

CS 5435

Isolation, Confinement, Sandboxing

Vitaly Shmatikov

Intrusion Detection Techniques

Misuse detection

- Use attack “signatures” - need a model of the attack
- Must know in advance what attacker will do (how?)
- Can only detect known attacks

Anomaly detection

- Using a model of normal system behavior, try to detect deviations and abnormalities
- Can potentially detect unknown attacks

Which is harder to do?

Reference Monitor

Observes execution of the program/process

- At what level? Instructions, memory accesses, system calls, network packets...

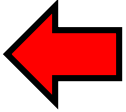
Halts or confines execution if the program is about to violate the security policy

- What's a "security policy"?
- Which system events are relevant to the policy?

Cannot be circumvented by the monitored process

Level of Monitoring

Which types of events to monitor?

- OS system calls 
- Command line
- Network data (e.g., from routers and firewalls)
- Keystrokes
- File and device accesses
- Memory accesses

Auditing / monitoring should be scalable

Enforceable Security Policies

[Schneider 1998]

Reference monitors can only enforce
safety policies

- 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

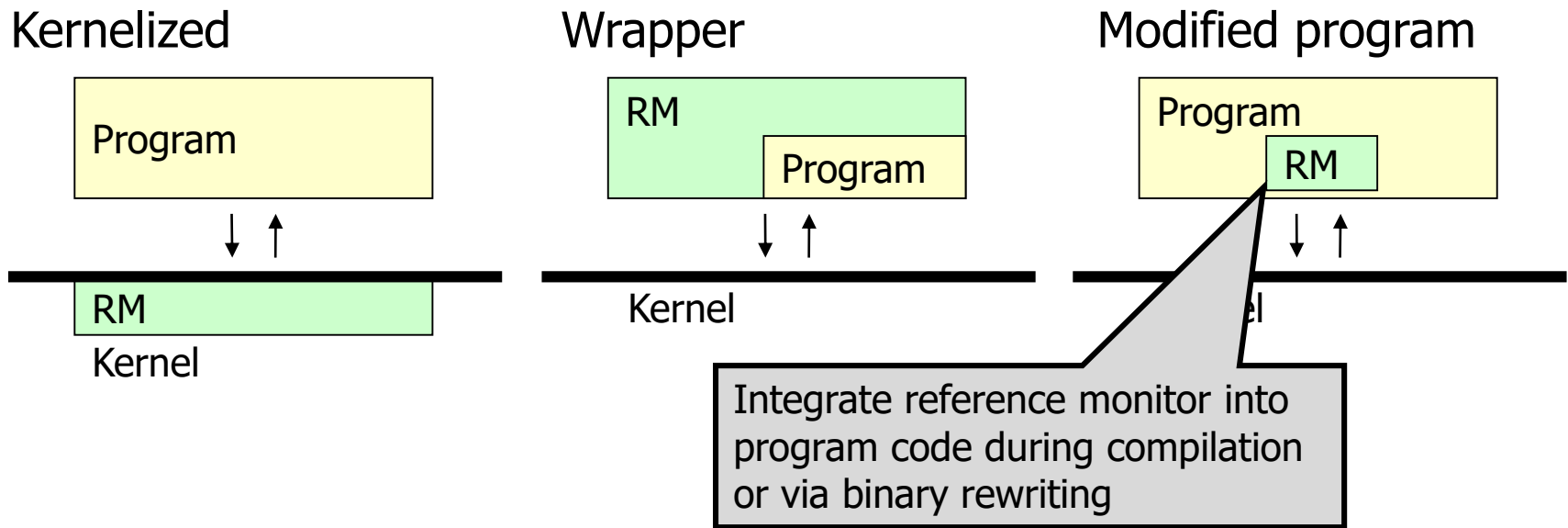
Not policies that require knowledge of the future

- “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

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

chroot() Jail

In Unix, chroot() changes root directory

- Confines code to limited portion of file system
- Sample use:

```
chdir /tmp/ghostview
```

```
chroot /tmp/ghostview
```

```
su tmpuser (or su nobody)
```

now “/tmp/ghostview” is prepended to all paths

Potential problems

- chroot changes root directory, but not current dir
 - If forget chdir, program can escape from changed root
- If you forget to change UID, process could escape

Only Root Should Execute chroot()

Otherwise, jailed program can escape

```
mkdir(/temp)          /* create temp directory      */
chroot(/temp)          /* now current dir is outside jail    */
chdir("../..../.")     /* move current dir to true root dir */
OS prevents traversal only if current root is on the path... is it?
chroot(".")            /* out of jail                        */
```

Otherwise, anyone can become root

- Create fake password file `/tmp/etc/passwd`
- Do `chroot("/tmp")`
- Run `login` or `su` (if available in chroot jail)
 - Instead of seeing real `/etc/passwd`, it will see the forgery

Many Ways to Escape Jail as Root

Create device that lets you access raw disk

Send signals to non chrooted process

Reboot system

Bind to privileged ports

jail()

First appeared in FreeBSD

- `jail jail-path hostname IP-addr cmd`

Stronger than chroot()

- No “../..” escape
- Can only bind to sockets with specified IP address and authorized ports
- Can only communicate with processes in the same jail
- Root is limited, e.g. cannot load kernel modules

Extra Programs Needed in Jail

Files needed for /bin/sh

- /usr/ld.so.1 shared object libraries
- /dev/zero clear memory used by shared objs
- /usr/lib/libc.so.1 general C library
- /usr/lib/libdl.so.1 dynamic linking access library
- /usr/lib/libw.so.1 Internationalization library
- /usr/lib/libintl.so.1 Internationalization library

Files needed for perl

- 2610 files and 192 directories

jailkit

Auto-builds files, libs, and dirs needed in jail environment

- **jk_init:** creates jail environment
- **jk_check:** checks jail env for security problems
 - checks for any modified programs,
 - checks for world-writable directories, etc.
- **jk_lsh:** restricted shell to be used inside jail

Problems with Chroot and Jail

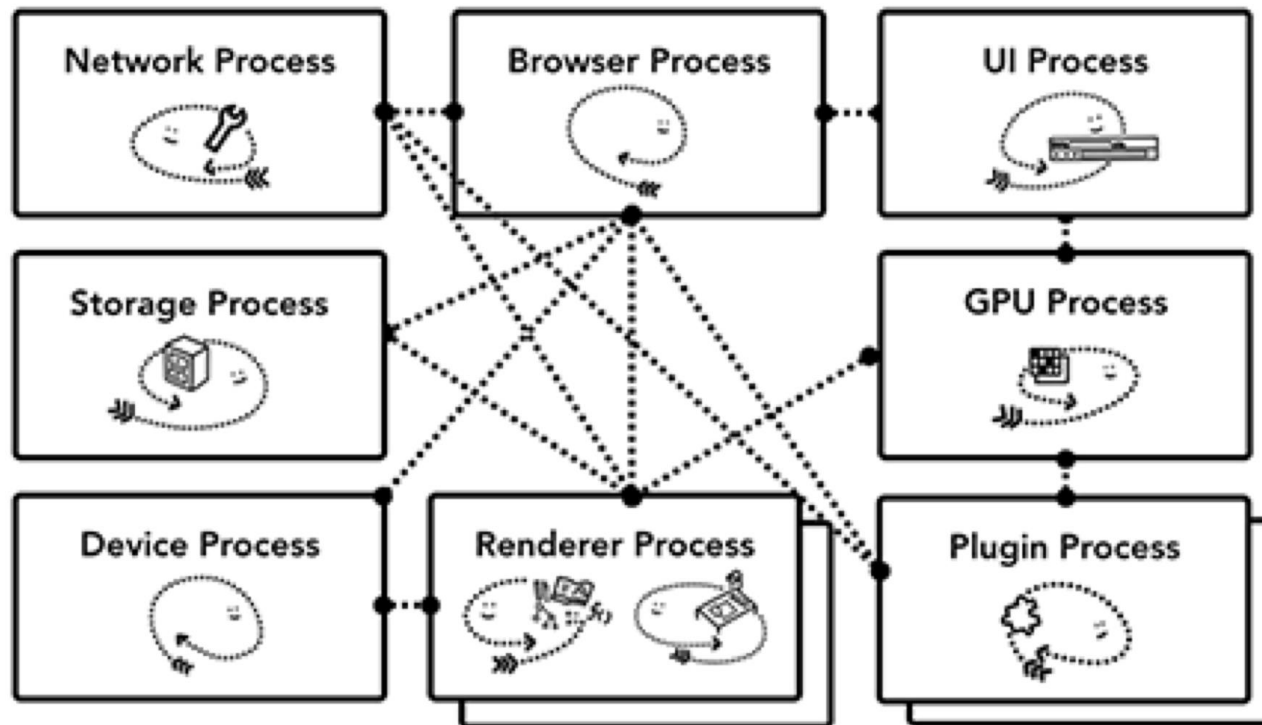
Too coarse

- All or nothing access to parts of file system
- Inappropriate for apps like a web browser
 - Needs read access to files outside jail (e.g., for sending attachments in Gmail)

Does not prevent malicious apps from...

- Accessing network and messing with other machines
- Trying to crash host OS

Chrome Architecture



Chrome Processes

Browser Process

Controls "chrome" part of the application like address bar and, bookmarks. Also handles the invisible, privileged parts of a web browser like network requests.

Renderer Process

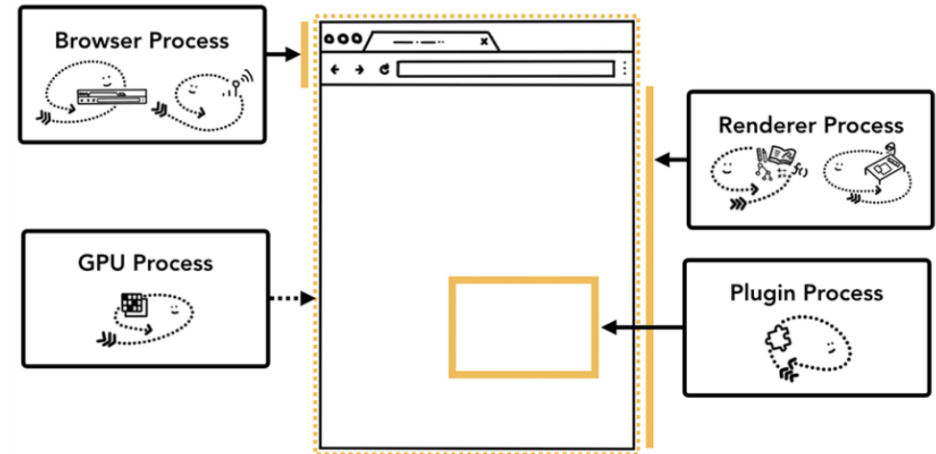
Controls anything inside of the tab where a website is displayed.

Plugin Process

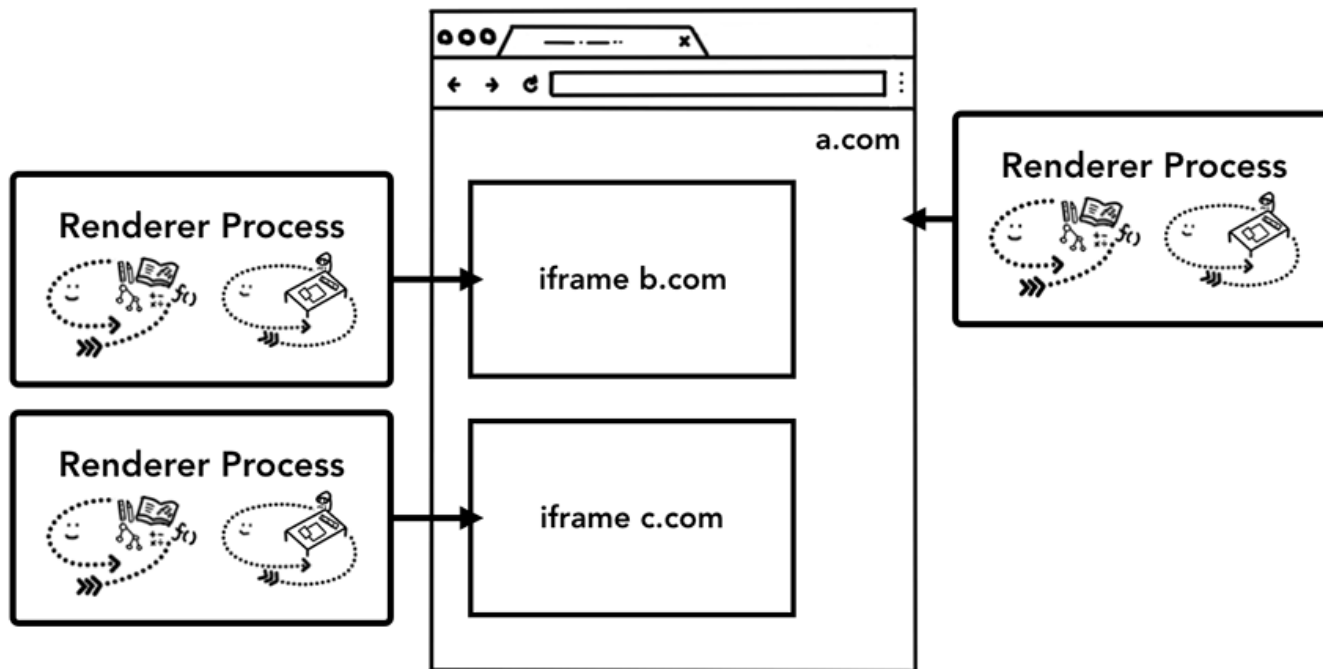
Controls any plugins used by the website, for example, flash.

GPU Process

Handles GPU tasks in isolation from other processes. It is separated into different process because GPUs handles requests from multiple apps and draw them in the same surface



Process-Based Site Isolation

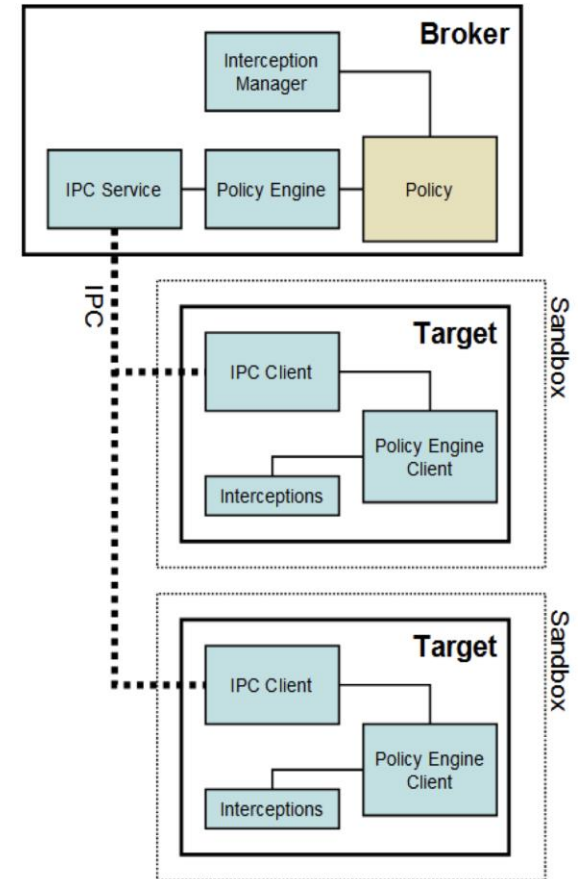


Chrome Architecture

Broker (Main Browser)

Privileged controller/supervisor of the activities of the sandboxed processes

Renderer's only access to the network is via its parent browser process and file system access can be restricted



Restricted Security Context

Chrome calls **CreateRestrictedToken** to create a token that has a subset of the user's privileges

Assigns the token the user and group **S-1-0-0 Nobody**, removes access to nearly every system resource

As long as the disk root directories have non-null security, no files (even with null ACLs) can be accessed

No network access

Windows “Job” Object

Renderer runs as a “Job” object rather than an interactive process.

Eliminates access to:

- desktop and display settings
- clipboard
- creating subprocesses
- access to global atoms table

Alternate Windows Desktop

Windows on the same desktop are effectively in the same security context because the sending and receiving of window messages is not subject to any security checks.

Sending messages across desktops is not allowed.

Chrome creates an additional desktop for target processes

Isolates the sandboxed processes from snooping in the user's interactions

System Call Interposition (SCI)

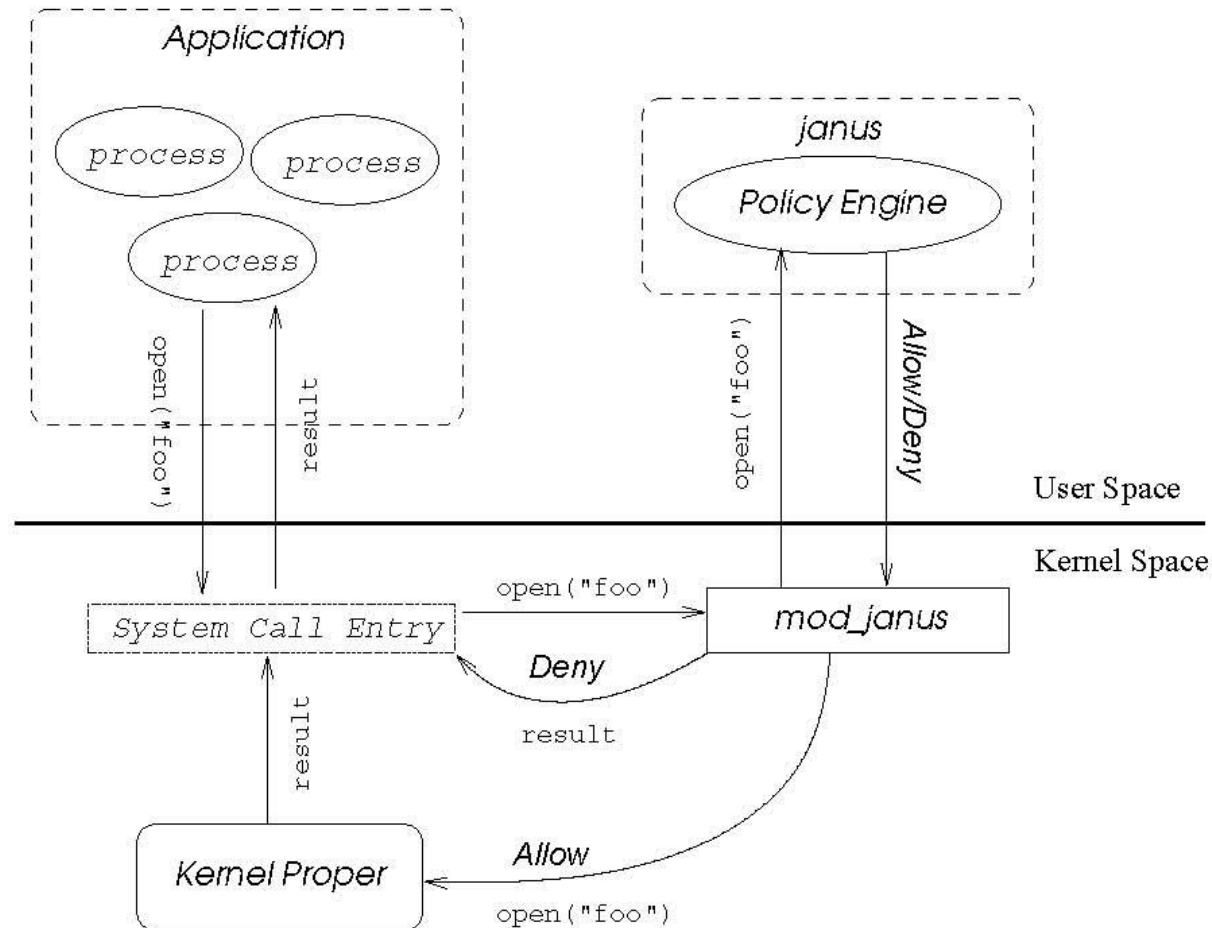
Observation: all sensitive system resources (files, sockets, etc.) are accessed via OS system calls

Idea: monitor all system calls and block those that violate security policy

- Inline, as part of the program's code
- Language-level
 - Example: Java runtime environment inspects the stack of the function attempting to access a sensitive resource to check whether it is permitted to do so
- Common OS-level approach: **system call wrapper**
 - Want to do this without modifying OS kernel (why?)

Janus

[Berkeley project, 1996]



Policy Design

Designing a good system call policy is not easy

When should a system call be permitted and when should it be denied?

Example: policy for a PDF reader

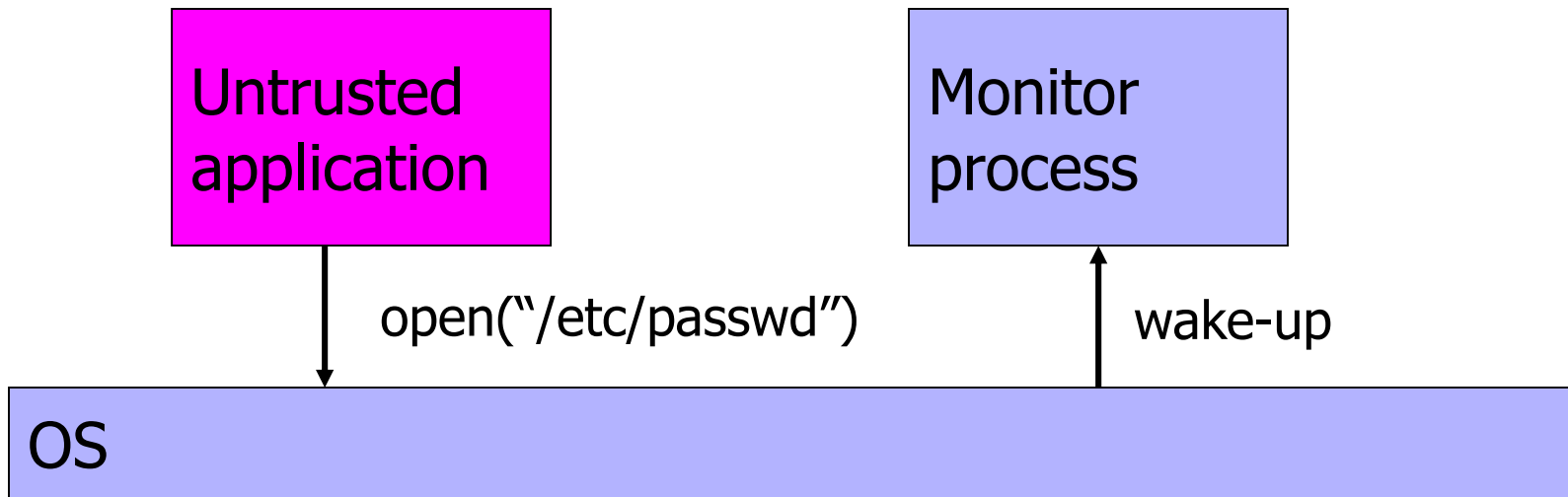
path allow /tmp/

path deny /etc/passwd

network deny all

Manually written policies tedious and error-prone

Trapping System Calls: ptrace()



ptrace() – can register a callback that will be called whenever process makes a system call

- Coarse: trace all calls or none
- If traced process forks, must fork the monitor, too

Note: Janus used ptrace initially, later discarded...

Traps and Pitfalls

[Garfinkel. "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools". NDSS 2003]

Incorrectly mirroring OS state

Overlooking indirect paths to resources

- Inter-process sockets, core dumps

Race conditions (TOCTTOU)

- Symbolic links, relative paths, shared thread meta-data

Unintended consequences of denying OS calls

- Process dropped privileges using setuid but didn't check value returned by setuid... and monitor denied the call

Bugs in reference monitors and safety checks

- What if runtime environment has a buffer overflow?

Incorrectly Mirroring OS State

[Garfinkel]

Policy: “process can bind TCP sockets on port 80,
but cannot bind UDP sockets”

`X = socket(UDP, ...)`

Monitor: “X is a UDP socket”

`Y = socket(TCP, ...)`

Monitor: “Y is a TCP socket”

`close(Y)`

`dup2(X,Y)`

Monitor's state now inconsistent with OS

`bind(Y, ...)`

Monitor: “Y is a TCP socket, Ok to bind”

Oops!

TOCTTOU in Syscall Interposition

User-level program makes a system call

- Direct arguments in stack variables or registers
- Indirect arguments are passed as pointers

Wrapper enforces some security policy

- Arguments are copied into kernel memory and analyzed and/or substituted by the syscall wrapper

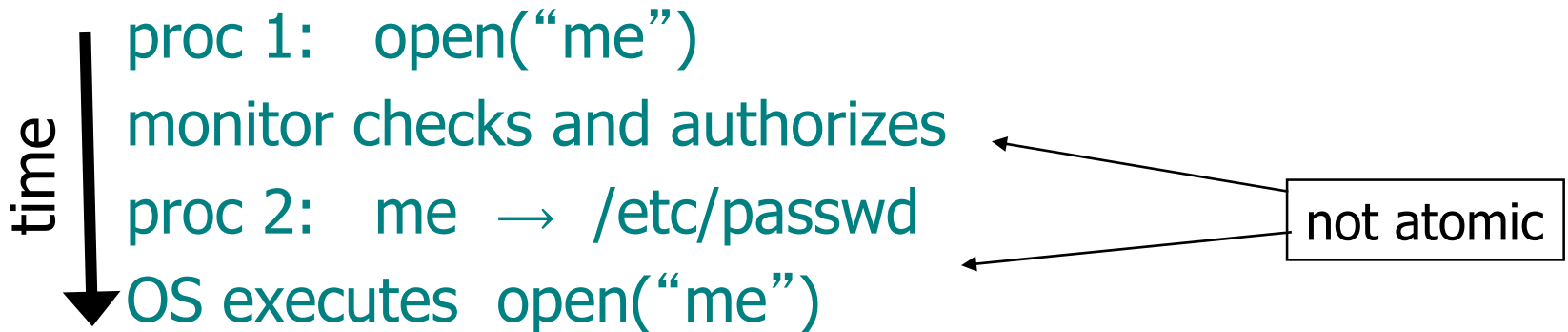
What if arguments change right here?

If permitted by the wrapper, the call proceeds

- Arguments are copied into kernel memory
- Kernel executes the call

Example of a TOCTTOU Attack

symlink: me → mydata.dat



Exploiting TOCTTOU Conditions

[Watson. "Exploiting Concurrency Vulnerabilities in System Call Wrappers". WOOT 2007]

Forced wait on disk I/O

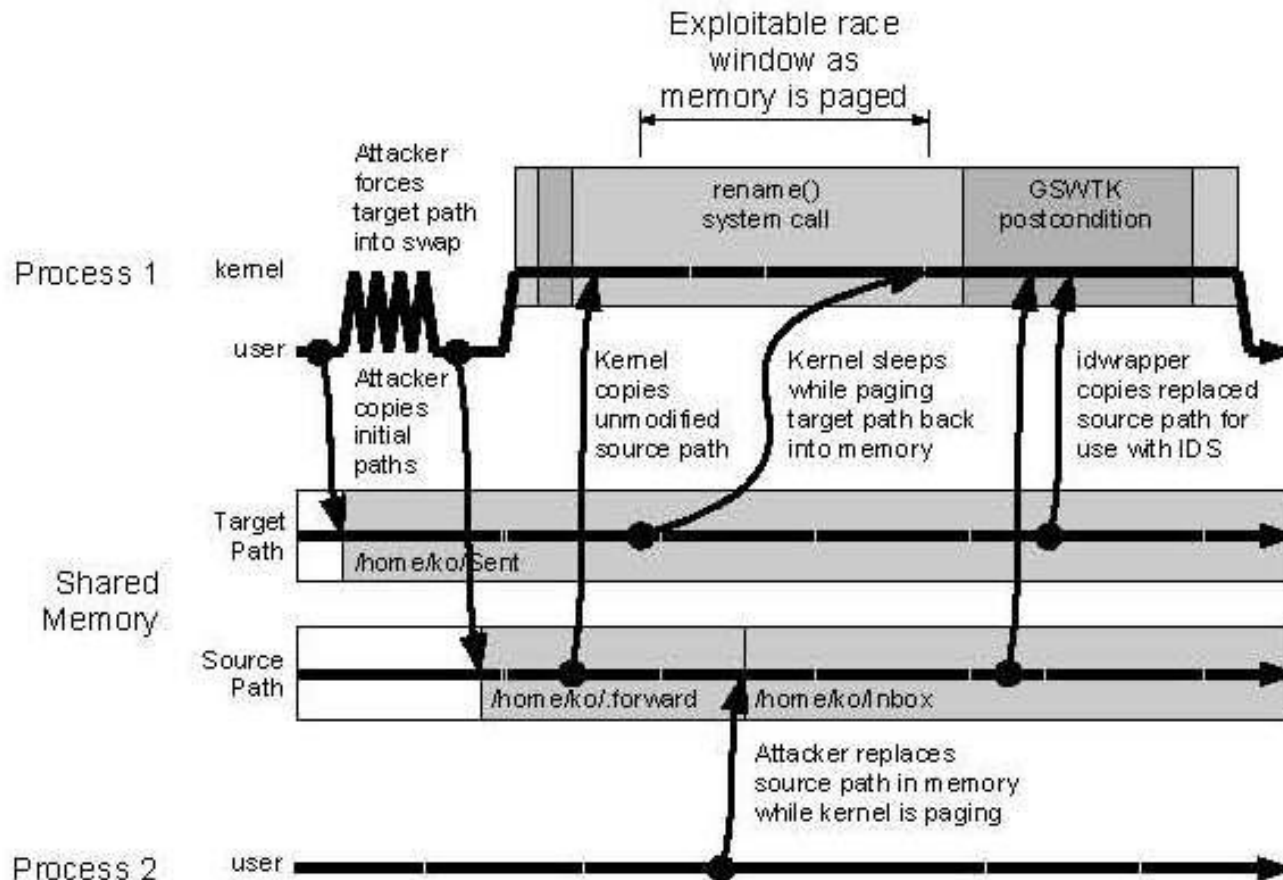
- Example: `rename()`
 - Attacker causes the target path of `rename()` to page out to disk
 - Kernel copies in the source path, then waits for target path
 - Concurrent attack process replaces the source path
 - Postcondition checker sees the replaced source path

Voluntary thread sleeps

- Example: `TCP connect()`
 - Kernel copies in the arguments
 - Thread calling `connect()` waits for a TCP ACK
 - Concurrent attack process replaces the arguments

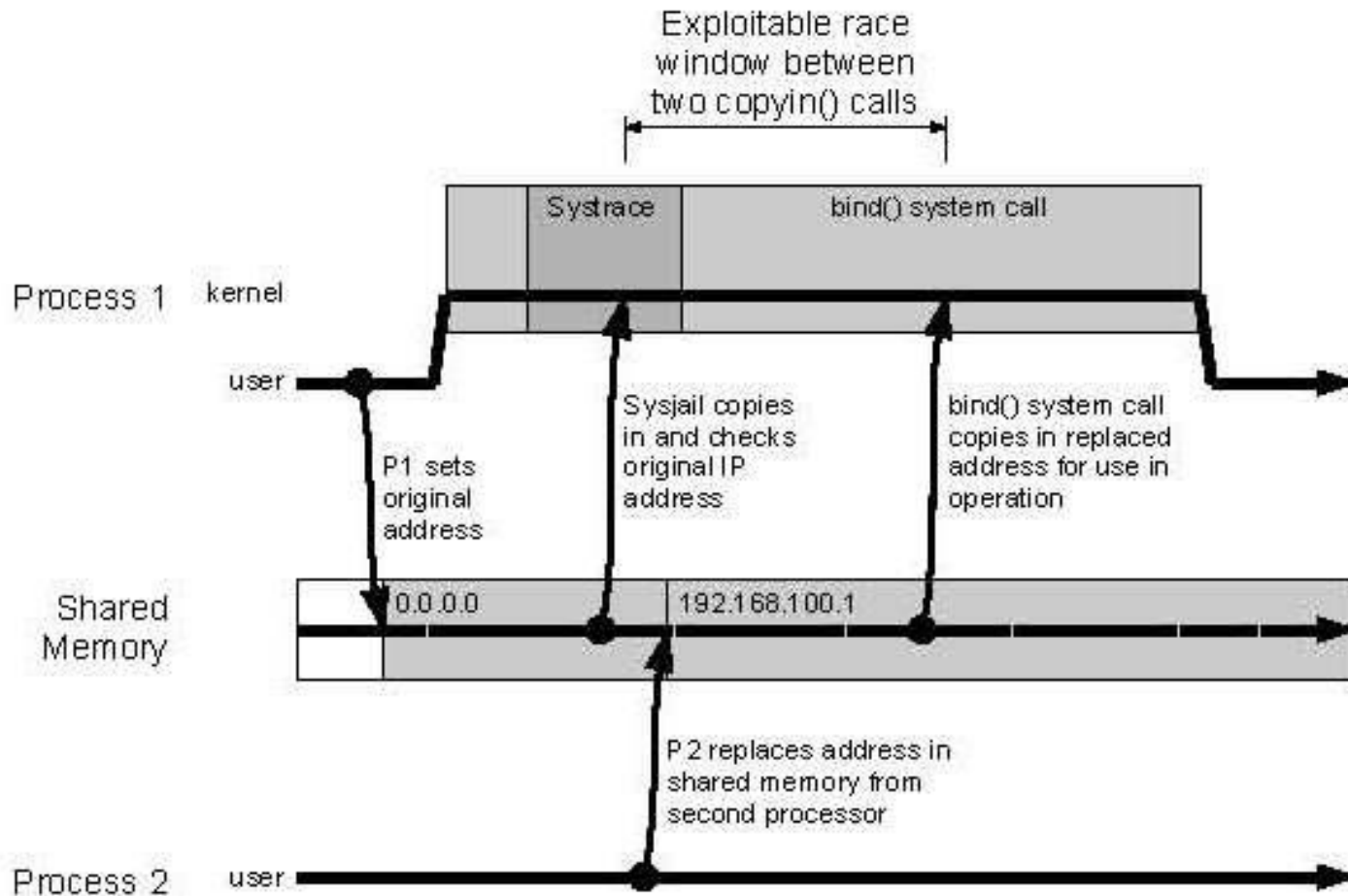
TOCTTOU via a Page Fault

[Watson]



TOCTTOU on Sysjail

[Watson]



Mitigating TOCTTOU

Make pages with syscall arguments read-only

- Tricky implementation issues
- Prevents concurrent access to data on the same page

Avoid shared memory between user process, syscall wrapper and the kernel

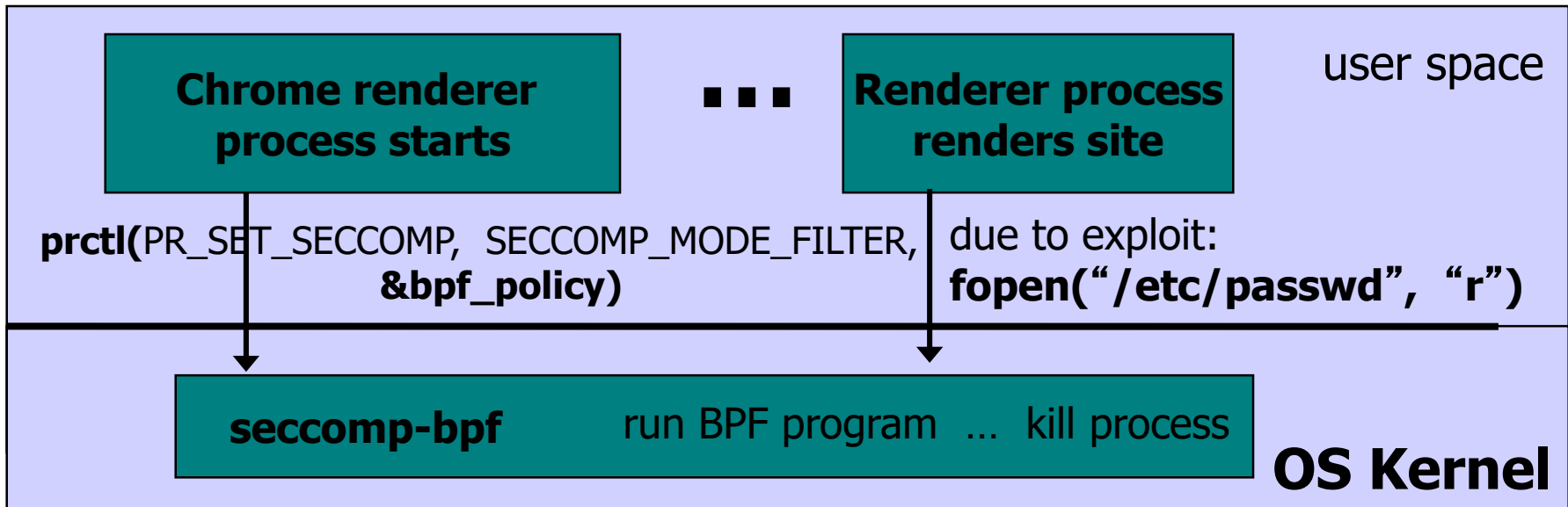
- Argument caches used by both wrapper and kernel
- Message passing instead of argument copying (why does this help?)

Atomicity using system transactions

Integrate security checks into the kernel?

SCI in Linux

Seccomp-BPF: Linux kernel facility used to filter process sys calls
Sys-call filter written in the BPF language (use BPFC compiler)
Used in **Chromium**, in **Docker containers**, ...



BPF Filters (Policy Programs)

Process can install multiple BPF filters:

- once installed, filters cannot be removed, executed on every system call
 - if program forks, child inherits all filters
 - if program calls `execve`, all filters are preserved
-

BPF filter input: syscall number, syscall args, architecture (x86 or ARM)

Filter returns one of:

- `SECCOMP_RET_KILL`: kill process
- `SECCOMP_RET_ERRNO`: return specified error to caller
- `SECCOMP_RET_ALLOW`: allow syscall

Installing a BPF Filter

- Must be called before setting BPF filter.
- Ensures set-UID, set-GID ignored on subsequent `execve()`
⇒ attacker cannot elevate privilege

```
int main (int argc , char **argv ) {  
    prctl(PR_SET_NO_NEW_PRIVS , 1);  
    prctl(PR_SET_SECCOMP,  SECCOMP_MODE_FILTER, &bpff_policy)  
    fopen("file.txt", "w");  
    printf("... will not be printed. \n" );  
}
```

Kill if call `open()` for write

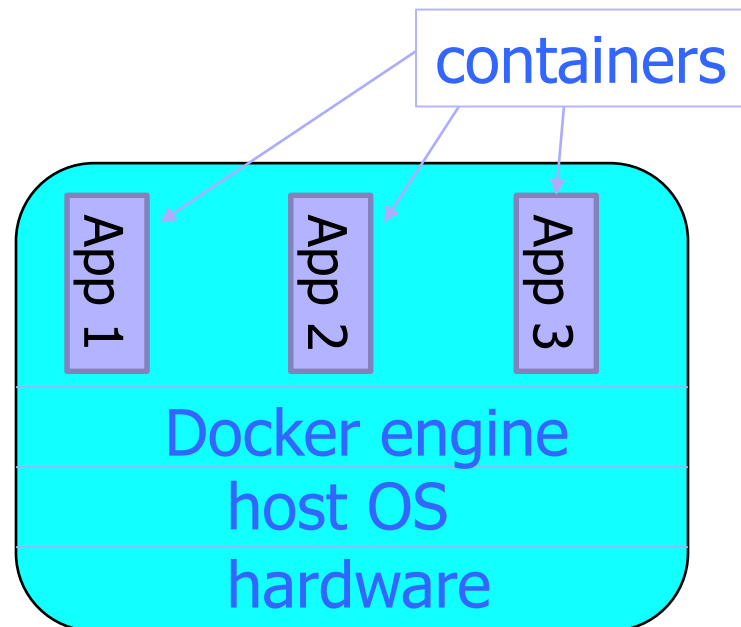
Isolating Docker Containers

Container: process level isolation

Container prevented from making sys calls filtered by secomp-BPF

Whoever starts container can specify BPF policy

- default policy blocks many syscalls, including ptrace



Docker System Call Filtering

Run nginx container with a specific filter called filter.json:

```
$ docker run --security-opt seccomp=filter.json nginx
```

Example filter:

```
"defaultAction": "SCMP_ACT_ERRNO",    // deny by default
"syscalls": [
  { "names": ["accept"],              // sys-call name
    "action": "SCMP_ACT_ALLOW",        // allow (whitelist)
    "args": [ ] },                    // what args to allow
  ...
]
```

Interposition + Static Analysis

1. Analyze the program to determine its expected system call behavior
2. Monitor actual behavior
3. Flag an intrusion if there is a deviation from the expected behavior
 - System call trace of the application is constrained to be consistent with the source or binary code

Callgraph Model

[Wagner and Dean. "Intrusion Detection via Static Analysis". Oakland 2001]

Build a **control-flow graph** of the application by static analysis of its source or binary code

Result: **non-deterministic finite-state automaton (NFA)** over the set of system calls

- Each vertex executes at most one system call
- Edges are system calls or empty transitions
- Implicit transition to special "Wrong" state for all system calls other than the ones in original code; all other states are accepting

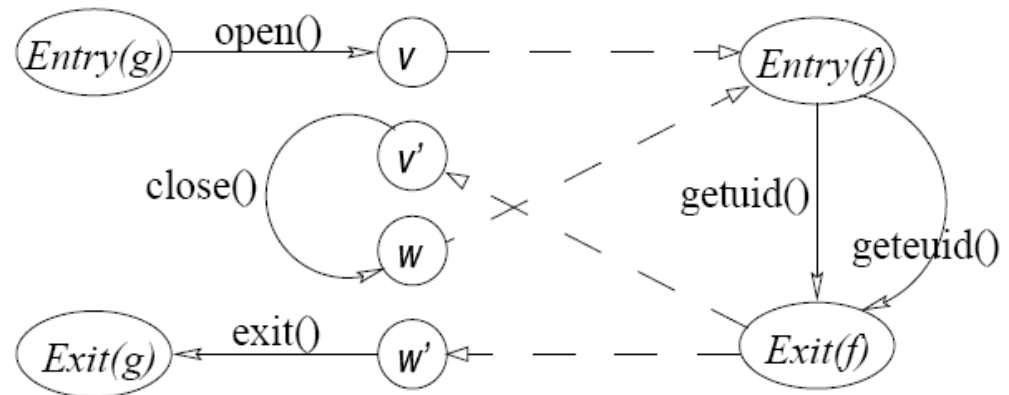
System call automaton is conservative

- **Zero false positives!**

NFA Example

[Wagner and Dean]

```
f(int x) {  
  x ? getuid() : geteuid();  
  x++;  
}  
g() {  
  fd = open("foo", O_RDONLY);  
  f(0); close(fd); f(1);  
  exit(0);  
}
```



Monitoring is $O(|V|)$ per system call

Problem: attacker can exploit impossible paths

- The model has no information about stack state!

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)

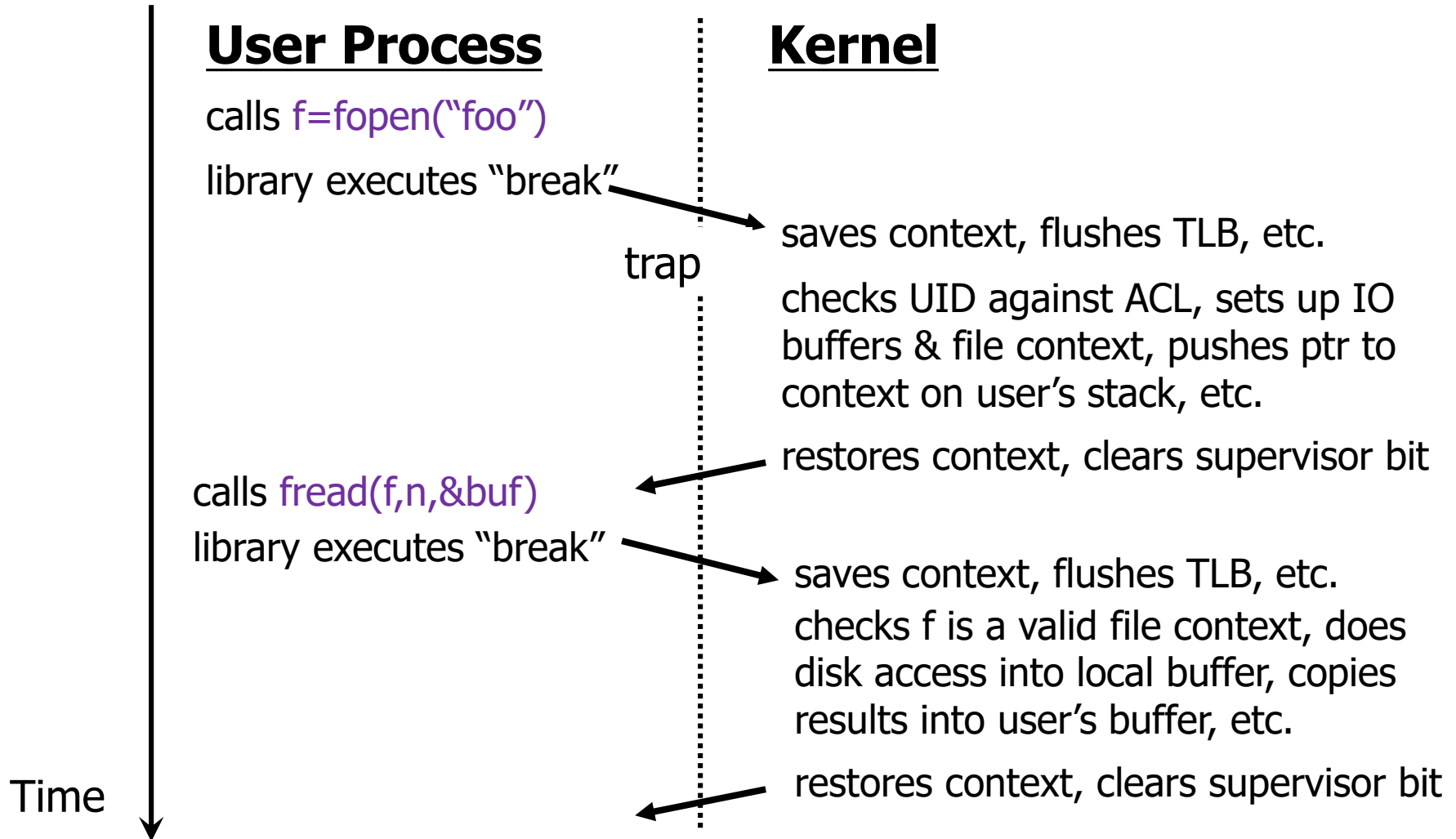
TLB miss raises a page fault exception

- 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]



Modern Hardware Meets Security

Modern hardware: large number of registers, big memory pages

Isolation \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 1993]

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

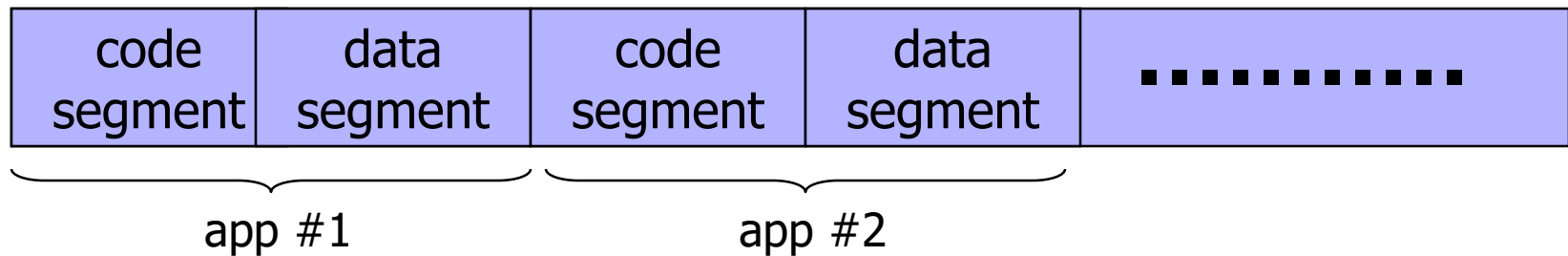
- Identified by a unique pattern of upper bits
- Code is separate from data (heap, stack, etc.)
- Think of a fault domain as a “sandbox”

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?

SFI: Memory Partitioning



Locate unsafe instructions: **jmp, load, store**

- At compile time, add guards before unsafe instructions
- When loading code, ensure all guards are present

Segment Matching

Designed for MIPS processor (many registers available)

dr1, dr2: dedicated registers not used by binary

- Compiler pretends these registers don't exist
- **dr2** contains segment ID

Indirect load instruction **R12**

Guard ensures code does not load data from another segment

```
dr1 ← R34
scratch-reg ← (dr1 >> 20)
compare scratch-reg and dr2
trap if not equal
R12 ← [dr1]
```

: get segment ID
: validate segment ID
: do load

Address Sandboxing

dr2: holds segment ID

Indirect load instruction **$R12 \leftarrow [R34]$** becomes:

$dr1 \leftarrow R34 \ \& \ \text{segment-mask}$

: zero out seg bits

$dr1 \leftarrow dr1 \mid dr2$

: set valid seg ID

$R12 \leftarrow [dr1]$

: do load

Fewer instructions than segment matching

... but does not catch offending instructions

Similar guards placed on all unsafe instructions

Simple SFI Example

Fault domain = from 0x1200 to 0x12FF

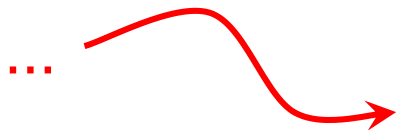
Original code: `write x`

Naïve SFI:

`x := x & 00FF`

`x := x | 1200`

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



`write x`

What if the code jumps right here?

Better SFI:

`tmp := x & 00FF`

`tmp := tmp | 1200`

`write tmp`

Verifying Jumps and Stores

If target address can be determined statically, mask it with the segment's upper bits

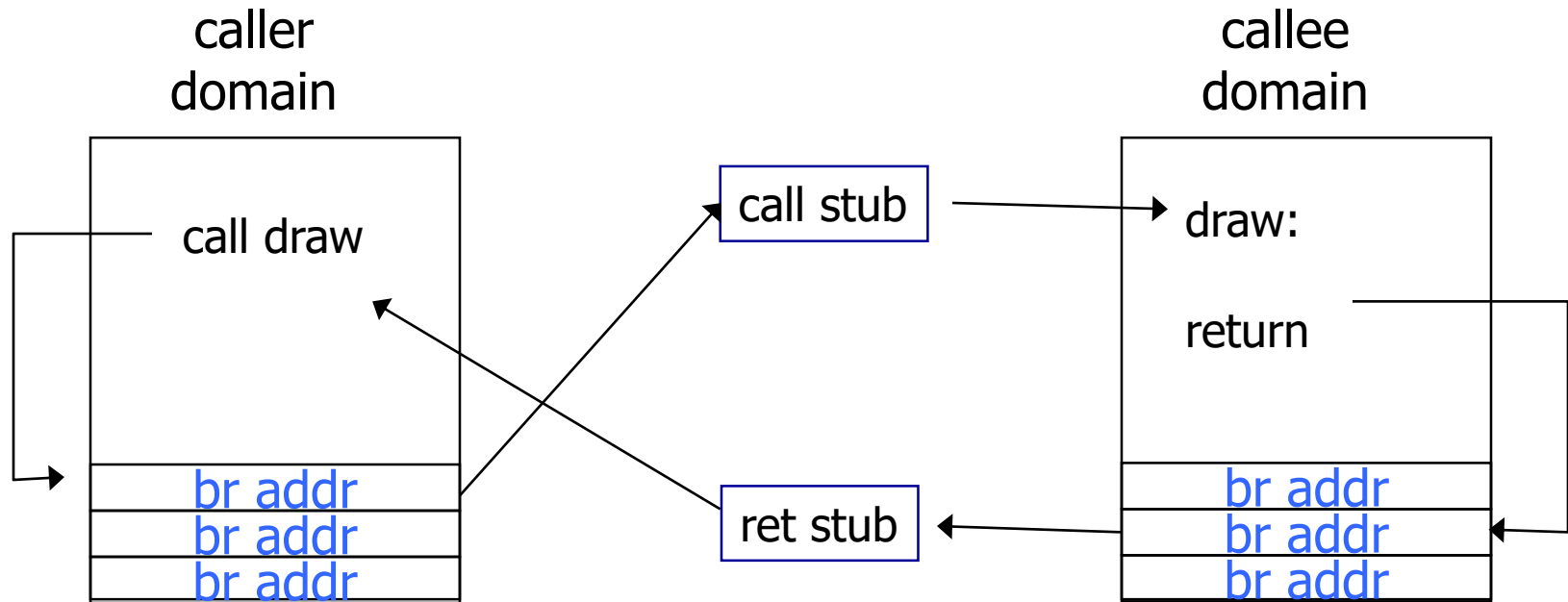
- Crash, but won't stomp on another process's memory

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

Cross-Domain Calls



Only stubs allowed to make cross-domain jumps

Jump table contains allowed exit points

- Addresses are hard coded, read-only segment

SFI Summary

Good performance

- 4% slowdown for mpeg_play

Harder to implement on x86

- Variable length instructions: unclear where to put guards
- Few registers: can't dedicate three to SFI
- Many instructions affect memory: more guards needed

Inline Reference Monitor

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

CFI: Control-Flow Integrity

[Abadi et al. "Control-Flow Integrity". CCS 2005]

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-to-libc)

- Secure even if the attacker has complete control over the thread's address space

CFI: Example of Instrumentation

Original code

Opcode bytes	Source Instructions	Opcode bytes	Destination Instructions
FF E1	jmp ecx ; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst

Instrumented code

B8 77 56 34 12	mov eax, 12345677h	; load ID-1	3E 0F 18 05	prefetchnta	; label
40	inc eax	; add 1 for ID	78 56 34 12	[12345678h]	; ID
39 41 04	cmp [ecx+4], eax	; compare w/dst	8B 44 24 04	mov eax, [esp+4]	; dst
75 13	jne error_label	; if != fail	...		
FF E1	jmp ecx	; jump to label			

Jump to the destination only if the tag is equal to "12345678"

Abuse an x86 assembly instruction to insert "12345678" tag into the binary

WebAssembly



Goal: safely run (almost) **native code in browser**

Portable binary instruction format

- Bytecode for a stack-based virtual machine
- 20x faster than JavaScript

Generated by ahead-of-time or JIT compilation

C++

```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

WebAssembly

```
get_local 0           // n  
i64.const 0           // 0  
i64.eq                // n==0 ?  
if i64  
    i64.const 1       // 1  
else  
    get_local 0       // n  
    get_local 0       // n  
    i64.const 1       // 1  
    i64.sub           // n-1  
    call 0            // f(n-1)  
    i64.mul           // n*f(n-1)  
end
```

WebAssembly Sandbox: CFI

No runtime instrumentation

Modules declare all accessible functions and their types at load time (even w/ dynamic linking)

Direct function calls: only to valid entries in the function table

Indirect function calls: typechecked at runtime

Returns: protected call stack

ROP only possible at function-level granularity

- “Gadgets” impossible, every call target must be a function entry declared at load time

WebAssembly Sandbox: Memory

Local variables with fixed scope in protected call stack, global variables in global index space

- Fixed-size, addressed by index, can't be overflowed

Local variables with unclear static scope in separate stack in special memory region

- References to this region computed with infinite precision to avoid wrapping, simplify bounds checking

Any abnormal operation traps to a JS exception

Native Client

[Yee et al. "Native Client". Oakland 2009]

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

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, 0xffffffff`

`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 \Rightarrow 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