

CSc 466/566

Computer Security

9 : Operating Systems — Introduction

Version: 2012/03/06 12:17:36

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2012 Christian Collberg

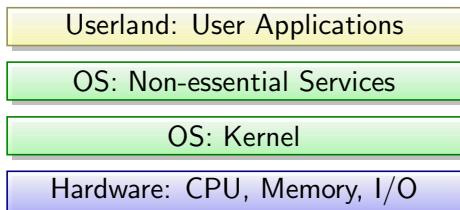
Christian Collberg

Outline

- 1 Introduction
 - Processes
 - Inter-Process Communication
 - Memory management
 - Virtual machines
- 2 Process Security
- 3 Authenticated boot
 - Trusted boot
 - Taking measurements
 - The TPM
 - The challenge
 - Privacy issues
 - Applications and Controversies
- 4 Pioneer
- 5 XOM
 - The XOM architecture
 - Preventing replay attacks
 - Fixing a leaky address bus
 - Fixing a leaky data bus
 - Discussion
- 6 Filesystem Security
 - SetUID
 - File Descriptors
 - Symbolic Links
- 7 Summary

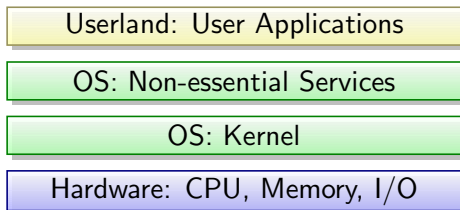
- The OS manages **hardware devices** (CPU, memory, network interfaces, output devices).
- The OS manages **multiple users** with different access rights.
- The OS manages multiple **concurrent processes** (multitasking) with different access rights.
- Users and processes should not be allowed to damage shared resources.

OS Layers



- **Kernel**: manages low-level resources (memory, processors, I/O devices).
- **Non-essential OS Services**: printing, ...
- **I/O**: USB, network interfaces, ..., are managed by **device drivers**.

System Calls



- User processes ask for services from the OS by issuing **system calls** (**syscalls**):
 - 1 The user application signals the processor by issuing a **software interrupt**;
 - 2 the CPU switches control to an **interrupt handler**;
 - 3 we enter **kernel mode** (the process **traps**);
 - 4 the system call executes.

- **Processes** are created by and managed by the kernel.
- **Time slicing**: The kernel gives each process a fair amount of time.
- A **parent process** creates a **child process** by **forking**:
 - The child has the same **privileges** as the parent.
 - The child inherits the parent's file descriptors.
- The **init** process is the root of the **process tree**.

```
> pstree
```

```
--+ 00001 root /sbin/launchd
|--- 00012 root /usr/libexec/kextd
|--- 00013 root /usr/libexec/UserEventAgent -l System
|--+ 00054 root /usr/sbin/cupsd -l
| |--- 00202 _lp HP_LaserJet_1012 9 collberg xfig-batch.nTr527 1 finishings=3
| | \--- 00203 _lp usb://Hewlett-Packard/hp%20LaserJet%201012?serial=00CNFB31376
|--+ 00055 root /usr/sbin/httpd -D FOREGROUND -D WEBSHARING_ON
| |--- 00199 _www /usr/sbin/httpd -D FOREGROUND -D WEBSHARING_ON
| |--- 52411 _www /usr/sbin/httpd -D FOREGROUND -D WEBSHARING_ON
| | \--- 54604 _www /usr/sbin/httpd -D FOREGROUND -D WEBSHARING_ON
|--- 00056 root /usr/sbin/cron
|--+ 00223 collberg /sbin/launchd
| |--- 00233 collberg /usr/sbin/distnoted agent
| |--- 00240 collberg /usr/libexec/UserEventAgent -l Aqua
| |--+ 00249 collberg /Applications/iTerm.app/Contents/MacOS/iTerm -psn_0_3277
| | | |--+ 83311 root login -fp collberg
| | | | \--+ 83312 collberg -tcsh
| | | | \--- 06649 collberg /usr/local/bin/email -tls -smtp-auth login -smtp-s
| | |--+ 79257 root login -fp collberg
| | | \--+ 79258 collberg -tcsh
| | | |--- 07961 collberg /Applications/Emacs.app/Contents/MacOS/Emacs slide
| | | \--+ 08109 collberg pstree
| | | \--- 08110 root ps -axwwo user,pid,ppid,pgid,command
```

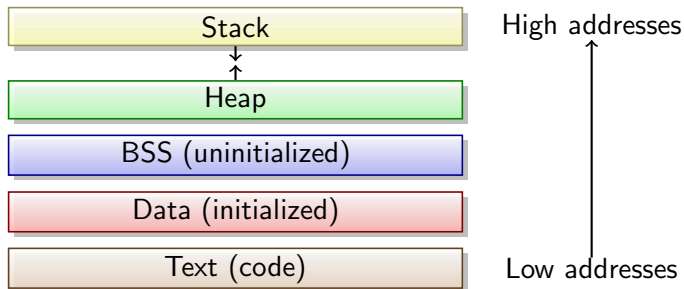
Process Privileges

- Each process has a **uid** (user ID) and **gid** (group ID) that identifies the user/group for the process.
- **Effective User ID** (euid) — used when deciding a process' access privileges.

Inter-Process Communication

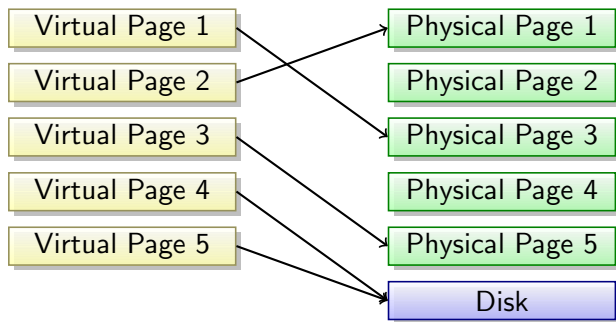
- **Sockets** and **pipes**
- **Signals** — asynchronous communication between processes. When a process receives a signal, the process is interrupted and a **handler** is invoked.
- **Remote Procedure Call** (RPC) — One process invokes a procedure in another process.

Memory management



- Each executing process' **address space** is separated into five regions.
- Each region has access restrictions (read, execute, write).
- The OS enforces address space boundaries between processes.

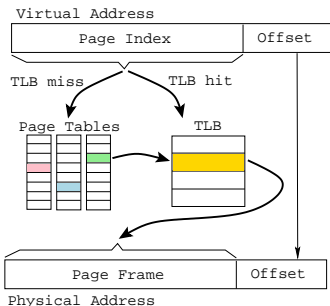
Virtual memory



- Each process sees a **virtual address space**. Virtual pages are mapped onto physical pages.
- From the process' point of view, memory is large and contiguous.
- Not currently needed pages are **paged out** to disk.

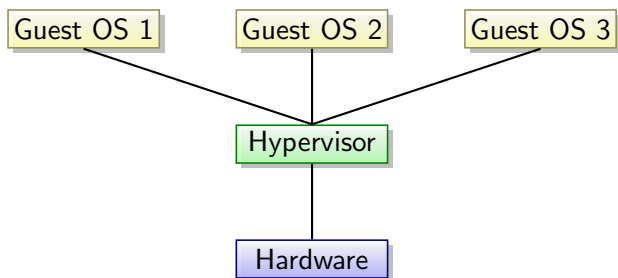
Virtual memory...

- Typical memory management system:



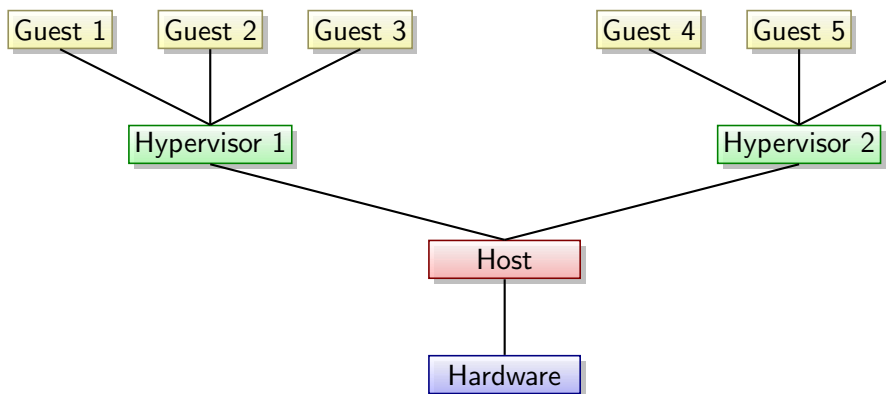
- On a memory access, the **Memory Management Unit** (MMU) looks up real address from the virtual address.
- On a **TLB miss** walk the page tables (slow), and update the TLB with the new virtual-to-physical address mapping.

Virtual machines



- A **Virtual Machine** (VM) provides a simulated environment.
- **Hypervisor** (**Virtual Machine Monitor** (VMM)) — software layer that provides the environment.
- **Native virtualization** — The VMM runs on the bare hardware.
- **Guest OS** — the OS running inside the VM.

Virtual machines...

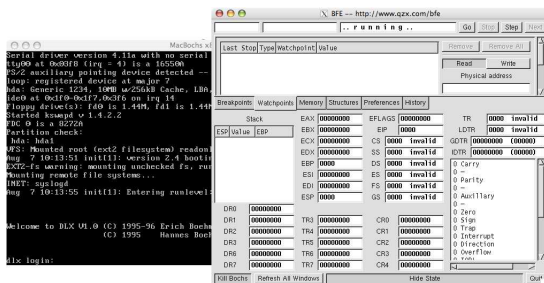


- **Hosted virtualization** — The VMM runs inside the **Host OS**.

Implementing virtual machines — Advantages

- **Hardware efficiency** — multiple OSs on the same machine.
- **Portability** — snapshot the state of the OS and move to other hardware.
- **Security** — the guest OS runs in a **sandbox**, on a breach it is easy to shut it down without affecting any other services.

Implementing virtual machines — Emulation



- **Emulation** — instructions are translated on-the-fly.
- Linux operating system (to the left) booting inside the Bochs x86 emulator (right) while running on a PowerPC Macintosh computer.
- Performance issues.


```

MacBochs x86
Serial driver version 4.11a with no serial
tty00 at 0x03f8 (irq = 4) is a 16550A
PS/2 auxiliary pointing device detected --
loop: registered device at major 7
hda: Generic 1234, 10MB w/256kB Cache, LBA,
ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
Floppy drive(s): fd0 is 1.44M, fd1 is 1.44M
Started kswapd v 1.4.2.2
FDC 0 is a 8272A
Partition check:
 hda: hda1
VFS: Mounted root (ext2 filesystem) readonly.
Aug 7 10:13:51 init[1]: version 2.4 bootin
EXT2-fs warning: mounting unchecked fs, run
Mounting remote file systems...
INET: syslogd
Aug 7 10:13:55 init[1]: Entering runlevel:
Welcome to DLX V1.0 (C) 1995-96 Erich Boehm
(C) 1995 Hannes Boehm
dlx login:

```

BFE -- http://www.qzx.com/bfe

.. running ..

Last Stop	Type	Watchpoint	Value

Breakpoints Watchpoints Memory Structures Preferences History

Stack			EAX	EFLAGS
ESP	Value	EBP	00000000	00000000
			EBX	EIP
			00000000	0000
			ECX	CS
			00000000	0000 invalid
			EDX	SS
			00000000	0000 invalid
			EBP	DS
			0000	0000 invalid
			ESI	ES
			00000000	0000 invalid
			EDI	FS
			00000000	0000 invalid
			ESP	GS
			0000	0000 invalid

DR0	00000000	TR3	00000000	CR0	00000000
DR1	00000000	TR4	00000000	CR1	00000000
DR2	00000000	TR5	00000000	CR2	00000000
DR3	00000000	TR6	00000000	CR3	00000000
DR6	00000000	TR7	00000000	CR4	00000000
DR7	00000000				

Kill Bochs Refresh All Windows Hide State

Implementing virtual machines — Virtualization

- **Virtualization** — Host and guest OSs run on the same hardware.
- Virtual and physical interfaces are matched.
- The hypervisor can insert actions on system calls, for example.

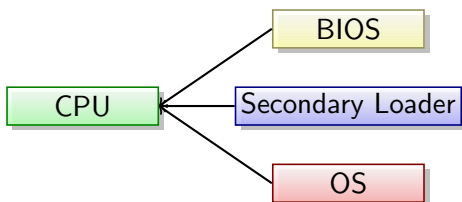
Outline

- 1 Introduction
 - Processes
 - Inter-Process Communication
 - Memory management
 - Virtual machines
- 2 **Process Security**
- 3 Authenticated boot
 - Trusted boot
 - Taking measurements
 - The TPM
 - The challenge
 - Privacy issues
 - Applications and Controversies
- 4 Pioneer
- 5 XOM
 - The XOM architecture
 - Preventing replay attacks
 - Fixing a leaky address bus
 - Fixing a leaky data bus
 - Discussion
- 6 Filesystem Security
 - SetUID
 - File Descriptors
 - Symbolic Links
- 7 Summary

Process Security

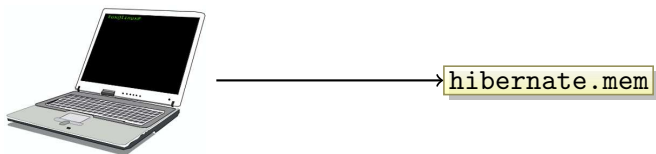
- We must monitor the processes running on a computer.
- Our trust depends on all the processes that are running:
 - 1 the first one when the computer starts up,
 - 2 the processes it starts,
 - 3 the processes *they* start,
 - 4 ...

The Boot Sequence



- On system startup:
 - 1 the BIOS firmware is executed;
 - 2 the BIOS loads the second-stage boot loader;
 - 3 the second-stage boot loader loads the rest of the OS;
 - 4 control is passed to the OS.
- A malicious user could take control at any point.
- A BIOS password stops the second-stage boot loader from executing.

Hibernation



- When **hibernating**, the OS stores the memory image to file.
- Attack:
 - 1 Restart the computer with a LiveCD.
 - 2 Extract passwords etc. from hibernation file.
- Defense: hard disk encryption.

Event Logging and Process Monitoring

- Monitor log files in `/var/log` for unusual events.
- Monitor processes (MacOSX: Activity Monitor, Windows: Process Explorer).
- Detect attacks:
 - 1 Locate malware with the same name as a real application, but in the wrong location.

Outline

- 1 Introduction
 - Processes
 - Inter-Process Communication
 - Memory management
 - Virtual machines
- 2 Process Security
- 3 Authenticated boot**
 - Trusted boot
 - Taking measurements
 - The TPM
 - The challenge
 - Privacy issues
 - Applications and Controversies
- 4 Pioneer
- 5 XOM
 - The XOM architecture
 - Preventing replay attacks
 - Fixing a leaky address bus
 - Fixing a leaky data bus
 - Discussion
- 6 Filesystem Security
 - SetUID
 - File Descriptors
 - Symbolic Links
- 7 Summary

Authenticated boot

- We've assumed the program executes in an untrusted environment. The adversary can
 - ① examine (reverse engineer),
 - ② modify (tamper),
 - ③ copy (pirate),our program.

Authenticated boot

- We've assumed the program executes in an untrusted environment. The adversary can
 - ① examine (reverse engineer),
 - ② modify (tamper),
 - ③ copy (pirate),our program.
- What if we could trust the client to run trusted
 - ① hardware,
 - ② operating system,
 - ③ applications?

Authenticated boot

- We've assumed the program executes in an untrusted environment. The adversary can
 - ① examine (reverse engineer),
 - ② modify (tamper),
 - ③ copy (pirate),our program.
- What if we could trust the client to run trusted
 - ① hardware,
 - ② operating system,
 - ③ applications?
- **Authenticated (or trusted) boot**: before you agree to communicate with a system (to allow it to buy a program from you, for example) you ask it to prove to you that it won't do anything bad.

Authenticated boot

- **Trusted Platform Module** (TPM) — Some PCs (such as IBM laptops) already have a TPM soldered onto the motherboard.
- There are research systems built on top of the TPM to provide trust.
- However, at the present time, no actual deployed systems.

Authenticated boot: Scenario

- 1 You want your program to run on Bob's computer.

Authenticated boot: Scenario

- ① You want your program to run on Bob's computer.
- ② Your program contains a secret, so you encrypt it.

Authenticated boot: Scenario

- ① You want your program to run on Bob's computer.
- ② Your program contains a secret, so you encrypt it.
- ③ You tell Bob to boot an operating system *eOS* that handles encrypted executables.

Authenticated boot: Scenario

- ① You want your program to run on Bob's computer.
- ② Your program contains a secret, so you encrypt it.
- ③ You tell Bob to boot an operating system *eOS* that handles encrypted executables.
- ④ But, is Bob actually running *eOS*? Could he be cheating?

Authenticated boot: Scenario

- 1 You want your program to run on Bob's computer.
- 2 Your program contains a secret, so you encrypt it.
- 3 You tell Bob to boot an operating system *eOS* that handles encrypted executables.
- 4 But, is Bob actually running *eOS*? Could he be cheating?
- 5 Bob could tell you he booted into the version of *eOS* you gave him, while in fact he first hacked it to leak the encryption keys.

Authenticated boot: Scenario

- 1 You want your program to run on Bob's computer.
- 2 Your program contains a secret, so you encrypt it.
- 3 You tell Bob to boot an operating system *eOS* that handles encrypted executables.
- 4 But, is Bob actually running *eOS*? Could he be cheating?
- 5 Bob could tell you he booted into the version of *eOS* you gave him, while in fact he first hacked it to leak the encryption keys.
- 6 You ask Bob compute a cryptographic hash over the kernel, and send it to you. Unless he lies about the hash, you should be fine!

Authenticated boot: Scenario

- 1 You want your program to run on Bob's computer.
- 2 Your program contains a secret, so you encrypt it.
- 3 You tell Bob to boot an operating system *eOS* that handles encrypted executables.
- 4 But, is Bob actually running *eOS*? Could he be cheating?
- 5 Bob could tell you he booted into the version of *eOS* you gave him, while in fact he first hacked it to leak the encryption keys.
- 6 You ask Bob compute a cryptographic hash over the kernel, and send it to you. Unless he lies about the hash, you should be fine!
- 7 Now, the kernel was started by a bootloader, and Bob could have hacked *it*, to modify the *eOS* image before loading it.

Authenticated boot: Scenario

- 1 You want your program to run on Bob's computer.
- 2 Your program contains a secret, so you encrypt it.
- 3 You tell Bob to boot an operating system eOS that handles encrypted executables.
- 4 But, is Bob actually running eOS? Could he be cheating?
- 5 Bob could tell you he booted into the version of eOS you gave him, while in fact he first hacked it to leak the encryption keys.
- 6 You ask Bob compute a cryptographic hash over the kernel, and send it to you. Unless he lies about the hash, you should be fine!
- 7 Now, the kernel was started by a bootloader, and Bob could have hacked *it*, to modify the eOS image before loading it.
- 8 So, you ask Bob to send you a hash of the bootloader.

Authenticated boot: Scenario

- 1 You want your program to run on Bob's computer.
- 2 Your program contains a secret, so you encrypt it.
- 3 You tell Bob to boot an operating system eOS that handles encrypted executables.
- 4 But, is Bob actually running eOS? Could he be cheating?
- 5 Bob could tell you he booted into the version of eOS you gave him, while in fact he first hacked it to leak the encryption keys.
- 6 You ask Bob compute a cryptographic hash over the kernel, and send it to you. Unless he lies about the hash, you should be fine!
- 7 Now, the kernel was started by a bootloader, and Bob could have hacked *it*, to modify the eOS image before loading it.
- 8 So, you ask Bob to send you a hash of the bootloader.
- 9 Of course, the bootloader was started by the BIOS, and *it* could have been hacked!

Authenticated boot: Scenario

- 1 You want your program to run on Bob's computer.
- 2 Your program contains a secret, so you encrypt it.
- 3 You tell Bob to boot an operating system *eOS* that handles encrypted executables.
- 4 But, is Bob actually running *eOS*? Could he be cheating?
- 5 Bob could tell you he booted into the version of *eOS* you gave him, while in fact he first hacked it to leak the encryption keys.
- 6 You ask Bob to compute a cryptographic hash over the kernel, and send it to you. Unless he lies about the hash, you should be fine!
- 7 Now, the kernel was started by a bootloader, and Bob could have hacked *it*, to modify the *eOS* image before loading it.
- 8 So, you ask Bob to send you a hash of the bootloader.
- 9 Of course, the bootloader was started by the BIOS, and *it* could have been hacked!
- 10 So, Bob sends you a hash of *it*, but. . .

Algorithm: Authenticated boot

- Before you agree to communicate with a system you ask it to prove to you that it won't do anything bad.

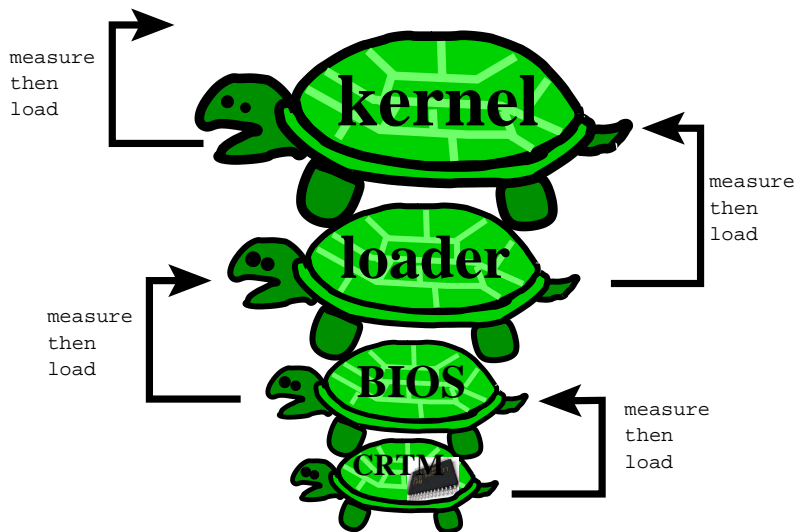
Algorithm: Authenticated boot

- Before you agree to communicate with a system you ask it to prove to you that it won't do anything bad.
- *Anything* that's running on Bob's computer could, potentially, affect whether you should trust it:
 - 1 OS,
 - 2 BIOS,
 - 3 bootloader,
 - 4 application programs,
 - 5 firmware, . . .

Authenticated boot: Basic idea

- The idea of trusted μ boot is to *measure every* potentially hostile piece of software running on a computer, and compare it against a library of **known-good-measurements**.
- **Measure**: “compute a cryptographic hash of”
- **Known-good-measurement**: a hash of a program you trust.
- If Bob can convince you that his computer, from the ground up, only runs code you trust, then you should have no problem handing him (or selling him) a program to run.

Authenticated boot



Authenticated boot: Basic idea

- At the very bottom of the stack of turtles there are two things you need to trust:
 - 1 **Core Root of Trust Measurement** (CRTM): piece of code contained in the BIOS which measures itself and the BIOS, before letting the BIOS execute.
 - 2 **Tamperproof Module**: this is where the measurements are stored.

Authenticated boot: Basic idea

- At the very bottom of the stack of turtles there are two things you need to trust:
 - 1 **Core Root of Trust Measurement** (CRTM): piece of code contained in the BIOS which measures itself and the BIOS, before letting the BIOS execute.
 - 2 **Tamperproof Module**: this is where the measurements are stored.
- The BIOS measures the bootloader, stores away the measurement and lets the bootloader execute.

Authenticated boot: Basic idea

- At the very bottom of the stack of turtles there are two things you need to trust:
 - 1 **Core Root of Trust Measurement** (CRTM): piece of code contained in the BIOS which measures itself and the BIOS, before letting the BIOS execute.
 - 2 **Tamperproof Module**: this is where the measurements are stored.
- The BIOS measures the bootloader, stores away the measurement and lets the bootloader execute.
- The bootloader measures the OS kernel, saves the result, and loads the OS.

Authenticated boot: Basic idea

- At the very bottom of the stack of turtles there are two things you need to trust:
 - 1 **Core Root of Trust Measurement** (CRTM): piece of code contained in the BIOS which measures itself and the BIOS, before letting the BIOS execute.
 - 2 **Tamperproof Module**: this is where the measurements are stored.
- The BIOS measures the bootloader, stores away the measurement and lets the bootloader execute.
- The bootloader measures the OS kernel, saves the result, and loads the OS.
- The OS measures kernel modules, configuration files, stores them in the TPM, then lets applications run.

Secure boot vs. Authenticated boot

- **Secure boot**: only ever boot a system consisting of code that you trust.
- **Authenticated boot**:
 - Bob can boot whatever system he wants!
 - But, he cannot lie about what system he's booted!

Taking measurements

- 1 User hits power button.

Taking measurements

- 1 User hits power button.
- 2 The computer's POST (*Power On and Self Test*) function is invoked.

Taking measurements

- ① User hits power button.
- ② The computer's POST (*Power On and Self Test*) function is invoked.
- ③ The BIOS is loaded and run.

Taking measurements

- ① User hits power button.
- ② The computer's POST (*Power On and Self Test*) function is invoked.
- ③ The BIOS is loaded and run.
- ④ The BIOS initializes the TPM.

Taking measurements

- ① User hits power button.
- ② The computer's POST (*Power On and Self Test*) function is invoked.
- ③ The BIOS is loaded and run.
- ④ The BIOS initializes the TPM.
- ⑤ The BIOS invokes the CRTM (contained inside the BIOS).

Taking measurements

- 1 User hits power button.
- 2 The computer's POST (*Power On and Self Test*) function is invoked.
- 3 The BIOS is loaded and run.
- 4 The BIOS initializes the TPM.
- 5 The BIOS invokes the CRTM (contained inside the BIOS).
- 6 The CRTM compute a SHA-1 hash over the BIOS.

Taking measurements

- 1 User hits power button.
- 2 The computer's POST (*Power On and Self Test*) function is invoked.
- 3 The BIOS is loaded and run.
- 4 The BIOS initializes the TPM.
- 5 The BIOS invokes the CRTM (contained inside the BIOS).
- 6 The CRTM compute a SHA-1 hash over the BIOS.
- 7 The BIOS compute a SHA-1 hash over the BootLoader.

Taking measurements

- 1 User hits power button.
- 2 The computer's POST (*Power On and Self Test*) function is invoked.
- 3 The BIOS is loaded and run.
- 4 The BIOS initializes the TPM.
- 5 The BIOS invokes the CRTM (contained inside the BIOS).
- 6 The CRTM compute a SHA-1 hash over the BIOS.
- 7 The BIOS compute a SHA-1 hash over the BootLoader.
- 8 The BIOS runs the BootLoader.

Taking measurements

- 1 User hits power button.
- 2 The computer's POST (*Power On and Self Test*) function is invoked.
- 3 The BIOS is loaded and run.
- 4 The BIOS initializes the TPM.
- 5 The BIOS invokes the CRTM (contained inside the BIOS).
- 6 The CRTM compute a SHA-1 hash over the BIOS.
- 7 The BIOS compute a SHA-1 hash over the BootLoader.
- 8 The BIOS runs the BootLoader.
- 9 The BootLoader measures and calls the OSKernel.

Taking measurements

- 1 User hits power button.
- 2 The computer's POST (*Power On and Self Test*) function is invoked.
- 3 The BIOS is loaded and run.
- 4 The BIOS initializes the TPM.
- 5 The BIOS invokes the CRTM (contained inside the BIOS).
- 6 The CRTM compute a SHA-1 hash over the BIOS.
- 7 The BIOS compute a SHA-1 hash over the BootLoader.
- 8 The BIOS runs the BootLoader.
- 9 The BootLoader measures and calls the OSKernel.
- 10 The OSKernelh measures and calls an application.

Stored Measurement Lists

```
static class OSKernel {  
    public static LinkedList [] SML;  
    public static void run() {  
        SML = new LinkedList [TPM.NumberOfPCRs];  
        for (int i=0; i<TPM.NumberOfPCRs; i++)  
            SML[i] = new LinkedList ();  
        SML[0].addLast (BIOS.BIOSHash);  
        SML[1].addLast (BIOS.BootLoaderHash);  
        SML[2].addLast (BIOS.OSKernelHash);  
        TPM.extend (10, TPM.SHA1 (Application.code));  
        SML[10].addLast (TPM.SHA1 (Application.code));  
        TPM.extend (10, TPM.SHA1 (Application.input));  
        SML[10].addLast (TPM.SHA1 (Application.input));  
        Application.run ();  
    }  
}
```

Stored Measurement Lists

- Each measurement is stored in two places:
 - ① on the TPM (using the `TPM.extend()` call);
 - ② in the kernel in an array of lists, *Stored Measurement List* (SML).
- When you challenge the computer to prove that it's benign it will return the SML so that you can check that all the measurements correspond to programs you trust.
- The SML is stored in the kernel (not on the TPM) because it could be large.

Stored Measurement Lists...

- Just storing the hashes in the kernel isn't enough — the kernel could be malicious and lie about them!
- The TPM therefore stores a “summary” of the hashes — a “digest-of-the-digests” — in on-chip registers.
- There are 16 *Platform Configuration Registers* (PCR), 20 bytes long, the size of a SHA-1 hash.

The TPM

```
static class TPM {  
    public static final int NumberOfPCRs = 16;  
    private static byte[][] PCR;  
    public static void extend (int i, byte[] b) {  
        int len = 1 + PCR[i].length + add.length;  
        ubyte[] res = new byte[len];  
        res[0] = (byte)i;  
        System.arraycopy(  
            PCR[i], 0, res, 1, PCR[i].length);  
        System.arraycopy(  
            add, 0, res, PCR[i].length, add.length);  
        return SHA1(res);  
    }  
}
```

The TPM...

- The `extend(i,b)` function doesn't just assign a new value to `PCR[i]`.
- It *extends* it by computing

$$\text{PCR}[i] = \text{SHA1}(i \parallel \text{PCR}[i] \parallel b).$$

- Each `PCR[i]` register becomes a combination of all hashes ever assigned to it and preserves the order in which the measurements were added.

The TPM: Data and Functions

```
static class TPM {  
    private static KeyPair EK;  
    private static byte [][] PCR;  
    private static byte [] ownerSecret;  
    public static void extend (int i, byte [] b)  
    public static Object [] quote(byte [] nonce)  
    public static byte [] SHA1 (byte [] b)  
    public static byte [] signRSA(  
        Object data, PrivateKey key)  
    public static KeyPair generateRSAKeyPair()  
    public static byte [] RND(int size)  
    public static void atManufacture()  
    public static void takeOwnership(  
        byte [] password)  
    public static void POST()  
}
```

The TPM: Data and Functions

```
public static void atManufacture() {  
    EK = generateRSAKeyPair();  
}  
public static void takeOwnership(  
    byte[] password) {  
    ownerSecret = password;  
}  
public static void POST() {  
    PCR = new byte[NumberOfPCRs][];  
    for(int i=0; i<NumberOfPCRs; i++)  
        PCR[i] = new byte[20];  
}
```


The TPM: Data and Functions...

- At manufacturing time the TPM gets a unique identity, an RSA key pair (the *Endorsement Key* (EK)).
- When the owner takes possession of the computer he gives the TPM with a secret password.
- At system startup time, the PCR registers are zeroed out.

Server

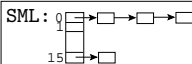
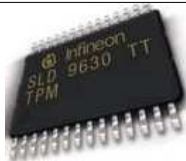
1. nonce ← RND()
send nonce

Client

SHA-1()

EK

PCR[0]
PCR[1]
...
PCR[15]



kernel

Server

1. nonce ← RND()
send nonce

Client

2. receive nonce

SHA-1()

EK

PCR[0]

PCR[1]

...

PCR[15]



SML: 0 → [] → [] → []

15 → []

kernel

Server

1. $\text{nonce} \leftarrow \text{RND}()$
send nonce

Client

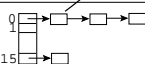
2. receive nonce
3. send (quote, SML)

SHA-1()

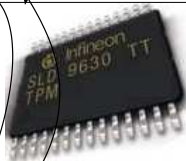
PCR[0]
PCR[1]

PCR[15]

$\text{quote} \leftarrow \text{sig}\{\text{PCR}, \text{nonce}\}_{EK_{priv}}$

SML: 

kernel



Server

1. $\text{nonce} \leftarrow \text{RND}()$
send nonce
4. receive (quote,SML)

Client

2. receive nonce
3. send (quote,SML)

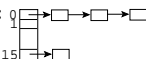
SHA-1()

EK

PCR[0]
PCR[1]
...
PCR[15]



$\text{quote} \leftarrow \text{sig}\{\text{PCR}, \text{nonce}\}_{\text{EK}_{\text{priv}}}$

SML: 

kernel

Server

1. $\text{nonce} \leftarrow \text{RND}()$
send nonce
4. receive (quote, SML)
5. $\text{cert}(EK_{\text{pub}})$ OK?

Client

2. receive nonce
3. send (quote, SML)

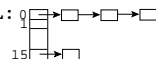
SHA-1()

EK

PCR[0]
PCR[1]
...
PCR[15]



$\text{quote} \leftarrow \text{sig}\{\text{PCR}, \text{nonce}\}_{EK_{\text{priv}}}$

SML:  kernel

Server

1. $\text{nonce} \leftarrow \text{RND}()$
send nonce
4. receive (quote, SML)
5. $\text{cert}(EK_{pub})$ OK?
6. $\text{sig}\{\text{PCR}, \text{nonce2}\}_{EK_{priv}}$ valid?

Client

2. receive nonce
3. send (quote, SML)

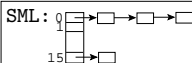
SHA-1()

EK

PCR[0]
PCR[1]
...
PCR[15]



$\text{quote} \leftarrow \text{sig}\{\text{PCR}, \text{nonce}\}_{EK_{priv}}$



kernel

Server

1. $\text{nonce} \leftarrow \text{RND}()$
send nonce
4. receive (quote, SML)
5. $\text{cert}(EK_{pub})$ OK?
6. $\text{sig}\{\text{PCR}, \text{nonce2}\}_{EK_{priv}}$ valid?
7. nonce2 fresh?
SML not tampered?
measurements OK?

Client

2. receive nonce
3. send (quote, SML)

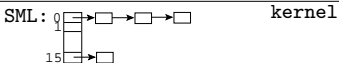
SHA-1()

EK

PCR[0]
PCR[1]
...
PCR[15]



$\text{quote} \leftarrow \text{sig}\{\text{PCR}, \text{nonce}\}_{EK_{priv}}$



The Challenge

```
public static Object [] quote(byte [] nonce) {
    Object [] data = {nonce, PCR};
    byte [] sig = signRSA(data, EK.getPrivate());
    return new Object [] {nonce, PCR, sig};
}
```

- To avoid replay attacks, the challenger starts by creating a random value, a **nonce**, and sends it in a challenge to the computer.
- The computer asks the TPM for a *quote*, i.e. the values of the PCRs, signed with the endorsement private key EK_{priv} (actually, a different key is used, for privacy reasons).

The Challenge. . .

- The computer collects the stored measurement list SML, packages it all up, and returns the package to the challenger.
- The challenger looks up a certificate corresponding to the TPM's public key EK_{pub} , and verifies that the TPM belongs to a trusted computer.
- The challenger validates the signature on the package it received from the remote computer (only the TPM knows the private key EK_{priv}).
- The challenger walks through the stored measurement lists and, merges the values together by simulating the TPM's `extend()` function. If the aggregate measurements match those of the PCRs you can be sure that the SML wasn't tampered with.
- Finally, check the measurements against a white- or blacklist.

Privacy issues

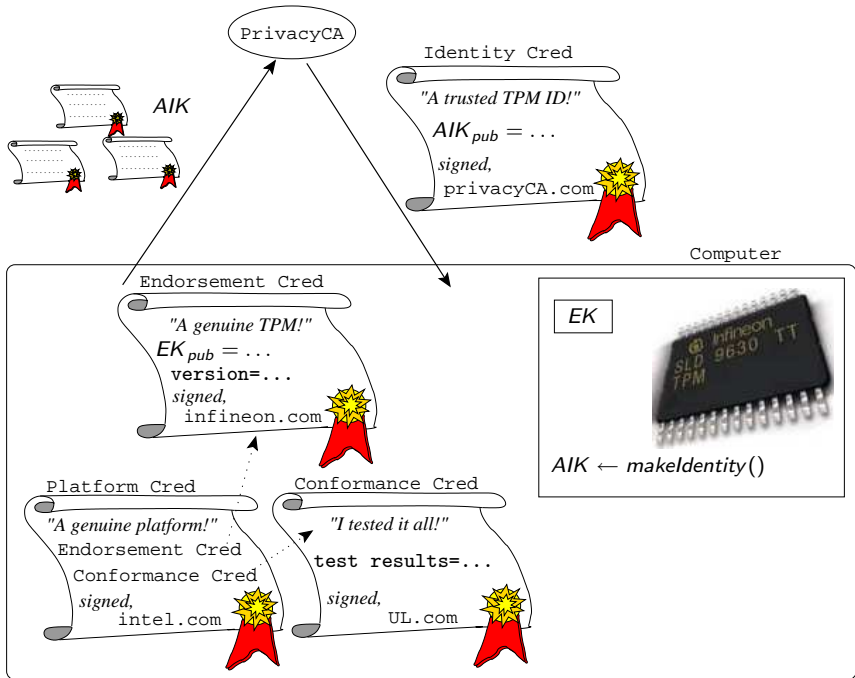
- We said that the computer is using the endorsement key EK to identify itself.
- We lied.
- Using the EK is a really bad idea because it has the potential to compromise your privacy.
- The EK public key uniquely identifies the TPM, and hence *you*.
- You want the challenger to learn that you're using a TPM-enabled trusted platform but not which one, or else all your transactions could be linked to each other.

Attestation Identity Key

- Instead we use a special RSA keypair, the *Attestation Identity Key* (AIK) that the TPM manufactures before engaging in the challenge protocol.
- No one knows that the AIK and the EK actually represent the same computer, except a trusted third party, the **Privacy Certification Authority**, (PrivacyCA).
- The PrivacyCA knows all the public keys EK_{pub} of all manufactured TPMs.
- The computer and the PrivacyCA engage in a protocol to manufacture an *identity credential* that the computer can use to prove that it's a compliant tamperproof platform, but without revealing exactly *who* it is.

Protocol to manufacture an identity credential. . .

- The computer holds three credentials:
 - ① **endorsement credential** — the TPM is genuine, the EK_{pub} is the public part of the endorsement keypair that was squirted into it during manufacture (signed by the TPM manufacturer);
 - ② *platform credential* — the TPM and the motherboard it's been soldered onto make up a trusted system (signed by the manufacturer of the platform);
 - ③ *conformance credential* — the platform has been tested by a third party and found to conform to certain security properties (signed by a testing lab).



Protocol to manufacture an identity credential. . .

- To obtain an identity credential the computer
 - ① asks the TPM to manufacture a new AIK key-pair;
 - ② bundles up AIK_{pub} together with all the credentials;
 - ③ sends everything off to the PrivacyCA.
- The PrivacyCA
 - ① convinces itself that the credentials belong to a genuine tamperproof platform;
 - ② issues the identity credential and returns it to the computer.

Social trust

- The endorsement, platform, and conformance credentials are published in the form of digital certificates.
- The certificates are signed by companies willing to put their good name behind a guarantee that a particular computer can be trusted.
- This is a form of **social trust**.
- You trust that the companies which manufactured and tested it are trustworthy.
- If they didn't take great care in ensuring that no security flaws were allowed to creep in — their brand name could be damaged.

Trusted third party limitation

- Having to rely on a trusted third party (TTP) such as the PrivacyCA is limiting:
 - ① the PrivacyCA will be involved in every transaction;
 - ② if the PrivacyCA's security is compromised the identity of every TPM will be revealed.
- There are other protocols which avoid the TTP.

- Hatred for Digital Rights Management.
- Privacy issues.
- Technical doubts that authenticated boot has any hope of succeeding in the real world.

Technical issues: IBM's implementation

- Database of measurements for Redhat Fedora: 25000 measurements.
- A typical SML: 700-1000 measurements.

Technical issues: IBM's implementation

- Database of measurements for Redhat Fedora: 25000 measurements.
- A typical SML: 700-1000 measurements.
- How to collect “good” measurements:
 - ① boot a “trusted system”
 - ② measure all modules, config files, scripts ⇒ whitelist of hashes

Technical issues: IBM's implementation

- Database of measurements for Redhat Fedora: 25000 measurements.
- A typical SML: 700-1000 measurements.
- How to collect “good” measurements:
 - 1 boot a “trusted system”
 - 2 measure all modules, config files, scripts ⇒ whitelist of hashes
- How to collect “bad” measurements:
 - 1 boot a compromised system (root kits, trojans, ...)
 - 2 measure infected files ⇒ blacklist of hashes
- How do we keep these lists up-to-date?!?!

Uses and Abuses

- Help digital rights management players to run untampered on PCs.
- Before you're allowed you to buy and download a movie, the movie studio will verify that only approved software and hardware is installed on your computer.
- If you have the *SoftICE* debugger on your harddisk, or you're running an out-of-date kernel, or your media, you're out of luck.
- If you *are* approved, the movie will be encrypted with your public key AIK_{pub} so that only an approved player running on an approved OS on a computer with an approved TPM can decrypt and play it.
- Actually, the movie will now only play on *your* computer since it's been tied directly to your TPM.

- If your OS happens to be localized to a part of the world where the movie has yet to appear in theaters, the studio may decide to refuse the download.
- The OS can't lie about any of the files on the harddisk, including any configuration files, it can't lie about in which part of the world it's running.

- Disney encrypts Nemo with a special **sealing key** *Seal*.

- Disney encrypts Nemo with a special **sealing key** *Seal*.
 - *Seal* depends on
 - the values in the PCRs
 - the TPM itself.
- ⇒ your friend with an identical computer can't watch your copy of Nemo!

- Disney encrypts Nemo with a special **sealing key** *Seal*.
 - *Seal* depends on
 - the values in the PCRs
 - the TPM itself.
- ⇒ your friend with an identical computer can't watch your copy of Nemo!
- If you reboot with slightly hacked OS
 - the PCRs will have changed
 - ⇒ the *Seal* will be different
 - ⇒ you can't decrypt Nemo!

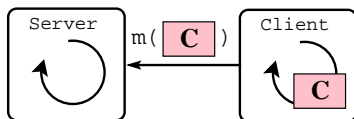
- Disney encrypts Nemo with a special **sealing key** *Seal*.
- *Seal* depends on
 - the values in the PCRs
 - the TPM itself.

⇒ your friend with an identical computer can't watch your copy of Nemo!
- If you reboot with slightly hacked OS
 - the PCRs will have changed
 - ⇒ the *Seal* will be different
 - ⇒ you can't decrypt Nemo!
- Microsoft could do the same to protect Office.

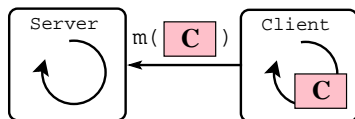
Outline

- 1 Introduction
 - Processes
 - Inter-Process Communication
 - Memory management
 - Virtual machines
- 2 Process Security
- 3 Authenticated boot
 - Trusted boot
 - Taking measurements
 - The TPM
 - The challenge
 - Privacy issues
 - Applications and Controversies
- 4 **Pioneer**
- 5 XOM
 - The XOM architecture
 - Preventing replay attacks
 - Fixing a leaky address bus
 - Fixing a leaky data bus
 - Discussion
- 6 Filesystem Security
 - SetUID
 - File Descriptors
 - Symbolic Links
- 7 Summary

- In a *very* restricted environment you can *measure* aspects of the untrusted client to verify that it is running the correct software:

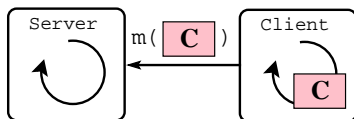


- In a *very* restricted environment you can *measure* aspects of the untrusted client to verify that it is running the correct software:



- Assume
 - 1 The the client's hardware configuration is known;
 - 2 The client-server latency is known;
 - 3 The client can only communicate with the server.

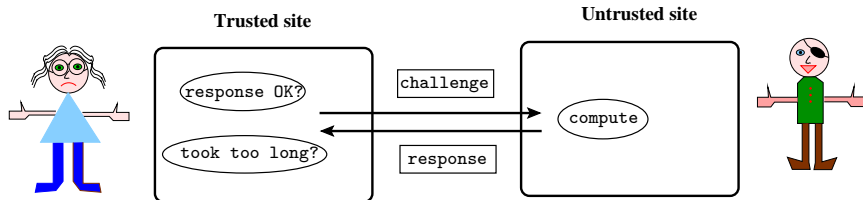
- In a *very* restricted environment you can *measure* aspects of the untrusted client to verify that it is running the correct software:



- Assume
 - 1 The client's hardware configuration is known;
 - 2 The client-server latency is known;
 - 3 The client can only communicate with the server.
- Applications:
 - 1 Check cell phone/PDA/smartcard for viruses;
 - 2 Check voting machine code;
 - 3 Check for rootkits on machines on your LAN.

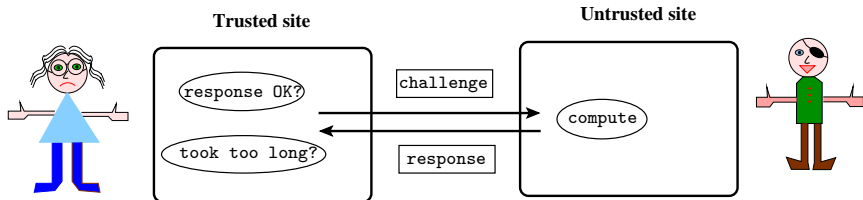
Pioneer...

- **Measure** the untrusted client code:



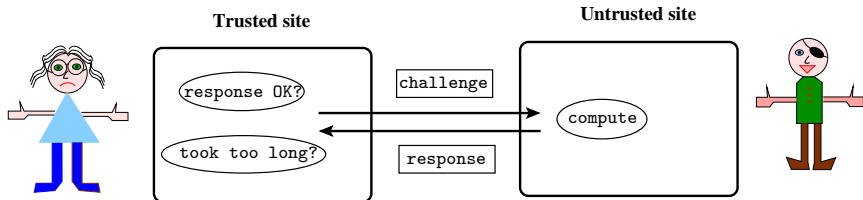
Pioneer...

- **Measure** the untrusted client code:



- Assume a very **restricted environment**:
 - 1 The the client's hardware configuration is known;
 - 2 The client-server latency is known;
 - 3 The client can only communicate with the server.

- **Measure** the untrusted client code:



- Assume a very **restricted environment**:
 - 1 The the client's hardware configuration is known;
 - 2 The client-server latency is known;
 - 3 The client can only communicate with the server.
- Applications:
 - 1 Check cell phone/PDA/smartcard for viruses;
 - 2 Check voting machine code;
 - 3 Check for rootkits on machines on your LAN.

Pioneer: Protocol

- Basic idea: ask client for a hash of its code.
- If
 - 1 the hash is the wrong value, or
 - 2 the computation took too longthe client has cheated!
- The hash function is constructed such that it can't be computed quicker.

Server

```
1.  $t_1 \leftarrow \text{currentTime}()$   
    $\text{nonce} \leftarrow \text{random}()$   
   send nonce
```

Client

V:

hash6()

send()

SHA-1()

E:

executable

Server

```
1.  $t_1 \leftarrow \text{currentTime}()$   
   nonce  $\leftarrow \text{random}()$   
   send nonce
```

Client

```
2. receive nonce  
    $c \leftarrow \text{hash6}(\text{nonce}, V)$   
   send  $c$ 
```

V:

```
hash6()
```

```
send()
```

```
SHA-1()
```

E:

```
executable
```

Server

```
1.  $t_1 \leftarrow \text{currentTime}()$   
   nonce  $\leftarrow \text{random}()$   
   send nonce  
  
3. receive  $c$  ←  
    $t_2 \leftarrow \text{currentTime}()$   
   if  $t_2 - t_1 > \Delta t$  or  
      $c$  is wrong then  
     FAIL
```

Client

```
2. receive nonce  
    $c \leftarrow \text{hash6}(\text{nonce}, V)$   
   send  $c$ 
```

V:

hash6()

send()

SHA-1()

E:

executable

Server

1. $t_1 \leftarrow \text{currentTime}()$
 $\text{nonce} \leftarrow \text{random}()$
 send nonce
3. receive c
 $t_2 \leftarrow \text{currentTime}()$
 if $t_2 - t_1 > \Delta t$ or
 c is wrong then
 FAIL

Client

2. receive nonce
 $c \leftarrow \text{hash6}(\text{nonce}, V)$
 send c

V :

hash6()

send()


SHA-1()

4. $h \leftarrow \text{SHA-1}(\text{nonce}||E)$
 send h

E :

executable

Server

1. $t_1 \leftarrow \text{currentTime}()$
nonce $\leftarrow \text{random}()$
send nonce
3. receive c
 $t_2 \leftarrow \text{currentTime}()$
if $t_2 - t_1 > \Delta t$ or
 c is wrong then
 FAIL
5. receive h 
if h is wrong then
 FAIL

Client

2. receive nonce
 $c \leftarrow \text{hash6}(\text{nonce}, V)$
send c

V :

hash6()

send()

SHA-1()

4. $h \leftarrow \text{SHA-1}(\text{nonce}||E)$
send h

E :

executable

Server

1. $t_1 \leftarrow \text{currentTime}()$
nonce $\leftarrow \text{random}()$
send nonce
3. receive c
 $t_2 \leftarrow \text{currentTime}()$
if $t_2 - t_1 > \Delta t$ or
 c is wrong then
FAIL
5. receive h
if h is wrong then
FAIL

Client

2. receive nonce
 $c \leftarrow \text{hash6}(\text{nonce}, V)$
send c

V :

hash6()

send()

SHA-1()

4. $h \leftarrow \text{SHA-1}(\text{nonce}||E)$
send h

6. $r \leftarrow \text{execute } E$
send r

E :

executable

Server

1. $t_1 \leftarrow \text{currentTime}()$
nonce $\leftarrow \text{random}()$
send nonce
3. receive c
 $t_2 \leftarrow \text{currentTime}()$
if $t_2 - t_1 > \Delta t$ or
 c is wrong then
FAIL
5. receive h
if h is wrong then
FAIL
7. receive r ←

Client

2. receive nonce
 $c \leftarrow \text{hash6}(\text{nonce}, V)$
send c

V:

hash6()

send()

SHA-1()

4. $h \leftarrow \text{SHA-1}(\text{nonce}||E)$
send h

6. $r \leftarrow \text{execute } E$
send r

E:

executable

Pioneer: Protocol

- The hash function must be **time optimal**, if not
 - the client can use the time he saved to execute his own instructions without the server noticing.

Pioneer: Protocol

- The hash function must be **time optimal**, if not
 - the client can use the time he saved to execute his own instructions without the server noticing.
- Others have tried to extend the protocol to general scenarios — **highly controversial**.

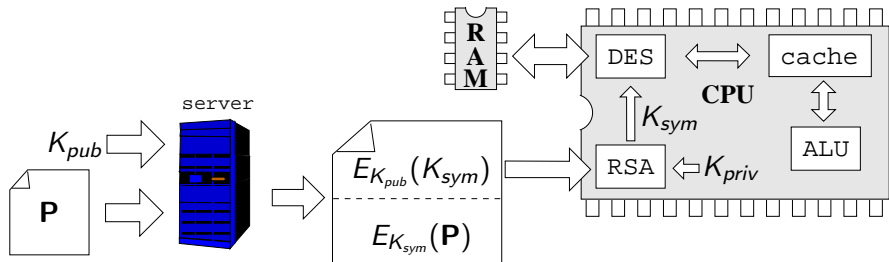
Outline

- 1 Introduction
 - Processes
 - Inter-Process Communication
 - Memory management
 - Virtual machines
- 2 Process Security
- 3 Authenticated boot
 - Trusted boot
 - Taking measurements
 - The TPM
 - The challenge
 - Privacy issues
 - Applications and Controversies
- 4 Pioneer
- 5 **XOM**
 - The XOM architecture
 - Preventing replay attacks
 - Fixing a leaky address bus
 - Fixing a leaky data bus
 - Discussion
- 6 Filesystem Security
 - SetUID
 - File Descriptors
 - Symbolic Links
- 7 Summary

Encrypted execution

- Idea:
 - Encrypt the program.
 - Keep a (unique) secret key in the CPU.
 - Decrypt inside the CPU.
- Protect algorithms (privacy)!
- Protect from tampering (integrity)!
- Protect from cloning (piracy)!
- Assume the CPU cannot be tampered with.

XOM Overview



- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .

XOM Overview...

- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .
- 2 K_{priv} is stored inside your CPU.

XOM Overview...

- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .
- 2 K_{priv} is stored inside your CPU.
- 3 The server generates a symmetric session key K_{sym} and encrypts P .

XOM Overview...

- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .
- 2 K_{priv} is stored inside your CPU.
- 3 The server generates a symmetric session key K_{sym} and encrypts P .
- 4 It then encrypts the session key with K_{pub} .

XOM Overview...

- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .
- 2 K_{priv} is stored inside your CPU.
- 3 The server generates a symmetric session key K_{sym} and encrypts P .
- 4 It then encrypts the session key with K_{pub} .
- 5 The server creates a new executable file consisting of the encrypted code and a file header containing the encrypted session key, i.e.

$$[E_{K_{pub}}(K_{sym}) || E_{K_{sym}}(P)].$$

XOM Overview...

- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .
- 2 K_{priv} is stored inside your CPU.
- 3 The server generates a symmetric session key K_{sym} and encrypts P .
- 4 It then encrypts the session key with K_{pub} .
- 5 The server creates a new executable file consisting of the encrypted code and a file header containing the encrypted session key, i.e.

$$[E_{K_{pub}}(K_{sym}) || E_{K_{sym}}(P)].$$

- 6 You download this file and give it to the OS to execute.

XOM Overview...

- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .
- 2 K_{priv} is stored inside your CPU.
- 3 The server generates a symmetric session key K_{sym} and encrypts P .
- 4 It then encrypts the session key with K_{pub} .
- 5 The server creates a new executable file consisting of the encrypted code and a file header containing the encrypted session key, i.e.

$$[E_{K_{pub}}(K_{sym}) || E_{K_{sym}}(P)].$$

- 6 You download this file and give it to the OS to execute.
- 7 The OS gives the CPU $E_{K_{pub}}(K_{sym})$ and the CPU decrypts it with K_{priv} .

XOM Overview...

- 1 To buy a program P from the server you start by handing it your public RSA key K_{pub} .
- 2 K_{priv} is stored inside your CPU.
- 3 The server generates a symmetric session key K_{sym} and encrypts P .
- 4 It then encrypts the session key with K_{pub} .
- 5 The server creates a new executable file consisting of the encrypted code and a file header containing the encrypted session key, i.e.

$$[E_{K_{pub}}(K_{sym}) || E_{K_{sym}}(P)].$$

- 6 You download this file and give it to the OS to execute.
- 7 The OS gives the CPU $E_{K_{pub}}(K_{sym})$ and the CPU decrypts it with K_{priv} .
- 8 The CPU can now decrypt and execute the encrypted code.

- The DES engine sits between external RAM and the cache so that every piece of code or data that gets read or written will first pass through it.
- In other words:
 - external RAM is always encrypted,
 - internal data and code is always in the clear,
 - the private key K_{priv} and the decrypted symmetric K_{sym} never leave the CPU.

Encrypted execution — Consequences

- Solves the copy protection problem — each program is tied to one CPU!
- Solves the privacy problem — code or data always encrypted off-chip!
- Solves the integrity problem — changing encrypted code is hard!
- Actually: There are still attacks!

The XOM architecture

- Stanford design.
- Never implemented in silicon.
- Simulated in software.
- Operating system, *XOMOS*, runs on top of it.

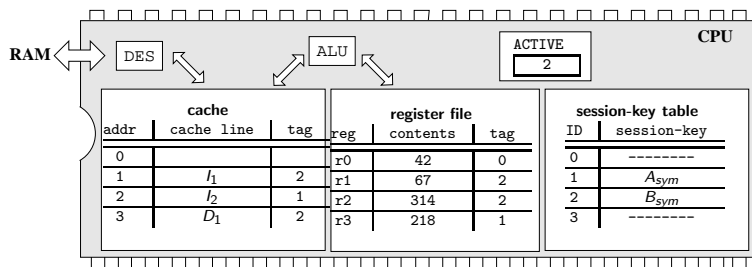
The XOM architecture — Compartments

- Each process may run in different security mode.
- Encrypted programs are slow!
- Programs may switch between encrypted and cleartext execution.
- CPU has 4 **Compartments**:
 - logical containers
 - protect one process from being observed or modified by another process.
 - the OS is untrusted: runs in its own compartment!
- Even a severely subverted operating system won't be allowed to examine or manipulate another protected process!

The XOM architecture — Compartments

- **Compartment 0**: code runs unencrypted
- **ACTIVE** register: current executing compartment
- **Session key table**: Maps compartment ID to key.
- Each **register** is tagged with compartment key.
- Each **cache line** is tagged with compartment key.
- On-chip data is in cleartext.
- On cache flush: encrypt!

The XOM architecture — Example

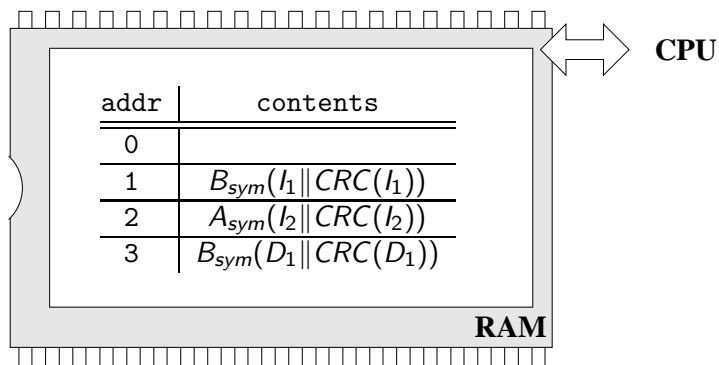


- Currently, two programs A and B are running.

The XOM architecture — Example. . .

- Two compartments, 1 and 2, for programs A and B .
- Compartment 2 is the currently active.
- Session-key table: $1 \mapsto A_{sym}, 2 \mapsto B_{sym}$.
- Registers $r1$ and $r2$ belong to compartment 2, register $r3$ to compartment 1.
- Register $r0$ is unprotected \Rightarrow compartments 1 and 2 could use $r0$ for insecure communication.

The XOM architecture — Example...



The XOM architecture — Example

The CPU tries to load data value D_1 at address 3 into register $r0$:

- 1 Look in the cache line: empty!

The XOM architecture — Example

The CPU tries to load data value D_1 at address 3 into register $r0$:

- 1 Look in the cache line: empty!
- 2 Cache miss, read $B_{sym}(D_1 || CRC(D_1))$ from address 3.

The XOM architecture — Example

The CPU tries to load data value D_1 at address 3 into register $r0$:

- 1 Look in the cache line: empty!
- 2 Cache miss, read $B_{sym}(D_1 || CRC(D_1))$ from address 3.
- 3 Look up key for the active compartment: B_{sym} .

The XOM architecture — Example

The CPU tries to load data value D_1 at address 3 into register $r0$:

- 1 Look in the cache line: empty!
- 2 Cache miss, read $B_{sym}(D_1 || CRC(D_1))$ from address 3.
- 3 Look up key for the active compartment: B_{sym} .
- 4 Decrypt the cache-line!

The XOM architecture — Example

The CPU tries to load data value D_1 at address 3 into register r0:

- 1 Look in the cache line: empty!
- 2 Cache miss, read $B_{sym}(D_1 || CRC(D_1))$ from address 3.
- 3 Look up key for the active compartment: B_{sym} .
- 4 Decrypt the cache-line!
- 5 Adversary could have swapped D_1 it for another encrypted value from some other part of the code!
 - Store CRC hash of each cache line.
 - If CRC doesn't match \Rightarrow exception!
 - Otherwise, load D_1 into register r0
 - Set r0's tag to 2.

The XOM architecture — ISA modifications

`secure_load reg, addr:`

- On a cache-miss, load the cache-line at address *addr*.
- Decrypt using the session key of the currently active process.
- If the hash doesn't validate, throw an exception.
- Store the decrypted value in *reg*
- Tag *reg* with the tag of the active process.

The XOM architecture — ISA modifications. . .

`secure_store reg, addr:`

- *reg*'s tag \neq active tag, exception!
- Store *reg* to *addr*'s cache-line
- Set the cache mine tag to the tag of the active process.
- On a cache-flush:
 - 1 Compute a hash of the cache line and its virtual memory address,
 - 2 Encrypt the cache-line and the hash with the session key,
 - 3 Write to memory.

The XOM architecture — ISA modifications. . .

Insecure load and store instructions work on data in the null compartment:

load reg, addr:

- 1 Load the value at address *addr* into register *reg*
- 2 Set its tag to 0.

store reg, addr:

- 1 If *reg* isn't tagged with 0, exception!
- 2 Write it to address *addr*.

The XOM architecture — ISA modifications. . .

`move_to_null reg`: If `reg`'s tag is different from the active tag, throw an exception. Otherwise, set `reg`'s tag to 0.

`move_from_null reg`: If `reg`'s tag is isn't 0, throw an exception. Otherwise, set `reg`'s tag to the tag of the currently active process.

- The OS needs to move data from a device into a user process.
- Two processes can insecurely exchange data: process 1 loads the data into a register, changes its tag to 0, process 2 changes the tag from 0 to its own tag value.

The XOM architecture — ISA modifications. . .

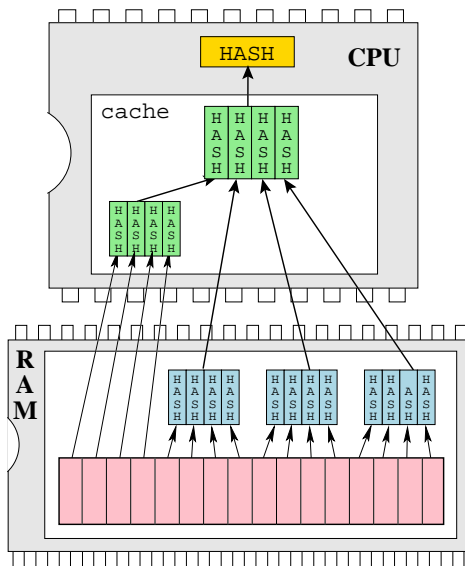
Enter and exit protected mode:

- enter *addr*:** The encrypted session-key is stored at address *addr*. If it's already been loaded into a slot in the session-key table, just set the ACTIVE register to the slot tag. Otherwise, load the key, decrypt it with the CPU's private key, store into an empty slot in the session-key table, and set the ACTIVE register. Start fetching and decrypting instructions.
- exit:** Set ACTIVE to 0. Stop decrypting instructions.

Attack: Replay!

- The adversary can save a value from a particular location and then later write it back into the same location!
- Check that a value you write into memory hasn't been changed the next time you read from the same location.

Replay attacks — Merkle Tree



Replay attacks — Merkle Tree...

- The leaves of the tree are chunks of memory that you want to protect.
- Internal nodes are hashes of child nodes.
- The root of the tree is a hash stored protected on the CPU.
- A replay attack won't work: the attacker can change a word in memory, and the hash of the word, and the hash of the hash — but not the root!

Replay attacks — Merkle Tree...

- Performance...
- Extra space: $\frac{1}{4}$ th of memory is taken up by hashes.
- Extra cost in memory bandwidth: a balanced m -ary tree of memory of size $N \Rightarrow \log_m(N)$ hash checks per read.
- Solution: cache parts of the tree on chip in the L2 cache \Rightarrow only 20% performance hit.

Attack: Watching the address bus!

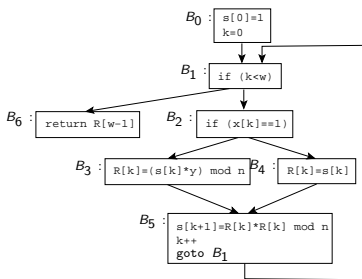
- OK, so maybe the adversary can't directly examine an executable since it's encrypted.
- He may still be able to extract information from it by examining its control flow execution pattern.
- In this attack the adversary examines the addresses going across the bus.

Modular exponentiation routine

- Consider the *modular exponentiation* routine used in RSA and Diffie-Hellman.
- x is the private key, w bits long.
- In the XOM architecture code and data doesn't move.
- Blocks of code are encrypted but reside in the same location in memory throughout execution.

Modular exponentiation routine. . .

```
s[0] = 1;
for(k=0; k<w; k++) {
    if (x[k] == 1)
        R[k] = (s[k]*y) mod n;
    else
        R[k] = s[k];
    s[k+1] = R[k]*R[k] mod n
}
return R[w-1];
```



Examining runtime execution patterns

- The encrypted blocks could be laid out in memory like this:

000	100	200	300	400	500	600
$E_k(B_0)$	$E_k(B_1)$	$E_k(B_2)$	$E_k(B_3)$	$E_k(B_4)$	$E_k(B_5)$	$E_k(B_6)$

- An adversary monitors the address bus while a secret message is being decrypted would see:

```
⟨000,  
    100, 200, 300, 500,  
    100, 200, 300, 500,  
    100, 200, 400, 500,  
    100, 200, 300, 500,  
        ...  
    100,  
600⟩
```

Examining runtime execution patterns

- The adversary can draw several conclusions:
 - ① There's a loop involving B_1 and B_5 .
 - ② From B_2 control either goes to B_3 or B_4 , and from B_3 and B_4 we always proceed to $B_5 \Rightarrow$ if-then-else-statement!
 - ③ Reconstruct the control-flow graph!
 - ④ Pattern-match to see that this is the modular exponentiation routine!
- He still doesn't know what's inside the blocks.
- Examine the trace:
 - A branch $B_2 \rightarrow B_3 \Rightarrow$ a 0!
 - A branch $B_2 \rightarrow B_4 \Rightarrow$ a 1!
 - (or possibly the opposite).

Side-channel attacks

- This is a form of a **side-channel attack**.
- Variants to distinguish between B_3 and B_4 :
 - use execution time,
 - use energy consumption.
- Noisy data: run multiple experiments.

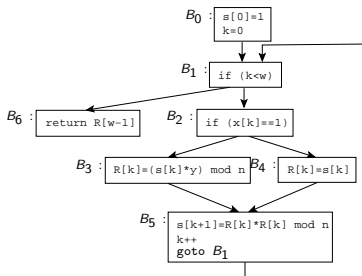
Side-channel attacks. . .

- Our own measurements are watching addresses go by on the address bus.
- These can be noisy too.
- For example, if a routine is small enough to at least partially fit in on-chip caches, then some branches *will not* be exposed!
- The adversary can turn off caching altogether.
- On the Intel X86: set bit 30 of status register CR0.
- So, we can assume that all branches are exposed.

Fixing a leaky address bus!

- Idea: blocks must be constantly permuted in memory!
- CPU has a **shuffle buffer** that keeps some memory blocks.
- When a block is needed from memory it's swapped with a random block from the shuffle buffer.
- A block is a cache line.
- In other words:
 - 1 a block M is read from memory,
 - 2 a block S is selected randomly from the shuffle buffer,
 - 3 M replaces S in the buffer,
 - 4 S is written back to M 's location.

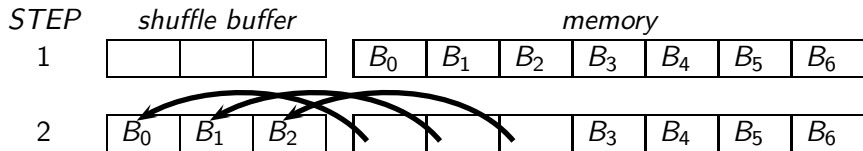
Shuffle Buffer Example



- 7 blocks from our example program, all residing in memory.
- The CPU has a 3-slot shuffle buffer.

Shuffle Buffer Example. . .

First, three blocks are selected from memory and brought in to populate the shuffle buffer.



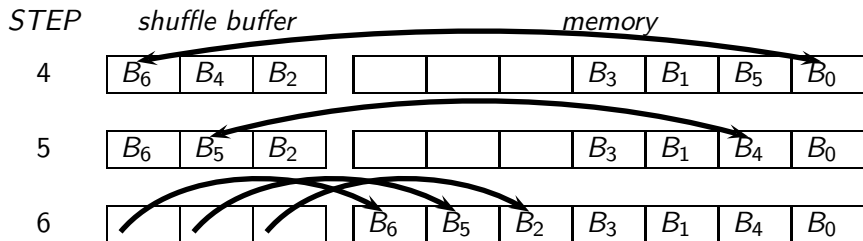
Shuffle Buffer Example. . .

Next, B_4 is needed so it's brought in from memory, replaces block B_1 which was selected randomly from the shuffle buffer, and B_1 is written back to memory, in B_4 's place:

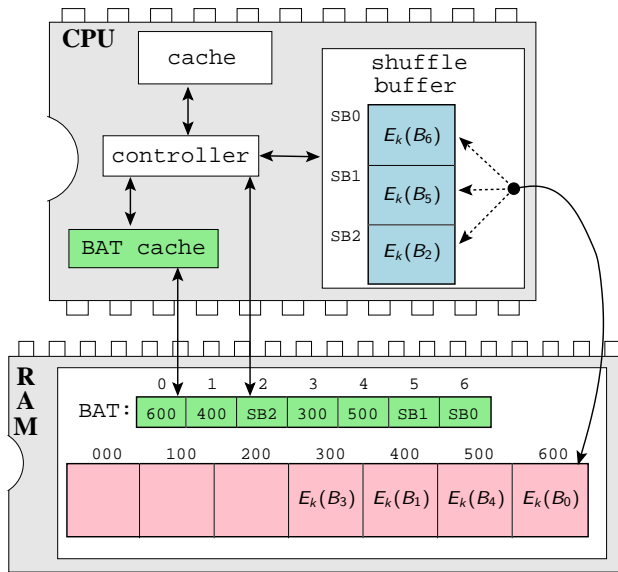


Shuffle Buffer Example. . .

Next, B_0 is swapped with B_6 , B_4 with B_5 , and, finally, as the program is finishing executing, the blocks in the buffer are written back to memory:



Shuffle Buffer Architecture



Shuffle Buffer Architecture. . .

- The *Block Address Table* (BAT) keeps track of where blocks currently reside (address in memory or shuffle buffer index).
- On a cache miss, the controller queries the BAT for the current location of the block. If the block is in the shuffle buffer already, it's returned to the cache. If, on the other hand, it's in memory, it's loaded and stored in both the cache and the shuffle buffer.
- Whatever block was evicted from the buffer gets written back to the location of the loaded block.
- An on-chip *BAT cache* reduces the latency of checking the BAT.

Attack: Watching the data bus!

- After a while our adversary has noticed that blocks are being continuously relocated.
- So, he gets tired of watching the address bus.
- Then can take to watching the **data bus** instead!

Example

- Assume these cipher-texts:

000	100	200	300	400	500	600
$E_k(B_0) =$ 0000	$E_k(B_1) =$ 1000	$E_k(B_2) =$ 2000	$E_k(B_3) =$ 3000	$E_k(B_4) =$ 4000	$E_k(B_5) =$ 5000	$E_k(B_6) =$ 6000

- Watch the cipher-texts going past on the data bus!

\langle 0000,
1000, 2000, 3000, 5000,
1000, 2000, 3000, 5000,
1000, 2000, 4000, 5000,
1000, 2000, 3000, 5000,
...
1000,
6000 \rangle

Fixing a leaky data bus

- Watching the blocks go by on the data bus reveals as much as watching addresses go by on the address bus!
- Easy to fix:
 - When a block is written back to a new location make sure that it has a different ciphertext.
 - \Rightarrow xor the cleartext block with its new address prior to encrypting it.

The XOM architecture — Discussion

- If you really want to protect your program you have to make sure that you hide *everything* that's going on inside the CPU and you have to protect *every* piece of code and data that gets stored off-chip.
- You cannot
 - ① leak any information in the address stream,
 - ② leak any information on the data bus,
 - ③ can't let the adversary change or replay a single bit in memory without you detecting it.
- Performance. . .
- **system-on-a-chip**: If the CPU and RAM both reside in the same physical capsule then there's no need to worry about anyone snooping on the bus.

Outline

- 1 Introduction
 - Processes
 - Inter-Process Communication
 - Memory management
 - Virtual machines
- 2 Process Security
- 3 Authenticated boot
 - Trusted boot
 - Taking measurements
 - The TPM
 - The challenge
 - Privacy issues
 - Applications and Controversies
- 4 Pioneer
- 5 XOM
 - The XOM architecture
 - Preventing replay attacks
 - Fixing a leaky address bus
 - Fixing a leaky data bus
 - Discussion
- 6 **Filesystem Security**
 - SetUID
 - File Descriptors
 - Symbolic Links
- 7 Summary

Virtual Memory Security

- On some systems, a **swap file** holds all virtual memory pages.
- Attack:
 - 1 Abruptly power down the computer;
 - 2 Re-boot with a LiveCD;
 - 3 Search swap file for passwords, keys, etc.
- Defense: Don't store passwords in cleartext in memory.

- See the book!

- To change password:
 - ① run the `passwd` program;
 - ② change `/etc/passwd` file.
- But, processes inherit the permissions of their parents, thus:
 - ① `passwd` runs with the user's privileges!
 - ② `/etc/passwd` is owned by root!

- Unix permissions include a **setuid** bit.
- If prog has `setuid=1` \Rightarrow
 - prog runs with **effective uid** (euid) of its owner.
- Example:
 - 1 `passwd` is owned by root.

- Unix permissions include a **setuid** bit.
- If prog has `setuid=1` \Rightarrow
 - prog runs with **effective uid** (euid) of its owner.
- Example:
 - 1 passwd is owned by root.
 - 2 passwd has `setuid=1`.

- Unix permissions include a **setuid** bit.
- If prog has `setuid=1` \Rightarrow
 - prog runs with **effective uid** (euid) of its owner.
- Example:
 - 1 passwd is owned by root.
 - 2 passwd has `setuid=1`.
 - 3 Bob runs passwd.

- Unix permissions include a **setuid** bit.
- If prog has `setuid=1` \Rightarrow
 - prog runs with **effective uid** (euid) of its owner.
- Example:
 - 1 passwd is owned by root.
 - 2 passwd has `setuid=1`.
 - 3 Bob runs passwd.
 - 4 Bob's passwd process runs with root permissions.

- Unix permissions include a **setuid** bit.
- If prog has setuid=1 \Rightarrow
 - prog runs with **effective uid** (euid) of its owner.
- Example:
 - 1 passwd is owned by root.
 - 2 passwd has setuid=1.
 - 3 Bob runs passwd.
 - 4 Bob's passwd process runs with root permissions.
 - 5 \Rightarrow Bob can change /etc/passwd!

- Attack:
 - ① Bob makes `setuid` program run arbitrary code.
 - ② \Rightarrow Bob gets privileges of the program's owner!
- **Privilege escalation** scenario.
- `setuid` programs must be safe!

SetUID: Example

```
static uid_t  euid , uid ;

int main(int argc , char* argv []) {
    uid = getuid ();
    euid = geteuid ();
    seteuid(uid); // Drop privileges
    // Do something ...
    seteuid(euid); // Raise privileges
    FILE *file = fopen("/var/log" ,"a" );
    seteuid(uid); // Drop privileges
    fprintf(file ," ..." ); // print using permissions
    fclose(file );
    return 0;
}
```

SetUID: Example. . .

- The example program runs with the user's permissions, most of the time.
- It raises permissions to the owner's in order to write to the log file.

SetUID: Vulnerability

```
int main() {  
    system("ls");  
    return 0;  
}
```

- Assume this program has `setuid=1`.
- `system()` invokes `/bin/sh`.
- The shell is told to execute `"ls"`.
- But, the shell uses the `PATH` variable to look up `"ls"`!
- \Rightarrow The user can manipulate `PATH` to make `system` execute the wrong program.
- Defense: `system("/usr/bin/ls")`.

File descriptors

- `open("filename", read/write/append/...)`:
 - ① kernel checks if the process has the right permissions;
 - ② kernel opens the file;
 - ③ kernel returns a **file descriptor**.
- `write(file_descriptor, "...")`:
 - ① kernel checks that the `file_descriptor` has *write* permissions.
 - ② kernel writes, or returns *error*.

File descriptors. . .

- NOTE: The OS only checks *read/write/...* permissions on *open!*
- NOTE: On *read()/write()* calls, the OS only checks the file descriptor was opened with those permissions!
- NOTE: File descriptors can be passed between processes!
- NOTE: Child processes inherit open file descriptors from parents!

File descriptor leaks

```
int main(int argc, char *argv[]) {  
    FILE *pw = fopen("/etc/passwd", "r");  
    // Read passwords...  
    // Oops, forgot to close pw!  
    execl("/home/bob/shell", "shell", NULL);  
}
```

- Bob's child process inherits open file descriptors.
- He can use `fcntl()` functions to access the open file.

Symbolic links

- `ln -s source-file target-file:`
 - A symbolic link points to the original copy.
- `open("filename", read/write/append/..., flags):`
 - `O_NOFOLLOW`: do not follow symlinks
 - `O_SYMLINK`: allow open of symlinks

```
int main(int argc, char *argv[]) {
    if (strcmp(argv[1], "/etc/passwd")==0)
        abort();
    else {
        FILE *pw = fopen(argv[1], "r");
        ...
    }
}
```

- The attacker could pass a symlink to `/etc/passwd` instead...

Outline

- 1 Introduction
 - Processes
 - Inter-Process Communication
 - Memory management
 - Virtual machines
- 2 Process Security
- 3 Authenticated boot
 - Trusted boot
 - Taking measurements
 - The TPM
 - The challenge
 - Privacy issues
 - Applications and Controversies
- 4 Pioneer
- 5 XOM
 - The XOM architecture
 - Preventing replay attacks
 - Fixing a leaky address bus
 - Fixing a leaky data bus
 - Discussion
- 6 Filesystem Security
 - SetUID
 - File Descriptors
 - Symbolic Links
- 7 Summary

Readings and References

- Chapter 3 in *Introduction to Computer Security*, by Goodrich and Tamassia.
- Section 9.7.3 in *Introduction to Computer Security*, by Goodrich and Tamassia.

Acknowledgments

Material and exercises have also been collected from these sources:

- ① Christian Collberg, Jasvir Nagra, *Surreptitious Software, Obfuscation, Watermarking, and Tamperproofing for Software Protection*,

<http://www.amazon.com/Surreptitious-Software-Obfuscation-Watermarking-Tamperproofing/dp/0321549252>

- ② Lie, Thekkath, Mitchell, Lincoln, Boneh, Mitchell, Horowitz, *Architectural support for copy, tamper resistant software*, ASPLOS-IX, 2000.
- ③ Zhuang, Zhang, Pande, *HIDE: an infrastructure for efficiently protecting information leakage on the address bus*, ASPLOS-XI, 2004.
- ④ Gomathisankaran, *Architecture Support for 3D Obfuscation*, IEEE Trans. Comput., Vol. 55, No. 5, pp. 497–507, 2006.