CS 161 Fall 2005 — Computer Security — Joseph/Tygar/Vazirani/Wagner — Notes 29

1 Operating System Security

Goals:

- Protecting different applications running on the same machine at the same time from each other
  - Keep malicious/buggy user programs from crashing OS
  - Keep malicious/buggy user programs from crashing each other
- Control over what applications run on a platform
  - Need a secure environment from HW to OS levels

Today's topics:

- Hardware support for protection
- Creating secure systems

2 Hardware support for protection

Hardware provides two things to help *isolate a program's effects to within just that program*:

- Address translation
  - Non-executable regions
- Dual mode operation

2.1 Address translation

What is an Address Space?

- Literally, all the memory addresses a program can touch.
- All the state that a program can affect or be affected by.

Achieve protection by restricting what a program can touch!

Hardware translates every memory reference from virtual addresses to physical addresses; software sets up and manages the mapping in the translation box (see Figure 1).

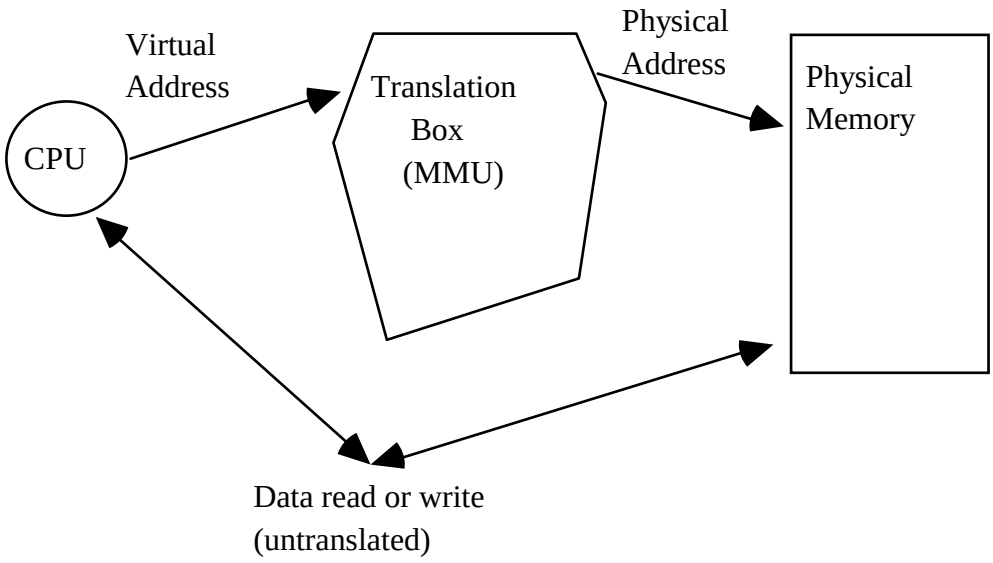

Figure 1: Address Translation in Modern Architectures.

Two views of memory:

- View from the CPU — what program sees, virtual memory
- View from memory — physical memory

Translation box (also called a *memory management unit*) converts between the two views.

Translation helps implement protection because there is no way for a program to even talk about other program's addresses; no way for it to touch operating system code or data (see Figure 2).

Translation also helps with the issue of how to stuff multiple programs into memory.

Translation is implemented using some form of table lookup. Separate table for each user address space.

2.2 Dual mode operation

Can an application modify its own translation tables? If it could, then it could get access to all of physical memory. Has to be restricted somehow.

Dual-mode operation

- When in the OS, can do anything (called "kernel mode", "supervisor mode", or "protected mode")
- When in a user program, restricted to only touching that program's memory (user-mode)

Implemented by setting a hardware-provided bit. Restricted operations can only be performed when the "kernel-mode" bit is set. Only the operating system itself can set and clear this bit.

HW requires CPU to be in **kernel-mode** to modify address translation tables.

Isolate each address space so its behavior can't do any harm, except to itself.

Several issues:

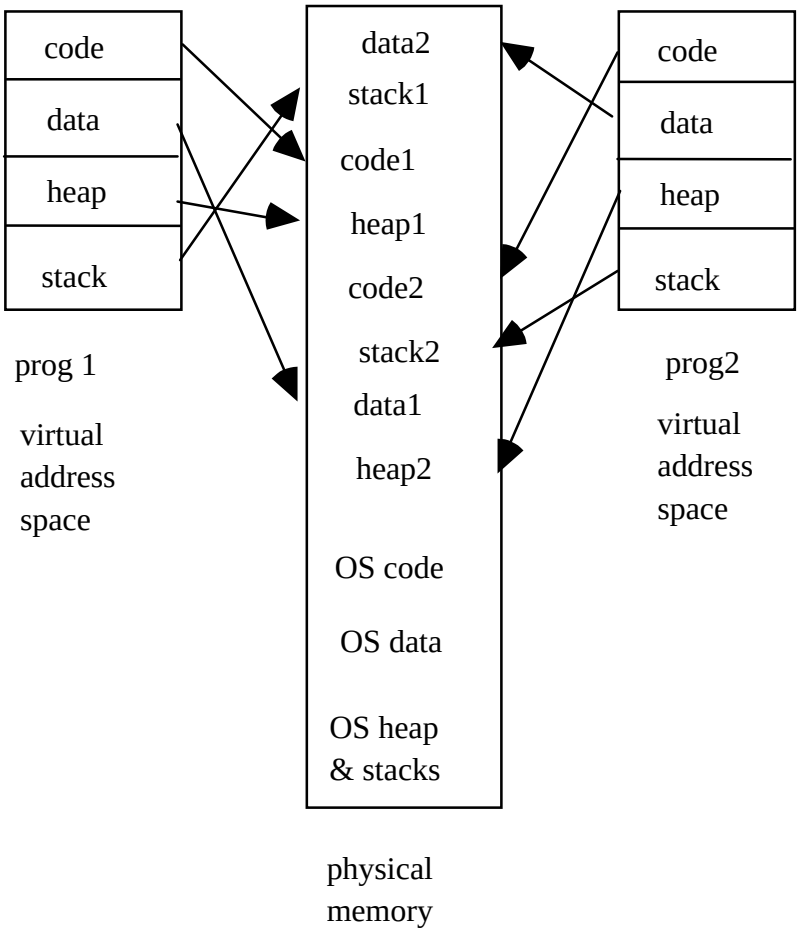

Figure 2: Example of Address Translation.

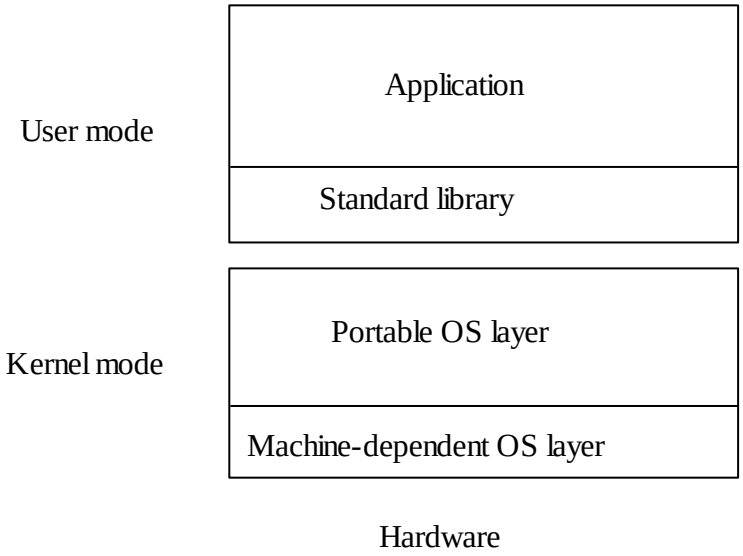

Figure 3: Typical UNIX Operating System Structure.

1. Do we need the kernel–application boundary if system is dedicated to a single application?
2. How to share CPU between kernel and user programs?
3. How do programs interact?
4. How does one switch between kernel and user modes when the CPU gets shared between the OS and a user program?
    - OS → user (kernel → user mode)
    - User → OS (user mode → kernel mode)

2.2.1 Kernel → User

To run a user program, create a thread to:

- Allocate and initialize address space control block
- Read program off disk and store in memory
- Allocate and initialize translation table (point to program memory)
- Run program (or to return to user level after calling the OS with a system call):
    - Set machine registers
    - Set hardware pointer to translation table
    - Set processor status word (from kernel mode to user mode)
    - Jump to start of program

2.2.2 User → Kernel

How does the user program get back into the kernel, or switch to another user program?

Voluntarily user→kernel System call — special instruction to jump to a specific operating system handler. Just like doing a procedure call into the operating system kernel — program asks OS kernel, please do something on procedure's behalf.

Can the user program call any routine in the OS?

No. Just specific ones the OS says are OK. Always start running handler at same place, otherwise, problems!

How does OS know that system call arguments are as expected?

It can't — OS kernel has to check all arguments (registers and memory) — otherwise, bug in user program can crash kernel.

TOCTOU vulnerabilities are common: Time of Check, Time of Use

```
open(filename, mode)
   check permissions for filename and mode
   create filehandle for filename
   return filehandle
```

Race condition between check and create steps, and malicious user.

Involuntarily user→kernel Hardware interrupt, also program exception

Examples of program exceptions:

- Bus error (bad address *e.g.*, unaligned access)
- Segmentation fault (out of range address)
- Page fault (important for providing illusion of infinite memory)

On system call, interrupt, or exception: hardware atomically

- Sets processor status to kernel mode
- Changes execution stack to an OS kernel stack
- Saves current program counter
- Jumps to handler routine in OS kernel
- Handler saves previous state of any registers it uses

Context switching between programs Same as with threads, except now also save and restore pointer to translation table. To resume a program, re-load registers, change hardware pointer to translation table, and jump to old PC.

2.3 Communication between address spaces

How do two address spaces communicate? Can't do it directly if address spaces don't share memory.

Instead, all inter-address space (in UNIX, inter-process) communication has to go through kernel, via system calls.

Models of inter-address space communication:

- Byte stream producer/consumer. For example, communicate through pipes connecting stdin/stdout.
- Message passing (send/receive). We can use this to build remote procedure call (RPC) abstraction, so that you can have one program call a procedure in another.
- File system (read and write files). File system is shared state! (Even though it exists outside of any address space.)
- "Shared Memory" — Alternately, on most UNIXes, can ask kernel to set up address spaces to share a region of memory, but that violates the whole notion of why we have address spaces — to protect each program from bugs in the other programs.

In any of these, once you allow communication, malicious actions or bugs from one program can propagate to those it communicates with, unless each program verifies that its input is as expected. Also, have to check for TOCTOU vulnerabilities.

2.4 Other hardware support

Problem: How to use hardware support to prevent buffer overflows?

Each Page Table Entry (or segment table entry) contains flag bits:

- Read-Only
- Dirty?
- LRU bits (time since last access)
- etc.

Can add another flag to mark individual memory areas as non-executable.

- No Execute (NX) support (AMD's Opteron and Athlon 64 CPUs).
- Execute Disable (XD) support (Intel x86)
- Alpha, SPARC, PowerPC, Itanium IA-64, and Transmeta.

Requires OS support (Linux and Sun's Sparc/Solaris, recently added to Windows XP in Service Pack 2) to mark stack and heap areas as non-executable. Any attempts to execute code from pages marked as non-executable results a program exception.

Possible to implement with software emulation, but incurs overhead.

Does this prevent buffer overflow exploits?

No. It only prevents buffer overflow exploits that try to execute code they send. Won't prevent overwriting of return program counter, which could be used to execute an existing procedure (*e.g.*, in a shared library or system call). A well crafted attack could consist of a payload with the return address for *execve* and some malicious parameters. The result is execution of the parameters with the privileges of the exploited program...

3 Secure Systems

Steps to building a secure system (only runs "known/certified" applications):

1. Start with secure hardware platform.
    - Manufacturer embeds their public key in machine's ROM.
    - Machine cryptographically hashes BIOS (in Flash RAM) and verifies hash (and thus, BIOS) integrity using public key.
    - Additional protection: BIOS is encrypted using manufacturer's private key.
2. Add a secure boot process.
    - Verified BIOS uses public key to verify cryptographic hash of root file system (or individual files).
    - Verified BIOS loads secure OS.

3. Run secure OS. Examines hashes for applications and loads secure applications.

Examples: Microsoft Xbox gaming console, TiVo Personal Video Recorder Series 2.

Very difficult to design truely secure systems — both have been "cracked" by hackers.

Attacks:

- Hardware modifications
  - Add "mod chip" to Xbox. Cat and mouse game to defeat...
- Cryptoanalysis attacks
  - Brute force attack to determine manufacturer's private key (not successful).
  - Flaw in some versions of Xbox hash algorithm enabled substitution of BIOS code.
- "Secure" program flaws
  - Exploit buffer overflow vulnerability in OS or game code (successful Xbox attack).
  - Exploit flaw in OS boot time parameters (enabled TiVo hackers to instruct secure kernel to load their kernel).