OS Security Principles

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Dicii
Università di Roma
Tor Vergata

Fundamental Security Principles

1. Systems must be usable by legitimate users only
2. Access is granted on the basis of authorization, according to the rules that are established by some system administrator

- an attacker's target could be rendering systems unusable by legitimate users (so called DOS – Denial of Service attacks)

Identifying Users

User Authentication

- Users login via passwords
- The passwords' database is stored in two distinct files:
  - `/etc/passwd`
  - `/etc/shadow`
- `/etc/passwd` can be accessed by any user
- `/etc/shadow` can be accessed only by root

/etc/passwd

- `/etc/passwd` has the following format:
  - `username:passwd:UID:GID:full_name:directory:shell`

`username:Npge08pfz4wuk:503:100:TheUser:/home/username:/bin/sh`

- `Np` represents the salt (16 bit) and `ge08pfz4wuk` is the encrypted password
- When using shadowing, `/etc/passwd` has the format: `username:x:503:100:full_name:/home/username:/bin/sh`
- `x` is a placeholder: `/etc/passwd` no longer contains passwords

/etc/shadow

- `/etc/shadow` has the format:
  `username:passwd:ult:can:must:note:exp:disab:reserved`

- where:
  1. `username` is the user
  2. `passwd` is the encrypted password
  3. `ult` are the days from 1/1/1970 since the last password change
  4. `can` day interval after which it is possible to change the password
  5. `must` day interval after which the password must be changed
  6. `Note` day interval after which the user is prompted for password update
  7. `exp` days after which the account is disabled if password expires
  8. `disab` days from 1/1/1970 after which the account will be disabled
  9. `reserved` no usage – a reserved field

User IDs in Unix

- The username is only a placeholder
- What determines which user is running a program is the UID
- The same is for GID
- Any process is at any time instant associated with three different UIDs/GIDs:
  - *Real*: this tells who you are
  - *Effective*: this tells what you can actually do
  - *Saved*: this tells who you can become again

UID/GID management system calls

- `setuid()/seteuid()`: available only to UID/EUID equal to 0 (root)
- `getuid()/geteuid()`: queries available to all users
- Similar services exist for managing GID
- `setuid()` is "non reversible" in the value of the saved UID: it overwrites all the three used IDs
- `seteuid()` is reversible and does not prevent restoring a saved UID
- An EUID-root user can temporarily become a different EUID user and then resume EUID-root identity
- UID and EUID values are not forced to correspond to those registered in `/etc/passwd`

`su` and `sudo`

- Both these commands are setuid-root
- They enable starting with the EUID-root identity
- If a correct input password is given by the user, they move the real UID to root or the target user (in case of `su`)
- After moving the UID to root, `sudo` executes the target command

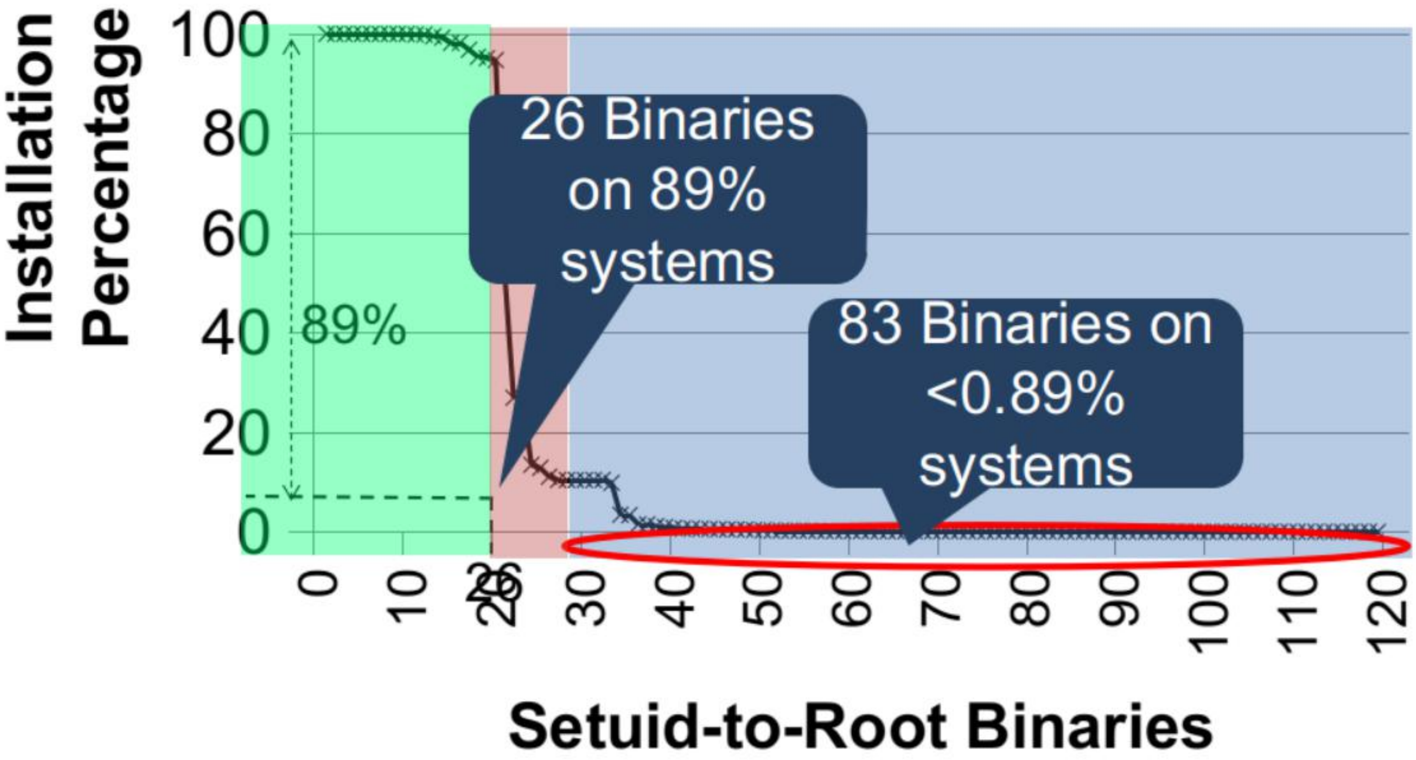
- ...it's now time to break Linux!
- ...how was all that possible?

Principle of Least Privilege

- In a particular abstraction layer of a computing environment, entities must be able to access only the information and resources that are necessary for its legitimate purpose
    - Applies to processes, users, programs, virtual instances, ...

- *Better system stability*: it is easier to test possible actions of the applications/users and interactions with other applications.
- *Better system security*: vulnerabilities in one application cannot be used to exploit the rest of the machine.
- *Ease of deployment*: the fewer privileges an application requires the easier it is to deploy within a larger environment.

Number of SUID root binaries in live systems



Access Control Fundamentals

- Control the ability for a subject to perform an action on an object.

Component	Description
Subject	The entity making the request (a *process* or *user*)
Action	The requested operation (read, write, execute)
Object	The target of the action (a file, network connection, hardware)

Purpose of Access Control

- Limit the damage caused by:
    - Malicious applications
    - Malicious users
    - Compromised applications
    - Badly written applications
    - User mistakes

Security policies

- A security policy is named **discretionary** (DAC) if ordinary users (including the administrator) are involved in the definition of security attributed (e.g. protection domains)

- A security policy is named **mandatory** (MAC) if its logics and the actual definition of security attributes is demanded to a security policies' administrator (who is not an actual user/administrator of the system)

- Security policies could benefit both *internal* and *external* security

Fine-Grained Access Control

- Can be either DAC or MAC
- Refers to the granularity of permissions
    - System wide?
    - Directory?
    - File?
    - Action on file?
    - Byte offset of file?

POSIX ACLs

- A DAC scheme
- File permissions based on the owner, group, and everyone else.
  - Owner can always modify permissions on the file.

```
# file: audio
# owner: gwurster
# group: audio
user::rwx
group::r-x
group:powerdev:r-x
mask::r-x
other::---
```

Granting permissions with ACL

- **if** the user ID of the process is the owner, the owner entry determines access
- **else if** the user ID of the process matches the qualifier in one of the named user entries, this entry determines access
- **else if** one of the group IDs of the process matches the owning group and the owning group entry contains the requested permissions, this entry determines access
- **else if** one of the group IDs of the process matches the qualifier of one of the named group entries and this entry contains the requested permissions, this entry determines access
- **else if** one of the group IDs of the process matches the owning group or any of the named group entries, but neither the owning group entry nor any of the matching named group entries contains the requested permissions, this determines that access is denied
- **else** the other entry determines access.

Granting permissions with ACL

- **if** the matching entry resulting from this selection is the owner or other entry and it contains the requested permissions, access is granted
- **else if** the matching entry is a named user, owning group, or named group entry and this entry contains the requested permissions and the mask entry also contains the requested permissions (or there is no mask entry), access is granted
- **else** access is denied.

Capabilities

Linux Capabilities

- Traditional UNIX implementations distinguish only two categories of processes:
  - *privileged* processes (EUID = 0)
  - *unprivileged* processes (all the others)
- Privileged processes bypass all kernel permission checks
- Unprivileged processes are subject to full permission checks (EUID, EGID, ACL)
- Capabilities have been designed to allow processes to drop root privileges.
  - Certain privileges were traditionally reserved for the root account.
  - Capabilities split the permissions given to root.
- There is a withdrawn standard (POSIX.1e) which tried to govern capabilities

Some Linux Capabilities

- `linux-headers/include/linux/capability.h`

Name	Description
`CAP_CHOWN`	Allow changing file ownership, overrides DAC
`CAP_KILL`	Send signals to other processes
`CAP_NET_RAW`	Allow using raw sockets (e.g., ping)
`CAP_NET_BIND_SERVICE`	Allow binding to ports below 1024
`CAP_SYS_NICE`	Allow raising process priority
`CAP_SYS_TIME`	Allow setting the system clock
`CAP_SYS_ADMIN`	The "new" root (it's an overloaded capability)

Capabilities implementation

- For all privileged operations, the kernel checks whether the thread has the required capability in its effective set
- The kernel provides system calls allowing a thread's capability set to be changed and retrieved
  - `int capget(cap_user_header_t hdrp, cap_user_data_t datap)`
  - `int capset(cap_user_header_t hdrp, const cap_user_data_t datap);`
  - there is the `libcap` library that provides higher-level wrappers to these system calls
- The filesystem must support attaching capabilities to an executable file, so that a process gains those capabilities when the file is executed

- In Linux, all these three aspects are supported since kernel 2.6.24

Thread Capability Sets

- Each thread in Linux has the following capability sets:
  - *Permitted*: a limited superset for the effective capabilities that a thread may assume. If a capability is dropped from the permitted set, it can never be reacquired.
  - *Inheritable*: the set of capabilities preserved upon an execve() executed by a privileged application.
  - *Effective*: the set of capabilities used by the kernel to perform permission checks for the thread.
  - *Ambient*: the set of capabilities that are preserved across an execve() of a program that is not privileged.

- Any child created via fork() inherits copies of the parent's capability sets

File Capabilities

- Capabilities are stored in files in their *extended attributes.*
- The set of file capabilities, along with the capability sets of the thread, determine the capabilities of a thread after an execve()
- There are two capability sets associated with files:
  - *Permitted*: these capabilities are automatically permitted to a thread, regardless of the thread's inheritable capabilities.
  - *Inheritable*: this set is ANDed with the thread's inheritable set to determine which inheritable capabilities are enabled in the permitted set of the thread after the execve()
- There is also the *Effective* sticky flag: if set, during an execve() all of the new permitted capabilities for the thread are raised in the effective set, otherwise no new permitted capability is in the effective set.

Transformation of capabilites during an execve()

- P'(ambient) = (file is privileged) ? 0 : P(ambient)
- P'(permitted) = (P(inheritable) & F(inheritable)) |
  (F(permitted) & cap_bset) | P'(ambient)

- P'(effective) = F(effective) ? P'(permitted) : P'(ambient)
- P'(inheritable) = P(inheritable) [i.e., unchanged]

- cap_bset is the value of the capability bounding set

Capability bounding set

- A per-thread security mechanism that can be used to limit the capabilities that can be gained during an execve().
- The bounding set is used in the following ways:
  - During an execve(), the capability bounding set is ANDed with the file permitted capability set, and the result of this operation is assigned to the thread's permitted capability set.
  - The capability bounding set thus places a limit on the permitted capabilities that may be granted by an executable file.
  - The capability bounding set acts as a limiting superset for the capabilities that a thread can add to its inheritable set using capset().
  - This means that if a capability is not in the bounding set, then a thread can't add this capability to its inheritable set, even if it was in its permitted capabilities, and thereby cannot have this capability preserved in its permitted set when it execve()s a file that has the capability in its inheritable set.

Security Modules

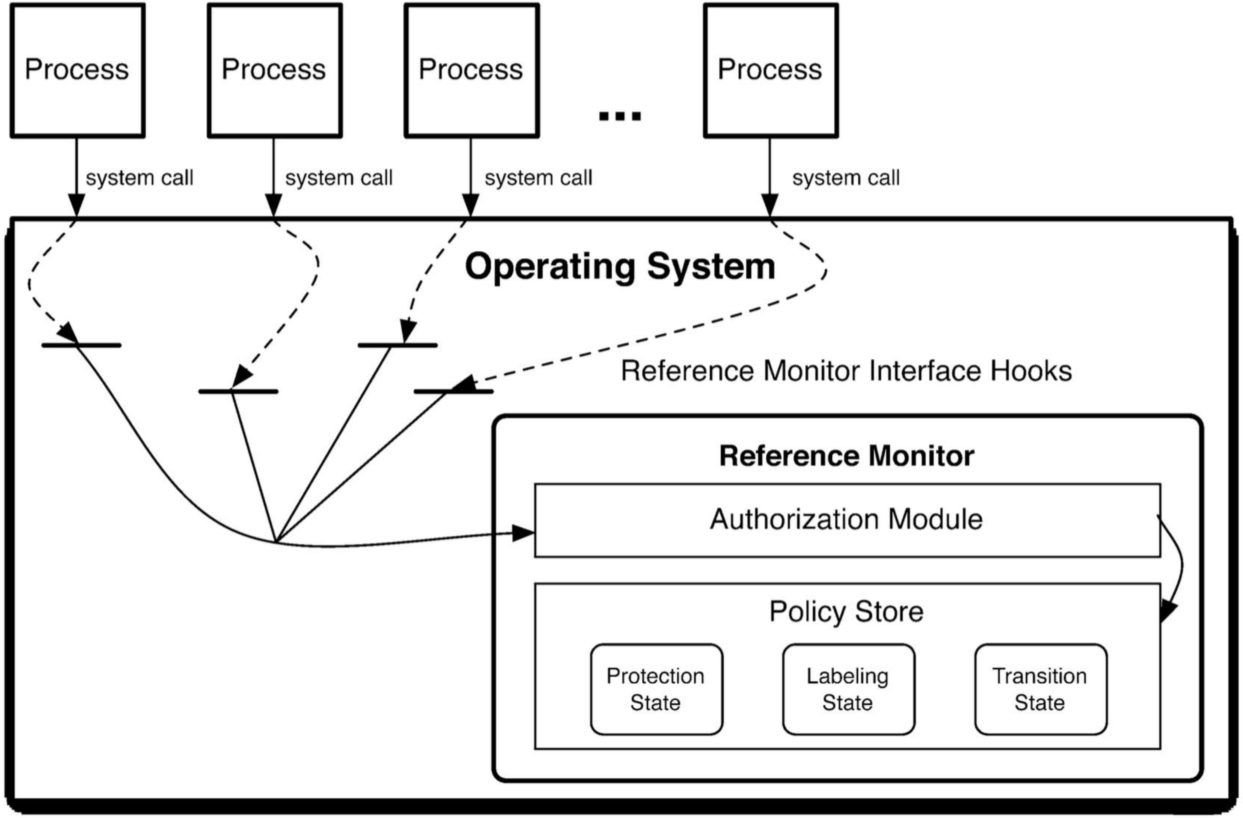
Security Modules

- They are an attempt to provide a *security framework* in the kernel
- They have been implemented as a response to the *lack of agreement* in the cybersec community on how to improve the security of an operating system
- By default, Linux offers only DAC mechanisms
- Security modules can be loaded in the kernel to implement some sort of MAC
- Some examples:
  - SELinux
  - SMACK
  - AppArmor
  - Tomoyo

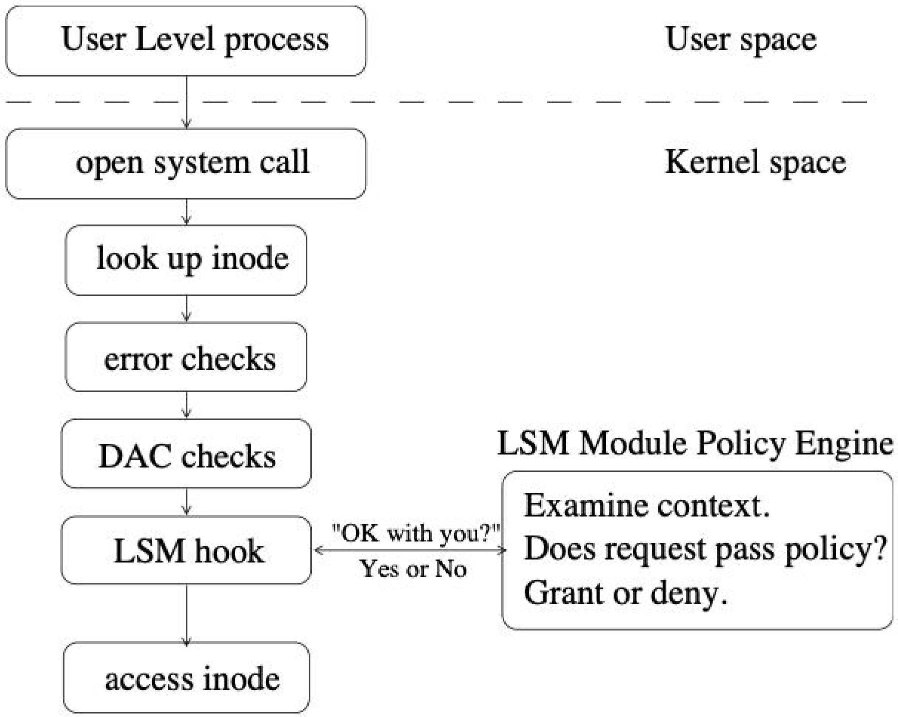
Security Modules

- The Linux Security Modules framework offers a minimum-overhead generic implementation the support for MAC:
  - a different security model can be implemented only changing the loaded *security module*
  - the invasivity in the kernel sources is minimal
  - can increase the adherence to POSIX.1e

Reference Monitor



Example of the Mechanism: File Access



LSM Hooks

- LSM hooks are *restrictive*
- They give access to LSM capabilities *only* if the thread executing the operation has already access to the resource
- They are therefore meant to *reduce* access privileges to resources

- These hooks are kept in the security_ops structure, which is initialized at boot time with default DAC functions related to capabilities
- Upon load, a security module can replace the hooks of interest using the register_security() internal function
- Only one security module can be loaded at a time in the kernel

register_security()

```
int register_security(struct security_operations *ops)
{
        if (verify(ops)) {
                printk(KERN_DEBUG "%s could not verify "
                       "security_operations structure.\n", __func__);
                return -EINVAL;
        }
        if (security_ops != &default_security_ops)
                return -EAGAIN;
        security_ops = ops;
        return 0;
}
```

Security fields

- Several data structures in the kernel have been modified so as to support LSM, introducing generic pointers to additional data structures

Data Structure	Object
`task_struct`	threads
`linux_binprm`	Program being loaded
`super_block`	Filesystem
`inode`	Pipe, file, or socket
`file`	Open file
`sk_buff`	Network buffer (packet)
`net_device`	Network device
`kern_ipc_perm`	Semaphore, shared memory segment, or message queue
`msg_msg`	Individual message

security_operations

- These are tables of function pointers used to manage *security fields*, which are different for each implementation of a LSM

```
struct security_operations {
        ...
        int (*ptrace) (struct task_struct *parent, struct task_struct *child);
        ...
        int (*inode_setattr) (struct dentry *dentry, struct iattr *attr);
        ...
};
```

Registering a Security Module

- Upon load, a kernel module must only register the function pointers that it will be using to support the enhanced security policies
- For example, within a module load function:

```
my_inode_setattr(struct dentry *dentry, struct iattr *iattr)
{...}

static struct security_operations my_security_ops = {
      inode_setattr = my_inode_setattr,
};
```

Security Hooks Categories

- Security hooks can be classified into several categories:
  - *Task hooks*: they provide a control interface for functions which manage processes and their operations
  - *Program loading hooks*: they provide an interface to decide upon which programs can be launched
  - *IPC hooks*: they allow to manage the access policies to IPC functions
  - *Filesystem hooks*: they enable more granular access control on file system objects
  - *Network hooks*: they provide an interface to manage both devices and network objects

An Example: File System Hooks

```
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;
```

An Example: File System Hooks

```
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();
```

An Example: File System Hooks

```
    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;
}
```

MAC on Linux

S.M.A.C.K. (Simplified Mandatory Access Control Kernel)

- It is based on *labels* associated with *subjects* and *objects*
  - labels are only strings, with no actual required meaning
- A subject can access an object only if the labels match
- There must be an explicit access rule describing a label match:
  - <subject-label> <object-label> <access>
- <access> is in the traditional Unix octal form (rwx)
- Labels are stored in files' extended attributes
  - A process must have `CAP_MAC_ADMIN` to change any of these attributes.

- Smack has been criticized for being written as a new LSM module instead of an SELinux security policy

Default Labels

- There are three default labels in SMACK:
  - _ (*floor*)
  - * (*star*)
  - ^ (*hat*)
- They are used with the following semantics:
  - Any access from a subject identified by * is denied
  - Any access from a subject identified by ^ is granted either in read mode, or in execute mode
  - Any object labeled with _ can be accessed in read mode or executed
  - Any object labeled with * can be accessed in any mode

SMACKFS

- It's a pseudo filesystem mounted in /sys/fs/smack
- There are several files that can be used to configure the system:
  - `access2`: can be used to query access permission (write a rule, read the response)
  - `change-rule`: write to this file to change access permission associated with label matches
  - `load2`: specify new access permissions
  - `onlycap`: specify additional labels which are required by processes to exploit the `CAP_MAC_ADMIN` capability
  - `revoke-subject`: remove all permissions associated with a certain subject

Tomoyo Linux

- Tomoyo is a MAC module that does not use labels to specify access grants
- Filepaths are taken as labels
  - the goal is to have an always *correct* and *explicit* labeling
  - in case of processes, every time that a fork occurs, *domain transition* is immediately captured by concatenating paths of the processes
- ACLs can be specified depending on the domain

Domain ACL example

<kernel> /usr/sbin/sshd /bin/bash

allow_execute /bin/ls

allow_read /home/pellegrini/.bashrc

allow_read/write /home/pellegrini/.bash_history

...

<kernel> /usr/sbin/sshd /bin/bash /bin/ls

allow_read /etc/group

allow_read /etc/nsswitch.conf

allow_read /etc/passwd

AppArmor

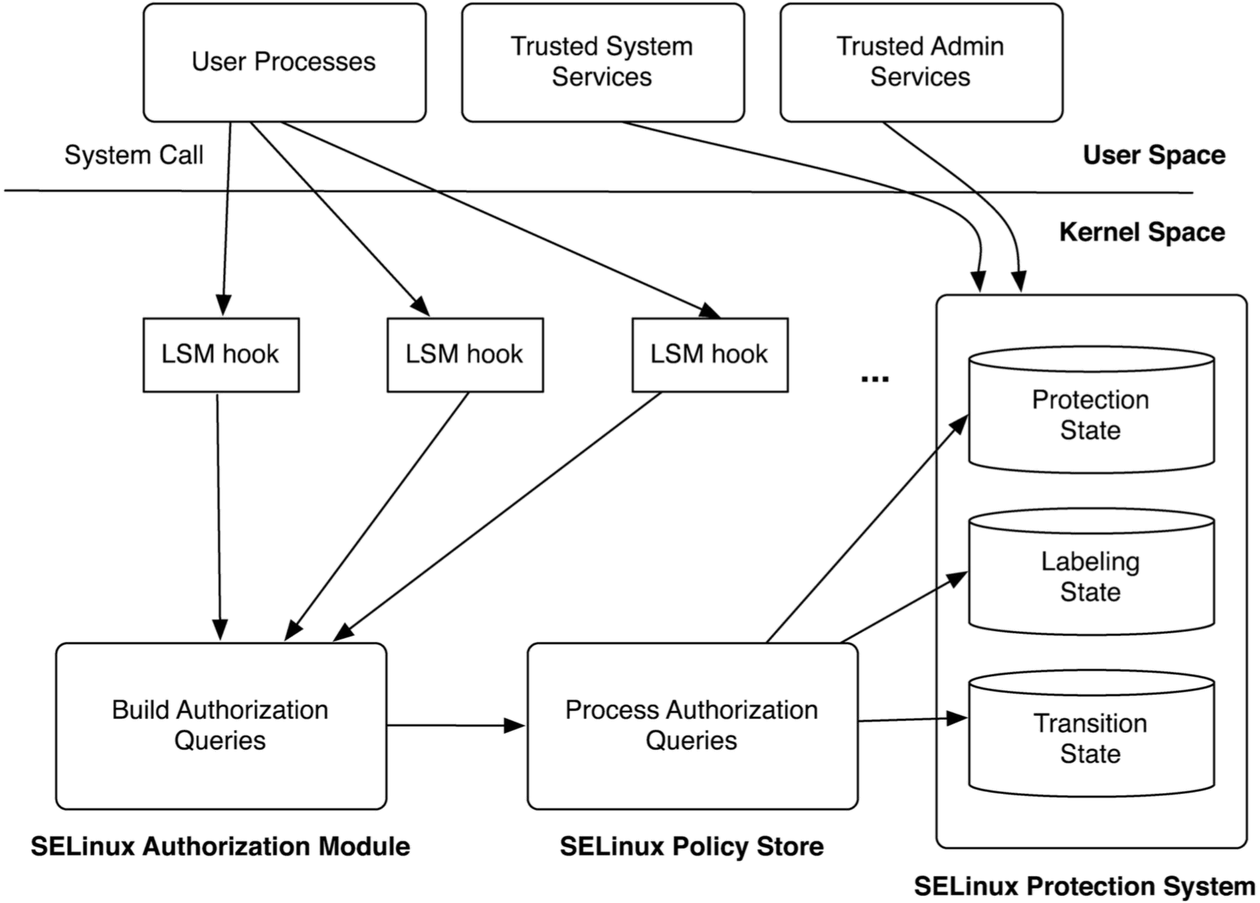
- It implements a task-centered MAC policy, where individual programs can have their own profile
    - A profile is a confinement of a program to set of files, capabilities, network access and rlimits
- Task profiles are created and loaded from userspace
- Tasks that do not have a profile run in an unconfined state which is equivalent to traditional Linux DAC (*selective confinement*)

AppArmor Policies

- Stored in `/etc/apparmor.d`
- Policies are expressed in the AppArmor Profile Language, which is then "compiled" in a binary representation in the kernel

```
/bin/bash {
    /bin/ls cx -> childprofile,
    network raw,
    profile childprofile {
        capability setuid,
        /home/pellegrini/**  rw,
    }
}
```

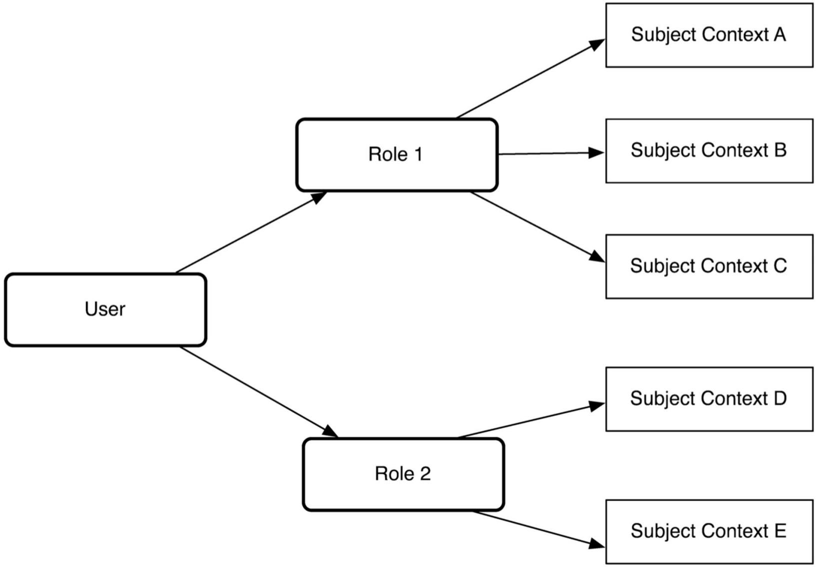
Security-Enhanced Linux (SELinux)



Step 1: Convert call to LSM hooks to authorization queries

- Parameters to an LSM call
  - Subject: the current process that is making the call
  - Object: inode
  - Operations requested

- Convert subject and object to labels
  - Called “context” in SELinux
  - Stored in kernel
  - Each object also has a “data type”



Step 2: Retrieve SELinux Policy Entry for the access request

- Example policy statement:

```
allow <subject_type> <object_type>:<object_class> <operation_set>

allow      user_t    passwd_exec_t:file          execute
allow    passwd_t    shadow_t:file               {read write}
```

SELinux Labeling State

- Map users/systems resources to labels

- Labeling state defines how newly created processes and resources are labeled
  - File context specification: define mapping from file paths to object context

```
<file path expr>          <context>
/etc/shadow.*             system_u:object_r:shadow_t:s0
/etc/*.*                  system_u:object_r:etc_t:s0
```

SELinux Transition State

- Defines under what conditions labels of subjects/objects may change

```
type_transition  <creator_type> <default_type>:<class> <resultant_type>

type_transition  passwd_t       etc_t:file             shadow_t
```

- A process with `passwd_t` label creates a file that would have `etc_t`, but with this policy the file will have the `shadow_t` label

SELinux Transition State

- Defines under what conditions labels of subjects/objects may change

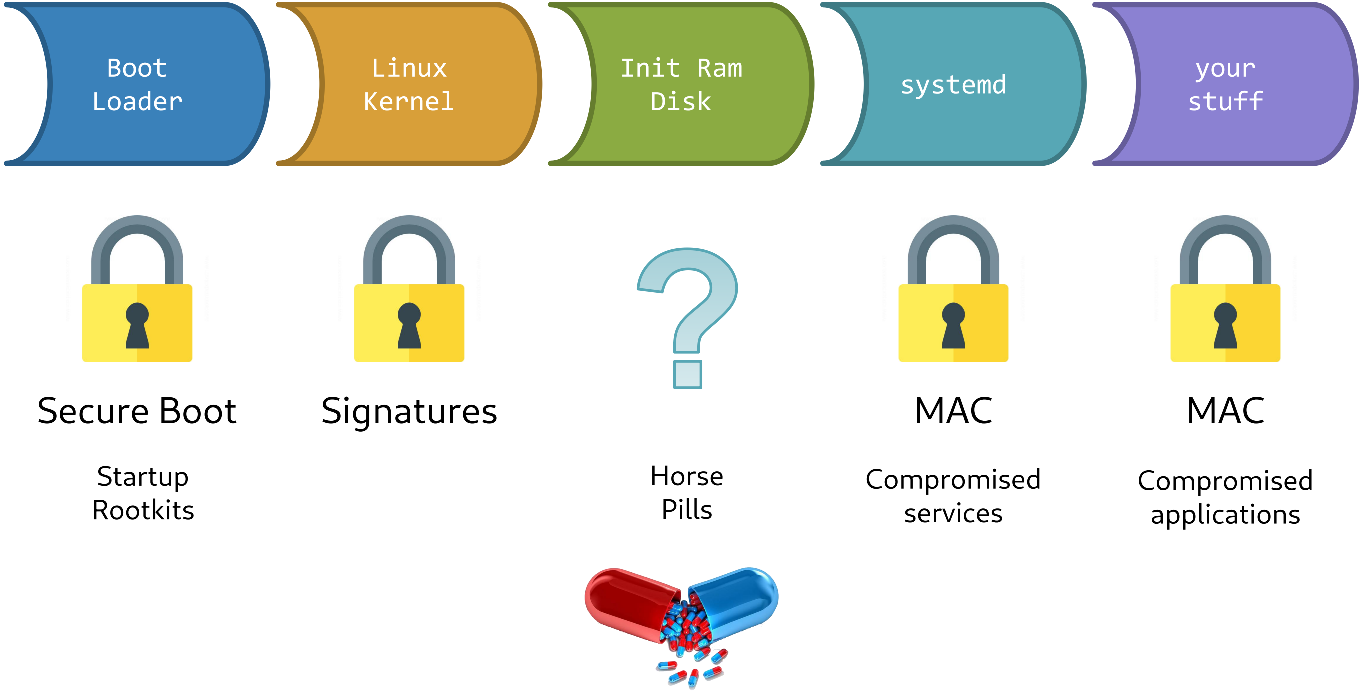
```
type_transition <current_type> <executable_file_type>:process <resultant_type>

type_transition user_t         passwd_exec_t:process         passwd_t
```

- A process with `user_t` label will change to `passwd_t` when executing a progra with the `passwd_exec_t` label

Boot Time Security

How your Computer Boots

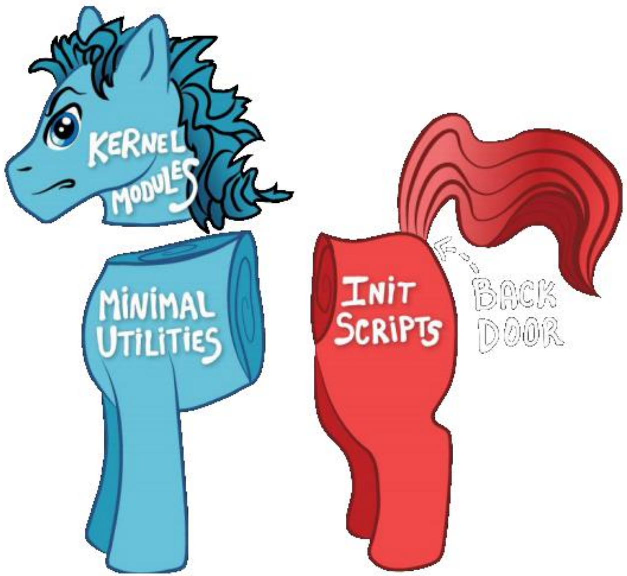


What your Ramdisk is Supposed to do?

- Load necessary modules
- Respond to hotplug events
- Cryptsetup (optional)
- Find and mount rootfs
- Clean up initrd
- Exec init

Horse Pills

- A boot-time attack which is based on init scripts loaded into a ramdisk and the usage of namespaces
- An infected ramdisk can easily take control of the machine



Horse Pills

- What an infected ramdisk could do:
  - load modules
  - cryptsetup
  - find and mount rootfs
  - enumerate kernel threads
  - **`clone(CLONE_NEWPID, CLONE_NEWNS)`**
  - remount root
  - mount scratch space
  - `fork()`
    - hook initrd updates
    - backdoor shell
  - `waitpid()`
  - shutdown/reboot

- remount /proc
- make fake kernel threads
- clean up initrd
- exec init

Mitigations?

- Detection
  - `/proc/<pid>/ns` links
  - Kernel therads proc entries (ppid != 0)
  - Audit
  - External examination
- How to prevent?
  - Do not assemble ramdisks on systems
  - Mostly unfeasible: it's how distros are able to target different machines!

OS-level Network Security

Address-based Service Enabling

- Based on the concept of Access Control List (ACL)
- Addresses of enabled users are explicitly specified
- It is useful for services exposed on a network
- An approach used in architectures such as:
  - super-servers:
    - **inetd**: the internet daemon
    - **xinetd**: the extended internet demon
  - TCP containers (e.g. **tcpd**)

UNIX inetd

- It controls services running on specific port numbers
- Upon connection or request arrival, it starts the actual target service and redirects sockets to stdout, stdin, stderr
- Association between port number and actual service has been based on the file `/etc/services`, with format:
    - ......
    - `ftp-data          20/tcp`
    - `ftp               21/tcp`
    - `telnet            23/tcp`
    - ......
- The **inetd** daemon was initially conceived as a means for resource usage optimization
- It has been then extended to cope with security

inetd Configuration

- Configuration information for **inetd** is typically kept by `/etc/inetd.conf`
- Each managed service is associated with one line structure as
  - Service name, as expressed in `/etc/services`
  - Socket type (e.g. stream)
  - Socket protocol (e.g. TCP)
  - Service flag (wait/nowait) which determines the execution mode (concurrent or not)
  - The user id to be associated with the running service instance (e.g. root)
  - The executable file path (e.g. `/usr/sbin/telnetd`) and its arguments (if any)

xinetd Features

- It provides an extension of **inetd** relying on
  - Address based access control
  - Time frame based access control
  - Full log of run-time events
  - DOS prevention by putting limitation on
    - Maximum number of per-service instances
    - Maximum number of total server instances
    - Log file size
    - Per machine source-connections
- Its configuration file is `/etc/xinetd.conf`
- It can be generated relying on the PERL utility `xconv.pl`

TCP daemons: tcpd

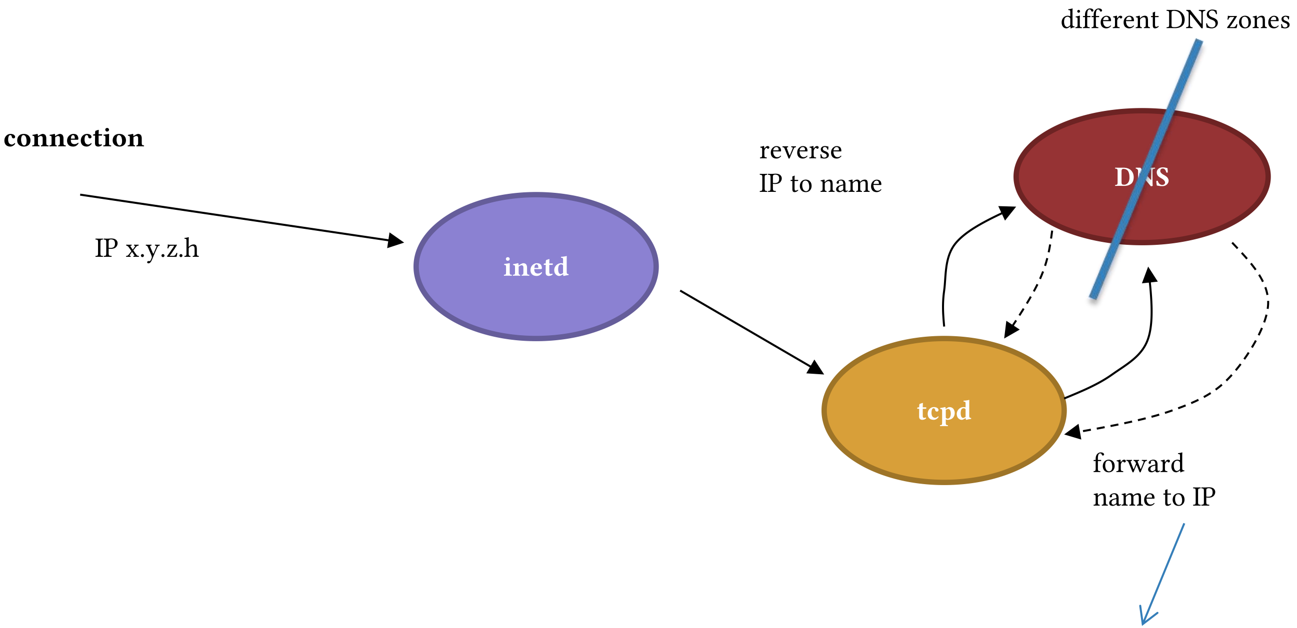
- The **tcpd** daemon wraps the services managed via **inetd**, so as to support access control rules
- **tcpd** is the actual server that is activated upon a request accepted by **inetd**
- **tcpd** receives as input the service specification
- Service management takes place by relying on rules coded in `/etc/hosts.deny` and `/etc/hosts.allow`
- Here we can find the specification of allowed or denied sources for a given service
- Each line is structured as `daemon_list : client_list`
- `ALL` is used to identify the whole set of managed services and all the hosts
- An example (access to all **inetd** services allowed from the local host):

```
# /etc/hosts.allow
ALL:  127.0.0.1
```

Reverse DNS tampering

- Usually host/domain specification occurs via symbolic names, rather than IP addresses
- Upon receiving a request/connection, **tcpd** checks with the source IP and executes a *reverse DNS* (rDNS) query to get the symbolic name of the source host
- An attacker can tamper with the reverse DNS query so as to reply with an allowed host/domain name
- To cope with this attack, **tcpd** typically performs both forwards DNS and reverse DNS queries so as to determine whether there is matching

An example scheme



if equal to x.y.z.h access is granted, otherwise it is not

DNS Zones

- DNS is defined in zones
- The owner of a zone maps different addresses to different domain names in their zone
- writing www.example.com accesses the example.com zone, and associates via an A (alias) record the hostname www to a certain IP
- rDNS is based on the PTR record
- a PTR record is stored in a special zone called .in-addr.arpa. This zone is adminstrated by whoever owns the block of IP addresses.