Securing Linux

Hyungjoon Koo and Anke Li
Outline

• Overview
  – Background: necessity & brief history
  – Core concepts

• LSM (Linux Security Module)
  – Requirements
  – Design

• SELinux
  – Key elements
  – Security context: identity (SID), role, type/domain

• AppArmor
  – Key elements
  – Application policy profile

• SELinux vs AppArmor
Why a new access control model

• Limited traditional access control for Linux
  – Discretionary Access Control (DAC)
    • Provide only a coarse access control
    • 9 bits model (rwx per owner, group and others)
    • Has setuid, setgid and sticky bit - not enough

• Cases when a fine-grained access control needs
  – Does passwd require root access to printers?
  – Suppose I have a secret diary and the app to read it
    • Can I restrict my app from reading/writing a socket over network?
  – Alice might have multiple roles
    • Surfing the web, writing a report, and managing a firewall
Brief history

• Increasing the demand for reference monitor in Linux
  – A mechanism to enforce access control
  – Originate from orange book from the NSA: too generic

• Adopting LSM in Linux Kernel
  – Originally a set of kernel modules in 2.2, updated in 2.4
  – LSM (Linux Security Module) Feature in 2.6
    • SELinux developed by the NSA and released in 2001
    • Default choice for Fedora/RedHat Linux

• Lots of early works
  – Subdomain (AppArmor), Flask (SELinux), OpenWall, ...
Reference monitor

- A component that authorizes access requests at the RMI defined by individual hooks which invokes module to submit a query to the policy store

From Operating System Security (Fig 2.3)
Core concepts

• Idea: Define policies to decide if applications/users have the privilege to proceed a given operation
  – MAC: Mandatory access control
  – Least Privileges

• Broadly covered security policy
  – To all subjects, all objects and all operations
  – As everything in Linux is represented as a FILE
    • files, directories, devices, sockets, ports, pipes, and IPCs
Linux Security Module (LSM)

• Implementation of a reference monitor
• Requirements
  – Modularized security
  – Loadable modules
  – Centralized MAC
  – LSM API
LSM design

• Definition
  – How to invoke permission check?
    • By calling the initiated function pointers in `security_ops`
    • Aka LSM hooks
  – One hook is shown below:

```c
static inline int security_inode_create (struct inode *dir,
                                        struct dentry *dentry,
                                        int mode)
{
    if (unlikely (IS_PRIVATE (dir)))
        return 0;
    return security_ops->inode_create (dir, dentry, mode);
}
```

• Placement
• Implementation
LSM design - hooking

- Simple diagram of hooking

```
<table>
<thead>
<tr>
<th>System call</th>
<th>Func ptr</th>
<th>Hook to LSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>0xffffaaaa</td>
<td>lsm_open(), 0xfffffbbb</td>
</tr>
<tr>
<td>read()</td>
<td>0xfffffaaba</td>
<td>lsm_read(), 0xfffffbbcb</td>
</tr>
<tr>
<td>write()</td>
<td>0xfffffaaca</td>
<td>lsm_write(), 0xfffffbbdb</td>
</tr>
<tr>
<td>getdents()</td>
<td>0xfffffaada</td>
<td>lsm_getdents(), 0xfffffbbbeb</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

LSM
- lsm_open()
- lsm_read()
- lsm_write()
- lsm_getdents()
LSM design

• Definition

• Placement
  – Where to place those hooks?
    • Entry of system call (not all of them)
    • Determined by source code analysis
  – Inline function
    • E.g., security_inode_create

• Implementation
LSM design – hooking example

- **open()** hook process
  - Process *syscall* in user
    - file path
    - operation
  - Invoke *syscall* in kernel
  - Lookup *inode*
  - Check DAC
  - Hook & check MAC
  - Grant access

From *Operating System Security (Fig 9.2)*
LSM design

• Definition
• Placement

• Implementation
  – Where to find the function which hooks point to?
  – SELinux, AppArmor, LIDS, etc.
  – Does placement need to change in different LSMs?
    • Theoretically yes
    • Practically, the placement of hooks is stabilized
SELinux at a glance

• Security Policies
  – Centralized store for access control
  – Can be modified by the SELinux system administrator
  – Determined by security contexts (=user, role, type)
  – Specification of permissions
  – Labeled with information for each file

• Based on TE (Type Enforcement) and RBAC model

• Operations to objects for subjects
  – append, create, rename, rwx, (un)link, (un)lock, ...

• Object classes
  – file, IPC, network, object, ...
Some valid questions

• How can SELinux internally incorporate with DAC?
  – DAC then MAC

• Who writes the policy?
  – Admin

• Isn't it hard to write a policy?
  – Indeed, and complicated (for SELinux)

• What happens if there is wrong policy?
  – Hell

API names are admittedly confusing
Security context

• Consist of three security attributes
  – User identity (SID, Security identifier)
    • SELinux user account associated with a subject or object
    • Different from traditional UNIX account (i.e. /etc/passwd)
  – Type or domain
  – Role (RBAC)
Security context

• Consist of three security attributes
  – User identity (SID, Security identifier)
  – Type or domain
    • Postfix_t (i.e. user_t, passwd_t, shadow_t, ...)
    • Divide subjects and objects into related groups
    • Typically type is assigned to an object, and domain to a process
    • Primary attribute to make fine-grained authorization decisions
  – Role (RBAC)
Security context

• Consist of three security attributes
  – User identity (SID, Security identifier)
  – Type or domain
  – Role (RBAC)
    • Postfix _r (i.e. sysadm_r, user_r, object_r, ...)
    • User might have multiple roles
    • Associate the role with domains (types) that it can access
      • Not assign permissions directly
    • Limits a set of permission ahead of time
    • If role is not authorized to enter a domain then denied
Security context example

• Putting all together
  – Alice wants to change her password
    • SID alice with the user role, $user_r$
    • Role permitted to run typical user processes
    • Any process with $user_t$ to execute the $passwd_exec_t$ label

```
role user_r  types {user_t user_firefox_t}

<perm> <sub_type> <obj_type>:<obj_class> <op_set>
Allow user_t   passwd_exec_t:file  execute
Allow passwd_t shadow_t:file       {read write}

<file_path_expr> <obj_context>
/usr/bin/passwd  system_u:object_r:passwd_exec_t
/etc/shadow.*    system_u:object_r:shadow_t
```
Decision making with policy

- **Access decision**
  - Based on security context
  - *allow*, *auditallow*, *dontaudit*, and *neverallow*

- **Q: how can we decide policy for a temporary object?**
  - temp processes (i.e. fork) and files

- **A: transition decision**
  - Process creation: *domain* transmission
  - File creation: *type* transmission (labelling)

<table>
<thead>
<tr>
<th>type_transition</th>
<th>&lt;curr_type&gt;</th>
<th>&lt;exe_file_type&gt;:process</th>
<th>&lt;res_type&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>type_transition</td>
<td>user_t</td>
<td>passwd_exec_t:process</td>
<td>passwd_t</td>
</tr>
</tbody>
</table>
Transition decision examples

• Process creation
  – *Domain* decision

• File creation
  – *Type* decision

From *SELinux Ch2*
Implementation

• Policy sources
  – .te files (type enforcement)
    • Define rules and macros (m4) & assign permissions
  – .fc files (file context)
    • Define file contexts, supporting regular expression
  – RBAC files
  – User declarations

• Makefile (target: policy, install, ...)

• Policy compiler
  – Merge all policies to policy.conf
  – Generate policy binary, centralized policy storage
AppArmor at a glance

• Another mainstream of LSM implementation
• Much simpler framework than SE Linux
  – Targeted policy
  – An “application security system”
  – Pathname based
  – Work in two modes:
    • enforce mode and complain mode
  – One policy file per application
• Used by some popular Linux distributions
  – Ubuntu, openSUSE, etc.
How AppArmor works?

• Designed to be a complement to DAC
  – Can’t provide complete access control

• Born to be targeted policy
  – unconfined_t in SELinux

• Application based access control
  – One policy file per application
  – Protect system against applications

• File + POSIX capabilities
AppArmor profile

- **Capability rules:**
  ```
  capability setuid,
capability dac_override,
  ```

- **Network rules:**
  ```
  network (read, write) inet,
deny network bind inet,
  ```

- **File rules:**
  ```
  /path/to/file rw,
  /dir/** r,
  ```
SELinux vs AppArmor

- Whole system vs. only a set of applications
- Types & domains vs. defining permission directly
- Strict MAC implementation vs. Partially implement
- Extended attributes vs. pathname
- Difficulty to configure
  - SELinux needs 4x bigger conf. file than AppArmor
- Overhead?
  - 7% vs. 2%
Conclusion

• SELinux and AppArmor can both greatly enhance OS security.
• Choice depends on what you need.