# A few introductory remarks

- **Most courses teach things we know how to do**
  - E.g., build an operating system, network, VLSI chip, etc.
- **But we can't teach you how to achieve security**
  - Security is a *property* of systems, algorithms
  - Worse yet, security is a *negative property*—the absence of attacks
- **In fact, computer security is largely an open problem**
  - Very few systems have adequate security
  - Really secure systems tend not to see widespread use
- **But we do hope to achieve at least 2 things w. CS155**
  - Give you and arsenal of security techniques you can use
  - Help you achieve a security "mindset"
    (by developing your intuition of where things go wrong)

# CS155 Goals

- **Developing an arsenal of techniques**
  - Learn about prevalent mechanisms and techniques
  - Also look at more esoteric systems with good ideas
- **Developing a security mindset**
  - Vulnerabilities often arise in unexpected places
  - Can concentrate on better door, but attacker will use window
  - Learn to be suspicious of any reasoning
- **My lectures intentionally contain false statements!**
  - Don't fall asleep or tune out during lecture
  - Try to find the flaws in what I'm saying and point them out
  - We learn the most from our mistakes

# View access control as a matrix

Objects

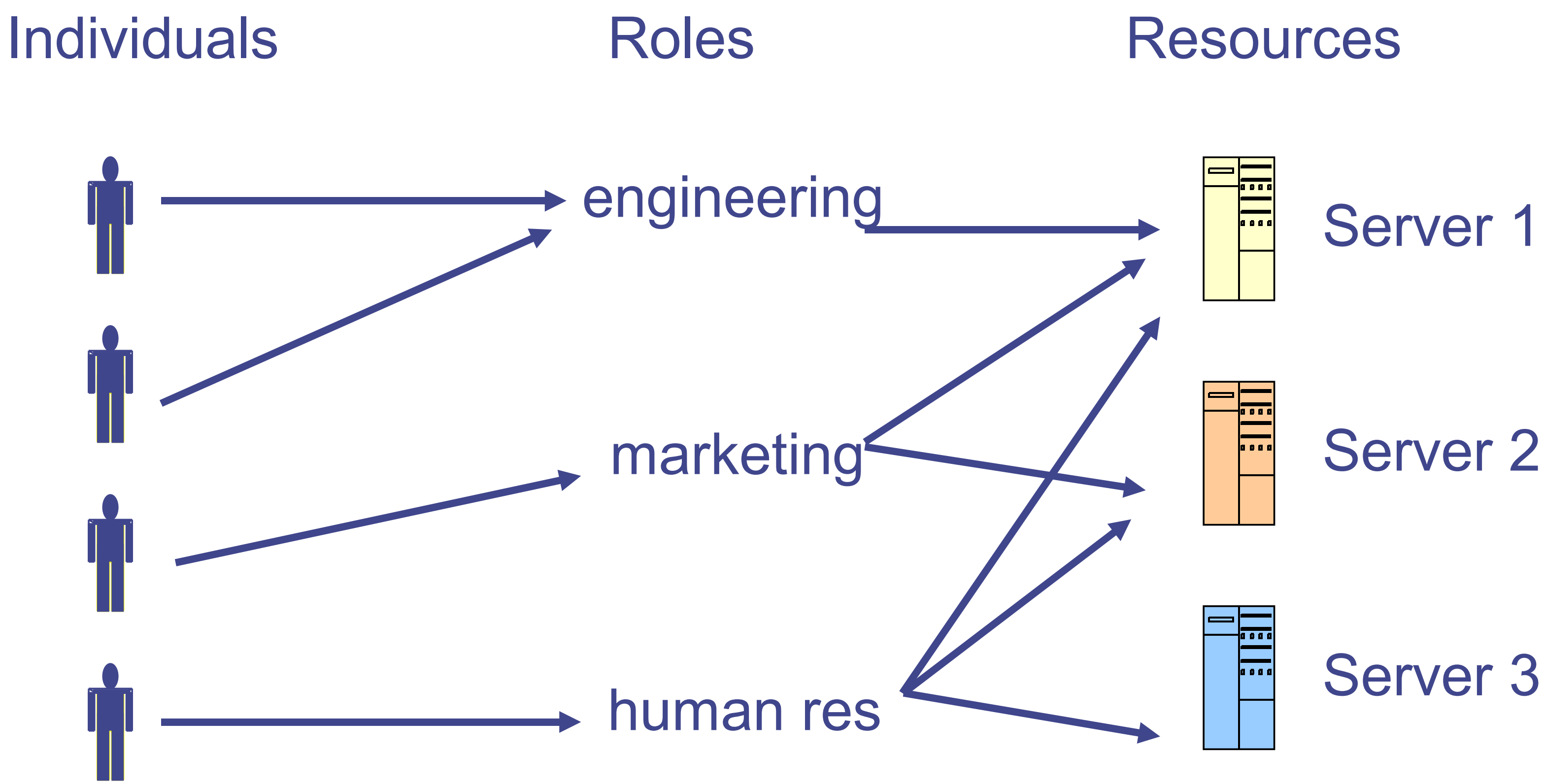
Subjects

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

# Specifying policy

- **Manually filling out matrix would be tedious**
- **Use tools such as groups or *role-based access control*:**



# 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…

# Example: Unix protection

- **Each process has a User ID & one or more group IDs**
- **System stores with each file:**
  - User who owns the file and group file is in
  - Permissions for user, any one in file group, and other
- **Shown by output of `ls -l` command:**

```
  user  group other owner group
- rwx   r-x   r-x   dm    cs155  ...  index.html
```

  - User permissions apply to processes with same user ID
  - Else, group permissions apply to processes in same group
  - Else, other permissions apply

# Unix continued

- **Directories have permission bits, too**
  - Need write perm. on directory to create or delete a file
- **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 can never delete files in `/etc`

# Unix continued

- **Directories have permission bits, too**
  - Need write perm. on directory to create or delete a file
- **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 can never delete files in `/etc`

    **Wrong:** Just need to convince root-owned process to do it

# Clearing old files in `/tmp`

- **Root deletes unused files in `/tmp` nightly**

  ```
  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** | **Attacker** |
| --- | --- |
| | creat ("/tmp/etc/passwd") |
| readdir ("/tmp") → "etc" | |
| lstat ("/tmp/etc") → DIRECTORY | |
| readdir ("/tmp/etc") → "passwd" | |
| | rename ("/tmp/etc" → "/tmp/x") |
| | symlink ("/etc", "/tmp/etc") |
| unlink ("/tmp/etc/passwd") | |

- **Time-of-check-to-time-of-use (TOCTTOU) bug**
  - find checks that `/tmp/etc` is not symlink
  - But meaning of file name changes before it is used

# Problem exacerbated by 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
- **Have to be very careful when writing setuid code**
  - Attackers can run setuid programs any time (no need to wait for once a day find job of last example)
  - Attacker controls many aspects of program's environment

# 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);
/* ... */
```

# 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);
/* ... */
```

- `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);
/* ... */
```

- `access` **call avoids dangerous security hole**

  **Wrong: Another TOCTTOU bug**

# An attack

| xterm | Attacker |
|---|---|
| | creat ("/tmp/X") |
| access ("/tmp/X") → OK | |
| | unlink ("/tmp/X") |
| | symlink ("/tmp/X" → "/etc/passwd") |
| open ("/tmp/X") | |

- **Attacker changes `/tmp/X` between check and use**
  - xterm unwittingly overwrites `/etc/passwd`
  - Time-of-check-to-time-of-use (TOCTTOU) bug
- **OpenBSD man page: "CAVEATS: access() is a potential security hole and should never be used."**

# SSH configuration files

- **SSH 1.2.12 – secure login program, runs as root**
  - Needs to bind TCP port under 1,024 (privileged operation)
  - Needs to read client private key (for host authentication)
- **Also needs to read & write files owned by user**
  - Read configuration file `~/.ssh/config`
  - Record server keys in `~/.ssh/known_hosts`
- **Author wanted to avoid TOCTTOU bugs:**
  - First binds socket & reads root-owned secret key file
  - Then drops all privileges before accessing user files
  - Idea: avoid using any user-controlled arguments/files until you have no more privileges than the user

# Trick question: ptrace bug

- **Dropping privs allows user to "debug" SSH**
  - Depends on OS, but at the time several were vulerable
- **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

# Non-file permissions

- **When can you send a process a signals?**
  - Need to kill processes you started, so should allow if real UIDs match, even if effective don't
  - But should restrict to certain signals (e.g., SIGALARM might mean something to application)
- **What about Ptrace (debugger system call)**
  - Ptrace lets one process modify another's memory
  - Setuid gives a program more privilege than invoking user
  - Don't let process ptrace more privileged process
  - But also must disable setuid if execing process ptraced

# A linux security hole

- **Some programs acquire then release privileges**
  - E.g., `su user` setuid, becomes user if password correct
- **Consider the following:**
  - A and B unprivileged processes owned by attacker
  - A ptraces B
  - A executes "`su user`" to its own identity
  - While `su` is superuser, B execs `su root`
    (A is superuser, so this is not disabled)
  - 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)
- **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

# 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 compiler bug?**
  - 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 from home files license
  - Which master is it serving on any given system call?
  - OS doesn't know if it just sees `open ("/sysx/bill", ...)`

# 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)

# Hydra

- **Machine & programing env. built at CMU in ′70s**
- **OS enforced object modularity with capabilities**
  - Could only call object methods with a capability
- **Agumentation 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

# KeyKOS

- **Capability system developed in the early 1980s**
- **Goal: Extreme security, reliability, and availability**
- **Structured as a "nanokernel"**
  - Kernel proper only 20,000 likes 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**
  - Partition system into many processes akin to objects
  - Capabilities like pointers to objects in OO languages

# 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

- **Refered 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

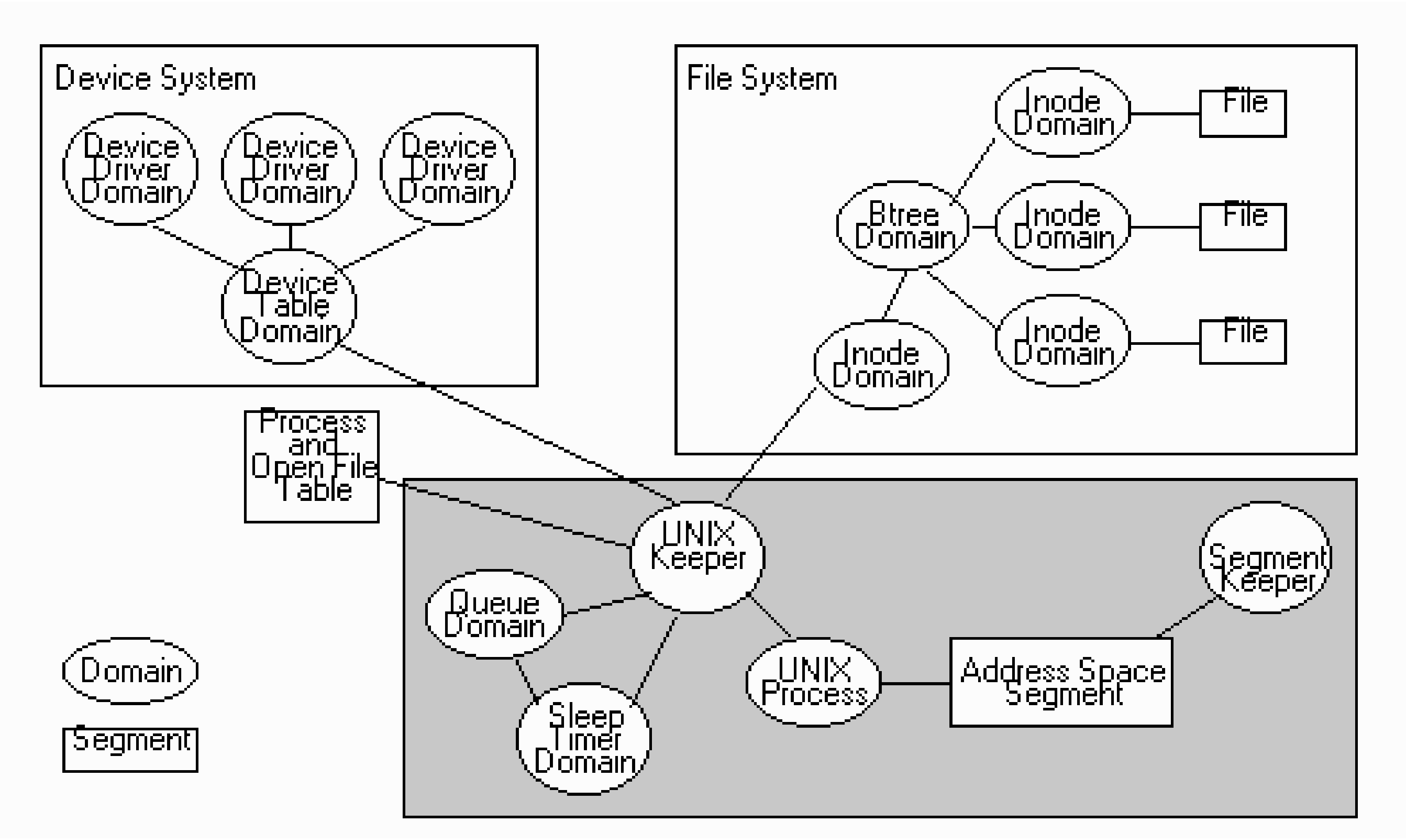
# Capability details

- **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
- **Proc’s kernel is it’s 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



# Keynix I/O

- **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: $\{\text{Object}, \text{Rights}, \textsc{mac}(\textit{check}, \text{Rights})\}$
- **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**

# 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 programming model never took off**
  - 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

# DAC vs. MAC

- **Most people familiar with discretionary access control (DAC)**
  - Unix permission bits are an example
  - Might set a file `private` so only group `friends` can read it
- **Discretionary means anyone with access can propagate information:**
  - `Mail sigint@enemy.gov < private`
- **Mandatory access control**
  - Security administrator can restrict propagation
  - Abbreviated MAC (NOT a message authentication code)

# Bell-Lapadula model

- **View the system as subjects accessing objects**
  - The system input is requests, the output is decisions
  - Objects can be organized in one or more hierarchies, *H* (a tree enforcing the type of decendents)
- **Four modes of access are possible:**
  - execute – no observation or alteration
  - read – observation
  - append – alteration
  - write – both observation and modification
- **The current access set, *b*, is (subj, obj, attr) tripples**
- **An access matrix *M* encodes permissible access types (as before, subjects are rows, objects columns)**

# Security levels

- **A *security level* is a $(c, s)$ pair:**
  - $c =$ classification – E.g., unclassified, secret, top secret
  - $s =$ category-set – E.g., Nuclear, Crypto
- $(c_1, s_1)$ ***dominates*** $(c_2, s_2)$ **iff** $c_1 \geq c_2$ **and** $s_2 \subseteq s_1$
  - $L_1$ *dominates* $L_2$ sometimes written $L_1 \sqsupseteq L_2$ or $L_2 \sqsubseteq L_1$
  - levels then form a lattice
- **Subjects and objects are assigned security levels**
  - level(S), level(O) – security level of subject/object
  - current-level(S) – subject may operate at lower level
  - level(S) bounds current-level(S) (current-level(S) $\sqsubseteq$ level(S))
  - Since level(S) is max, sometimes called S's *clearance*

# Security properties

- **The simple security or *ss-property*:**
  - For any $(S, O, A) \in b$, if $A$ includes observation, then level($S$) must dominate level($O$)
  - E.g., an unclassified user cannot read a top-secret document
- **The star security or **-property*:**
  - If a subject can observe $O_1$ and modify $O_2$, then level($O_2$) dominates level($O_1$)
  - E.g., cannot copy top secret file into secret file
  - More precisely, given $(S, O, A) \in b$:
    if $A = r$ then current-level($S$) $\sqsupseteq$ level($O$) ("no read up")
    if $A = a$ then current-level($S$) $\sqsubseteq$ level($O$) ("no write down")
    if $A = w$ then current-level($S$) $=$ level ($O$)

# Straw man MAC implementation

- **Take an ordinary Unix system**
- **Put labels on all files and directories to track levels**
- **Each user U has a security clearance (level(U))**
- **Determine current security level dynamically**
  - When U logs in, start with lowest curent-level
  - Increase current-level as higher-level files are observed (sometimes called a *floating label* system)
  - If U's level does not dominate current, kill program
  - If program writes to file it doesn't dominate, kill it
- **Is this secure?**

# No: Covert channels

- **System rife with *storage channels***
  - Low current-level process executes another program
  - New program reads sensitive file, gets high current-level
  - High program exploits covert channels to pass data to low
- **E.g., High program inherits file descriptor**
  - Can pass 4-bytes of information to low prog. in file offset
- **Other storage channels:**
  - Exit value, signals, file locks, terminal escape codes, …
- **If we eliminate storage channels, is system secure?**

# No: Timing channels

- **Example: CPU utilization**
  - To send a 0 bit, use 100% of CPU is busy-loop
  - To send a 1 bit, sleep and relinquish CPU
  - Repeat to transfer more bits
- **Example: Resource exhaustion**
  - High prog. allocate all physical memory if bit is 1
  - If low prog. slow from paging, knows less memory available
- **More examples: Disk head position, processor cache/TLB polution, …**

# Reducing covert channels

- **Observation: Covert channels come from sharing**
  - If you have no shared resources, no covert channels
  - Extreme example: Just use two computers
- **Problem: Sharing needed**
  - E.g., read unclassified data when preparing classified
- **Approach: Strict partitioning of resources**
  - Strictly partition and schedule resources between levels
  - Occasionally reapportion resources based on usage
  - Do so infrequently to bound leaked information
  - In general, only hope to bound bandwidth of covert channels
  - Approach still not so good if many security levels possible

# Declassification

- **Sometimes need to prepare unclassified report from classified data**
- **Declassification happens outside of system**
  - Present file to security officer for downgrade
- **Job of declassification often not trivial**
  - E.g., Microsoft word saves a lot of undo information
  - This might be all the secret stuff you cut from document