Secure Architecture Principles

- Mandatory access control
- Multi-level security
- SELinux

Most slides are from Prof. Michael Clarkson

Secure Architecture Principles

Mandatory access control

Review: DAC

- Discretionary access control (DAC)
  - Philosophy: users have the discretion to specify policy themselves
  - Commonly, information belongs to the owner of object
  - Model: access control relation
    - Set of triples (subj,obj,rights)
    - Sometimes described as access control "matrix"
- Implementations:
  - Access control lists (ACLs): each object associated with list of (subject, rights)
  - Capabilities: distributed ways of implementing privilege lists

MAC

- Mandatory access control (MAC)
  - not Message Authentication Code (applied crypto), nor Media Access Control (networking)
  - philosophy: central authority mandates policy
  - information belongs to the authority, not to the individual users
- Three case studies:
  1. Multi-level security (military)
  2. Chinese wall (consulting firm)
  3. Clark-Wilson (business)

Secure Architecture Principles

Multi-level security

Sensitivity

- Concern is confidentiality of information
- Documents classified according to sensitivity: risk associated with release of information

- In US:
  - Top Secret
  - Secret
  - Confidential
  - Unclassified

Compartments

- Documents classified according to compartment(s): categories of information (in fact, aka category)
  - Cryptography
  - nuclear
  - biological
  - reconnaissance
- Need to Know Principle:
  - access should be granted only when necessary to perform assigned duties (instance of Least Privilege)
  - {crypto,nuclear}: must need to know about both to access
  - {}: no particular compartments

Labels

- Label: pair of sensitivity level and set of compartments. e.g.,
  - (Top Secret, {crypto, nuclear})
  - (Unclassified, {})
- Users are labeled according to their clearance
- Document is labeled aka classified
  - Perhaps each paragraph labeled
  - Label of document is most restrictive label for any paragraph
- Labels are imposed by organization
- Notation: let L(X) be the label of entity X

Restrictiveness of labels

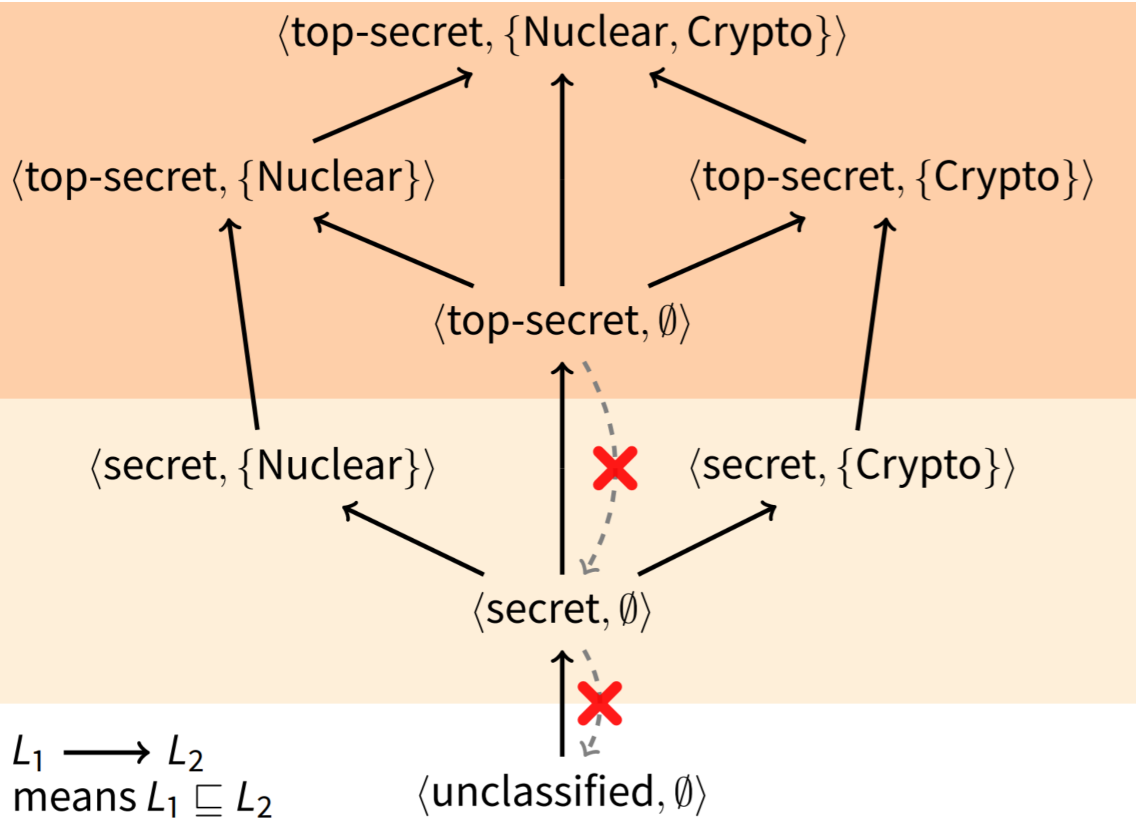
- Notation: L1 $\sqsubseteq$ L2
  - means L1 is no more restrictive than L2
    - less precisely: L1 is less restrictive than L2
    - another reading: information may flow from L1 to L2
    - also: L1 is dominated by L2
- e.g.,
  - (Unclassified,{}) $\sqsubseteq$ (Top Secret, {})
  - (Top Secret, {crypto}) $\sqsubseteq$ (Top Secret, {crypto,nuclear})

Restrictiveness of labels

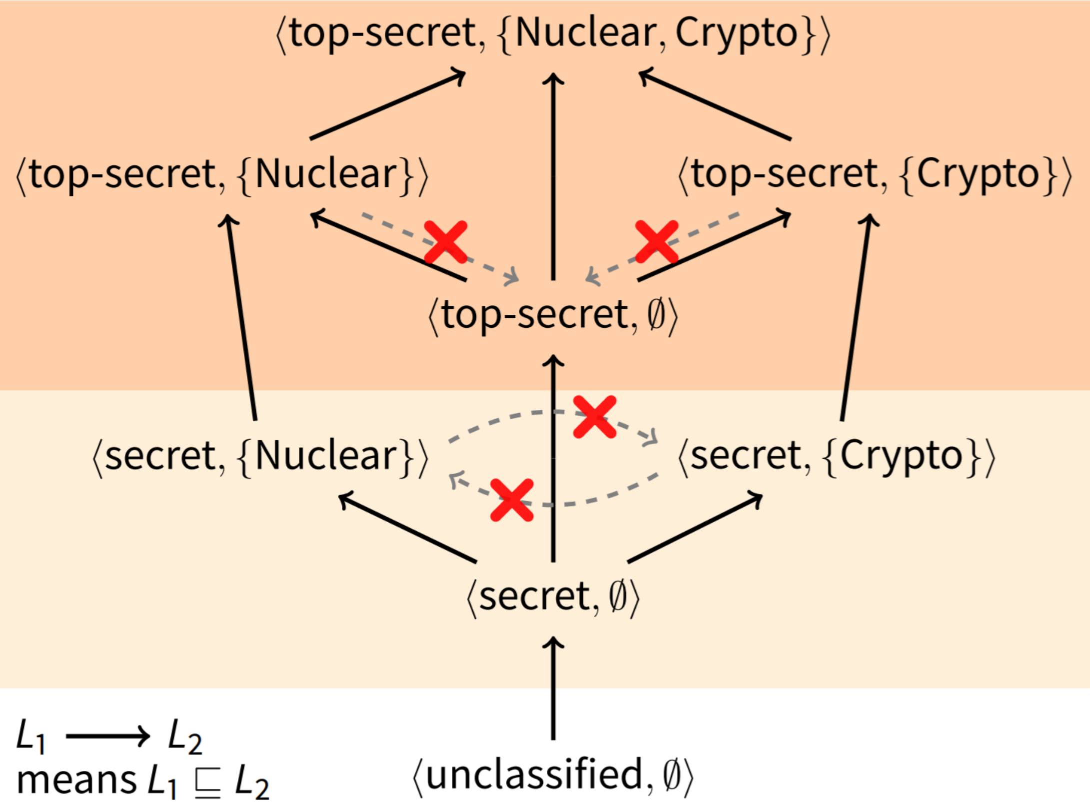
- Definition:
  - Let L1 = (S1, C1) and L2 = (S2, C2)
  - L1 $\sqsubseteq$ L2 iff S1 $\leq$ S2 and C1 $\subseteq$ C2
  - Where $\leq$ is order on sensitivity: Unclassified $\leq$ Confidential $\leq$ Secret $\leq$ Top Secret

- Partial order:
  - Some labels are incomparable
  - e.g. (Secret, {crypto}) vs. (Top Secret, {nuclear})

Labels from a lattice

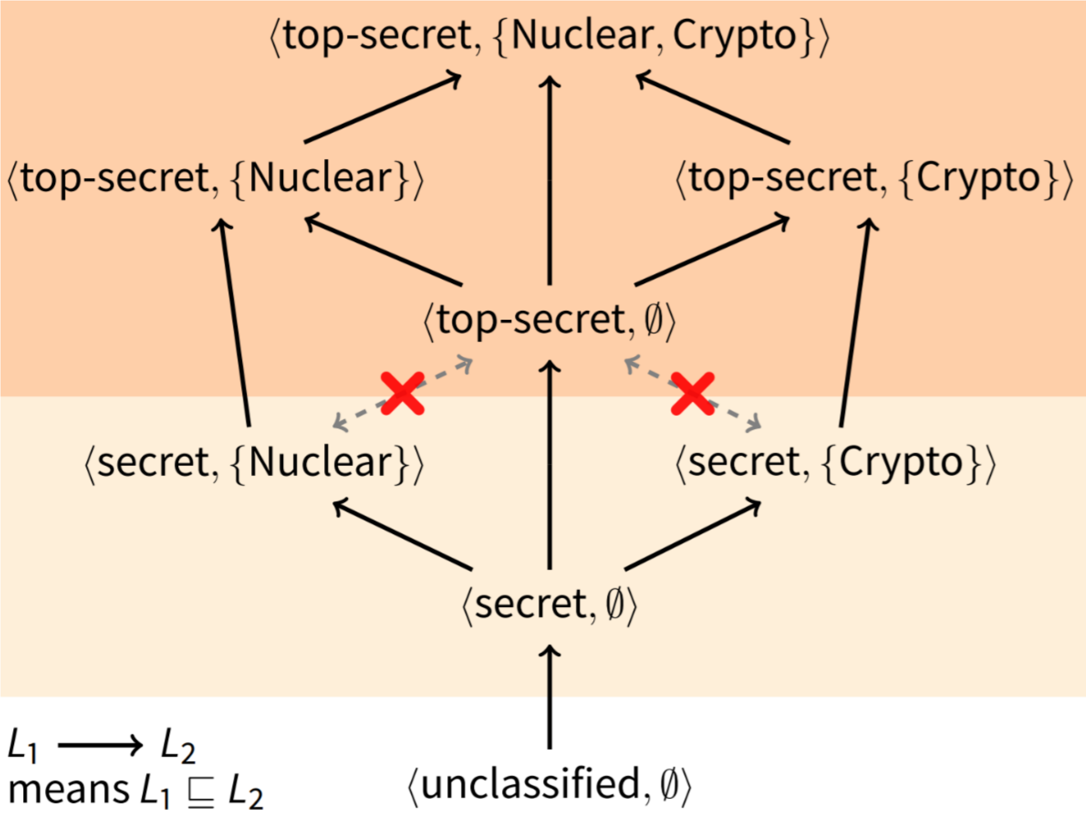


Labels from a lattice



$L_1 \longrightarrow L_2$ means $L_1 \sqsubseteq L_2$

Labels from a lattice



Access control with MLS

- When may a subject read an object? (Confidentiality)
  - S may read O iff $L(O) \sqsubseteq L(S)$
  - object's classification must be below (or equal to) subject's clearance
  - "no read up"
- When may a subject write an object? (Integrity)
  - S may write O iff $L(S) \sqsubseteq L(O)$
  - object's classification must be above (or equal to) subject's clearance
  - "no write down"
- Beautiful **symmetry** between these

Reading with MLS

- Scenario:
  - Colonel with clearance (Secret, {nuclear, Europe})
  - DocA with classification (Confidential, {nuclear})
  - DocB with classification (Secret, {Europe, US})
  - DocC with classification (Top Secret, {nuclear, Europe})
- Which documents may Colonel read?
  - Recall: S may read O iff $L(O) \sqsubseteq L(S)$
  - DocA: (Confidential, {nuclear}) $\sqsubseteq$ (Secret, {nuclear, Europe})
  - DocB: (Secret, {Europe, US}) not $\sqsubseteq$ (Secret, {nuclear, Europe})
  - DocC: (Top Secret, {nuclear, Europe}) not $\sqsubseteq$ (Secret, {nuclear, Europe})

Writing with MLS

- Scenario:
  - Colonel with clearance (Secret, {nuclear, Europe})
  - DocA with classification (Confidential, {nuclear})
  - DocB with classification (Secret, {Europe, US})
  - DocC with classification (Top Secret, {nuclear, Europe})
- Which documents may Colonel write?
  - Recall: S may write O iff L(S) $\sqsubseteq$ L(O)
  - DocA: (Secret, {nuclear, Europe}) not $\sqsubseteq$ (Confidential, {nuclear})
  - DocB: (Secret, {nuclear, Europe}) not $\sqsubseteq$ (Secret, {Europe, US})
  - DocC: (Secret, {nuclear, Europe}) $\sqsubseteq$ (Top Secret, {nuclear, Europe})

Prevention of laundering with MLS

- Laundering Scenario:
  - "subject with clearance Top Secret reads Top Secret information then writes it into an Unclassified file"
- More generally: S reads O1 then writes O2 where $L(O2) \sqsubset L(O1)$ regardless of L(S)

- Can't happen:
  - S read O1, so $L(O1) \sqsubseteq L(S)$
  - S wrote O2, so $L(S) \sqsubseteq L(O2)$
  - So $L(O1) \sqsubseteq L(S) \sqsubseteq L(O2)$
  - Hence $L(O1) \sqsubseteq L(O2)$
  - But combined with $L(O2) \sqsubset L(O1)$, we have $L(O1) \sqsubset L(O1)$
  - Contradiction

Perplexities of writing with MLS

- Blind write: subject may not read higher-security object yet may write it
  - Useful for logging
- Declassification violates the "no write down" rule
  - Unclassified output from Secret information (write down)
    - Encryption (secret input) → unclassified output
  - Traditional solution is trusted subjects who are not constrained by access control rules
    - Could introduces a potential vulnerability

Bell-La Padula model [1973]

- Formal mathematical model of MLS plus access control matrix
- Proof that information cannot leak to subjects not cleared for it
- "No read up": simple security property
- "No write down": *-property

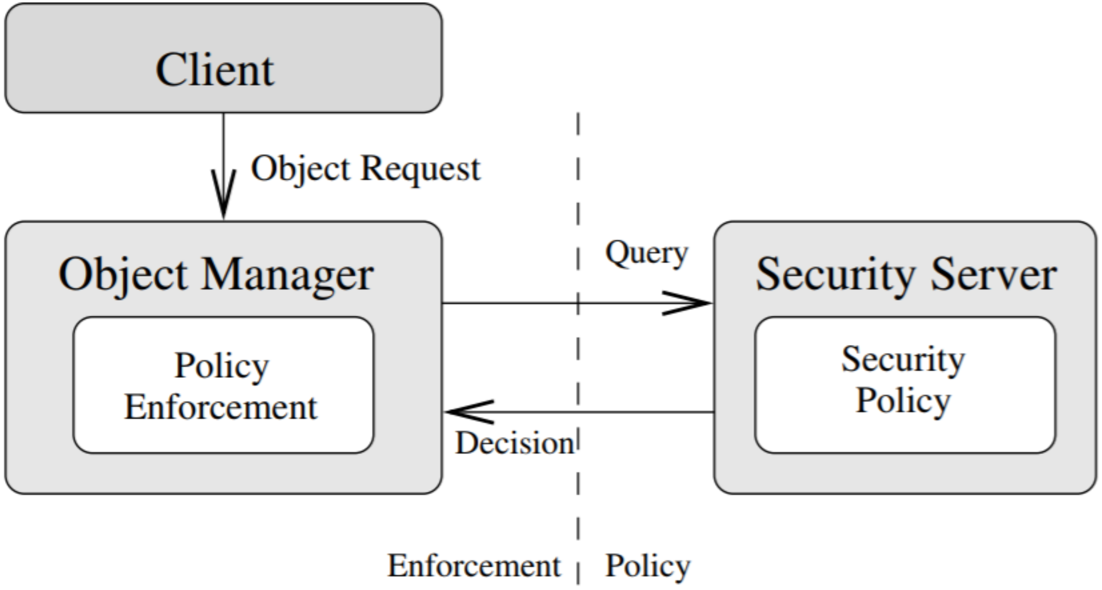
Secure Architecture Principles

SELinux

Flash 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

Architecture



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

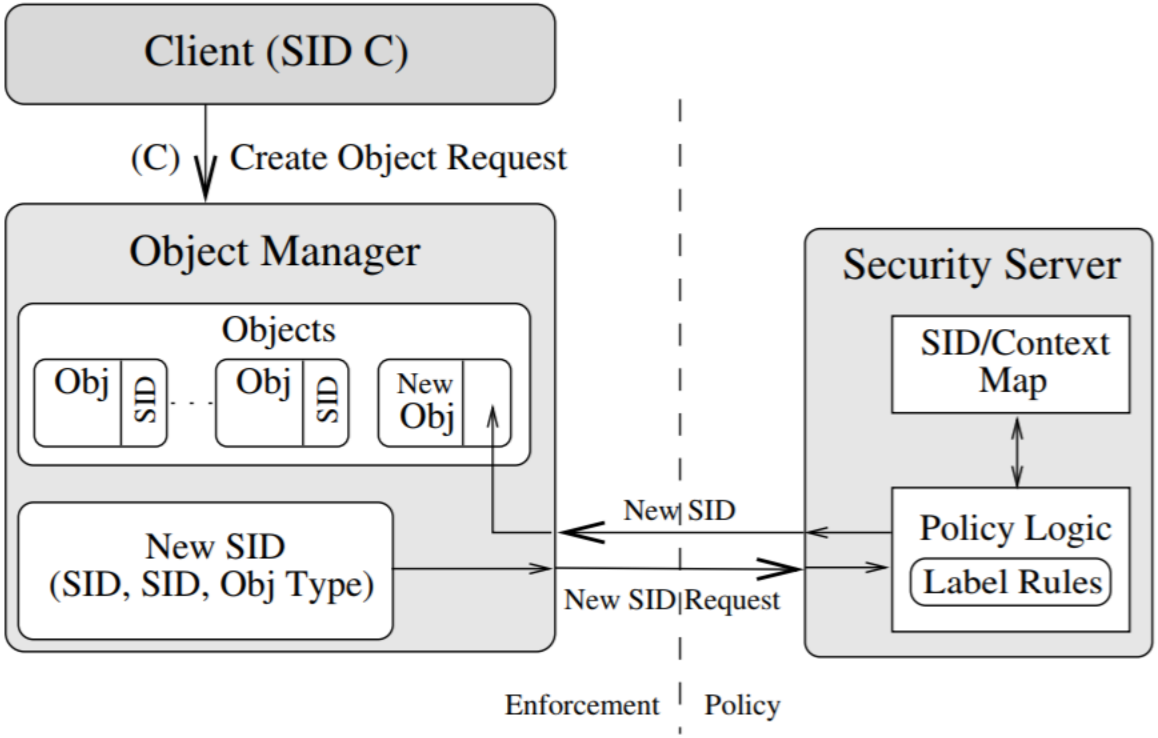
Challenges

- 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

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}

Creating new object



Security server interface

```
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);
```

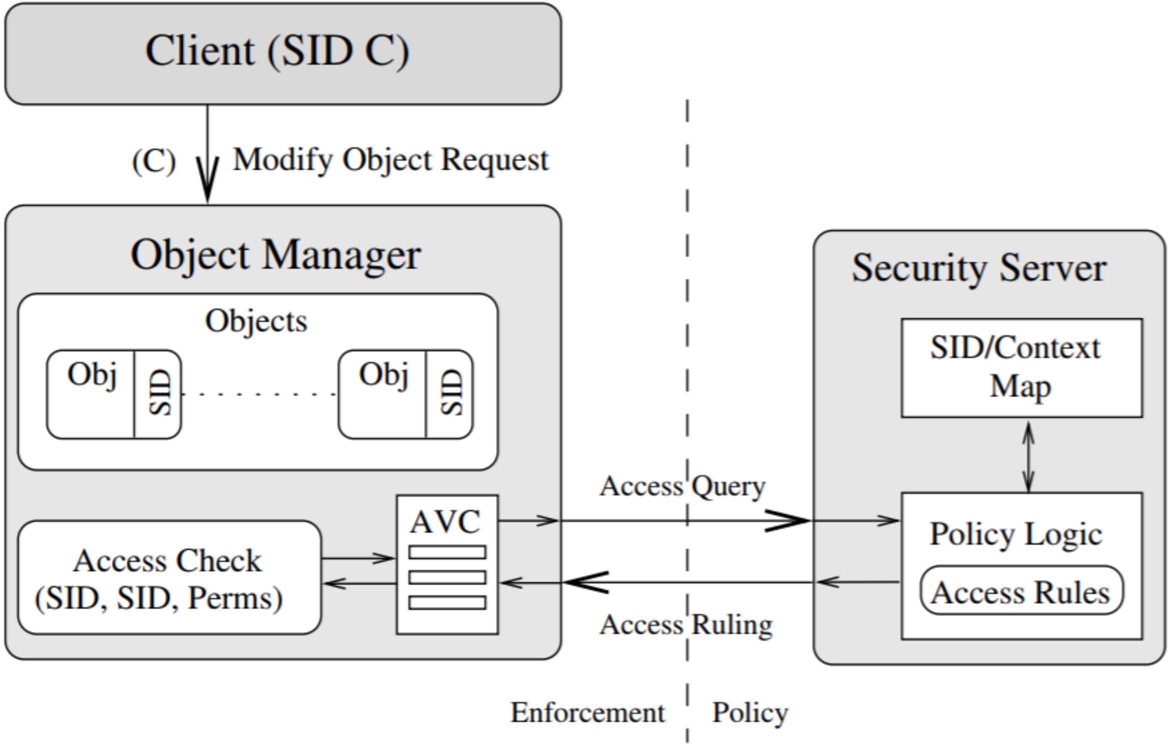
- ssid, tsid – source and target SIDs
- tclass – type of target
  - E.g., regular file, device, raw IP socket, TCP socket, ...
- Server can decide more than it is asked for
  - access_vector_t is a bitmask of permissions
  - decided can contain more than requested
  - Effectively implements decision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

Access vector cache

- 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:
  - X Andy can authorize expenses up to $999.99 %
  - X 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

AVC in a query



AVC interface

```
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);
```

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

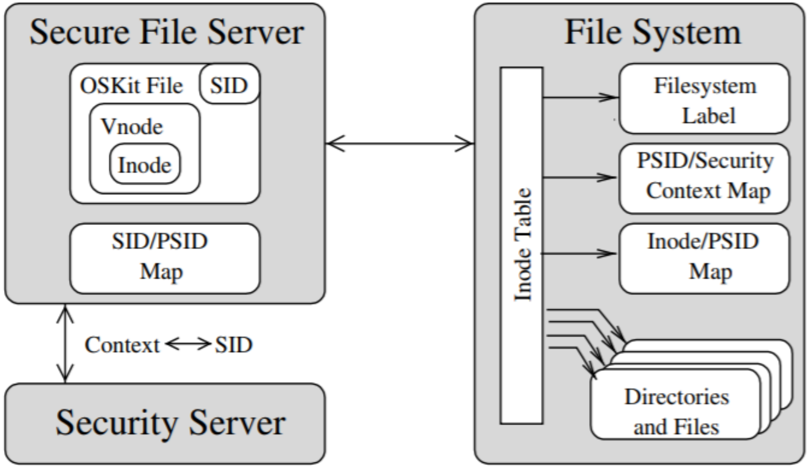
```
ret = avc_has_perm_ref(
        current->sid, sk->sid, sk->sclass,
        SOCKET__BIND, &sk->avcr);
```

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

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

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

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)

SElinux contexts

In practice, SElinux contexts have four parts:

user *role* *type* *level*

`system_u : system_r : sshd_t : s0`

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
```

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
```

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 to 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 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

Example: Loading kernel modules

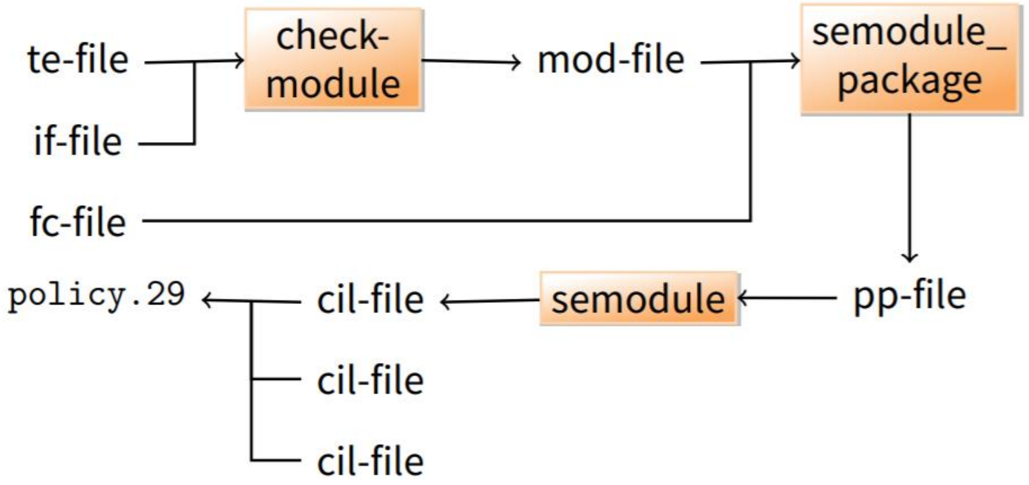
```
(1) allow sysadm_t insmod_exec_t:file x_file_perms;
(2) allow sysadm_t insmod_t:process transition;
(3) allow insmod_t insmod_exec_t:process { entrypoint execute };
(4) allow insmod_t sysadm_t:fd inherit_fd_perms;
(5) allow insmod_t self:capability sys_module;
(6) allow insmod_t sysadm_t:process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

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.

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)