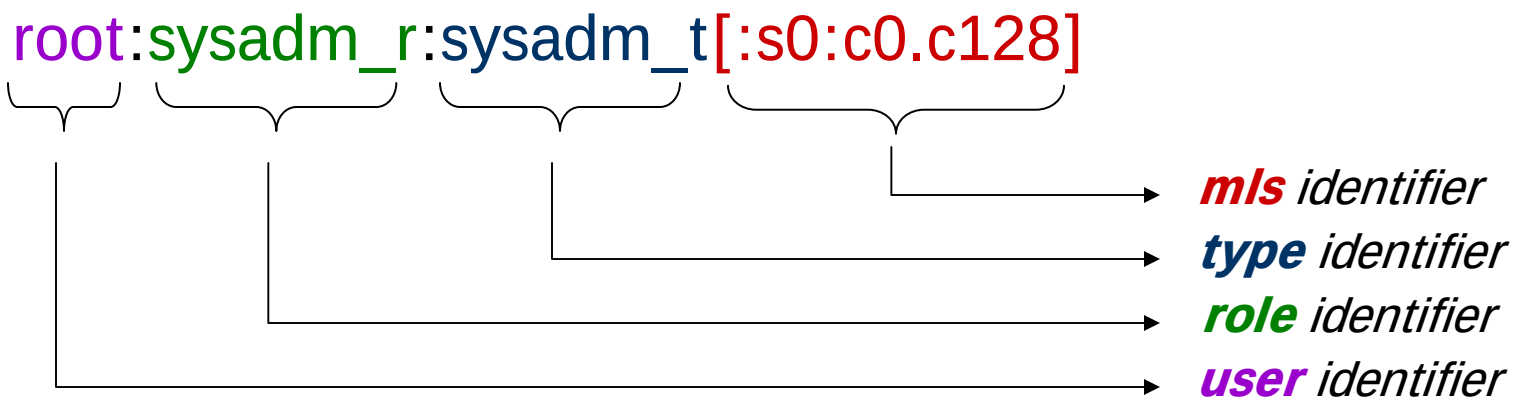
# SELinux Policy Concepts and Overview

## Security Policy Development Primer for Security Enhanced Linux

**(Module 3)**

# Access Control Attributes

- SELinux assigns subject and objects a security context:




- 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 | lnk_file | netlink_route_socket | security |
| blk_file | msg | netlink_selinux_socket | sem |
| capability | msgq | netlink_socket | shm |
| chr_file | netif | netlink_tcpdiag_socket | sock_file |
| dir | netlink_audit_socket | netlink_xfrm_socket | socket |
| fd | netlink_dnrt_socket | node | system |
| fifo_file | netlink_firewall_socket | packet_socket | tcp_socket |
| file | netlink_ip6fw_socket | passwd | udp_socket |
| filesystem | netlink_kobject_uevent_socket | process | unix_dgram_socket |
| ipc | netlink_nflog_socket | rawip_socket | unix_stream_socket |
| key_socket | | | |

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

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

- 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:

```
-r--------  root  root  system_u:object_r:shadow_t  /etc/shadow
```

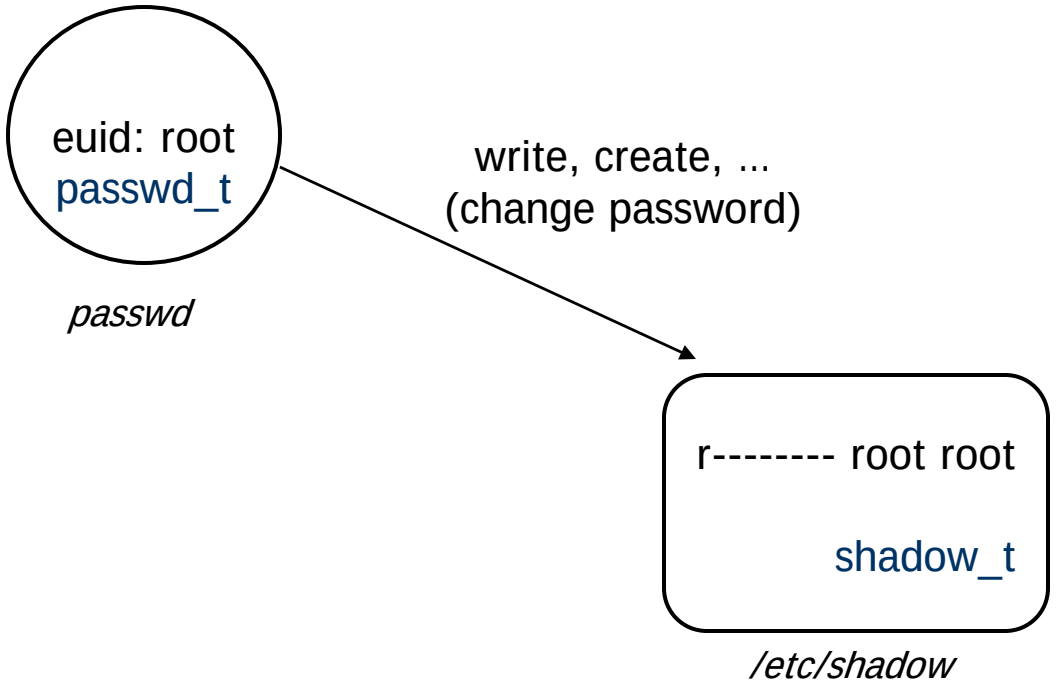
*standard Linux*

*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



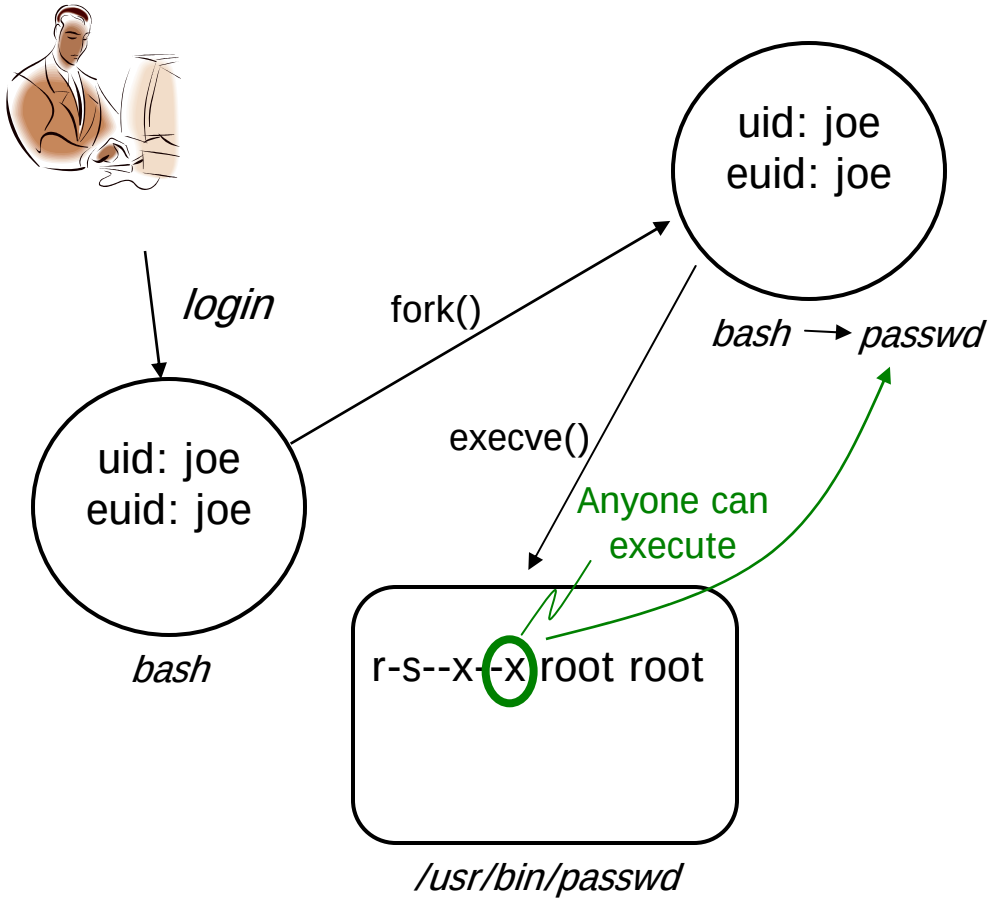

```
allow  passwd_t  shadow_t :  file
                { read getattr write setattr append };
```

# Problem of Domain Transitions

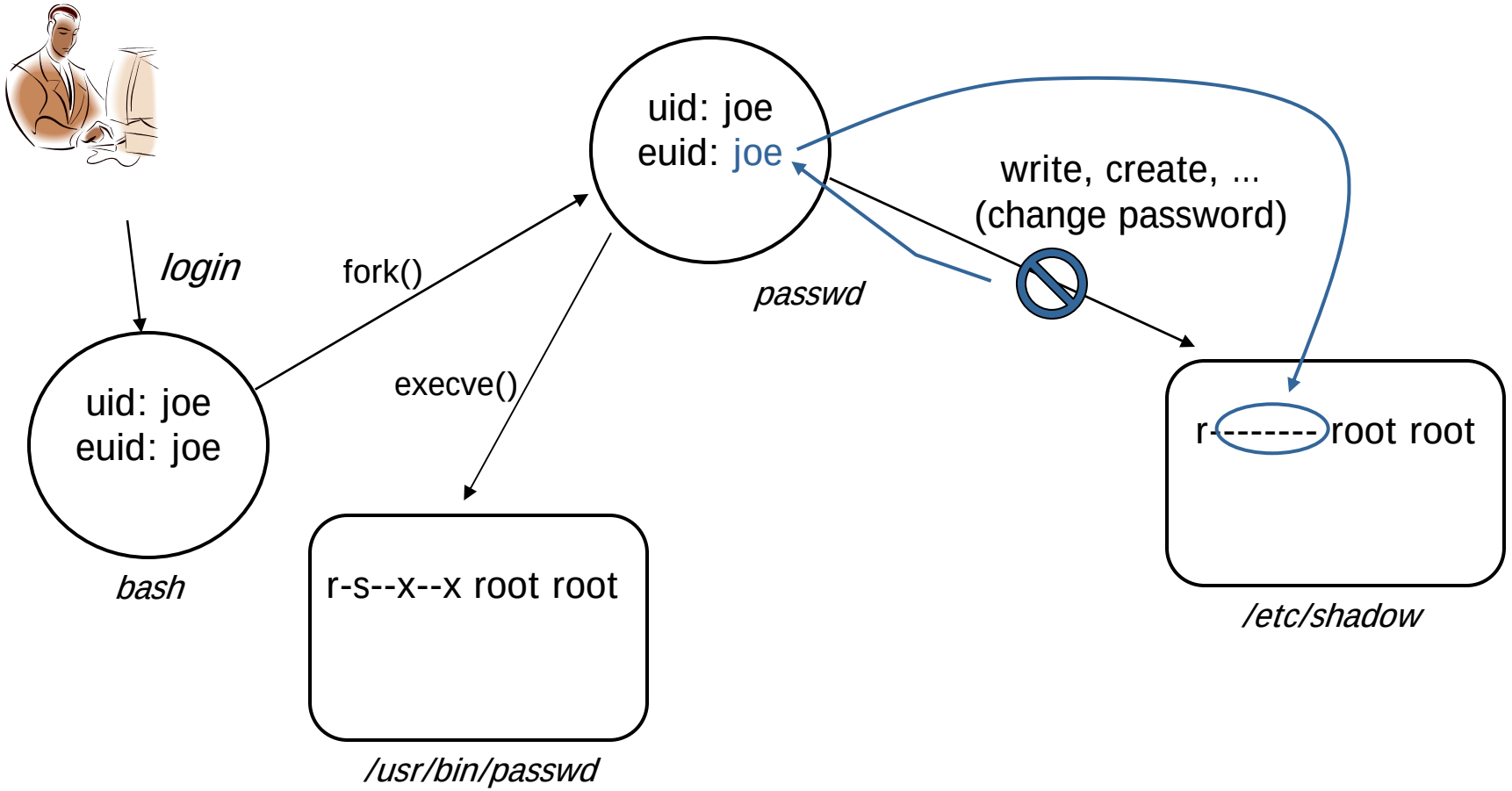



```
allow  passwd_t  shadow_t :  file
                { read getattr write setattr append };
```

# Standard Linux passwd Security



# Standard Linux passwd Security



# Standard Linux passwd Security

*login*

uid: joe
euid: joe

*bash*

fork()

uid: joe
euid: joe

*bash*

execve()

set uid

r-s--x--x root root

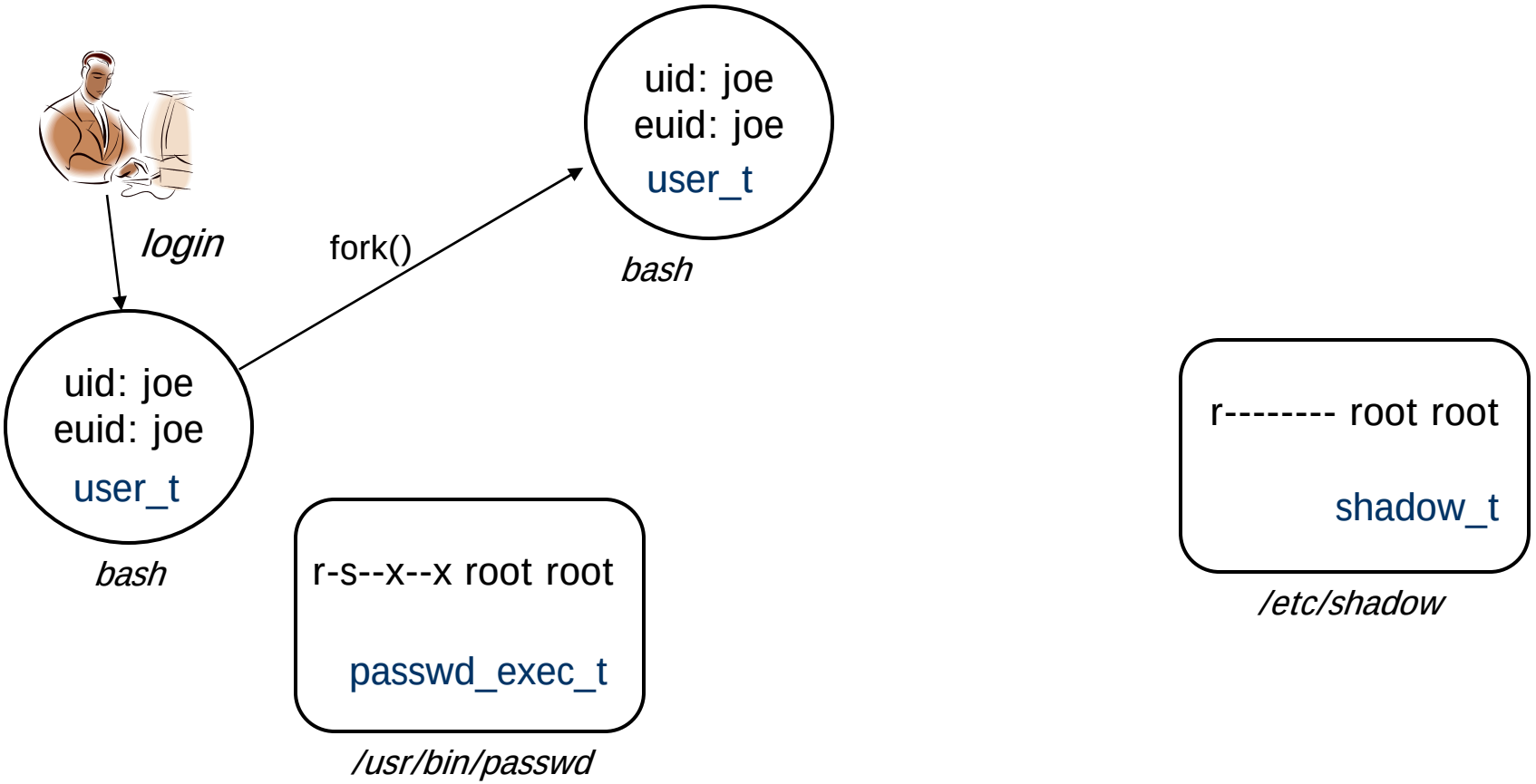
*/usr/bin/passwd*

r-------- root root

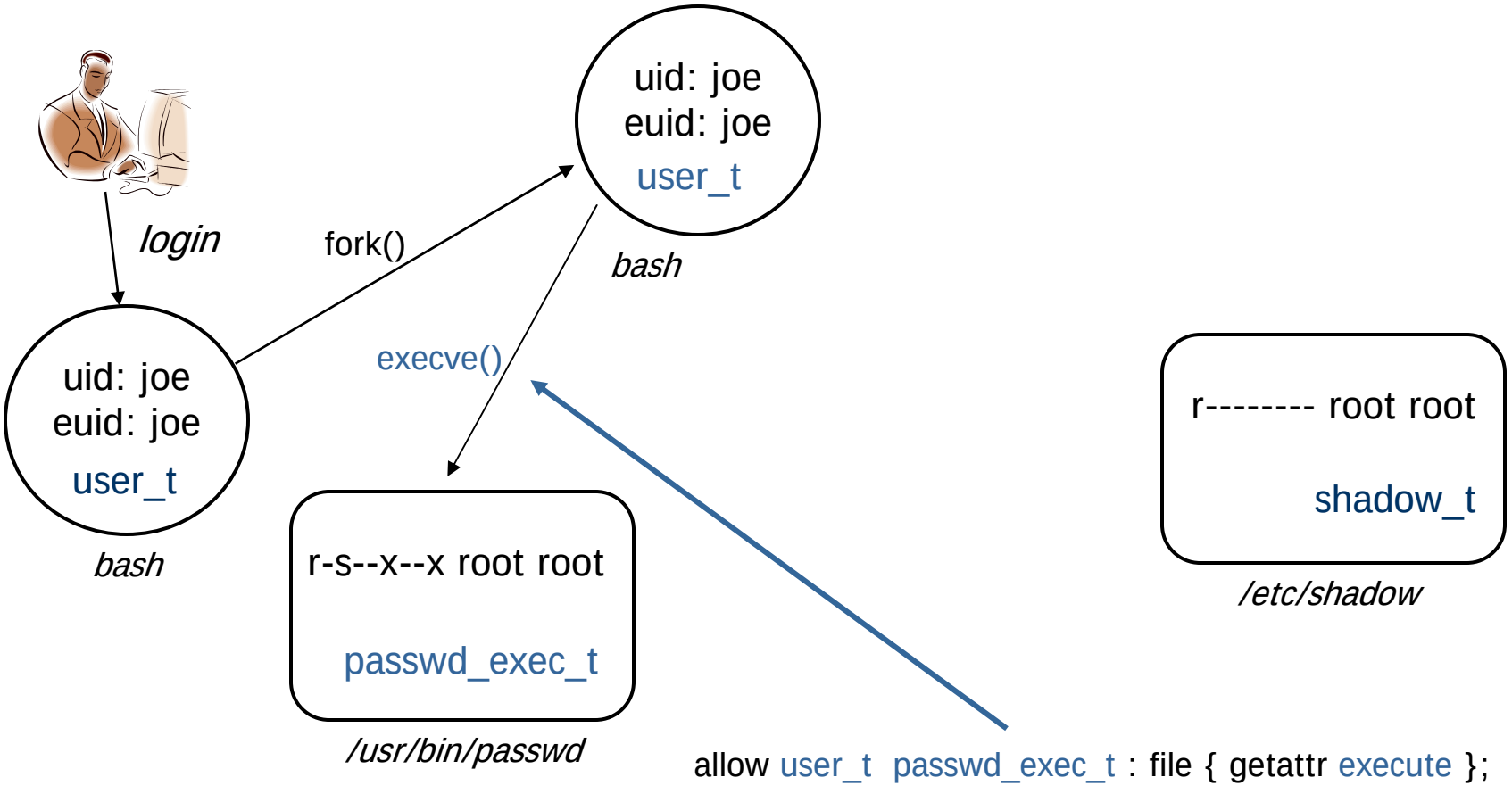
*/etc/shadow*

# Standard Linux passwd Security

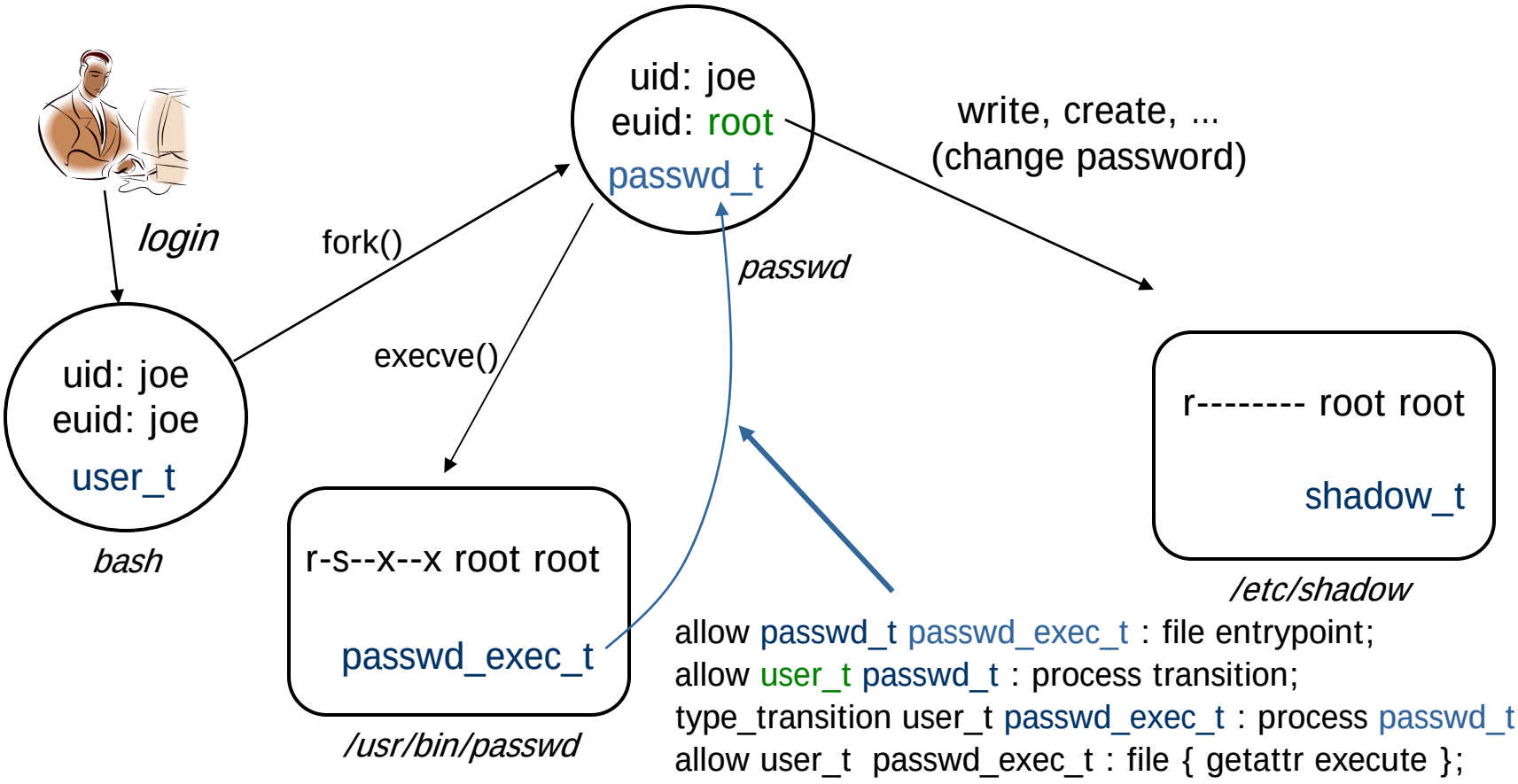
# SELinux Domain Transitions



# SELinux Domain Transitions



# SELinux Domain Transitions

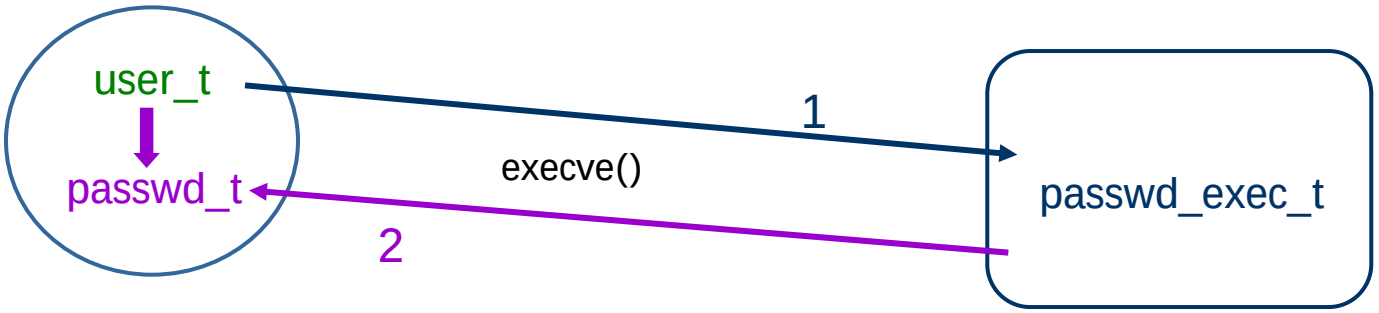


# Type Transition Statement

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

type_transition user_t passwd_exec_t : process passwd_t;

*source domain* — *file type of executable* — *object (process for domain trans)* — *new type for process*



# 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

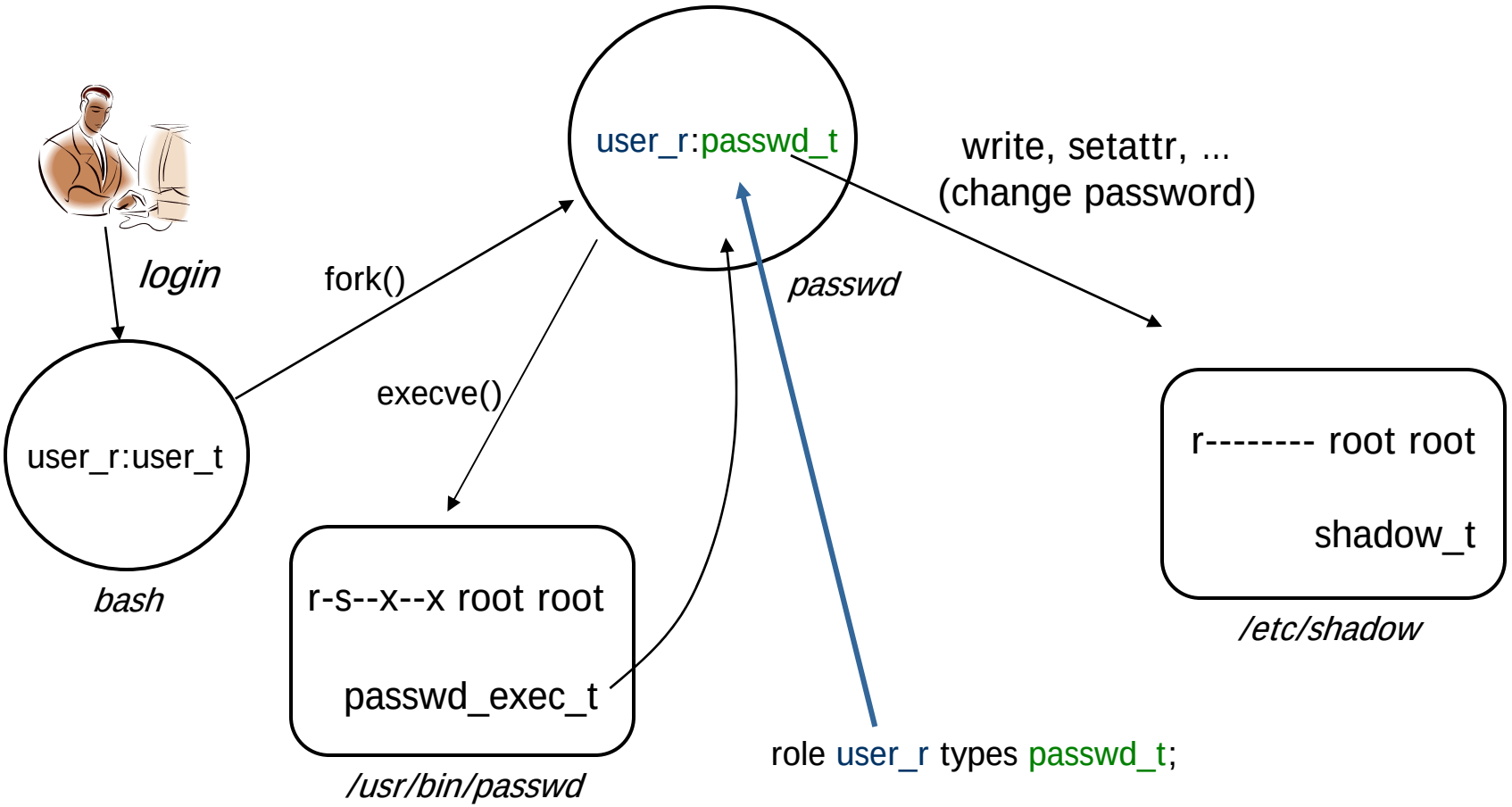
- Roles associates domains with users
  - further constrains process type transitions
    - process type allowed only if allowed by role definition
    - even if type enforcement allows it
- Role declaration statement

role user_r types passwd_t;

*role for which type is allowed*

*allowed type*

# Roles in Domain Transitions



# 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?