SELinux Policy Concepts and Overview

Security Policy Development Primer for Security Enhanced Linux

(Module 3)
Access Control Attributes

- SELinux assigns subject and objects a security context:

```plaintext
root:sysadm_r:sysadm_t[:s0:c0.c128]
```

- Security context is only access control attribute in SELinux
- Security Identifier (SID): number represents security context active within the kernel
Standard Linux vs SELinux

- **Subject (Process) Access Control Attributes**
  - Linux: real and effective user and group IDs
  - SELinux: security context (user:role:type)
  - Linux UIDs and SELinux UID are independent

- **Objects Access Control Attributes**
  - Linux: (files) access modes (rwx r-x r-x) and user and group IDs
  - SELinux: security context (user:role:type)
More on Security Contexts

- Linux and SELinux access controls are orthogonal
  - each mechanism uses its own access control attributes
  - two separate access checks; both must pass
- A process type is also called a “domain”
  - though object and subject contexts are identical
- Role and user are little used on objects
  - objects’ role usually “object_r”
- Type is most used part of a context (by far) in policies
  - emphasis on type enforcement in a policy
What is a Type?

- A type is an unambiguous identifier
  - created by the policy writer
  - applied to all subjects and objects and for access decisions
- Types group subjects and objects
  - signifies security equivalence
  - everything with the same type has the same access
  - policies have as few or as many types as needed
- Type “meaning” created through use
  - e.g. shadow_t only has meaning because of a policy rules
  - similar to a programmer giving meaning to variables
Type Enforcement Access Control

- Access specified between
  - subject type (e.g., process or domain)
  - and object type (e.g., file, dir, socket, etc.)

- Four elements in defining allowed access
  - source type(s)  aka domain(s)
  - target type(s)  objects to which access allowed
  - object class(es)  classes to which access applies
  - permission(s)  type of access allowed

- SELinux prevents access unless explicitly allowed
Object Classes and Permissions

- SELinux defines 41 kernel object classes

  - association
  - blk_file
  - capability
  - chr_file
  - dir
  - fd
  - fifo_file
  - file
  - filesystem
  - ipc
  - key_socket
  - netlink_route_socket
  - netlink_selinux_socket
  - netlink_socket
  - netlink_tcpdiag_socket
  - netlink_xfrm_socket
  - node
  - packet_socket
  - passwd
  - process
  - rawip_socket
  - security
  - sem
  - shm
  - socket
  - tcp_socket
  - udp_socket
  - unix_dgram_socket
  - unix_stream_socket

  Each with their own fine-grained permissions
  - For example, file object class has 20 permissions:

    - ioctl
    - create
    - lock
    - append
    - rename
    - quotaon
    - entrypoint
    - read
    - getattr
    - relabelfrom
    - unlink
    - execute
    - mountron
    - execmod
    - write
    - setattr
    - relabelto
    - link
    - swapon
    - execute_no_trans

  Documentation available at www.tresys.com/selinux
passwd Program Example

allow passwd_t shadow_t : file
    { create ioctl read getattr lock write setattr append link unlink rename };

- Allows processes with passwd_t domain type read, write, and create access to files with shadow_t type
  - Purpose: passwd program runs with passwd_t type, allowing it to change shadow password file (/etc/shadow)

- Shadow password file attributes:

  - standard Linux
    - only root allowed to create new copies of file

  - SELinux
    - only allows passwd_t domain (via above allow rule) to modify file
passwd Program Example

uid: root
passwd_t

write, create, … (change password)

passwd

r-------- root root
shadow_t
/etc/shadow

allow passwd_t shadow_t : file
{ read getattr write setattr append };
Problem of Domain Transitions

uid: joe
euid: root
user_t

uid: joe
euid: joe
user_t

allow passwd_t shadow_t: file
{ read getattr write setattr append };

© 2002 - 2005 Tresys Technology, LLC (www.tresys.com'selinux, selinux@tresys.com)
Standard Linux passwd Security

uid: joe  
euid: joe  

login

fork()

bash → passwd

execve()

Anyone can execute

r-s--x--x root root  

/etc/shadow

/usr/bin/passwd
Standard Linux passwd Security

uid: joe
euid: joe

write, create, ...
(change password)

uid: joe
euid: joe

fork()

execve()

login

bash

/usr/bin/passwd

r-s--x--x root root

/etc/shadow

r-------- root root

/usr/bin/passwd
Standard Linux passwd Security

uid: joe
euid: joe

login
fork()
bash

execve()
set uid

rs--x--x root root
/usr/bin/passwd

r-------- root root
/etc/shadow
Standard Linux passwd Security

uid: joe
euid: root

uid: joe
euid: root

write, create, …
(change password)

fork()
execve()
set uid

dash

r-------- root root

/etc/shadow

login

/usr/bin/passwd

/log

set uid

/usr/bin/passwd

/met/etc/shadow
SELinux Domain Transitions

uid: joe
euid: joe
user_t

login

fork()

uid: joe
euid: joe
user_t

bash

r-s--x--x root root
passwd_exec_t

r------- root root
shadow_t

/etc/shadow

/usr/bin/passwd
SELinux Domain Transitions

uid: joe
euid: joe
user_t

uid: joe
euid: joe
user_t

bash

fork()

eexecve()

r------- root root
shadow_t
/etc/shadow

r-s--x--x root root
passwd_exec_t
/usr/bin/passwd

allow user_t passwd_exec_t : file { getattrib execute };
SELinux Domain Transitions

uid: joe
euid: root

passwd_t

uid: joe
euid: joe
type_transition user_t passwd_exec_t : process passwd_t;
allow passwd_t passwd_exec_t : file entrypoint;
allow user_t passwd_t : process transition;
type_transition user_t passwd_exec_t : process passwd_t;
allow user_t passwd_exec_t : file { getattr execute };

write, create, ...
(change password)

/etc/shadow

uid: joe
euid: root

bash

fork()

execve()

allow user_t passwd_t : process transition;
allow passwd_t passwd_exec_t : file entrypoint;
Type Transition Statement

- First form: default **domain transition**
  - Causes a domain type transition to be attempted on `execve()`

```c
type_transition user_t passwd_exec_t : process passwd_t;

source domain file type of executable object new type for process
(sourcedomain for executing process domain trans)
```

![Diagram](image.png)
Type Transition Statement

- type_transition specifies default transition
  - Does **NOT** allow it!
  - Successful domain trans. requires access allowed
    - original domain execute access to executable file
    - original domain permission to transition to new domain
    - new domain permission to be entered via program
    - others...

- Second form: default object types on creation
  - to be discussed in later modules
The Role of Roles

- Roles associates domains with users
  - further constrains process type transitions
    - process type allowed only if allowed by role definition
    - even if type enforcement allows it

- Role declaration statement

```plaintext
role user_r types passwd_t;
```

- role for which type is allowed
- allowed type
Roles in Domain Transitions

```
login

user_r:user_t

bash

fork()

execve()

r-s--x--x root root
passwd_exec_t

/etc/bin/passwd

write, setattr, ...
(change password)

user_r:passwd_t

passwd

role user_r types passwd_t;

r------ root root
shadow_t

/etc/shadow
```
Why Type Enforcement

- Extremely configurable mandatory access control
  - flexible (not tied to a single security objective)
  - dynamic (loadable/conditional policy)
  - possible to be pragmatic within a policy
    - even necessary due to Linux legacy!
  - fine-grained access control
    - object classes and permissions, unlimited types and rules
- Useful for a large number of security goals and objectives
Security Goals TE can Implement

- System integrity, RVM/kernel self-protection
  - raw devices and resources
  - kernel configuration and binary files (e.g., modules)
  - daemon/services configuration and binary files
  - protection of SELinux policy itself
- Application integrity
  - configuration and binary files
  - inter-process communication
- Least privilege
  - preventive security engineering design
  - protection of privileged user environments
Security Goals TE can Implement

- Controlled execution domains
  - isolation of untrusted code (e.g., sandboxes)
  - prevention of malicious code in trusted domains

- System Hardening
  - confinement of error propagation (exploitations)
  - fine-grained access control

- Domain isolation
  - trusted from untrusted
  - application from application

- Information flow policies
  - Multilevel security and multiple security levels
  - Guards and other cross-domain solutions
  - Perimeter defense
Challenges with SELinux TE

- Policies are usually complex
  - Due to complexity of Linux kernel
    - legacy issues with Linux/Unix
    - need for Pragmatism
- Flexibility comes with a price!
  - 41 kernel object classes, hundreds of permissions
  - thousands of object instances
  - unlimited domain and object types
- Assurance of mechanism evolving
  - open source model helps
  - certainly no worse than Linux (or other mainstream OSs)
  - in fact much better with a good TE policy
Policy Concept Overview Summary

- Standard Linux and SELinux access control mechanisms are orthogonal
- SELinux security context: user:role:type
  - applied to both objects and subjects
  - type is the primary means of controlling access
- Fine grained access control
  - 41 kernel object classes, hundreds of permissions
- Access must be explicitly allowed in TE policy
  - all access denied by default
Policy Concept Overview Summary

- TE allow statement:
  ```
  allow domain_type object_type: classes permission ;
  ```
  specifies allowed access based on types

- TE domain transition:
  - changing of process type (domain) on execve()
  - type_transition specifies default transition

- Type enforcement flexible
  - can implement many security properties

- Roles further constrain domain transitions
QUESTIONS?