CSE509: (Intro to) Systems Security

Fall 2012

Radu Sion

Reference Monitors
Software Fault Isolation

Reference Monitors

Observes the execution of a program and halts the program if it's going to violate the security policy.

Common Examples:

- operating system (hardware-based)
- interpreters (software-based)
- firewalls

Claim: majority of today's enforcement mechanisms are instances of reference monitors.

Requirements

- Must have (reliable) access to information about what the program is about to do.
  - e.g., what instruction 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.
  - really, stopping isn't necessary, but transition to a "good" state.
- 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.

Types of Policies

Under quite liberal assumptions:

- there's a nice class of policies that reference monitors can enforce (safety properties).
- there are desirable policies that no reference monitor can enforce *precisely*.
  - rejects a program if *and only if* it violates the policy

Assumptions:

- monitor can have access to entire state of computation.
- monitor can have infinite state.
- but monitor can't guess the future – the predicate it uses to determine whether to halt a program must be computable.

Theory vs. Practice

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 arbitrary 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 small 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.

Operating Systems cca. 1975

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.
- Secrecy? Confidentiality? Access control?

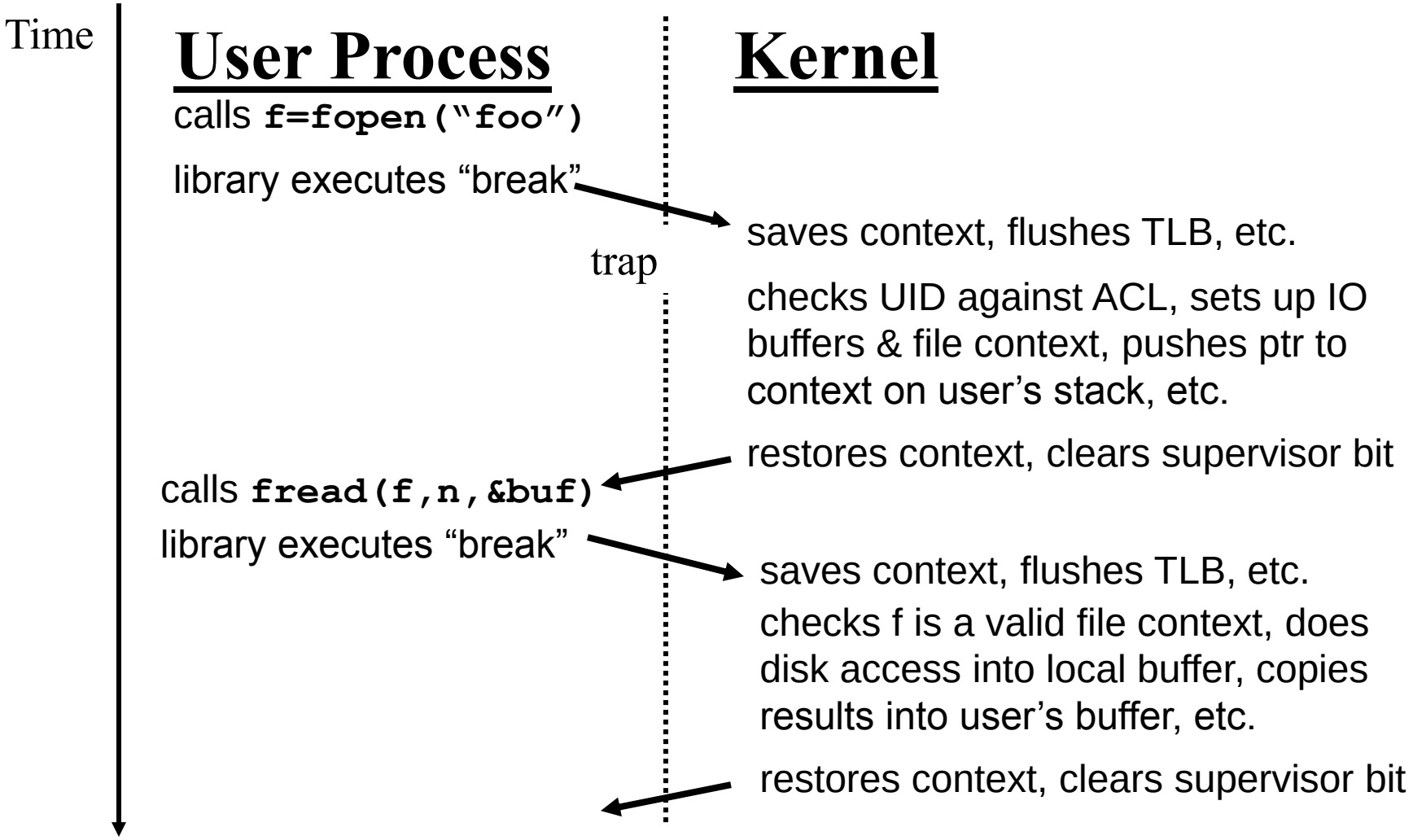
What Can go Wrong?

- 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
- OS or hardware is buggy ... Oops.

Hardware saves the day!

- 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 (see below)
    - 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.

Steps in Typical System Call

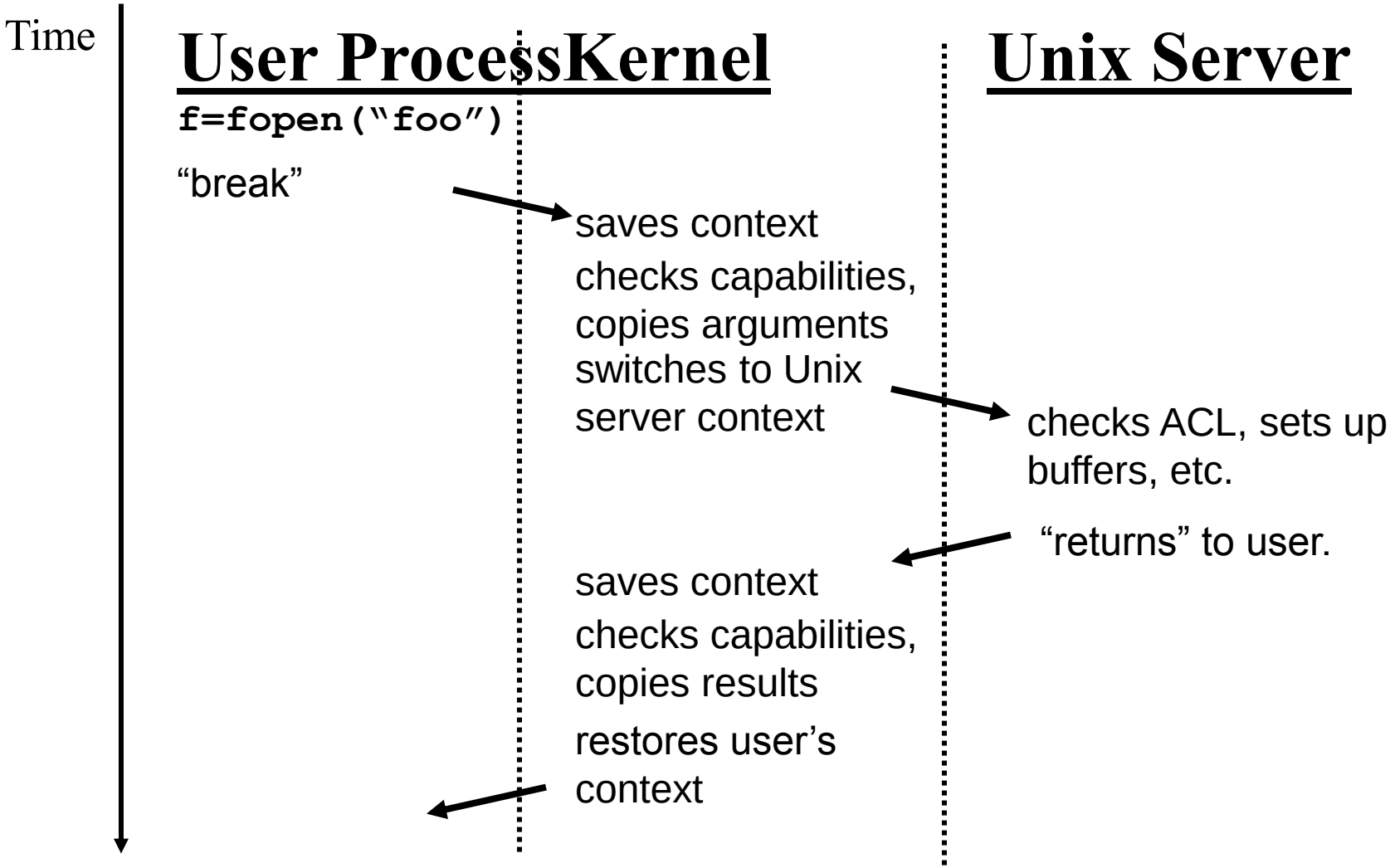


1980s fresh ideas

A big push for *microkernels*:

- Mach, Spring, etc.
- Only put the bare minimum into the kernel.
  - context switching code, TLB management
  - trap and interrupt handling
  - device access
- Run everything else as a process.
  - file system(s)
  - networking protocols
  - page replacement algorithm
- Sub-systems communicate via *remote procedure call* (RPC)
- Reasons: Increase Flexibility, Minimize the TCB

Syscall in Microkernels



Overheads!

Claim was that flexibility and increased assurance would win

- But performance overheads were non-trivial
- Many PhD's on minimizing overheads of communication
- Even highly optimized implementations of RPC cost 2-3 orders of magnitude more than a procedure call.

Result: a backlash against the approach.

- Windows, Linux, Solaris continue the monolithic tradition.
  - and continue to grow for performance reasons (e.g., GUI) and for functionality gains (e.g., specialized file systems.)
- Mac OS X, some embedded or specialized kernels (e.g., Exokernel) are exceptions. VMware achieves multiple personalities but has monolithic personalities sitting on top.

In real life performance matters

The hit of crossing the kernel boundary:

- Original Apache forked a process to run each CGI:
  - could attenuate file access for sub-process
  - protected memory/data of server from rogue script
  - i.e., closer to least privilege
- Too expensive for a small script: fork, exec, copy data to/from the server, etc.
- So current push is to run the scripts in the server.
  - i.e., throw out least privilege !!!

Similar situation with databases, web browsers, file systems, etc.

Thus the Big Question

From a least privilege perspective, many systems should be decomposed into separate processes. But if the overheads of communication (i.e., traps, copying, flushing TLB) are too great, programmers won't do it.

Can we achieve isolation *and* cheap communication?

Fun Idea: Software Fault Isolation

- Wahbe et al. (SOSP'93)
- Keep software components in same hardware-based address space.
- Use a software-based reference monitor to isolate components into logical address spaces.
  - conceptually: check each read, write, & jump to make sure it's within the component's logical address space.
  - hope: communication as cheap as procedure call.
  - worry: overheads of checking will swamp the benefits of communication.
- Note: doesn't deal with other policy issues
  - e.g., availability of CPU

Checked+Interpreted Execution

```
void interpretor(int pc, reg[], mem[], code[], memsz, codesz) {
   while (true) {
     if (pc >= codesz) exit(1);
     int inst = code[pc], rd = RD(inst), rs1 = RS1(inst),
         rs2 = RS2(inst), immed = IMMED(inst);
      switch (opcode(inst)) {
        case ADD: reg[rd] = reg[rs1] + reg[rs2]; break;
        case LD:  int addr = reg[rs1] + immed;
                  if (addr >= memsz) exit(1);
                  reg[rd] = mem[addr];
                  break;
        case JMP: pc = reg[rd]; continue;

       ...
   }
   pc++;
}}
```

Pros&Cons of Interpreter

Pros:

- easy to implement (small TCB.)
- works with binaries (high-level language-independent.)
- easy to enforce other aspects of OS policy

Cons:

- terribly execution overhead (x25? x70?)

but it's a start.

SFI in practice

Used a hand-written specializer or *rewriter*.

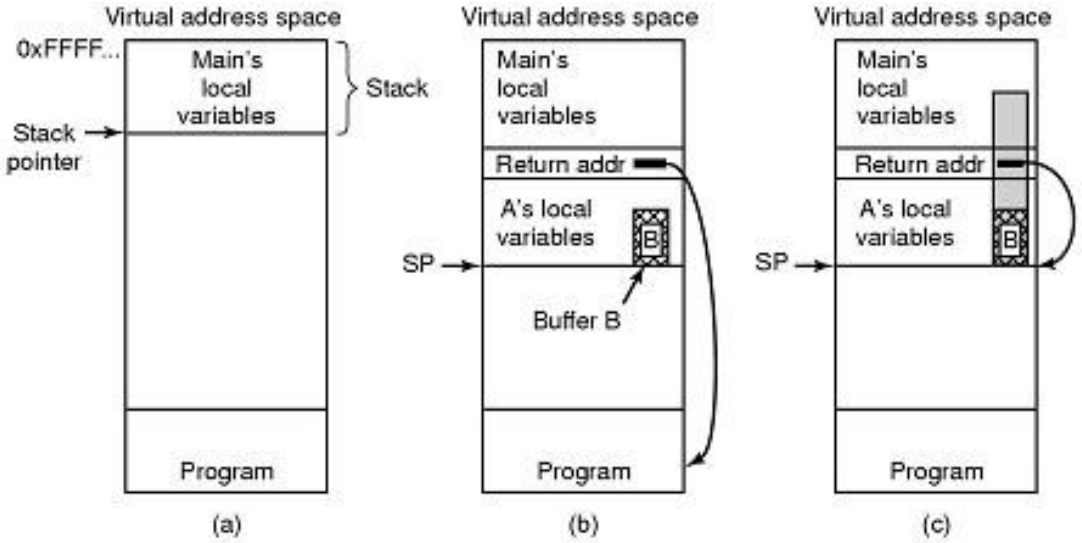
- Code and data for a domain in one contiguous segment.
  - upper bits are all the same and form a *segment id.*
  - separate code space to ensure code is not modified.
- Inserts code to ensure stores [optionally loads] are in the logical address space.
  - force the upper bits in the address to be the segment id
  - no branch penalty – just mask the address
  - may have to re-allocate registers and adjust PC-relative offsets in code.
  - simple analysis used to eliminate unnecessary masks
- Inserts code to ensure jump is to a valid target
  - must be in the code segment for the domain
  - must be the beginning of the translation of a source instruction
  - in practice, limited to instructions with labels.

More on Jumps

- PC-relative jumps are easy:
  - just adjust to the new instruction's offset.
- Computed jumps are not:
  - must ensure code doesn't jump *into* or *around* a check or else that it's *safe* for code to do the jump.
  - E.g., to ensure the latter:
    - a dedicated register is used to hold the address that's going to be written – so all writes are done using this register.
    - only inserted code changes this value, and it's always changed (atomically) with a value that's in the data segment.
    - so at all times, the address is "valid" for writing.
    - works with little overhead for almost all computed jumps.

Buffer Overflow Overview (Stack)

Buffer Overflow



- (a) Situation when main program is running
- (b) After program *A* called
- (c) Buffer overflow shown in gray

Lec 19
Fig 1

Time of Check To Time of Use (TOCTOU)

```
// Victim (installed setuid-root)
void main (int argc, char **argv)
{
  int fd;
  if (access(argv[1], R_OK) != 0)
    exit(1);
  fd = open(argv[1], O_RDONLY);
  // Do something with fd...
}
```

Q: How do we fix it ?

```
// Attacker
void main (int argc, char **argv)
{
  // Assumes directories and links:
  // dir0/lnk -> public_file
  // dir1/lnk -> /etc/shadow
  // activedir -> dir0

  // Let the victim run
  if (fork() == 0) {
    system("victim activedir/lnk");
    exit(0);
  }
  usleep(1); // yield CPU

  // Switch where target points
  unlink("activedir");
  symlink("dir1", "activedir");
}
```

SQL/Code Injection

A query string embedded somewhere in your application:

SELECT * FROM users WHERE name = '" + userName + "';

What if userName is coming from the attacker (e.g., web form):

a' or 't'='t

Then we have:

SELECT * FROM users WHERE name = 'a' OR 't'='t';

Or what if userName becomes:

a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't

We then get:

SELECT * FROM users WHERE name = 'a';
DROP TABLE users;
SELECT * FROM userinfo WHERE 't' = 't';