

Outline

- 1 Mandatory access control
- 2 Labels and lattices
- 3 LOMAC
- 4 SELinux

1 / 43

DAC vs. MAC

- Most people are familiar with *discretionary* access control (DAC)
 - Unix permission bits are an example
 - E.g., might set file `private` so that only group `friends` can read it:
`-rw-r-- 1 dm friends 1254 Feb 11 20:22 private`
 - Anyone with access to information can further propagate that information at his/her discretion:
`$ Mail sigint@enemy.gov < private`
- **Mandatory access control (MAC) can restrict propagation**
 - Security administrator may allow you to read but not disclose file
 - Not to be confused with Message Authentication Codes and Medium Access Control, also both "MAC"

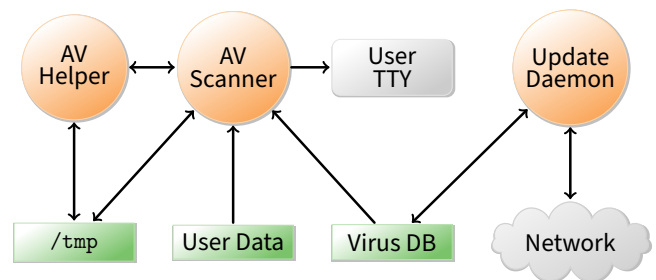
2 / 43

MAC motivation

- Prevent users from disclosing sensitive information (whether accidentally or maliciously)
 - E.g., classified information requires such protection
- Prevent software from surreptitiously leaking data
 - Seemingly innocuous software may steal secrets in the background
 - Such a program is known as a *trojan horse*
- Case study: Symantec AntiVirus 10
 - Contained a remote exploit (attacker could run arbitrary code)
 - Inherently required access to all of a user's files to scan them
 - Can an OS protect private file contents under such circumstances?

3 / 43

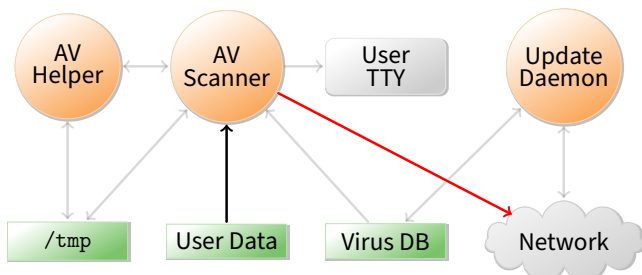
Example: Anti-virus software



- Scanner – checks for virus signatures
- Update daemon – downloads new virus signatures
- How can OS enforce security without trusting AV software?
 - Must not leak contents of your files to network
 - Must not tamper with contents of your files

4 / 43

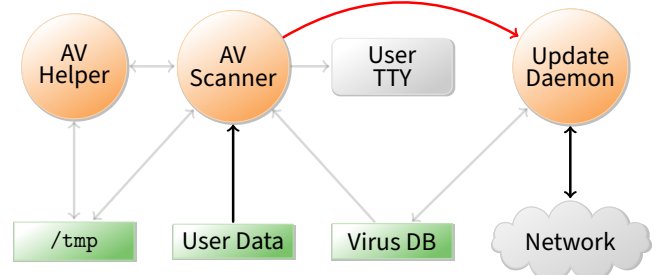
Example: Anti-virus software



- Scanner can write your private data to network
- Prevent scanner from invoking any system call that might send a network messages?

4 / 43

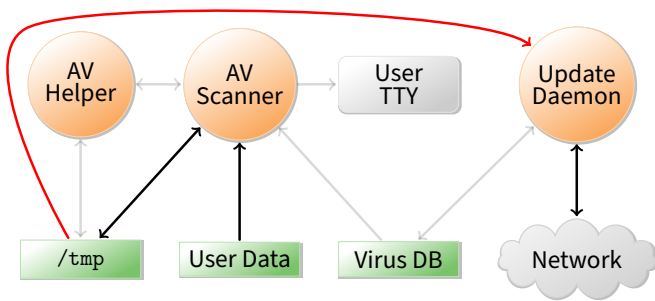
Example: Anti-virus software



- Scanner can send private data to update daemon
- Update daemon sends data over network
 - Can cleverly disguise secrets in order/timing of update requests
- Block IPC & shared memory system calls in scanner?

4 / 43

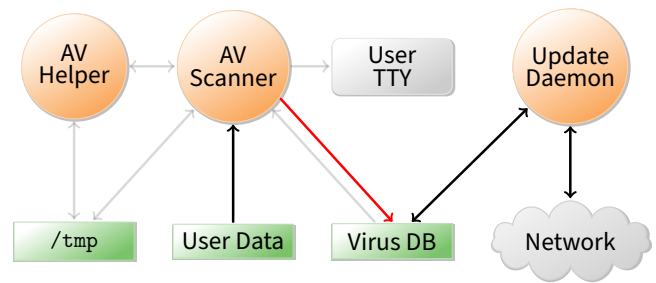
Example: Anti-virus software



- Scanner can write data to world-readable file in `/tmp`
- Update daemon later reads and discloses file
- Prevent update daemon from using `/tmp`?

4/43

Example: Anti-virus software



- Scanner can acquire read locks on virus database
 - Encode secret user data by locking various ranges of file
- Update daemon decodes data by detecting locks
 - Discloses private data over the network
- Have trusted software copy virus DB for scanner?

4/43

The list goes on

- Scanner can call `setproctitle` with user data
 - Update daemon extracts data by running `ps`
- Scanner can bind particular TCP or UDP port numbers
 - Sends no network traffic, but detectable by update daemon
- Scanner can relay data through another process
 - Call `ptrace` to take over process, then write to network
 - Use `sendmail`, `httpd`, or `portmap` to reveal data
- Disclose data by modulating free disk space
- Can we ever convince ourselves we've covered all possible communication channels?
 - Not without a more systematic approach to the problem

5/43

Outline

- 1 Mandatory access control
- 2 Labels and lattices
- 3 LOMAC
- 4 SELinux

6/43

Bell-La Padula model [BL]

- View the system as subjects accessing objects
 - Access control: take *requests* as input and output *decisions*
- Four modes of access are possible:
 - execute – no observation or alteration
 - read – observation
 - append – alteration
 - write – both observation and modification
- An access matrix M encodes permissible access types
 - As in last lecture, subjects are rows, objects are columns
- The current access set, b , is (subj, obj, attr) triples
 - Encodes accesses in progress (e.g., open files)
 - At a minimum, $(S, O, A) \in b$ requires A permitted by cell $M_{S,O}$

7/43

Security levels

- A security level or label is a pair (c, s) where:
 - c = classification – E.g., 1 = unclassified, 2 = secret, 3 = topsecret
 - s = category-set – E.g., Nuclear, Crypto, Russia, ...
- (c_1, s_1) dominates (c_2, s_2) iff $c_1 \geq c_2$ and $s_1 \supseteq s_2$
 - L_1 dominates L_2 is sometimes written $L_1 \times L_2$ or $L_1 \sqsupseteq L_2$
 - Labels then form a lattice (partial order with lub & glb)
- Inverse of dominates relation is *can flow to*, written \sqsubseteq
 - $L_1 \sqsubseteq L_2$ (“ L_1 can flow to L_2 ”) means L_2 dominates L_1
- 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

8/43

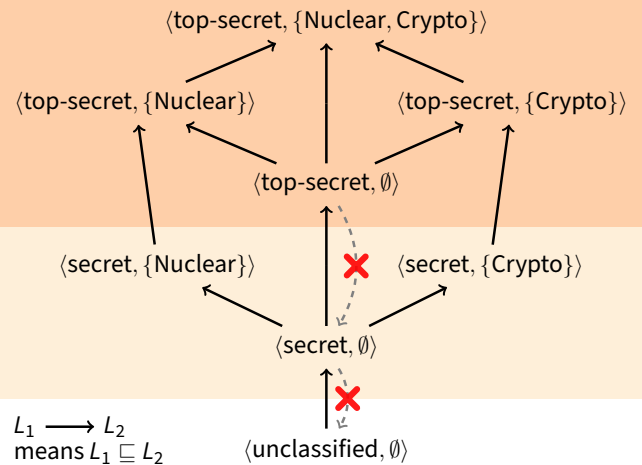
Security properties

Two access control properties with respect to labels:

- **The simple security or *ss-property* (DAC):**
 - For any $(S, O, A) \in b$, if A includes observation, then $\text{level}(S)$ must dominate $\text{level}(O)$, i.e., $\text{level}(O) \sqsubseteq \text{level}(S)$
 - E.g., an unclassified user cannot read a top-secret document
- **The star security or **-property* (MAC):**
 - If any subject both observes O_1 and modifies O_2 , then $\text{level}(O_2)$ dominates $\text{level}(O_1)$, i.e., $\text{level}(O_1) \sqsubseteq \text{level}(O_2)$.
 - E.g., no subject can read a top secret file, then write a secret file
 - More precisely, given $(S, O, A) \in b$:
 - if $A = r$ then $\text{level}(O) \sqsubseteq \text{current-level}(S)$ “no read up”
 - if $A = a$ then $\text{current-level}(S) \sqsubseteq \text{level}(O)$ “no write down”
 - if $A = w$ then $\text{current-level}(S) = \text{level}(O)$

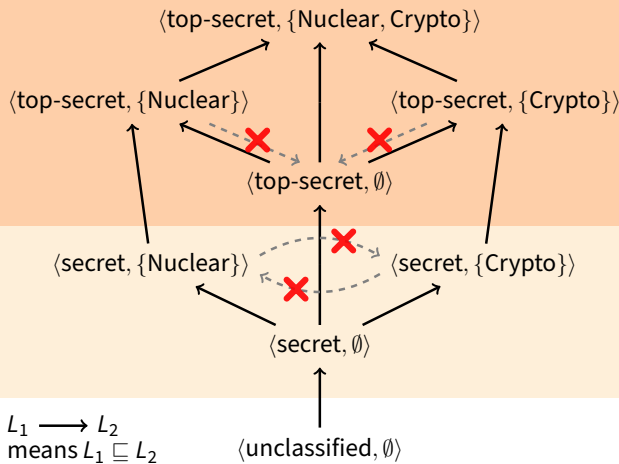
9 / 43

Labels form a lattice [Denning]



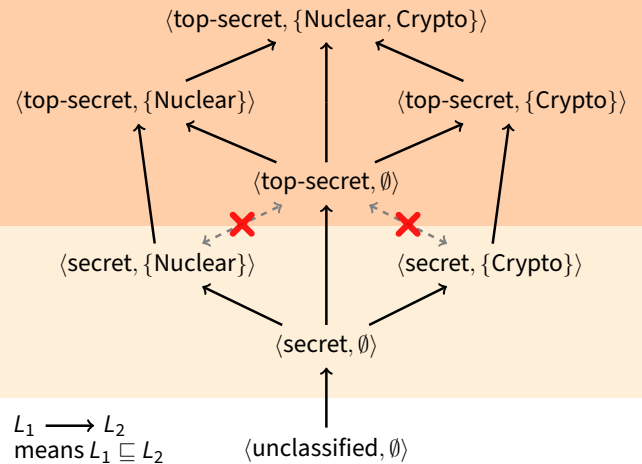
10 / 43

Labels form a lattice [Denning]



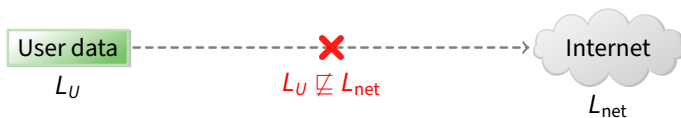
10 / 43

Labels form a lattice [Denning]



10 / 43

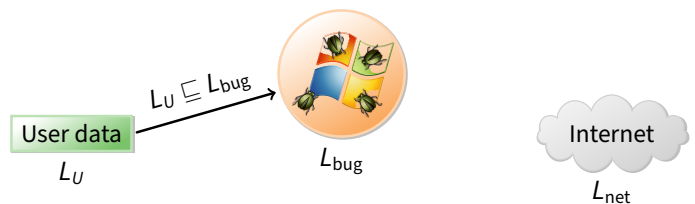
\sqsubseteq is transitive



- Transitivity makes it easier to reason about security
- Example: Label user data so it cannot flow to Internet
 - Policy holds regardless of what other software does

11 / 43

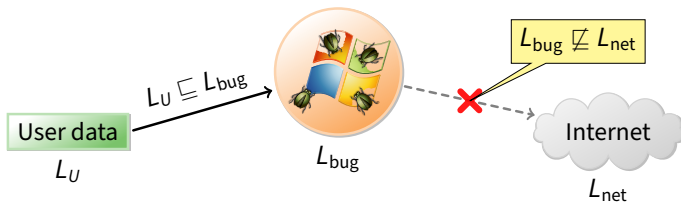
\sqsubseteq is transitive



- Transitivity makes it easier to reason about security
- Example: Label user data so it cannot flow to Internet
 - Policy holds regardless of what other software does
- Suppose untrustworthy software reads file
 - Process labeled L_{bug} reads file, so must have $L_U \sqsubseteq L_{bug}$

11 / 43

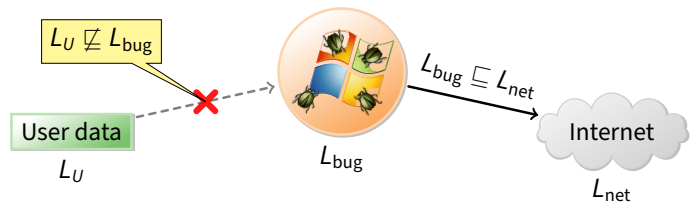
\sqsubseteq is transitive



- **Transitivity makes it easier to reason about security**
- **Example: Label user data so it cannot flow to Internet**
 - Policy holds regardless of what other software does
- **Suppose untrustworthy software reads file**
 - Process labeled L_{bug} reads file, so must have $L_U \sqsubseteq L_{\text{bug}}$
 - If $L_U \sqsubseteq L_{\text{bug}}$ and $L_U \not\sqsubseteq L_{\text{net}}$, it follows that $L_{\text{bug}} \not\sqsubseteq L_{\text{net}}$.

11 / 43

\sqsubseteq is transitive



- **Transitivity makes it easier to reason about security**
- **Example: Label user data so it cannot flow to Internet**
 - Policy holds regardless of what other software does
- **Conversely, a process that can write to the network cannot read the file**

11 / 43

Straw man MAC implementation

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

12 / 43

No: Covert channels

- **System rife with *covert 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 read-only file descriptor**
 - Can pass 4-bytes of information to low program in file offset
- **Other storage channels:**
 - Exit value, signals, file locks, terminal escape codes, ...
- **If we eliminate storage channels, is system secure?**

13 / 43

No: Timing channels

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

14 / 43

Reducing covert channels

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

15 / 43

Declassification

- Sometimes need to prepare unclassified report from classified data
- Declassification happens outside of traditional access control model
 - 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
 - Another bad mistake: Redact PDF using black censor bars over or under text, leaving text selectable (e.g., [Cluley1], [Cluley2])

16 / 43

Biba integrity model [Biba]

- **Problem: How to protect integrity**
 - Suppose text editor gets trojaned, subtly modifies files
 - Might mess up attack plans even without leaking anything
- **Observation: Integrity is the converse of secrecy**
 - In secrecy, want to avoid writing to lower-security files
 - In integrity, want to avoid writing higher-integrity files
- **Use integrity hierarchy parallel to secrecy one**
 - Now *security level* is a $\langle c, i, s \rangle$ triple, where i = integrity
 - $\langle c_1, i_1, s_1 \rangle \sqsubseteq \langle c_2, i_2, s_2 \rangle$ iff $c_1 \leq c_2$ and $i_1 \geq i_2$ and $s_1 \subseteq s_2$
 - Only trusted users can operate at higher integrity (which is visually lower in the lattice—opposite of secrecy)
 - If you read less authentic data, your current integrity level gets lowered (putting you up higher in the lattice), and you can no longer write higher-integrity files

17 / 43

Outline

- 1 Mandatory access control
- 2 Labels and lattices
- 3 LOMAC
- 4 SELinux

18 / 43

LOMAC [Fraser]

- MAC not widely accepted outside military
- LOMAC's goal: make MAC more palatable
 - Stands for Low water Mark Access Control
- Concentrates on Integrity
 - More important goal for many settings
 - E.g., don't want viruses tampering with all your files
 - Also don't have to worry as much about covert channels
- Provides reasonable defaults (minimally obtrusive)
- Has actually had impact
 - Originally available for Linux (2.2)
 - Now ships with FreeBSD
 - Windows introduced similar Mandatory Integrity Control (MIC)

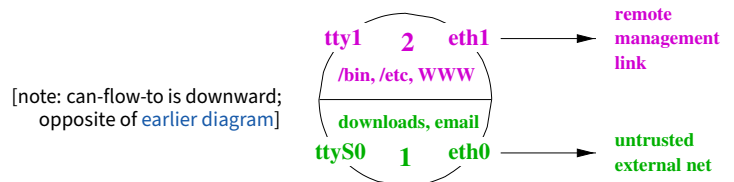
19 / 43

LOMAC overview

- **Subjects are jobs (essentially processes)**
 - Each subject labeled with an integrity number (e.g., 1, 2)
 - Higher numbers mean more integrity (so unfortunately $2 \sqsubseteq 1$ by earlier notation)
 - Subjects can be reclassified on observation of low-integrity data
- **Objects (files, pipes, etc.) also labeled w. integrity level**
 - Object integrity level is fixed and cannot change
- **Security: Low-integrity subjects cannot write to high integrity objects**
- **New objects have level of their creator**

20 / 43

LOMAC defaults



- **Two levels: 1 and 2**
- **Level 2 (high-integrity) contains:**
 - FreeBSD/Linux files intact from distro, static web server config
 - The console, trusted terminals, trusted network
- **Level 1 (low-integrity) contains**
 - NICs connected to Internet, untrusted terminals, etc.
- **Idea: Suppose worm compromises your web server**
 - Worm comes from network → level 1
 - Won't be able to muck with system files or web server config

21 / 43

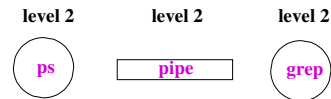
The self-revocation problem

- Want to integrate with Unix unobtrusively
- Problem: Application expectations
 - Kernel access checks usually done at file open time
 - Legacy applications don't pre-declare they will observe low-integrity data
 - An application can "taint" itself unexpectedly, revoking its own permission to access an object it created

22 / 43

Self-revocation example

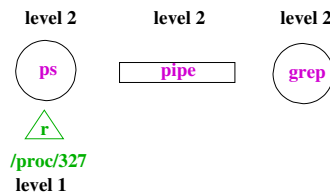
- User has high-integrity (level 2) shell
- Runs: `ps | grep user`
 - Pipe created before `ps` reads low-integrity data
 - `ps` becomes tainted, can no longer write to `grep`



23 / 43

Self-revocation example

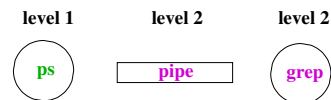
- User has high-integrity (level 2) shell
- Runs: `ps | grep user`
 - Pipe created before `ps` reads low-integrity data
 - `ps` becomes tainted, can no longer write to `grep`



23 / 43

Self-revocation example

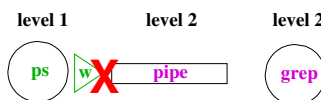
- User has high-integrity (level 2) shell
- Runs: `ps | grep user`
 - Pipe created before `ps` reads low-integrity data
 - `ps` becomes tainted, can no longer write to `grep`



23 / 43

Self-revocation example

- User has high-integrity (level 2) shell
- Runs: `ps | grep user`
 - Pipe created before `ps` reads low-integrity data
 - `ps` becomes tainted, can no longer write to `grep`



23 / 43

Solution

- Don't consider pipes to be real objects
- Join multiple processes together in a "job"
 - Pipe ties processes together in job
 - Any processes tied to job when they read or write to pipe
 - So will lower integrity of both `ps` and `grep`
- Similar idea applies to shared memory and IPC
- Summary: LOMAC applies MAC to non-military systems
 - But doesn't allow military-style security policies (i.e., with secrecy, various categories, etc.)

24 / 43

Outline

- 1 Mandatory access control
- 2 Labels and lattices
- 3 LOMAC
- 4 SELinux

25 / 43

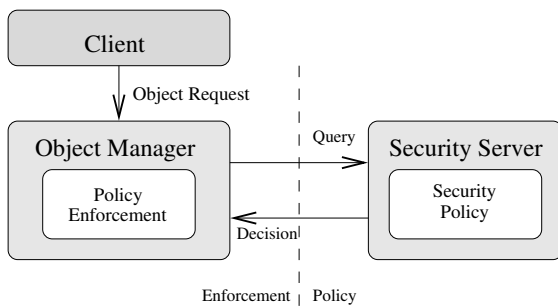
The flask security architecture

- **Problem: Military needs adequate secure systems**
 - How to create civilian demand for systems military can use?
- **Idea: Separate policy from enforcement mechanism**
 - Most people will plug in simple DAC policies
 - Military can take system off-the-shelf, plug in new policy
- **Requires putting adequate hooks in the system**
 - Each object has manager that guards access to the object
 - Conceptually, manager consults security server on each access
- **Flask security architecture prototyped in fluke**
 - Now part of SELinux

Following figures from [Spencer]

26 / 43

Architecture



- Kernel mediates access to objects at “interesting” points
- Kicks decision up to external (user-level) security server

27 / 43

Challenges

- **Performance**
 - Adding hooks on every operation
 - People who don't need security don't want slowdown
- **Using generic enough data structures**
 - Object managers independent of policy still need to associate data structures (e.g., labels) with objects
- **Revocation**
 - May interact in a complicated way with any access caching
 - Once revocation completes, new policy must be in effect
 - Bad guy cannot be allowed to delay revocation completion indefinitely

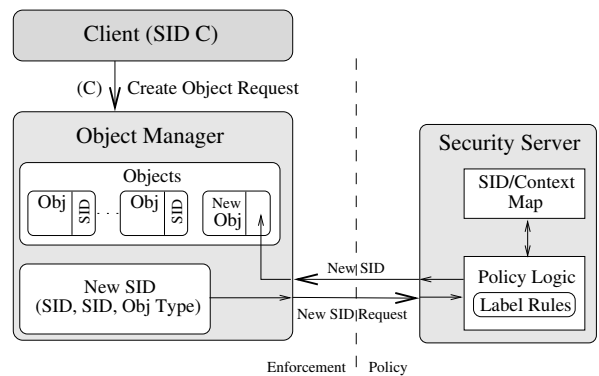
28 / 43

Basic flask concepts

- **All objects are labeled with a security context**
 - Security context is an arbitrary string—opaque to object manager in the kernel
- **Labels abbreviated with security IDs (SIDs)**
 - 32-bit integer, interpretable only by security server
 - Not valid across reboots (can't store in file system)
 - Fixed size makes it easier for object manager to handle
- **Queries to server done in terms of SIDs**
 - Create (client SID, old obj SID, obj type)? → SID
 - Allow (client SID, obj SID, perms)? → {yes, no}

29 / 43

Creating new object



30 / 43

Security server interface [Loscocco]

```
int security_compute_av(
    security_id_t ssid, security_id_t tsid,
    security_class_t tclass, access_vector_t requested,
    access_vector_t *allowed, access_vector_t *decided,
    __u32 *seqno);
```

- `ssid, tsid` – source and target SIDs
- `tclass` – type of target
 - E.g., regular file, device, raw IP socket, TCP socket, ...
- Server can decide more than it is asked for
 - `access_vector_t` is a bitmask of permissions
 - `decided` can contain more than `requested`
 - Effectively implements decision prefetching
- `seqno` used for revocation (in a few slides)

31 / 43

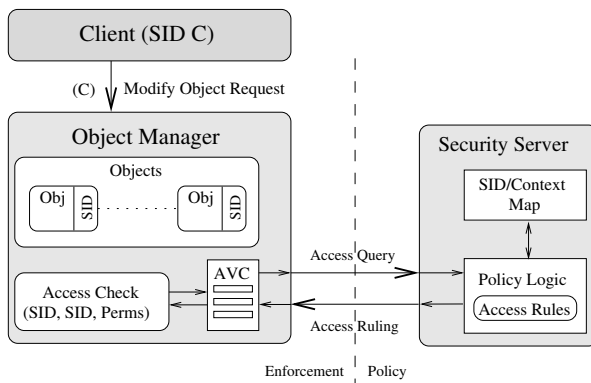
Access vector cache (AVC)

- Want to minimize calls into security server
- AVC caches results of previous decisions
 - Note: Relies on simple enumerated permissions
- Decisions therefore cannot depend on parameters:
 - ✗ Andy can authorize expenses up to \$999.99
 - ✗ Bob can run processes at priority 10 or higher
- Decisions also limited to two SIDs
 - Complicates file relabeling, which requires 3 checks:

Source	Target	Permission checked
Subject SID	Old file SID	Relabel-From
Subject SID	New file SID	Relabel-To
Old file SID	New file SID	Transition-From

32 / 43

AVC in a query



33 / 43

AVC interface

```
int avc_has_perm_ref(
    security_id_t ssid, security_id_t tsid,
    security_class_t tclass, access_vector_t requested,
    avc_entry_ref_t *aeref);
```

- `avc_entry_ref_t` points to cached decision
 - Contains `ssid, tsid, tclass, decision vec., & recently used info`
- `aeref` argument is hint
 - After first call, will be set to relevant AVC entry
 - On subsequent calls speeds up lookup
- Example: New kernel check when binding a socket:


```
ret = avc_has_perm_ref(
    current->sid, sk->sid, sk->sclass,
    SOCKET__BIND, &sk->avcr);
```

 - Now `sk->avcr` is likely to be speed up next socket op

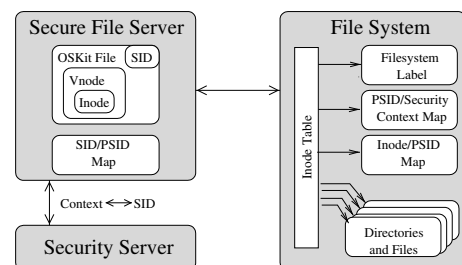
34 / 43

Revocation support

- Decisions may be cached in AVC entries
- Decisions may implicitly be cached in migrated permissions
 - E.g., Unix checks file write permission on `open`
 - But may want to disallow future writes even on open file
 - Write permission migrated into file descriptor
 - May also migrate into page tables/TLB w. `mmap`
 - Also may migrate into open sockets/pipes, or operations in progress
- AVC contains hooks for callbacks
 - After revoking in AVC, AVC makes callbacks to revoke migrated permissions
 - `seqno` can be used to ensure strict ordering of policy changes

35 / 43

Persistence



- Must label persistent objects in file system
 - Persistently map each file/directory to a security context
 - Security contexts are variable length, so add level of indirection
 - “Persistent SIDs” (PSIDs) – numbers local to each file system

36 / 43

Transitioning SIDs

- May need to relabel objects
 - E.g., files in file system
- Processes may also want to transition their SIDs
 - Depends on existing permission, but also on program
 - SELinux allows programs to be defined as *entrypoints*
 - Thus, can restrict with which programs users enter a new SID (similar to the way `setuid` transitions `uid` on program entry)

37 / 43

SELinux contexts

- In practice, SELinux contexts have four parts:

$\overbrace{\text{system_u}}^{\text{user}} : \overbrace{\text{system_r}}^{\text{role}} : \overbrace{\text{sshd_t}}^{\text{type}} : \overbrace{\text{s0}}^{\text{level}}$

- *user* is not Unix user ID, e.g.:

```
$ id
uid=1000(dm) gid=1000(dm) groups=1000(dm) 119(admin)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c255
$ /bin/su
Password:
# id
uid=0(root) gid=0(root) groups=0(root)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c255
# newrole -r system_r -t sysadm_t
Password:
# id -Z
unconfined_u:system_r:sysadm_t:s0-s0:c0.c255
```

38 / 43

Users, roles, types

- SELinux user is assigned on login, based on rules

```
# semanage login -l
Login Name      SELinux User  MLS/MCS Range
__default__    unconfined_u  s0-s0:c0.c255
root           root_u        s0-s0:c0.c255
```

- A user is allowed to assume different roles w. `newrole`
- But roles are restricted by SELinux (not Unix) users

```
# semanage user -l
SELinux User  ... SELinux Roles
root          staff_r sysadm_r system_r
unconfined_u system_r unconfined_r
user_u        user_r
```

39 / 43

Types

- Each role allows only certain *types*
 - Can check with `seinfo -x --role=name`
- Types allow non-hierarchical security policies
 - Each subject is assigned a *domain*, each object a *type*
 - Policy stated in terms of what each domain can do each type
- Example: Suppose you wish to enforce that each invoice undergoes the following processing:
 - Receipt of the invoice recorded by a clerk
 - Receipt of the merchandise verified by purchase officer
 - Payment of invoice approved by supervisor
- Can encode state of invoice by its type
 - Set transition rules to enforce all steps of process

40 / 43

Example: Loading kernel modules

```
(1) allow sysadm_t insmod_exec_t:file x_file_perms;
(2) allow sysadm_t insmod_t:process transition;
(3) allow insmod_t insmod_exec_t:process { entrypoint execute };
(4) allow insmod_t sysadm_t:fd inherit_fd_perms;
(5) allow insmod_t self:capability sys_module;
(6) allow insmod_t sysadm_t:process sigchld;
```

1. Allow `sysadm` domain to run `insmod`
2. Allow `sysadm` domain to transition to `insmod`
3. Allow `insmod` program to be entrypoint for `insmod` domain
4. Let `insmod` inherit file descriptors from `sysadm`
5. Let `insmod` use `CAP_SYS_MODULE` (load a kernel module)
6. Let `insmod` signal `sysadm` with `SIGCHLD` when done

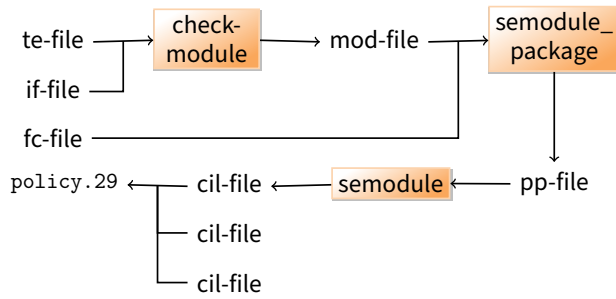
41 / 43

Policy specification

- Very complicated sets of rules
 - E.g., on Fedora, `sesearch --all | wc -l` shows 73K rules
 - Rules based mostly on types
- Allowed/restricted transitions very important
 - E.g., `init` can run `initscripts`, can run `httpd`
 - Nowadays `systemd` needs to be able to transition to arbitrary types
 - `httpd` program has special `httpd_exec_t` type, allows process to have `httpd_t` type.
 - Might label `public_html` directories so `httpd` can access them, but not access rest of home directory
- Can also use levels to enforce MLS
 - E.g., “`:s0-s0:c0.c255`” means process is at sensitivity `s0` with no categories, but has all categories in clearance.

42 / 43

Policy construction



- **Very low quality tooling around policy construction**
 - Broken build systems, incompatible kernel policy formats, ...
- **Hard to check** `/sys/fs/selinux/policy` **matches expectations**
 - No single-pass decompilation, tools seem to hang on real policies
 - Even rebuilding from source is hard (e.g., actual compilation happens during RPM install, using tons of spec macros)