CSC 256/456: Operating Systems

Attacks on Operating System Kernels

John Criswell
University of Rochester

aka

The “we’re doing it wrong” Lecture

Operating Systems as the Security Foundation

- Process abstraction provides isolation
- I/O abstractions control access to devices
- Access controls control sharing

Ideal Application Design

- Small programs with least privilege
  - Smaller programs less likely to be exploitable
  - Least privilege limits damage done by an attack

Monolithic Operating System is BAD Design

- OS kernel is *very* large
- OS kernel is *very* privileged
- OS kernel is written in C

If the operating system kernel is exploited, all security guarantees are *null* and *void*.

Methods of Exploitation

- Kernel bugs
  - Missing access control checks
  - Memory safety errors
- Exploitation of privileged programs
  - Load malicious code into kernel
  - Access kernel internal memory
  - Modify OS kernel image on disk

Kernel Bugs

Missing Access Control Checks

- OS kernel fails to check process/thread credentials
  - Permits processes to perform privileged operations
  - Potentially modify OS kernel behavior
    - Has occurred in the "real world"
- Static analysis solutions exist
  - Not used in practice

Memory Safety

- Generally speaking, accessing the "wrong" memory
  - Similar to a "segmentation fault" in applications
- Corrupt important kernel data structures
- Modify kernel code
- Change kernel control flow
  - Trick kernel into performing arbitrary computation

Three Kinds of Safety Violations

Three Kinds of Safety Violations

- Pointer-based errors
  - Array-bounds violations
  - Dangling pointers
  - Bad casts

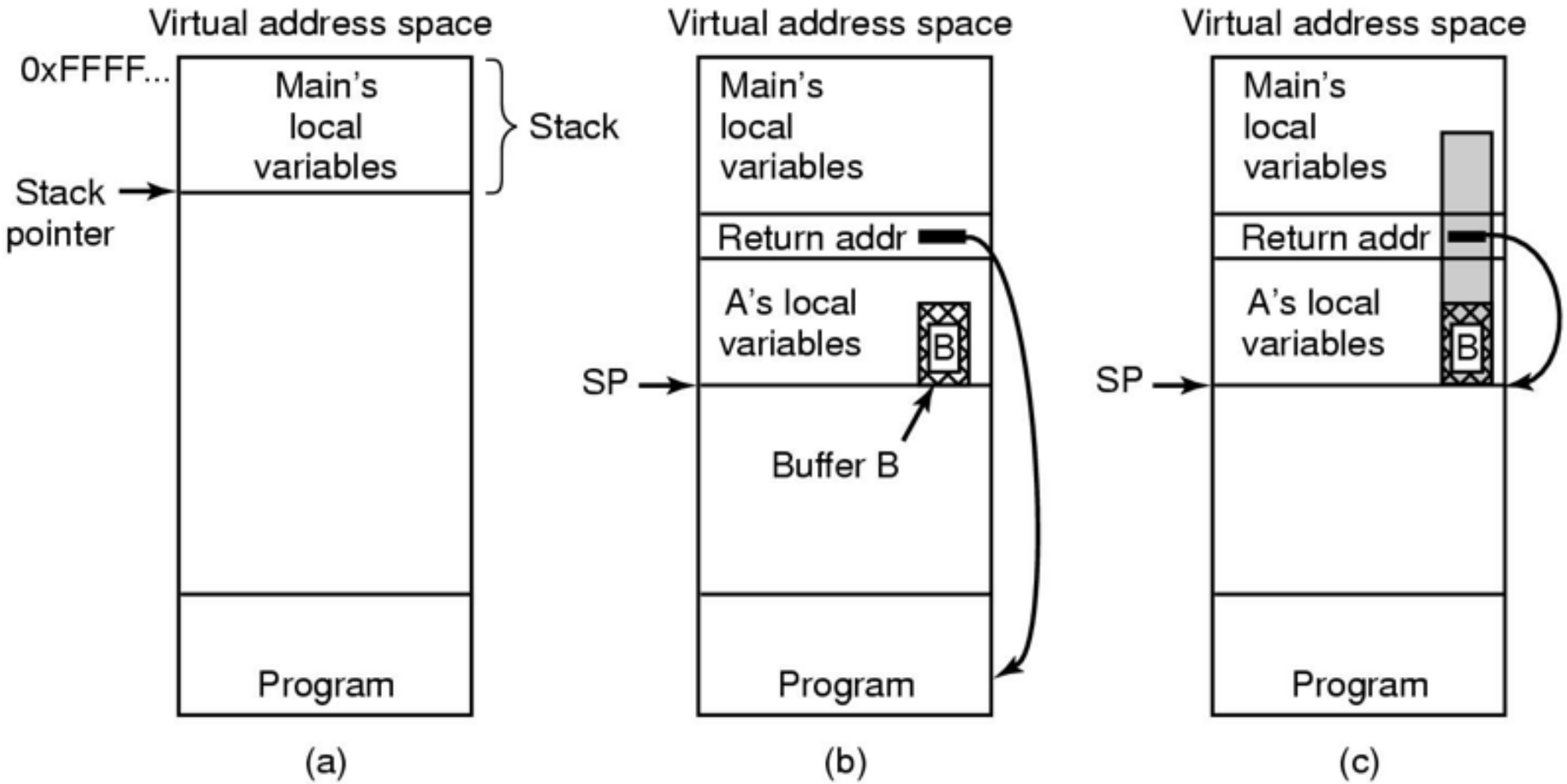
Three Kinds of Safety Violations

- Pointer-based errors
  - Array-bounds violations
  - Dangling pointers
  - Bad casts
- State manipulation errors
  - Signal handler dispatch on interrupted kernel state

Three Kinds of Safety Violations

- Pointer-based errors
  - Array-bounds violations
  - Dangling pointers
  - Bad casts
- State manipulation errors
  - Signal handler dispatch on interrupted kernel state
- Hardware configuration errors
  - Mapping code memory into the kernel heap

Buffer Overflow



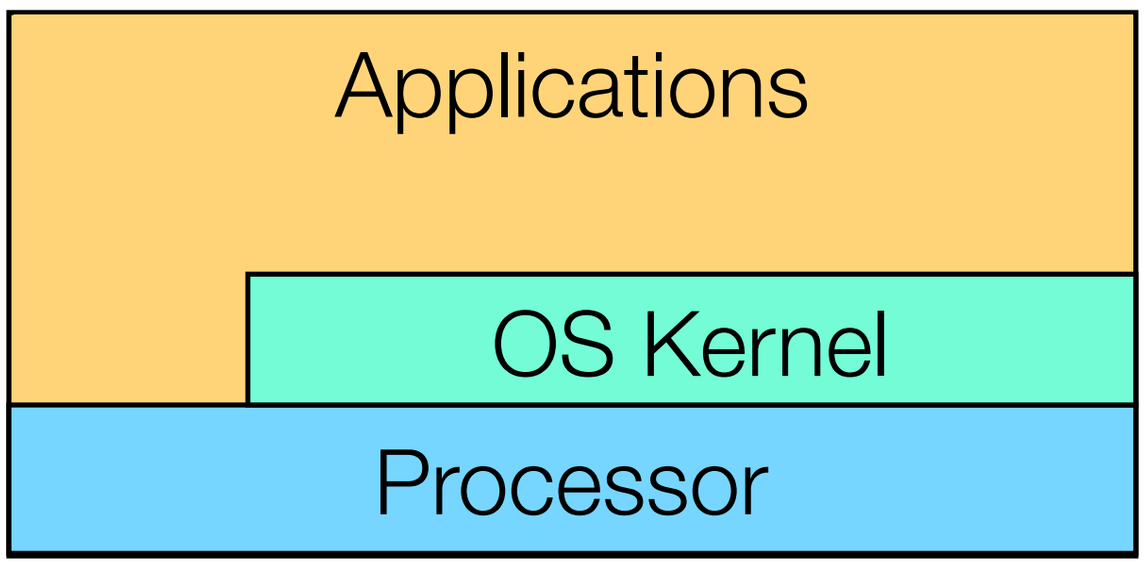
Two General Solutions

- Write OS code in type-safe programming language
- Instrument C code to detect errors at run-time

Writing OS in Type-Safe Language

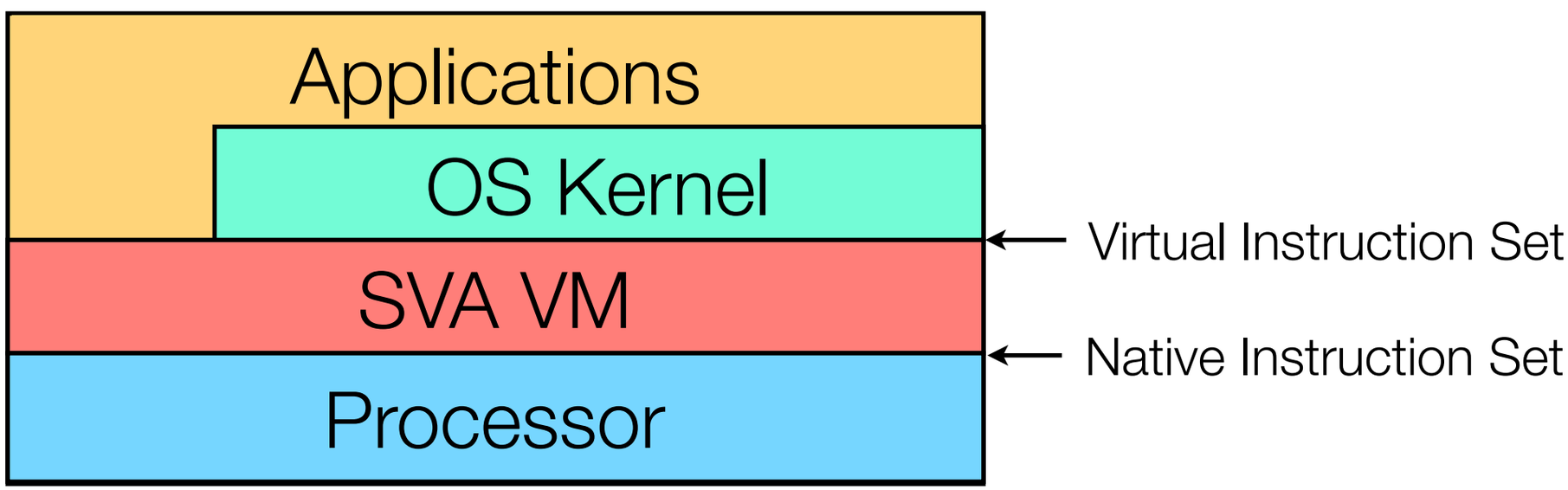
- Can be done (e.g., Singularity, Verve, SPIN, KaffeOS)
- Need to rewrite entire OS
- Need to deal with garbage collection
  - Can't "stop the world" during an interrupt
  - Ensure that garbage collection doesn't trap
  - May consume more resources
- Some code still written in C and assembler

Secure Virtual Architecture (SVA)



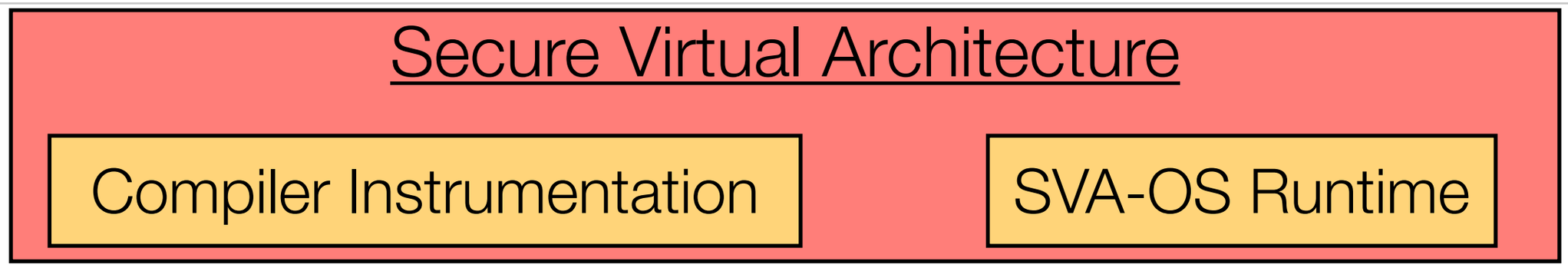
- Compile OS to virtual instruction set
  - Designed to be easy to analyze and instrument
  - Low-level instructions (SVA-OS) replace inline assembly
- Translate ahead-of-time, boot-time, or run-time

Secure Virtual Architecture (SVA)



- Compile OS to virtual instruction set
  - Designed to be easy to analyze and instrument
  - Low-level instructions (SVA-OS) replace inline assembly
- Translate ahead-of-time, boot-time, or run-time

SVA Virtual Instruction Set



- SVA-Core: Compiler Instrumentation
  - Based on LLVM IR: Typed, Explicit SSA form
  - Sophisticated compiler analysis and instrumentation
- SVA-OS: Runtime Library
  - OS-neutral instructions to support a commodity OS
  - Encapsulates & controls hardware and state manipulation
  - Keeps policy decisions within the OS

Memory Safety with SVA-M

Memory Safety with SVA-M

- Safe pointer arithmetic: no array bounds violations

Memory Safety with SVA-M

- Safe pointer arithmetic: no array bounds violations
- Loads & stores access correct *subset* of objects

Memory Safety with SVA-M

- Safe pointer arithmetic: no array bounds violations
- Loads & stores access correct *subset* of objects
- Control-flow integrity

Memory Safety with SVA-M

- Safe pointer arithmetic: no array bounds violations
- Loads & stores access correct *subset* of objects
- Control-flow integrity
- Type-safety for a subset of memory objects

Memory Safety with SVA-M

- Safe pointer arithmetic: no array bounds violations
- Loads & stores access correct *subset* of objects
- Control-flow integrity
- Type-safety for a subset of memory objects
- Dangling pointers made safe; no garbage collection

Memory Safety Compiler Instrumentation

- Add code to track the location of memory objects
  - Record object locations in “per-pool” data structures
  - Take advantage of type-safe memory pools
- Add checks on loads and stores
- Add checks on pointer arithmetic operations

Memory Safety for SVA-OS

- Restrict MMU configuration
  - Prevent double mappings that violate type safety
  - Prevent making code segment writeable
  - Prevent applications from accessing kernel memory
- Restrict state manipulation
  - Generally disallow changes to interrupted kernel state
  - Provide unwinding of interrupted kernel state

Exploitation of Privileged Programs

OS Kernel Configuration/Extension

- Administrator can
  - Load new code into the kernel
  - Write new code into the files containing kernel code
  - Bypass file system by writing to device files
  - Modify kernel memory through /dev/kmem device

Result:

Privileged programs are on par with the kernel!

Kernel-Level Malware

- Malicious code loaded into kernel
- Typically hides processes, files, network connections
  - Has access to all kernel data structures
- Also capable of
  - Snooping on applications
  - Modifying application memory
  - Altering application behavior

Kernel-Level Malware Hooking

- Modify system call table
  - Intercept all desired system calls
- Modify interrupt handler tables
  - Get executed periodically
- Modify function pointers to virtual file system layer
  - Intercept all file system calls

Kernel Level Malware Challenges

- Difficult to classify a kernel driver as “malicious”
- Difficult to control code running in lowest privilege layer
- Kernel-level malware has many, many options

Disclaimer

Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).