Advanced Systems Security: Security-Enhanced Linux

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

March 4, 2010

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

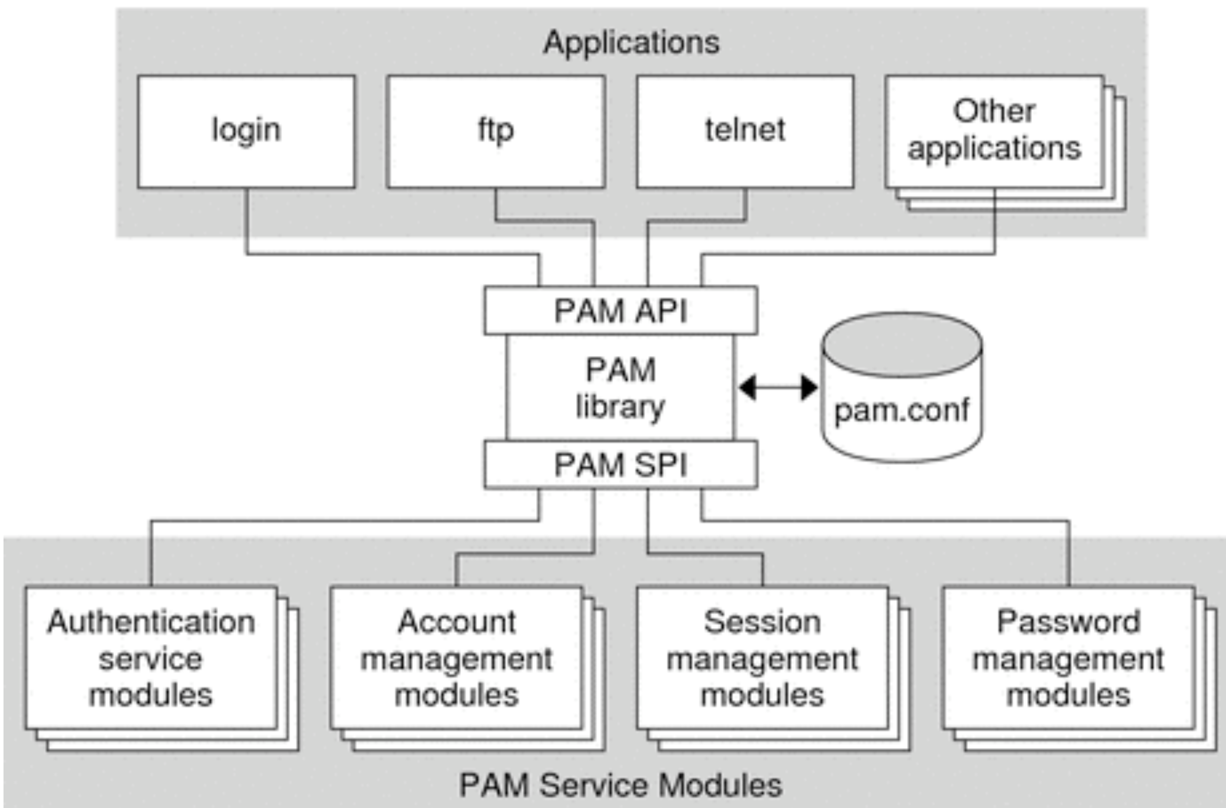
- What kind of security decisions are made by user-space services?
  - Authentication (e.g., sshd)
  - Access control (e.g., X windows, DBs, etc)
  - Configuration (e.g., policy build and installation)
- Also, many services need to be aware of SELinux to enable usability
  - E.g., Listing files/processes with SELinux contexts (ls/ps)

User-space Services

- Authentication
  - Various authentication services need to create a 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

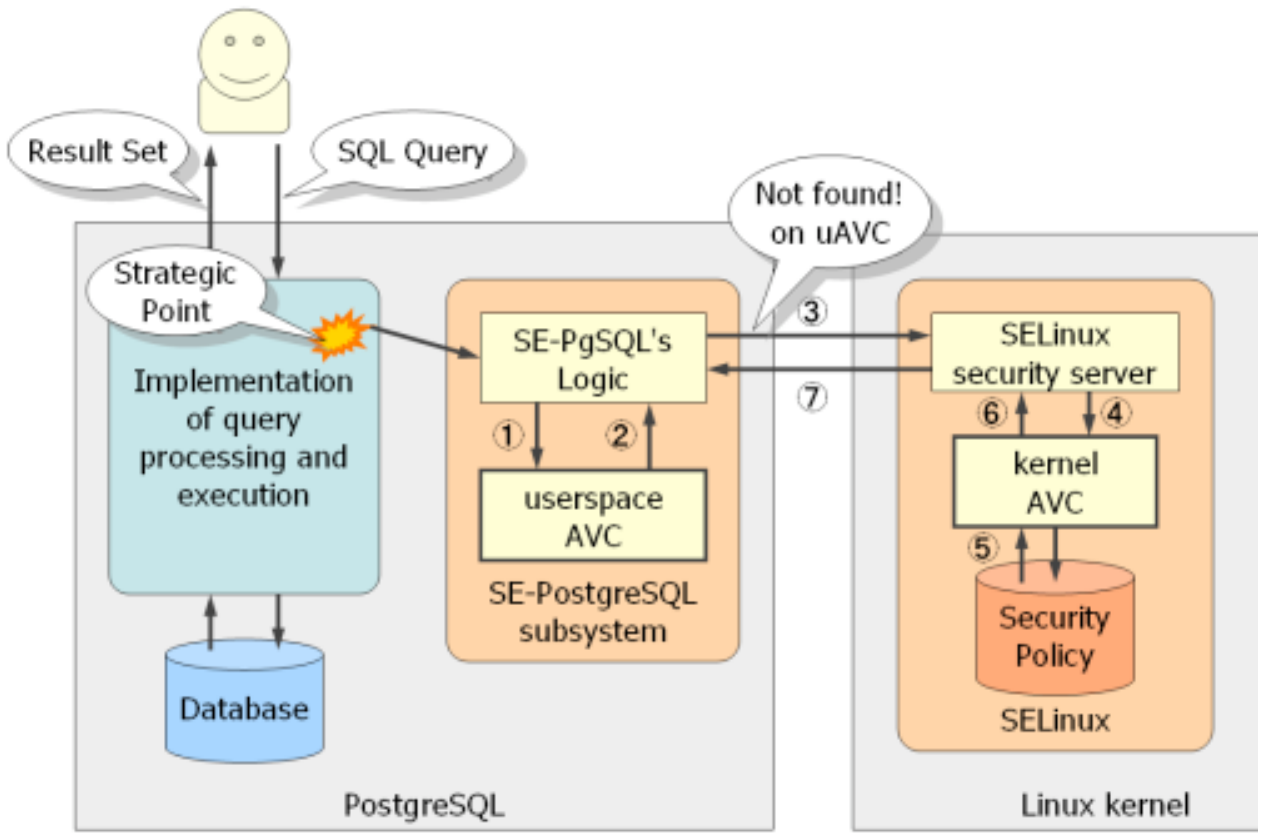


User-space Services

- Access Control
  - Many user-space services are shared among clients of different security
    - Problem: service may leak one client's secret to another
- If your SELinux policy allows multiple clients with different security requirements to talk to the same service, what can you do?

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

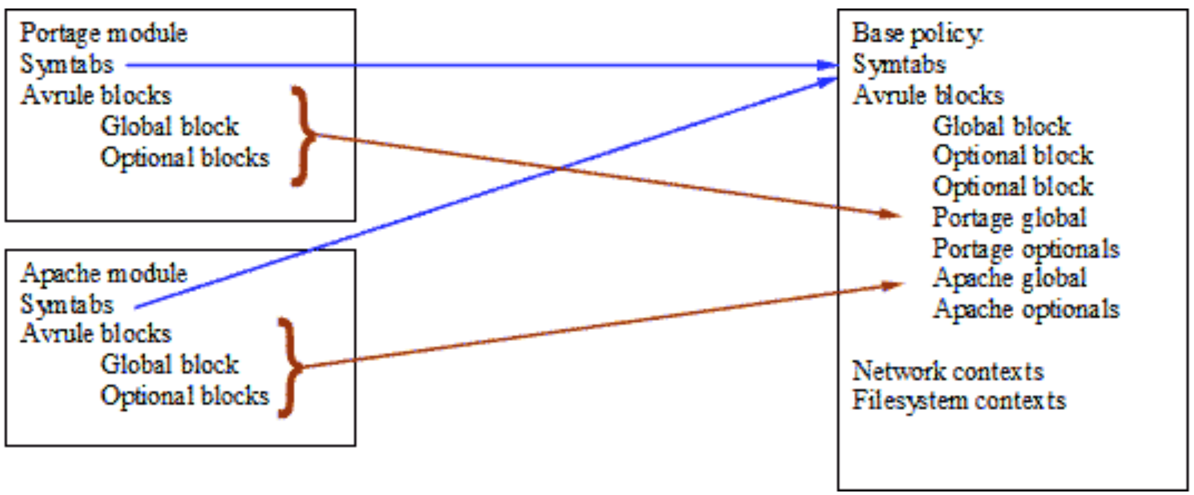


User-space Services

- 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

- sys_security system call rejected
  - Linux maintainers do not want to add system calls
  - The use of a void* input to the kernel will not be allowed
- Alternatives
  - */proc*
    - Supposed to be process-specific
  - *sysfs* -- special files for I/O with 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 userspace: libselinux/src/load_policy.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

```
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 kernel: security/selinux/selinuxfs.c

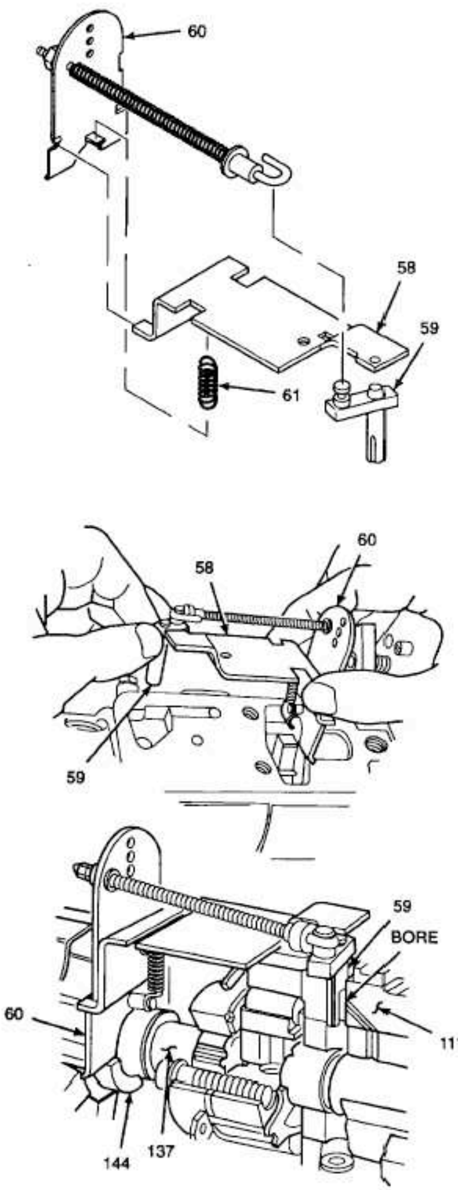
```
static ssize_t sel_write_load(struct file * file, const char __user * buf,
                              size_t count, loff_t *ppos)
{
…

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?



Threat: Remote Attackers

- How do we design policies if our threat is remote attackers?

"On the Internet, nobody knows you're a dog."

Goal: Confine Network Daemons

- Motivation for AppArmor the other major LSM (supported by SuSE and other Linux versions)
  - SELinux targeted policy has same aim
- Goal: keep a compromised daemon from compromising the system
- Challenge: some daemons must be trusted (e.g., SSH, DNS, DHCP)
- Result: Chen, Li, and Mao (NDSS 2009) found that AppArmor and SELinux (targeted) have attack paths from network daemons (SELinux has more)

Threat: Protect System Integrity

- How do we design policies to protect the system's trusted computing base?

Goal: Methodology to Find TCB

- Take the SELinux Example Policy and customize for the particular site (a security target)
- Goal: Find a trusted computing base from those processes in the trust model
- Challenge: Many policy rules allow interaction of trusted and untrusted processes
- Result: Develop a methodology for customizing a policy, but some leaps of faith result

SELinux Example Policy

- For Each Target Application
  - Definer has a threat model in mind
  - Definer specifies policy against that model
  - Definer and others test that the application runs given that policy
- For System
  - Aggregate of application policies
  - No coherent threat model
  - Application interactions not examined in detail

Experiment: Find SELinux TCB

- *Can we identify a TCB in SELinux Example Policy whose integrity protection can be managed?*
  - (1) Propose a TCB
  - (2) Identify Biba integrity violations
  - (3) "Handle" integrity violations
    - Classify integrity violations
    - Remove violations that can be managed (TP)
    - Revise TCB proposal
    - Revise SELinux policy

Propose a TCB

- Can use transition state graph (exec) to server programs (httpd_t) to identify base subject types
- Ones that provide TCB services (e.g., authentication)
- Others that have many transitions (hard to contain)

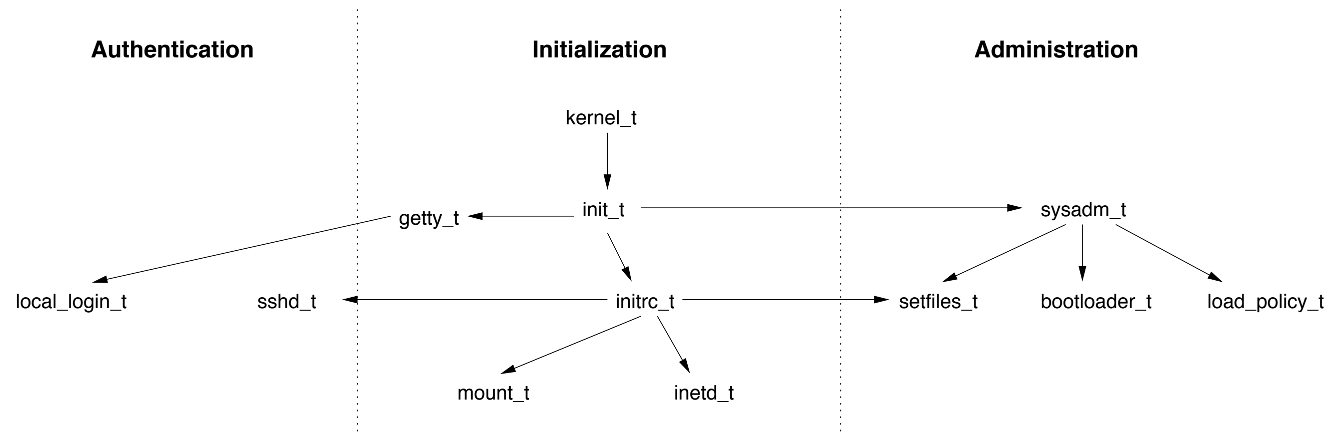


Figure 2: SELinux Example Policy's type transition hierarchy for our proposed TCB subject types.

Identify Integrity Violations

- Biba Integrity Analysis
  - TCB subject types → read/exec perms
    - Generate corresponding "integrity-sensitive write" perms
  - Others → write perms
    - Generate corresponding "integrity-sensitive read" perms
  - Analysis
    - Do Others' write to integrity-sensitive writes?
    - Do TCB subjects read using any integrity-sensitive read perms?
    - Equivalent, so do the more efficient

Are There Integrity Violations?

- Permissions
  - 129 perms used to “read down”
    - 57 socket perms, 25 fifo perms
  - 1583 perms used to “write up”
- Subjects
  - 28 high integrity subjects “read down”
    - 35 for sysadm_t, 4 for load_policy_t
  - 150 low integrity subjects “write up”

Example Conflicts

- Generic conflicts
  - Processes read from sockets
  - Processes read fifos
  - Trusted subjects are given broad access
- Specific issues
  - Files: /tmp, /etc, /etc/resolv.conf, /var
  - Logs: logfiles, backups
  - Others: ttys, devices

Classify Integrity Violations

- Classify Based on Possible Resolutions
  - Type classifications:
    - Upgrade low subject type to trusted – "should be trusted"
    - Exclude low subject type – "troublemaker"
    - Downgrade trusted subject type – "not possible to trust"
    - Exclude conflicting object type – "troublemaker objects"
  - Permission classifications:
    - Sanitize perm use (allow) – "filtering permissions"
    - Deny access to conflicting perms (deny) – "troublemaker perms"
    - Modify policy – "major surgery"

Classification Approach

- Reclassify/Exclude Subject Types
  - Exclude/trust writeup subjects that conflict with many readdown perms (sendmail)
  - Downgrade readdown subjects that conflict with many writeup perms
- "Sanitize" Readdown Perms
  - Read-write integrity vs read-only integrity
  - Small number of readdown subjects (fifos)
  - Assess permission type/use (sockets)

Classification Approach (con't)

- Exclude Conflicting Writeup Objects
  - Writeup perm that impacts several readdown perms
  - Remove excluded subject type perms (*)
- Deny Conflicting Writeup Perms
  - Find conflicting perm between readdown and writeup
    - Broad readdowns (user files, all files, …)
  - Test if can be denied
- Change The Policy
  - When all else fails…

Example Classifications

- Generic conflicts
  - Sanitize: Processes read from sockets
  - Sanitize: Processes read fifos
  - Deny: Broad access that conflicts
- Specific issues
  - Sanitize: /var, logfiles, backups
  - Exclude subjects: /etc, /etc/resolv.conf
  - Deny/change: sysadm_t, httpd_t
  - False: /tmp directory

Results

- 30 Trusted Subject Types
  - Obvious: kernel_t, init_t, getty_t, …
  - Admin: sysadm_t, load_policy_t, setfiles_t,
  - Auth: sshd_t, sshd_login_t, …
  - Less obvious: apt_t, hwclock_t, ipsec, cardmgr, …
- SELinux Core Subject Types (policy)
- 25 Excluded Subject Types (more since then)
- 4 Excluded Object Types (removable_dev)

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
- Turn a set of app policies into a coherent system
  - Prevent network threats and design for app integrity