Naïve SFI:  
  ```
  x := x & 00FF
  ...
  x := x | 1200
  write x
  ```

- Better SFI:  
  ```
  tmp := x & 00FF
  tmp := tmp | 1200
  write tmp
  ```

convert `x` into an address that lies within the fault domain

What if the code jumps right here?
• Generalize SFI to more general safety policies than just memory safety
  – Policy specified in some formal language
  – Policy deals with application-level concepts: access to system resources, network events, etc.
    • “No process should send to the network after reading a file”, “No process should open more than 3 windows”, ...

• Policy checks are integrated into the binary code
  – Via binary rewriting or when compiling

• Inserted checks should be uncircumventable
  – Rely on SFI for basic memory safety
Policy Specification in SASI

SASI policies are finite-state automata

- Can express any safety policy
- Easy to analyze, emulate, compile
- Written in SAL language (textual version of diagrams)
Policy Enforcement

• Checking before every instruction is an overkill
  – Check “No division by zero” only before DIV
• SASI uses partial evaluation
  – Insert policy checks before every instruction, then rely on static analysis to eliminate unnecessary checks
• There is a “semantic gap” between individual instructions and policy-level events
  – Applications use abstractions such as strings, types, files, function calls, etc.
  – Reference monitor must synthesize these abstractions from low-level assembly code
M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti

Control-Flow Integrity:
Principles, Implementations, and Applications

(CCS 2005)
CFI: Control-Flow Integrity

[Abadi et al.]

• Main idea: pre-determine control flow graph (CFG) of an application
  – Static analysis of source code
  – Static binary analysis ← CFI
  – Execution profiling
  – Explicit specification of security policy

• Execution must follow the pre-determined control flow graph
CFI: Binary Instrumentation

• Use binary rewriting to instrument code with runtime checks (similar to SFI)
• Inserted checks ensure that the execution always stays within the statically determined CFG
  – Whenever an instruction transfers control, destination must be valid according to the CFG
• Goal: prevent injection of arbitrary code and invalid control transfers (e.g., return-oriented-programming)
  – Secure even if the attacker has complete control over the thread’s address space
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort( a, len, lt );
    sort( b, len, gt );
}
CFI: Control Flow Enforcement

• For each control transfer, determine statically its possible destination(s)

• Insert a **unique bit pattern at every destination**
  – Two destinations are equivalent if CFG contains edges to each from the same source
    • This is imprecise *(why?)*
  – Use same bit pattern for equivalent destinations

• Insert binary code that at runtime will check whether the bit pattern of the target instruction matches the pattern of possible destinations
CFI: Example of Instrumentation

Original code

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Opcode bytes</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jmp ecx; computed jump</td>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4]; dst</td>
</tr>
</tbody>
</table>

Instrumented code

- Jump to the destination only if the tag is equal to “12345678”
- Abuse an x86 assembly instruction to insert “12345678” tag into the binary
CFI: Preventing Circumvention

• Unique IDs
  – Bit patterns chosen as destination IDs must not appear anywhere else in the code memory except ID checks

• Non-writable code
  – Program should not modify code memory at runtime
    • What about run-time code generation and self-modification?

• Non-executable data
  – Program should not execute data as if it were code

• Enforcement: hardware support + prohibit system calls that change protection state + verification at load-time
Improving CFI Precision

• Suppose a call from A goes to C, and a call from B goes to either C, or D (when can this happen?)
  – CFI will use the same tag for C and D, but this allows an “invalid” call from A to D
  – Possible solution: duplicate code or inline
  – Possible solution: multiple tags

• Function F is called first from A, then from B; what’s a valid destination for its return?
  – CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
  – Solution: shadow call stack
CFI: Security Guarantees

• Effective against attacks based on illegitimate control-flow transfer
  – Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge

• Does not protect against attacks that do not violate the program’s original CFG
  – Incorrect arguments to system calls
  – Substitution of file names
  – Other data-only attacks
Possible Execution of Memory

[Erlingsson]

Possible control flow destination
Safe code/data

Data memory

Code memory for function A

Code memory for function B

x86  x86/NX  RISC/NX  x86/CFI
Next Step: XFI

[Erlingsson et al. OSDI ‘06]

• Inline reference monitor added via binary rewriting
  – Can be applied to some legacy code
• CFI to prevent circumvention
• Fine-grained access control policies for memory regions
  – More than simple memory safety (cf. SFI)
• Relies in part on load-time verification
  – Similar to “proof-carrying code”
Two Stacks

• XFI maintains a separate “scoped stack” with return addresses and some local variables
  – Keeps track of function calls, returns and exceptions

• Secure storage area for function-local information
  – Cannot be overflowing, accessed via a computed reference or pointer, etc.
  – Stack integrity ensured by software guards
  – Presence of guards is determined by static verification when program is loaded

• Separate “allocation stack” for arrays and local variables whose address can be passed around
XFI: Memory Access Control

• Module has access to its own memory
  – With restrictions (e.g., shouldn’t be able to corrupt its own scoped stack)

• Host can also grant access to other contiguous memory regions
  – Fine-grained: can restrict access to a single byte
  – Access to constant addresses and scoped stack verified statically
  – Inline memory guards verify other accesses at runtime
    • Fast inline verification for a certain address range; if fails, call special routines that check access control data structures
**XFI: Preventing Circumvention**

- Integrity of the XFI protection environment
  - Basic control-flow integrity
  - “ Scoped stack” prevents out-of-order execution paths even if they match control-flow graph
  - Dangerous instructions are never executed or their execution is restricted
    - For example, privileged instructions that change protection state, modify x86 flags, etc.
- Therefore, XFI modules can even run in kernel