

Access Control in Operating Systems

Slides by Hussain Almohri

Access Control

- **Definition:** Access control is the process of mediating every request to data and services maintained by a system and determining whether the request should be granted or denied.
- Access control is the act of ensuring that a user accesses only what she is *authorized* to and no more.
- Components of an AC system
 - Security Policy
 - Security Model
 - Security Mechanism

[Vimercati, et al., 2005]

Security Policies

- A policy defines the (high-level) rules according to which access control must be regulated.
- A policy is specified in a policy specification language.
- It determines the applicable objects, subjects, and the permissible capabilities.

[Vimercati, et al., 2005]

- For example
 - allow program execution for all users
 - restrict installation in /usr/bin for all users except root

Security Model

- An access control model provides a formal representation of the access control security policy.
- The benefit of this formalization is to provide theoretical proof of correctness and limitations of the policy engine.

[Vimercati, et al., 2005]

Policy enforcement mechanisms

- Policies specified by administrators in policy files but enforced by the kernel.
- The kernel provides appropriate policy checking functions before executing operations that are designated for policy enforcement.
- For example
 - Check policies for all file system operations
 - Check policies for `execv` and `fork`
 - Check policies for `socket()`

Open vs. Closed Policies

- **Closed policy:** authorizations specify permissions for an access. The closed policy allows an access if there exists a positive authorization for it, and denies it otherwise.
- **Open Policy: (negative)** authorizations specify denials for an access. The open policy denies an access if there exists a negative authorization for it, and allows it otherwise.

Combination of the two

- Most system use closed policy and in some cases with a combination of the two.
- Example
 - A permission to be given to a group of users except Alice.
 - A closed policy is specified for the group and a negative policy for Alice.
- Problems: incompleteness and inconsistencies

Approaches to solve inconsistencies

- Denials first
- Most specific first
- Explicit priority levels
- Positional (whoever comes first)
- Time-dependent priorities

Policy examples

```
isAuthorizedToTransferBetweenAccounts(User, Amount, From, To) :-  
    Amount < 100000.00 ,  
    //The amount to be transferred must be less than 100,000.00
```

```
    getBalanceOfAccount(From, Balance),  
    //get the balance of account From
```

```
    Amount < Balance,  
    //The transferred amount must be less than the balance
```

```
    isValidTimeForTheAmount(Amount).  
    //Check the time validity
```

```
can_Perform_On_Resource(User, manager, update, Record) :-  
    managerCanUpdate(User, Record).
```

XML Policies

Policies could be specified with XML based languages

E.g., A user account can only be involved in a single session.

```
<Policy>
  <Authentication>
    <LoginSession:SingleUser>
    </LoginSession:SingleUser>
  </Authentication>
</Policy>
```

```
boolean authenticate(user) {
  boolean singleSession =
    checkSessionPolicy();
  if(sessionExists(user)&& singleSession) {
    return false;
  } ...
}
```

Enforcement Mechanisms

- Access control models
 - Discretionary Access Control (DAC)
 - Based on subjects and roles
 - Mandatory Access Control (MAC)
 - Based on capabilities

Discretionary Access Control

- Differs from Discretionary Access Controls (DAC), where users and ownership models are provided
- Methods such as assigning UIDs in UNIX are used to identify users.
- Works in a system of ownerships and permissions

DAC in UNIX-based systems

- The system is accessed and controlled by users and groups.
- A user can join multiple groups.
- Each object in the system has an ownership and a set of permissions for its owner, its group, and all others.
- Each object can only be owned by one user.

DAC in UNIX-based systems

- Since files are a critical part of UNIX-based systems, access control can be enforced on many operations.
- Network access, memory files, disk files, controlling I/O devices are among the many.
- The file system has a component to perform authorization of operations.

DAC applied to system objects

- Regular file
- d Directory
- l Symbolic link
- c Character special device
- b Block special device
- p FIFO
- s Socket

root users

- When the system is first installed, it will automatically create the root user that has virtually all permissions to everything.
- In macOS since (OX 10.11 El Capitan),
 - the root privileges are limited when it comes to a number of critical libraries.
 - The system restricts the root itself from changing directory permissions, adding files (in some cases), or deleting files from protected libraries.

User interactions

- root user can add new users and groups that are used to perform operational tasks.
- Users either interact with a terminal or with the graphical interface. In both cases, a user account must be associated with every action taken by the human user.
- To determine the currently active user

```
/home/user$ whoami  
user  
/home/user$ exit  
$ whoami  
root
```

IDs and Group Memberships

- To show the current user's id and groups

```
$ id
uid=0(root) gid=0(wheel)
groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),9(procm
od),12(everyone),20(staff),29(certusers),61(localaccounts),80(admin),33(_appsto
re),98(_lpadmin),100(_lpoperator),204(_developer),395(com.apple.access_ftp),398
(com.apple.access_screensharing),399(com.apple.access_ssh)
```

- To show the id for a particular user:
 - `$ id root`

File permissions and ownership

- File ownerships

| | | | | | | | | |
|------------|----|-----------|-----------|------|-----|----|-------|----------|
| drwxr-xr-x | 2 | root | wheel | 68 | Aug | 23 | 00:35 | netboot |
| drwxr-xr-x | 5 | _networkd | _networkd | 170 | Oct | 1 | 22:48 | networkd |
| drwxr-x--- | 5 | root | wheel | 170 | Oct | 18 | 2014 | root |
| drwxr-xr-x | 4 | root | wheel | 136 | Oct | 1 | 22:38 | rpc |
| drwxrwxr-x | 35 | root | daemon | 1190 | Oct | 13 | 07:22 | run |
| drwxr-xr-x | 2 | daemon | wheel | 68 | Aug | 23 | 00:35 | rwho |
| drwxr-xr-x | 7 | root | wheel | 238 | Sep | 17 | 10:04 | spool |
| drwxrwxrwt | 6 | root | wheel | 204 | Oct | 12 | 15:48 | tmp |

- The first column shows the permissions, the second is the number of links to the file, the third is the owner and the fourth is the group

Change file ownership

- To change file owners
 - `$ chown user file`
- To change file group
 - `$ chgrp group file`
 - `$ chown user:group file`
- Check groups
 - `$ groups`

Executable permissions

```
$ ls -al p.sh
-rw-r--r--  1 user  staff  0 Oct 13 08:05 p.sh
$ chmod +x p.sh
$ ls -al p.sh
-rwxr-xr-x  1 user  staff  0 Oct 13 08:05 p.sh
$ chmod o-x p.sh #removing permission from others
$ ls -al p.sh
-rwxr-xr--  1 hsn   staff  0 Oct 13 08:05 p.sh
$ chmod -x p.sh
$ chmod ug+x p.sh
$ ls -al p.sh
-rwxr-xr--  1 hsn   staff  0 Oct 13 08:05 p.sh
```

suid programs

- Files can optionally have a suid bit set.
- The suid bit is useful with executable files.
- Suppose user creates an executable file `p.sh` and then issues `chmod g+s`.
- Also, user allows the development group to be assigned as the file group.
- When a user from development group runs `p.sh`, it will run as if user was running it.

```
$ chmod g+s p.sh
$ ls -al p.sh
-rwxr-sr--  1 hsn  staff  0 Oct 13 08:05 p.sh
```

File creation mask

- UNIX/Linux systems use a default file creation mask to specify which permissions are *not granted* by default.
- Check your file creation defaults
 - `$ touch file; ls -al file`
 - `$ umask`
- To change umask such that new files are not given read/write permissions for g and o
 - `$ umask u=rwx, g=, o=`
 - `$ umask`
 - `0077`

From DAC to MAC

- DAC is limited to file ownership and permissions.
- We need a more general mechanism to enforce access control even when DACs are not appropriately set by the users.
- MAC is controlled often by a single user (an administrator) with root permissions.
- MAC specifies objects, subjects, and their capabilities (i.e., permissible operations)

Mandatory Access Control

- Limiting *subjects*' access to *objects*
- A subject can be a program binary, a process, or a thread
- An object is a structure such as a file, a port, a memory segment, or an I/O device
- The operating system makes sure MAC is enforced according to a *mechanism* and/or a *policy*

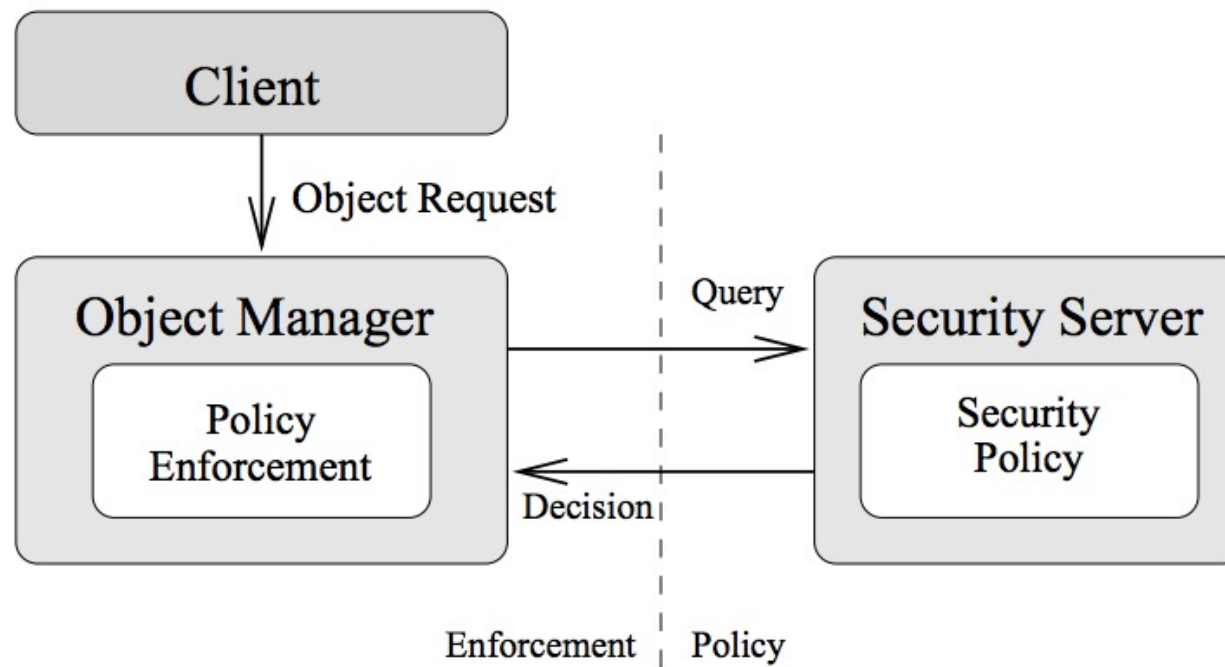
Why the need for a MAC system?

- General-purpose operating systems allow any binary to run
- No clear separation of processes and their privileges
- Often untrustworthy processes run on systems with virtually access to all system resources
- Processes have access to objects as far as the UID allows

First important attempt: FLASK

- Flux Advanced Security Kernel
- An architecture to completely separate *policies* from *enforcement*
- Flexible support for defining any level of security policies

FLASK Architecture



Concept: Type enforcement

- FLASK uses type enforcement to achieve its security goals
- It uses the binary executable to determine the appropriate access to objects
- Uses object labeling
 - Each object has an access label
 - Only one authorized party issues labels
 - Labels checked at access time

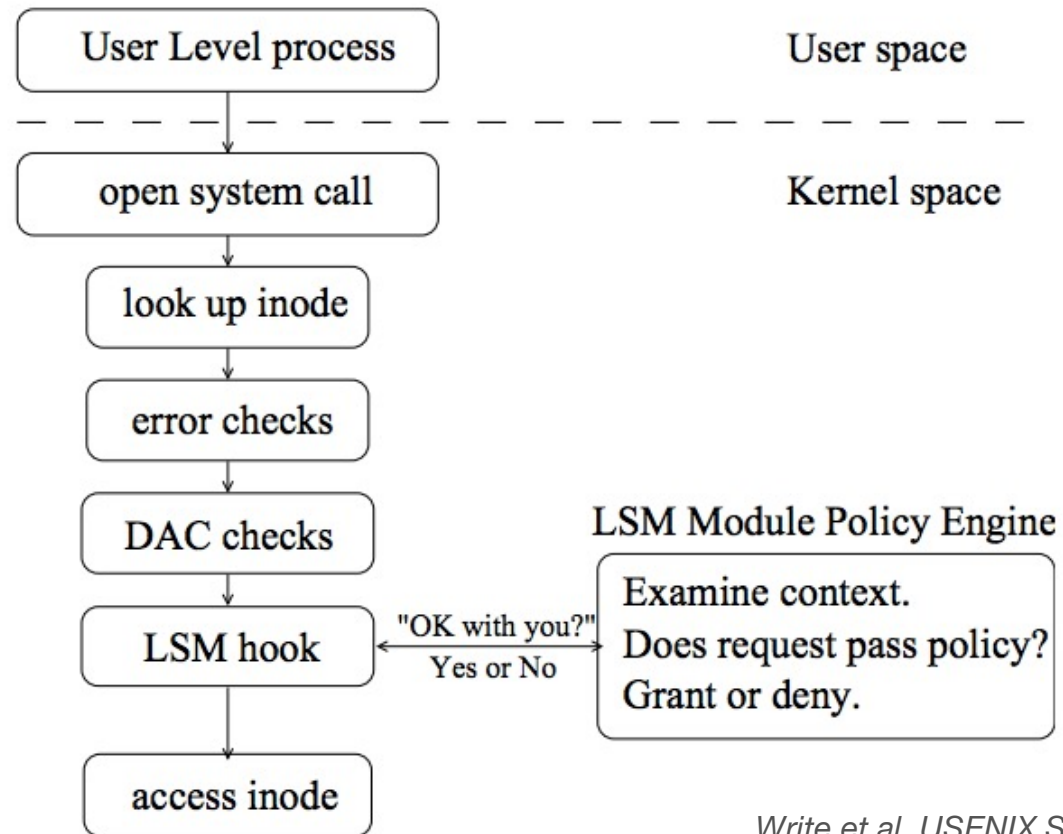
Implementation: Linux Security Modules

- An architecture for extending the Linux kernel to provide an implementation of *complete mediation*.
- It provides a code template for programmers to define the actions and protection mechanisms on the system resources.
- Work as loadable kernel modules that extend the security functionality of the system.

Design goals for LSM

- The security framework must be
 - truly generic, where using a different security model is merely a matter of loading a different kernel module;
 - conceptually simple, minimally invasive, and efficient; and
 - able to support the existing POSIX.1e capabilities logic as an optional security module.
- By Linus Torvalds

LSM hooks

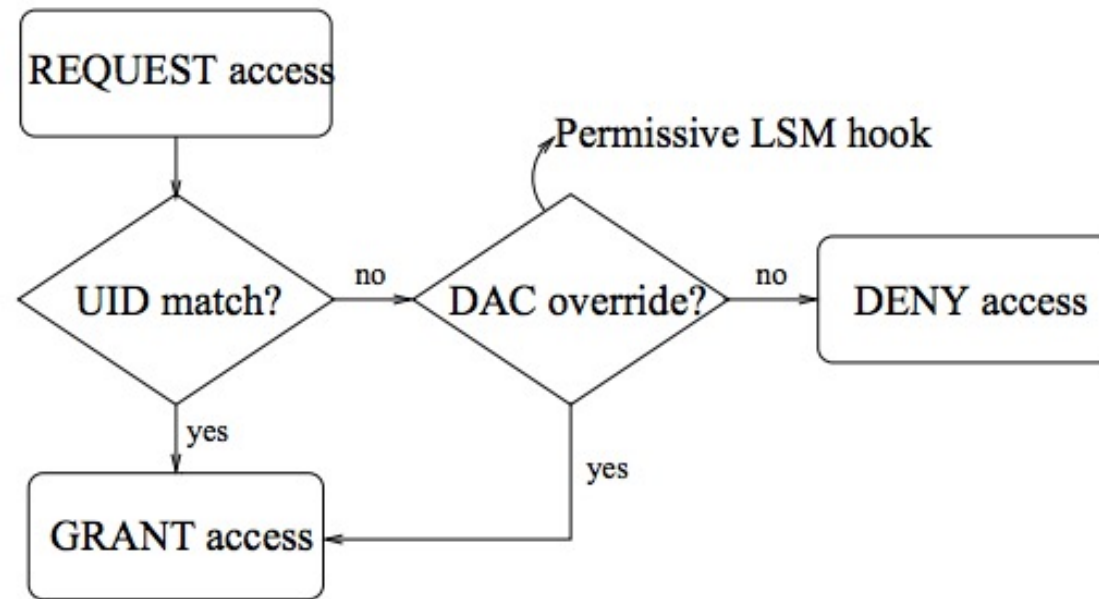


Write et al. USENIX SEC'12

LSM's simple strategy

- **Restrictive (most cases):** If the kernel decides to grant access, the security module may *deny access*, but when the kernel would deny access, the security module is ignored.
- **Permissive (some cases):** The security module can grant access when the kernel would *deny* it.

Permissive hooks



Write et al. USENIX SEC'12

How hooks are implemented?

```
int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode) {
    int error; down(&dir->i_zombie);
    error = may_create(dir, dentry);
    if (error)
        goto exit_lock;
    error = -EPERM;
    if (!dir->i_op || !dir->i_op->mkdir)
        goto exit_lock;
    mode &= (S_IRWXUGO|S_ISVTX);
    error = security_ops->inode_ops->mkdir(dir,dentry, mode);
    if (error)
        goto exit_lock;
    DQUOT_INIT(dir);
    lock_kernel();
    error = dir->i_op->mkdir(dir, dentry, mode);
    unlock_kernel();
    exit_lock:
    up(&dir->i_zombie);
    if (!error) {
        inode_dir_notify(dir, DN_CREATE);
        security_ops->inode_ops->post_mkdir(dir,dentry, mode);
    }
    return error;
}
```

Control new directory creation

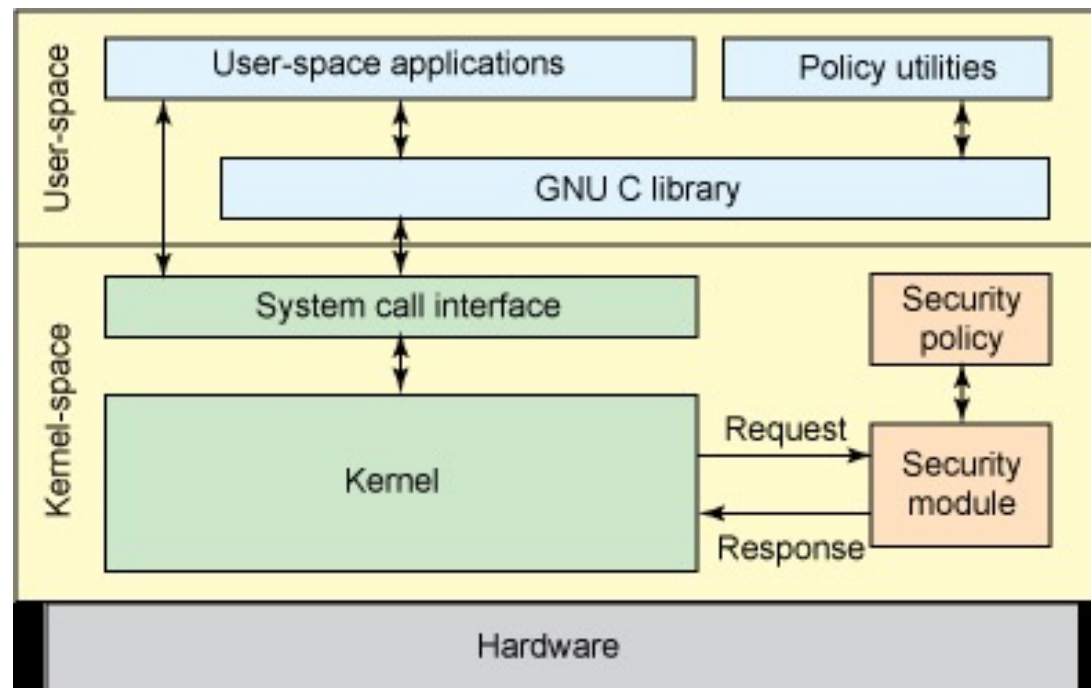
Write et al. USENIX SEC'12

Update the security field for the new directory

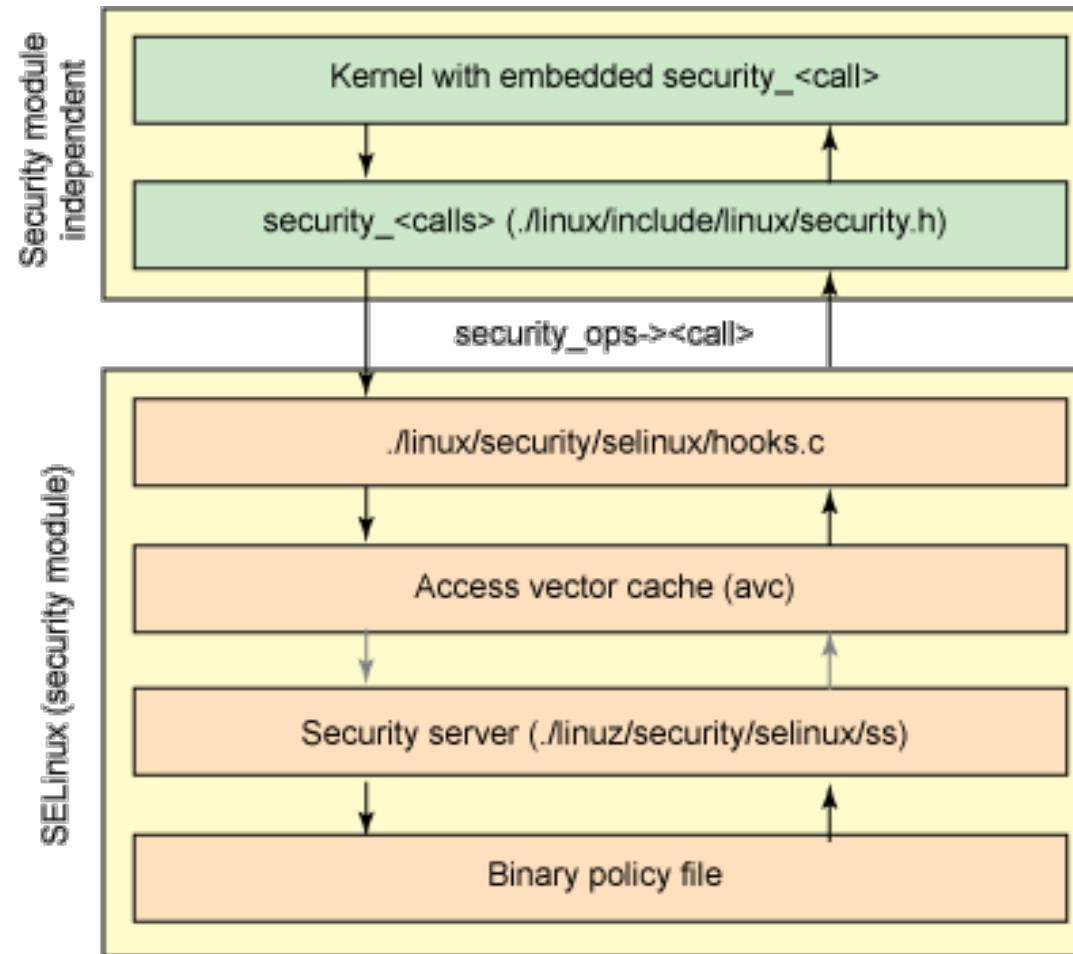
Security Enhanced Linux (SELinux)

- Uses the FLASK architecture for Linux
- Includes a security server that labels objects and checks for permissions
- Objects have security contexts and security identifiers
- Checks use SIDs and security contexts to determine if access should be granted
- Implemented in Linux Security Modules

Security Enhanced Linux (SELinux)



Security Enhanced Linux (SELinux)



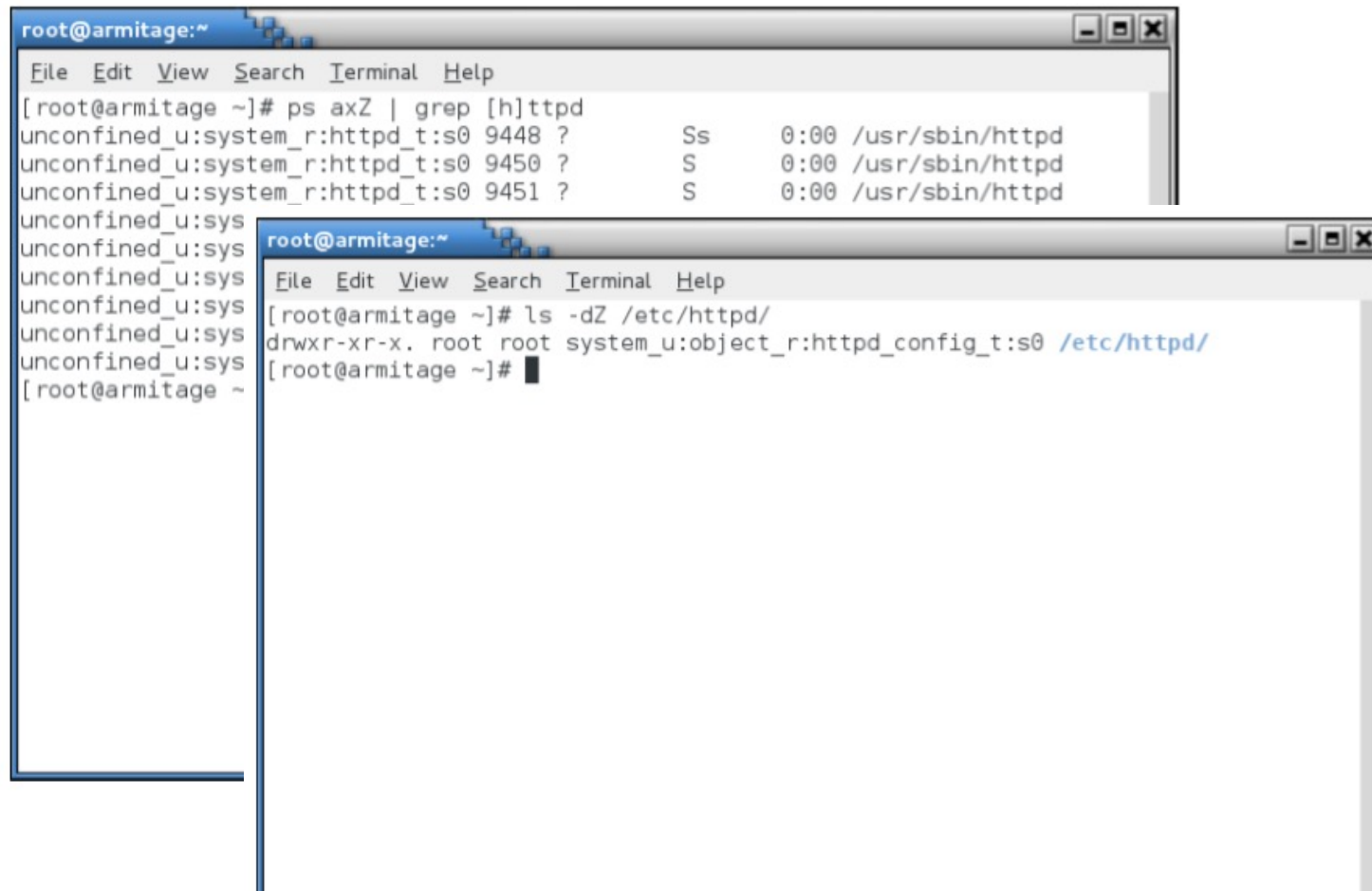
Labeling for MAC in SELinux

- Labeling is done for
 - files, processes, ports, sockets, etc.
 - Labels are stored as file system attributes.
 - For processes, labels are managed by the kernel.
- Labels are formatted as
 - user:role:type:level(optional)

How SELinux works? (Example)

- An Apache process is labeled with type httpd_t.
- Apache's configuration file is labeled with type httpd_config_t.
- A policy indicates that only a process with httpd_t can access httpd_config_t.

How SELinux works? (Example)



The image shows two terminal windows from a system named 'armitage'. The top window displays the output of the command 'ps axZ | grep [h]ttpd', showing three httpd processes with SELinux contexts of 'unconfined_u:system_r:httpd_t:s0'. The bottom window displays the output of 'ls -dZ /etc/httpd/', showing the directory's SELinux context as 'system_u:object_r:httpd_config_t:s0'.

```
root@armitage:~  
File Edit View Search Terminal Help  
[root@armitage ~]# ps axZ | grep [h]ttpd  
unconfined_u:system_r:httpd_t:s0 9448 ? Ss 0:00 /usr/sbin/httpd  
unconfined_u:system_r:httpd_t:s0 9450 ? S 0:00 /usr/sbin/httpd  
unconfined_u:system_r:httpd_t:s0 9451 ? S 0:00 /usr/sbin/httpd  
unconfined_u:sys  
unconfined_u:sys  
unconfined_u:sys  
unconfined_u:sys  
unconfined_u:sys  
unconfined_u:sys  
[root@armitage ~]  
root@armitage:~  
File Edit View Search Terminal Help  
[root@armitage ~]# ls -dZ /etc/httpd/  
drwxr-xr-x. root root system_u:object_r:httpd_config_t:s0 /etc/httpd/  
[root@armitage ~]#
```

Analysis of SELinux

- Pros
 - Theoretically can mediate access to any object
 - Recognizes basic execution units as subjects
 - Enforcement deep in the kernel
- Cons
 - Downgraded performance
 - Downgraded usability
 - Complicated policies may result in wrong decisions