Inline Reference Monitors: SFI, CFI, XFI, WIT, NaCl

Vitaly Shmatikov
Reading Assignment

- Akritidis et al. “Preventing memory error exploits with WIT” (Oakland 2008).
- Yee et al. “Native Client: a sandbox for portable, untrusted x86 native code” (Oakland 2009).
Reference Monitor

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

- Halts or contains 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

- Most enforcement mechanisms we will see are example of reference monitors
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

Kernelized

Program

Kernel

RM

Wrapper

RM

Program

Kernel

Integrate reference monitor into program code during compilation or via binary rewriting

Modified program

Program

RM

Kernel

Kernel

slide 5
What Makes a Process Safe?

- **Memory safety**: all memory accesses are “correct”
  - Respect array bounds, don’t stomp on another process’s memory, separation between code and data

- **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
    - …but wait until we see mimicry attacks

- **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

**User Process**
- calls `f=fopen("foo")`
- library executes "break"

**Kernel**
- trap
- 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

- calls `fread(f,n,&buf)`
- library executes "break"

- 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
Modern Hardware Meets Security

Modern hardware: large number of registers, big memory pages

Principle of least privilege $\Rightarrow$ 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 access 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 masked so that all memory writes are to addresses within the segment
    - Coarse-grained memory safety (viz. array bounds checking)
  - Code 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
- Mainly concerned with executing untrusted code
Simple SFI Example

- Fault domain = from 0x1200 to 0x12FF
- Original code: write x
- Naïve SFI:
  \[
  x := x \& 00FF \\
  x := x \mid 1200
  \]
- Better SFI:
  \[
  \text{tmp} := x \& 00FF \\
  \text{tmp} := \text{tmp} \mid 1200 \\
  \text{write tmp}
  \]

What if the code jumps right here?
Inline Reference Monitor

- Generalize SFI to more general safety policies than just memory safety
  - Policy specified in some formal language
  - Policy deal 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)

No division by zero

No network send after file read

¬ (op = “div” ∧ arg2 = 0)

¬ \text{read} \quad \text{send}

¬ \text{read} \quad \text{read}
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
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
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-to-libc)
  - 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

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Opcode bytes</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS 77 56 34 12</td>
<td>mov eax, 12345677h ; load ID-1</td>
<td>3E 0F 18 05</td>
<td>prefetchnta ; label</td>
</tr>
<tr>
<td>40</td>
<td>inc eax ; add 1 for ID</td>
<td>78 56 34 12</td>
<td>[12345678h] ; ID</td>
</tr>
<tr>
<td>39 41 04</td>
<td>cmp [ecx+4], eax ; compare w/dst</td>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4] ; dst</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label ; if != fail</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx ; jump to label</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- 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

- Inline reference monitor added via binary rewriting
  - Can be applied to some legacy code
- Uses CFI as a building block to prevent circumvention
- Supports 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 overflown, 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
WIT: Write Integrity Testing

Combines static analysis …

- For each memory write, compute the set of memory locations that may be the destination of the write
- For each indirect control transfer, compute the set of addresses that may be the destination of the transfer
- “Color table” assigns matching colors to instruction (write or jump) and all statically valid destinations
  - Is this sound? Complete?

… with dynamic enforcement

- Code is instrumented with runtime checks to verify that destination of write or jump has the right color

[Akritidis et al.]
WIT: Write Safety Analysis

◆ Start with off-the-shelf points-to analysis
  • Gives a conservative set of possible values for each ptr

◆ A memory write instruction is “safe” if…
  • It has no explicit destination operand, or destination operand is a temporary, local or global variable
    - Such instructions either modify registers, or a constant number of bytes starting at a constant offset from the frame pointer or the data segment (example?)
  • … or writes through a pointer that is always in bounds
    - How do we know statically that a pointer is always in bounds?

◆ Safe instructions require no runtime checks
◆ Can also infer safe destinations (how?)
WIT: Runtime Checks

- Statically, assign a distinct color to each unsafe write instruction and all of its possible destinations
  - What if some destination can be written by two different instructions? Any security implications?

- Add a runtime check that destination color matches the statically assigned color
  - What attack is this intended to prevent?

- Same for indirect (computed) control transfers
  - Except for indirect jumps to library functions (done through pointers which are protected by write safety)
  - How is this different from CFI? Hint: think RET address
WIT: Additional Protections

- Change layout of stack frames to segregate safe and unsafe local variables
- Surround unsafe objects by guards/canaries
  - What attack is this intended to prevent? How?
- Wrappers for malloc()/calloc() and free()
  - malloc() assigns color to newly allocated memory
  - free() is complicated
    - Has the same (statically computed) color as the freed object
    - At runtime, treated as an unsafe write to this object
    - Reset color of object to 0 – what attack does this prevent?
    - Several other subtle details and checks – read the paper!
WIT: Handling Libraries

◆ Basic WIT doesn’t work for libraries (why?)
◆ Instead, assign the same, standard color to all unsafe objects allocated by library functions and surround them by guards
  • Different from the colors of safe objects and guards
  • Prevents buffer overflows
  • What attack does this not prevent?
◆ Wrappers for memory copying functions
  • For example, memcpy() and strcpy()
  • Receive color of the destination as an extra argument, check at runtime that it matches static color
Native Client

◆ Goal: download an x86 binary and run it “safely”
  • Much better performance than JavaScript, Java, etc.
◆ ActiveX: verify signature, then unrestricted
  • Critically depends on user’s understanding of trust
◆ .NET controls: IL bytecode + verification
◆ Native Client: sandbox for untrusted x86 code
  • Restricted subset of x86 assembly
  • SFI-like sandbox ensures memory safety
  • Restricted system interface
  • (Close to) native performance

[Yee et al. - Google]
NaCl Sandbox

◆ Code is restricted to a subset of x86 assembly
  • Enables reliable disassembly and efficient validation
  • No unsafe instructions
    – syscall, int, ret, memory-dependent jmp and call, privileged instructions, modifications of segment state …

◆ No loads or stores outside dedicated segment
  • Address space constrained to 0 mod 32 segment
  • Similar to SFI

◆ Control-flow integrity
## Constraints for NaCl Binaries

| C1 | Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution. |
| C2 | The binary is statically linked at a start address of zero, with the first byte of text at 64K. |
| C3 | All indirect control transfers use a `nacljmp` pseudo-instruction (defined below). |
| C4 | The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4). |
| C5 | The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary. |
| C6 | All *valid* instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address. |
| C7 | All direct control transfers target valid instructions. |
Control-Flow Integrity in NaCl

- For each direct branch, statically compute target and verify that it’s a valid instruction
  - Must be reachable by fall-through disassembly
- Indirect branches must be encoded as
  
  ```
  and  %eax, 0xffffffffe0
  jmp  *%eax
  ```
  
  - Guarantees that target is 32-byte aligned
  - Works because of restriction to the zero-based segment
  - Very efficient enforcement of control-flow integrity

- No RET
  - Sandboxing sequence, then indirect jump
Interacting with Host Machine

◆ **Trusted runtime environment** for thread creation, memory management, other system services

◆ **Untrusted → trusted control transfer: trampolines**
  - Start at 0 mod 32 addresses (*why?*) in the first 64K of the NaCl module address space
    - First 4K are read- and write-protected (*why?*)
  - Reset registers, restore thread stack (outside module’s address space), invoke trusted service handlers

◆ **Trusted → untrusted control transfer: springboard**
  - Start at non-0 mod 32 addresses (*why*)
  - Can jump to any untrusted address, start threads
Other Aspects of NaCl Sandbox

- No hardware exceptions or external interrupts
  - Because segment register is used for isolation, stack appears invalid to the OS ⇒ no way to handle

- No network access via OS, only via JavaScript in browser
  - No system calls such as connect() and accept()
  - JavaScript networking is subject to same-origin policy

- IMC: inter-module communication service
  - Special IPC socket-like abstraction
  - Accessible from JavaScript via DOM object, can be passed around and used to establish shared memory