
Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Reference Monitor for Linux

• LSM provides a reference monitor interface for Linux
  ‣ Complete Mediation

• You need a module and infrastructure to achieve the other two goals
  ‣ Tamperproofing
  ‣ Verifiability

• SELinux is a comprehensive reference validation mechanism aiming at reference monitor guarantees
SELinux History

• Origins go back to the Mach microkernel retrofitting projects of the 1980s
  ‣ DTMach (1992)
  ‣ DTOS (USENIX Security 1995)
  ‣ Flask (USENIX Security 1999)
  ‣ SELinux (2000-…)

• Motivated by the security kernel design philosophy
  ‣ But, practical considerations were made
Inevitability of Failure

• Philosophy of the approach

• Flawed Assumption:
  ‣ That security can be managed by the application space without OS security support (*protection is not sufficient*)

• Paraphrase: Can’t build a secure system without a reference monitor and MPS
  ‣ And a secure operating system needs an entire ecosystem

• Come back to this later…
The Rest of the SELinux Story

• **Tamperproof**
  ‣ Protect the kernel
  ‣ Protect the trusted computing base
  ‣ *Use MPS to provide tamperproofing of TCB?*

• **Verifiability**
  ‣ Code correctness
  ‣ Policy satisfy a security goal
  ‣ *Use MPS to express secrecy and integrity requirements*
Design MPS

• Do not believe that classical integrity is achievable in practice
  ‣ Too many exceptions
  ‣ Commercial systems will not accept constraints of classical integrity

• Instead, focus on providing comprehensive control of access aiming for
  ‣ Confining root processes (tamperproof)
  ‣ Least privilege in general (verifiability)

• How does ‘least privilege’ affect security?
SELinux Policy Model

- See slides in 07-TypeEnforcement…
SELinux Policy Rules

- SELinux Rules express an MPS
  - Protection state – ALLOW subject-label object-label ops
  - Labeling state – TYPE_TRANSITION subject-label object-label new-label (at create – objects)
    - Default is to label to same state as creator
  - Transition state – TYPE_TRANSITION subject-label object-label new-label (at exec – processes)

- Tens of thousands of rules are necessary for a standard Linux distribution
  - Protect system processes from user processes
  - User data can be protected by MLS
SELinux “Setuid”

- How does SELinux enable a normal user to run a privileged (setuid) process, such as `passwd`?
SELinux Transition State

- For user to run passwd program
  - Only passwd should have permission to modify /etc/shadow
- Need permission to execute the passwd program
  - `allow user_t passwd_exec_t:file execute` (user can exec /usr/bin/passwd)
  - `allow user_t passwd_t:process transition` (user gets passwd perms)
- Must transition to passwd_t from user_t
  - `allow passwd_t passwd_exec_t:file entrypoint` (run w/ passwd perms)
  - `type_transition user_t passwd_exec_t:process passwd_t`
- Passwd can the perform the operation
  - `allow passwd_t shadow_t:file {read write}` (can edit passwd file)
SELinux Deployment

- You’ve configured your SELinux policy
  - Now what is left?
- Surprisingly, a lot
  - Many services must be aware of SELinux
  - Got to get the policy installed in the kernel
  - Got to manage all this policy
- And then there is the question of getting the policy to do what you want
User-space Services

- What kind of security decisions are made by user-space services?
User-space Services

• What kind of security decisions are made by user-space services?
  ‣ Authentication (e.g., sshd)
  ‣ Access control (e.g., X windows, DBs (servers), browsers (middleware), etc.)
  ‣ Configuration (e.g., policy build and installation)

• Also, many services need to be aware of SELinux to enable usability
  ‣ E.g., Listing files/processes with SELinux contexts (ls/ps)
User-space Services

- **Authentication**
  - Various authentication services need to create a “SELinux subject context” on a user login
  - Like login in general, except we set an SELinux context and a UID for the generated shell

- How do you get all these ad hoc authentication services to interact with SELinux?
Authentication for SELinux

- Pluggable Authentication Modules
  - There is a module for SELinux that various authentication services use to create a subject context
User-space Services

- **Access Control**
  - Many user-space services are shared among mutually untrusting clients
    - Problem: service may leak one client’s secret to another
  - If your SELinux policy allows multiple, mutually untrusting clients to talk to the same service, what can SELinux do to prevent exploits?
User-space Services

- Add SELinux support to the service
  - X Windows, postgres, dbus, gconf, telephony server
- E.g., Postgres with the SELinux user-space library
User-space Services

- **Configuration**
  - You need to get the SELinux policy constructed and loaded into the kernel
    - Without allowing attacker to control the system policy
    - And policy can change dynamically

- How to compose policies?
- How to install policies?
Compose Policies

- The SELinux policy is modular
  - Although not in a pure, object-oriented sense
  - Too much had been done
- **Policy management system** composes the policy from modules, linking a module to previous definitions and loads them
Installing Policies

- How would you enable user-space processes to push data (e.g., MPS configuration) into the kernel?
sysfs Background

• During the 2.5 development cycle, the Linux driver model was introduced to fix several shortcomings of the 2.4 kernel:
  ‣ No unified method of representing driver-device relationships existed.
  ‣ There was no generic hotplug mechanism.
  ‣ procfs was cluttered with lots of non-process information.

• Main uses
  ‣ Configure drivers
  ‣ Export driver information
sysfs Example: load_policy

From kernel: security/selinux/selinuxfs.c

enum sel_inos {
    SEL_ROOT_INO = 2,
    SEL_LOAD,   /* load policy */
    SEL_ENFORCE, /* get or set enforcing status */
}

static struct tree_descr selinux_files[] = {
    [SEL_LOAD] = {"load", &sel_load_ops, S_IRUSR|S_IWUSR},
    [SEL_ENFORCE] = {"enforce", &sel_enforce_ops,
                      S_IRUGO|S_IWUSR},

    static struct file_operations sel_load_ops = {
        .write = sel_write_load,
    };
};
sysfs Example: load_policy

From userspace: libselinux/src/load_policy.c

```c
int security_load_policy(void *data, size_t len)
{
    char path[PATH_MAX];
    int fd, ret;

    snprintf(path, sizeof path, "%s/load", selinux_mnt);
    fd = open(path, O_RDWR);
    if (fd < 0)
        return -1;

    ret = write(fd, data, len);
    close(fd);
}
```
sysfs Example: load_policy

From kernel: security/selinux/selinuxfs.c

```c
static ssize_t sel_write_load(struct file * file, const char __user * buf,
    size_t count, loff_t *ppos)
{
    ... 

    length = task_has_security(current, SECURITY__LOAD_POLICY);
    if (length)
        goto out;
    ... 

    if (copy_from_user(data, buf, count) != 0)
        goto out;
    length = security_load_policy(data, count); --- ss/services.c
    if (length)
        goto out;
```

When Are We Done?

- There is a significant configuration effort to get the SELinux system deployed
  - Who does this?
  - What happens if I want to change something?
  - Does it prevent the major threats?
Take Away

- **Problem**: Turn the SELinux policy into a working, usable reference monitor
  - Work with user-space services
  - Design the policy that you want

- There are many requirements for user-space services to provide authentication, access control, and policy configuration itself
  - PAM, Policy Mgmt, User-space access, Network support

- Design of MPS can only be semi-automated
  - Prevent network threats and design for app integrity