

View access control as a matrix

Objects

	File 1	File 2	File 3	...	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
...					
User m	read	write	read	write	read

Subjects

- **Subjects (processes/users) access objects (e.g., files)**
- **Each cell of matrix has allowed permissions**

Two ways to slice the matrix

- **Along columns:**
 - Kernel stores list of who can access object along with object
 - Most systems you've used probably do this
 - Examples: Unix file permissions, Access Control Lists (ACLs)
- **Along rows:**
 - Capability systems do this
 - More on these later...

Outline

- 1 Unix protection
- 2 Unix security holes
- 3 Capability-based protection
- 4 Microarchitectural attacks

Unix continued

- **Directories have permission bits, too**
 - Need write permission on a directory to create or delete a file
 - **E**xecute permission means ability to use pathnames in the directory, separate from **r**ead permission which allows listing
- **Special user `root` (UID 0) has all privileges**
 - E.g., Read/write any file, change owners of files
 - Required for administration (backup, creating new users, etc.)
- **Example:**
 - `drwxr-xr-x 56 root wheel 4096 Apr 4 10:08 /etc`
 - Directory writable only by root, readable by everyone
 - Means non-root users cannot directly delete files in `/etc`

Non-file permissions in Unix

- **Many devices show up in file system**
 - E.g., `/dev/tty1` permissions just like for files
- **Other access controls not represented in file system**
- **E.g., must usually be root to do the following:**
 - Bind any TCP or UDP port number less than 1024
 - Change the current process's user or group ID
 - Mount or unmount most file systems
 - Create device nodes (such as `/dev/tty1`) in the file system
 - Change the owner of a file
 - Set the time-of-day clock; halt or reboot machine

Example: Login runs as root

- **List of Unix users with accounts typically stored in files in `/etc`**
 - Files `passwd`, `group`, and often `shadow` or `master.passwd`
- **For each user, files contain:**
 - Textual username (e.g., “`dm`”, or “`root`”)
 - Numeric user ID, and group ID(s)
 - One-way hash of user’s password: $\{\text{salt}, H(\text{salt}, \text{passwd})\}$
 - Should have tunable difficulty d : $\{d, \text{salt}, H_d(\text{salt}, \text{passwd})\}$
 - Other information, such as user’s full name, login shell, etc.
- `/usr/bin/login` **runs as root**
 - Reads username & password from terminal
 - Looks up username in `/etc/passwd`, etc.
 - Computes $H(\text{salt}, \text{typed password})$ & checks that it matches
 - If matches, sets group ID & user ID corresponding to username
 - Execute user’s shell with `execve` system call

Setuid

- **Some legitimate actions require more privs than UID**
 - E.g., how should users change their passwords?
 - Stored in root-owned `/etc/passwd` & `/etc/shadow` files
- **Solution: Setuid/setgid programs**
 - Run with privileges of file's owner or group
 - Each process has *real* and *effective* UID/GID
 - *real* is user who launched setuid program
 - *effective* is owner/group of file, used in access checks
 - Actual rules and interfaces somewhat complicated [Chen]
- **Shown as “s” in file listings**
 - `-rws--x--x 1 root root 52528 Oct 29 08:54 /bin/passwd`
 - Obviously need to own file to set the setuid bit
 - Need to own file and be in group to set setgid bit

Setuid (continued)

- **Examples**

- `passwd` – changes user's password
- `su` – acquire new user ID (given correct password)
- `sudo` – run one command as root
- `ping` (historically) – uses raw IP sockets to send/receive ICMP

- **Have to be very careful when writing setuid code**

- Attackers can run setuid programs any time (no need to wait for root to run a vulnerable job)
- Attacker controls many aspects of program's environment

- **Example attacks when running a setuid program**

- Change `PATH` or `IFS` if setuid prog calls `system(3)`
- Set maximum file size to zero (if app rebuilds DB)
- Close `fd 2` before running program—may accidentally send error message into protected file

Linux capabilities

- Wireshark needs network access, not ability to delete all files
- Linux subdivides root's privileges into ~ 40 **capabilities**, e.g.:
 - `cap_net_admin` – configure network interfaces (IP address, etc.)
 - `cap_net_raw` – use raw sockets (bypassing UDP/TCP)
 - `cap_sys_boot` – reboot; `cap_sys_time` – adjust system clock
- Usually root gets all, but behavior can be modified by “securebits” (see `prctl(2)`)
- Capabilities *don't* survive `execve` unless bits are set in *both* thread & inode (exception: ambient capabilities)
- “*Effective*” bit in inode acts like `setuid` for capability

```
$ ls -al /usr/bin/dumpcap
-rwxr-xr-- 1 root wireshark 116808 Jan 30 06:23 /usr/bin/dumpcap
$ getcap /usr/bin/dumpcap
/usr/bin/dumpcap cap_dac_override,cap_net_admin,cap_net_raw=eip
[Oops, cap_dac_override ≈ root! needed for USB capture]
```
- See also: `getcap(8)`, `setcap(8)`, `capsh(1)`

Other permissions

- **When can process *A* send a signal to process *B* with *kill*?**
 - Allow if sender and receiver have same effective UID
 - But need ability to kill processes you launch even if `suid`
 - So allow if real UIDs match, as well
 - Can also send `SIGCONT` w/o UID match if in same session
- **Debugger system call *ptrace***
 - Lets one process modify another's memory
 - `Setuid` gives a program more privilege than invoking user
 - So don't let a process `ptrace` a more privileged process
 - E.g., Require sender to match real & effective UID of target
 - Also disable/ignore `setuid` if `ptraced` target calls `exec`
 - Exception: root can `ptrace` anyone

Outline

- 1 Unix protection
- 2 Unix security holes
- 3 Capability-based protection
- 4 Microarchitectural attacks

A security hole

- **Even without root or setuid, attackers can trick root owned processes into doing things...**
- **Example: Want to clear unused files in /tmp**
- **Every night, automatically run this command as root:**

```
find /tmp -atime +3 -exec rm -f -- {} \;
```
- **find identifies files not accessed in 3 days**
 - executes `rm`, replacing `{}` with file name
- `rm -f -- path` **deletes file *path***
 - Note “--” prevents *path* from being parsed as option
- **What’s wrong here?**

An attack

find/rm

```
readdir (“/tmp”) → “badetc”  
lstat (“/tmp/badetc”) → DIRECTORY  
readdir (“/tmp/badetc”) → “passwd”
```

```
unlink (“/tmp/badetc/passwd”)
```

Attacker

```
mkdir (“/tmp/badetc”)  
creat (“/tmp/badetc/passwd”)
```

An attack

find/rm

```
readdir (“/tmp”) → “badetc”  
lstat (“/tmp/badetc”) → DIRECTORY  
readdir (“/tmp/badetc”) → “passwd”  
  
unlink (“/tmp/badetc/passwd”)
```

Attacker

```
mkdir (“/tmp/badetc”)  
creat (“/tmp/badetc/passwd”)  
  
rename (“/tmp/badetc” → “/tmp/x”)  
symlink (“/etc”, “/tmp/badetc”)
```

- Time-of-check-to-time-of-use [TOCTTOU] bug
 - find checks that /tmp/badetc is not symlink
 - But meaning of file name changes before it is used

xterm command

- Provides a terminal window in X-windows
- Used to run with `setuid` root privileges
 - Requires kernel pseudo-terminal (pty) device
 - Required root privs to change ownership of pty to user
 - Also writes protected `utmp/wtmp` files to record users
- Had feature to log terminal session to file

```
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);  
/* ... */
```

- What's wrong here?

xterm command

- Provides a terminal window in X-windows
- Used to run with `setuid` root privileges
 - Requires kernel pseudo-terminal (pty) device
 - Required root privs to change ownership of pty to user
 - Also writes protected `utmp/wtmp` files to record users
- Had feature to log terminal session to file

```
if (access (logfile, W_OK) < 0)
    return ERROR;
```

```
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
/* ... */
```

- `xterm` is root, but shouldn't log to file user can't write
- `access` call avoids dangerous security hole
 - Does permission check with *real*, not *effective* UID

xterm command

- Provides a terminal window in X-windows
- Used to run with `setuid` root privileges
 - Requires kernel pseudo-terminal (pty) device
 - Required root privs to change ownership of pty to user
 - Also writes protected `utmp/wtmp` files to record users
- Had feature to log terminal session to file

```
if (access (logfile, W_OK) < 0)
    return ERROR;
```

```
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
/* ... */
```

- `xterm` is root, but shouldn't log to file user can't write
- `access` call avoids dangerous security hole
 - Does permission check with *real*, not *effective* UID
 - **Wrong: Another TOCTTOU bug**

An attack

xterm

access (“/tmp/log”) → OK

open (“/tmp/log”)

Attacker

creat (“/tmp/log”)

unlink (“/tmp/log”)

symlink (“/tmp/log” → “/etc/passwd”)

- **Attacker changes /tmp/log between check and use**
 - xterm unwittingly overwrites /etc/passwd
 - Another TOCTTOU bug
- **OpenBSD man page: “CAVEATS: access() is a potential security hole and should never be used.”**

Preventing TOCCTOU

- **Use new APIs that are relative to an opened directory fd**
 - openat, renameat, unlinkat, symlinkat, faccessat
 - fchown, fchownat, fchmod, fchmodat, fstat, fstatat
 - O_NOFOLLOW flag to open avoids symbolic links in last component
 - But can still have TOCTTOU problems with hardlinks
- **Lock resources, though most systems only lock files (and locks are typically advisory)**
- **Wrap groups of operations in OS transactions**
 - Microsoft supports for transactions on Windows Vista and newer [CreateTransaction](#), [CommitTransaction](#), [RollbackTransaction](#)
 - A few research projects for POSIX [[Valor](#)] [[TxOS](#)]

SSH configuration files

- **SSH 1.2.12 client ran as root for several reasons:**
 - Needed to bind TCP port under 1024 (privileged operation)
 - Needed to read client private key (for host authentication)
- **Also needed to read & write files owned by user**
 - Read configuration file `~/.ssh/config`
 - Record server keys in `~/.ssh/known_hosts`
- **Software structured to avoid TOCTTOU bugs:**
 - First bind socket & read root-owned secret key file
 - Second drop *all* privileges—set real, & effective UIDs to user
 - Only then access user files
 - Idea: avoid using any user-controlled arguments/files until you have no more privileges than the user
 - What might still have gone wrong?

Trick question: ptrace bug

- **Actually do have more privileges than user!**
 - Bound privileged port and read host private key
- **Dropping privs allows user to “debug” SSH**
 - Depends on OS, but at the time several had *ptrace* implementations that made SSH vulnerable
- **Once in debugger**
 - Could use privileged port to connect anywhere
 - Could read secret host key from memory
 - Could overwrite local user name to get privs of other user
- **The fix: restructure into 3 processes!**
 - Perhaps overkill, but really wanted to avoid problems
- **Today some linux distros restrict ptrace with [Yama](#)**

A Linux security hole

- **Some programs acquire then release privileges**
 - E.g., `su user` is `setuid root`, becomes `user` if password correct
- **Consider the following:**
 - A and B unprivileged processes owned by attacker
 - A ptraces B (works even with Yama, as B could be child of A)
 - A executes “`su user`” to its own identity
 - With effective UID (EUID) 0, `su` asks for password & waits
 - While A’s EUID is 0, B execs `su root`
(B’s exec honors `setuid`—not disabled—since A’s EUID is 0)
 - A types password, gets shell, and is attached to `su root`
 - Can manipulate `su root`’s memory to get root shell

Editorial

- **Previous examples show two limitations of Unix**
- **Many OS security policies *subjective* not *objective***
 - When can you signal/debug process? Re-bind network port?
 - Rules for non-file operations somewhat incoherent
 - Even some file rules weird (creating hard links to files)
 - Lots of complexities when composing these policies
- **Correct code is much harder to write than incorrect**
 - Delete file without traversing symbolic link
 - Read SSH configuration file (requires 3 processes??)
 - Write mailbox owned by user in dir owned by root/mail
- **Don't just blame the application writers**
 - Must also blame the interfaces they program to

Outline

- 1 Unix protection
- 2 Unix security holes
- 3 Capability-based protection
- 4 Microarchitectural attacks

Another security problem [Hardy]

- **Setting: A multi-user time sharing system**
 - This time it's not Unix
- **Wanted Fortran compiler to keep statistics**
 - Modified compiler `/sysx/fort` to record stats in `/sysx/stat`
 - Gave compiler “home files license”—allows writing to anything in `/sysx` (kind of like Unix `setuid`)
- **What's wrong here?**

A confused deputy

- **Attacker could overwrite any files in `/sysx`**
 - System billing records kept in `/sysx/bill` got wiped
 - Probably command like `fort -o /sysx/bill file.f`
- **Is this a bug in the compiler `fort`?**
 - Original implementors did not anticipate extra rights
 - Can't blame them for unchecked output file
- **Compiler is a “confused deputy”**
 - Inherits privileges from invoking user (e.g., read `file.f`)
 - Also inherits privileges from home files license
 - Which source of authority is it serving on any given system call?
 - OS doesn't know if it just sees `open ("/sysx/bill", ...)`

Recall access control matrix

Objects

	File 1	File 2	File 3	...	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
...					
User m	read	write	read	write	read

Subjects

Capabilities

- **Slicing matrix along rows yields capabilities**
 - E.g., For each process, store a list of objects it can access
 - Process explicitly invokes particular capabilities
- **Can help avoid confused deputy problem**
 - E.g., Must give compiler an argument that both specifies the output file and conveys the capability to write the file (think about passing a file descriptor, not a file name)
 - So compiler uses no *ambient authority* to write file
- **Three general approaches to capabilities:**
 - Hardware enforced (Tagged architectures like [M-machine](#))
 - Kernel-enforced ([Hydra](#), [KeyKOS](#))
 - Self-authenticating capabilities (like [Amoeba](#))
- **Good history in [\[Levy\]](#)**

Hydra [Wulf]

- **Machine & programming environment built at CMU in '70s**
- **OS enforced object modularity with capabilities**
 - Could only call object methods with a capability
- **Augmentation let methods manipulate objects**
 - A method executes with the capability list of the object, not the caller
- **Template methods take capabilities from caller**
 - So method can access objects specified by caller

- **Capability system developed in the early 1980s**
 - Inspired many later systems: [EROS](#), [Coyotos](#)
- **Goal: Extreme security, reliability, and availability**
- **Structured as a “nanokernel”**
 - Kernel proper only 20,000 lines of C, 100KB footprint
 - Avoids many problems with traditional kernels
 - Traditional OS interfaces implemented outside the kernel (including binary compatibility with existing OSes)
- **Basic idea: No privileges other than capabilities**
 - Means kernel provides purely *objective* security mechanism
 - As objective as pointers to objects in OO languages
 - In fact, partition system into many processes akin to objects

Unique features of KeyKOS

- **Single-level store**
 - Everything is persistent: memory, processes, ...
 - System periodically checkpoints its entire state
 - After power outage, everything comes back up as it was (may just lose the last few characters you typed)
- **“Stateless” kernel design only caches information**
 - All kernel state reconstructible from persistent data
- **Simplifies kernel and makes it more robust**
 - Kernel never runs out of space in memory allocation
 - No message queues, etc. in kernel
 - Run out of memory? Just checkpoint system

KeyKOS capabilities

- Referred to as “keys” for short
- Types of keys:
 - *devices* – Low-level hardware access
 - *pages* – Persistent page of memory (can be mapped)
 - *nodes* – Container for 16 capabilities
 - *segments* – Pages & segments glued together with nodes
 - *meters* – right to consume CPU time
 - *domains* – a thread context
- **Anyone possessing a key can grant it to others**
 - But creating a key is a privileged operation
 - E.g., requires “prime meter” to divide it into submeters

Capability details

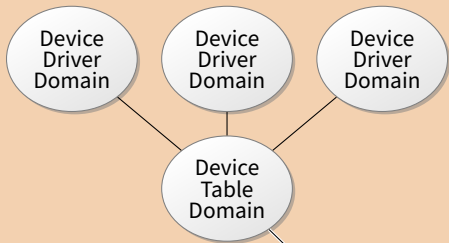
- **Each domain has a number of key “slots”:**
 - 16 general-purpose key slots
 - *address slot* – contains segment with process VM
 - *meter slot* – contains key for CPU time
 - *keeper slot* – contains key for exceptions
- **Segments also have an associated keeper**
 - Process that gets invoked on invalid reference
- **Meter keeper (allows creative scheduling policies)**
- **Calls generate return key for calling domain**
 - (Not required—other forms of message don't do this)

KeyNIX: UNIX on KeyKOS

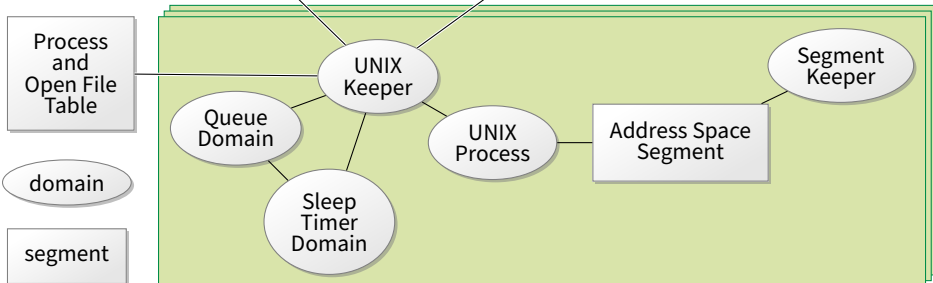
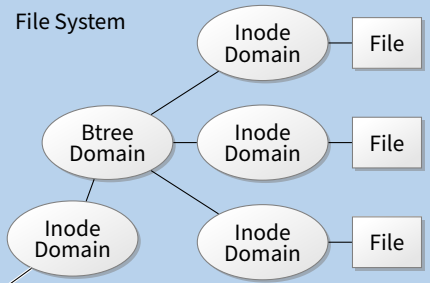
- **“One kernel per process” architecture**
 - Hard to crash kernel
 - Even harder to crash system
- **A process’s kernel is its keeper**
 - Unmodified Unix binary makes Unix syscall
 - Invalid KeyKOS syscall, transfers control to Unix keeper
- **Of course, kernels need to share state**
 - Use shared segment for process and file tables

KeyNIX overview

Device System



File System



- **Every file is a different process**
 - Elegant, and fault isolated
 - Small files can live in a node, not a segment
 - Makes the `namei()` function very expensive
- **Pipes require queues**
 - This turned out to be complicated and inefficient
 - Interaction with signals complicated
- **Other OS features perform very well, though**
 - E.g., `fork` is six times faster than Mach 2.5

Self-authenticating capabilities

- **Every access must be accompanied by a capability**
 - For each object, OS stores random *check* value
 - Capability is: {Object, Rights, MAC(*check*, Rights)}
(MAC = cryptographic *Message Authentication Code*)
- **OS gives processes capabilities**
 - Process creating resource gets full access rights
 - Can ask OS to generate capability with restricted rights
- **Makes sharing very easy in distributed systems**
- **To revoke rights, must change *check* value**
 - Need some way for everyone else to reacquire capabilities
- **Hard to control propagation**

Amoeba

- **A distributed OS, based on capabilities of form:**
 - server port, object ID, rights, check
- **Any server can listen on any machine**
 - Server port is hash of secret
 - Kernel won't let you listen if you don't know secret
- **Many types of object have capabilities**
 - Files, directories, processes, devices, servers (E.g., X windows)
- **Separate file and directory servers**
 - Can implement your own file server, or store other object types in directories, which is cool
- **Check is like a secret password for the object**
 - Server records check value for capabilities with all rights
 - Restricted capability's check is hash of old check, rights

Limitations of capabilities

- **IPC performance a losing battle with CPU makers**
 - CPUs optimized for “common” code, not context switches
 - Capability systems usually involve many IPCs
- **Capability model never fully took off as kernel API**
 - Requires changes throughout application software
 - Call capabilities “file descriptors” or “Java pointers” and people will use them
 - But discipline of pure capability system challenging so far
 - People sometimes quip that capabilities are an OS concept of the future and always will be
- **But real systems do use capabilities**
 - Firefox security based on language-level object capabilities
 - FreeBSD now ships with Capsicum, making capabilities available

Capsicum [Watson]

- **Capability API in FreeBSD 9**
- `cap_enter` **enters a process into capability mode**
 - Can no longer use absolute pathnames, “..”, etc.
- `cap_new` **turns file descriptors into restricted capabilities**
 - ~60 individual permissions can be restricted per capability
 - E.g., disallow `fchmod` (which works on read-only fds)
- **Used by various base system binaries**
- **Supported by a growing number of applications**
- **Patches exist to use Capsicum for Chrome's sandboxing**

Outline

- 1 Unix protection
- 2 Unix security holes
- 3 Capability-based protection
- 4 **Microarchitectural attacks**

Cache timing attacks

```
const char *table;  
  
int  
victim (int secret_byte)  
{  
    return table[secret_byte*64];  
}
```

- **Accessing memory based on secret data can leak the data**
- **Approach 1: Flush/Evict + Reload**
 - Share `table` with victim process (shared lib or deduplication)
 - Flush `table` from cache (`clflush` instruction, or overflow cache)
 - After `victim`, time reads of `table`, fast line tells you `secret_byte`
- **Approach 2: Prime + Probe**
 - No shared memory, but attacker primes cache with its own buffer
 - Victim's `table` access evicts one of attacker's cache lines
 - Slow cache line (+ cache mapping) reveals secret data

Speculative execution key to performance

```
unsigned char *array1, *array2;
int array1_size, array2_size;

int lookup (int input)
{
    if (input < array1_size)
        return array2[array1[input] * 4096];
    return -1;
}
```

- **CPU predicts branches to mask memory latency**
 - E.g., predict `input < array_size` even if `array1_size` not cached
 - Wait to get `array1_size` from memory before retiring instructions
 - *Squash* incorrectly predicted instructions by reverting registers
 - But can't revert cache state, only registers
- **Example: intel Haswell**
 - Specutatively executes up to 192 micro-ops
 - Indexes branch target buffer by bottom 31 bits of branch address

Spectre attack [Kocher]

```
unsigned char *array1, *array2;
int array1_size, array2_size;

int lookup (int input)
{
    if (input < array1_size)
        return array2[array1[input] * 4096];
    return -1;
}
```

- **Say attacker supplies** `input`, **wants to read** `array1[input]`
 - `input` can exceed bounds, reference any byte in address space
- **Ensure** `array1` **cached**, **but** `array1_size` **and** `array2` **uncached**
- **Flush+reload attack on** `array2` **now reveals** `array1[input]`
 - CPU will likely predict branch taken (don't usually overflow)
 - Speculatively load from `array2` before seeing `array1_size`
 - Reloaded cache line reveals `array1[input]`

Many more variants of Spectre

- **Attack on JavaScript JIT**
 - Malicious JavaScript reads secrets outside of JavaScript sandbox
- **eBPF compiles packet filters in kernel (e.g., for tcpdump)**
 - Can generate code to reveal arbitrary kernel memory
- **Can even use victim code that's not supposed to be executed**
 - Mistrain branch predictor on indirect branch
 - Speculatively execute arbitrary “spectre gadget” in victim process
 - Same cache impact even if gadget execution entirely squashed
 - Has been used to leak host memory from inside virtual machine
- **Use other speculation channels**
 - E.g., CPU predicts that previous store does not conflict with a load

Mitigation

- **Replace array bounds checks with index masking (used by Chrome)**
 - `return array2[array1[input&0xffff] * 4096]`
 - Limits distance of bounds violation
- **Place JavaScript sandbox in separate address space**
- **XOR pointers with type-dependent poison values (in JITs)**
 - Branch mispredictions on type checks XOR wrong values
- **Make CPUs a bit better about leaking state through side channels**
- **Insert “gratuitous” memory barriers to prevent speculation on sensitive data**
- **Unfortunately general solution still an open problem**