CSCI 4907/6545 Software Security

Fall 2026

Instructor: Jie Zhou

Department of Computer Science

George Washington University



Slides materials are partially credited to John Criswell of the University of Rochester.

Least Privilege Principle

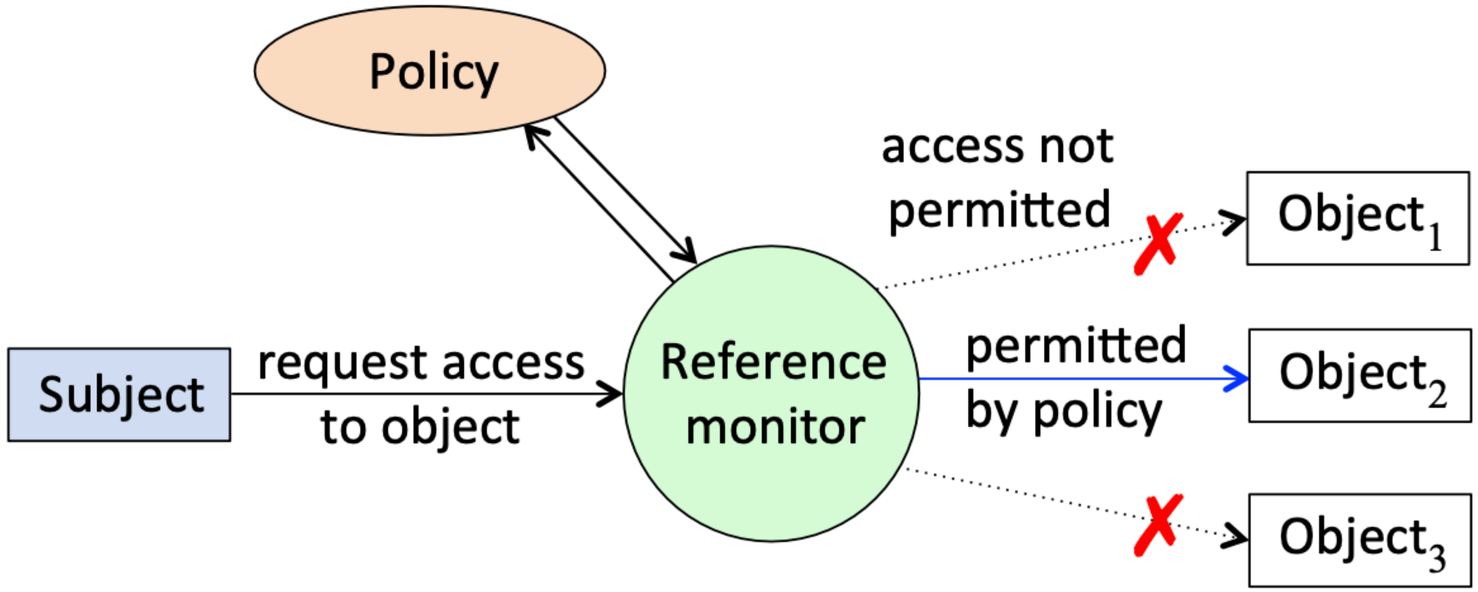
Outline: Principle of Least Privilege

- What are privileges?
- What problems do current systems have with privileges?
- What can we do to more safely use privileges?

At a given point in time, *what operations* are allowed on *which object*?

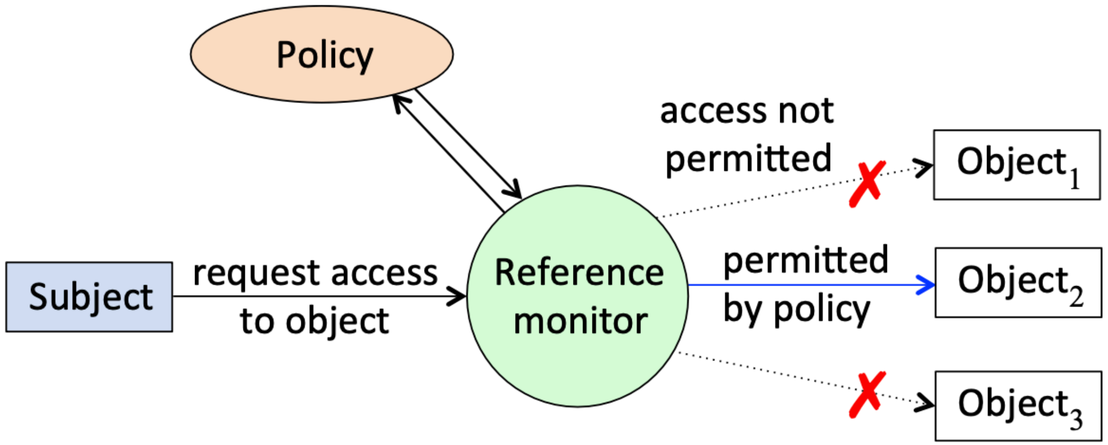
Reference Monitor

- All references by any program to objects (data, devices, etc.) are validated against a policy.



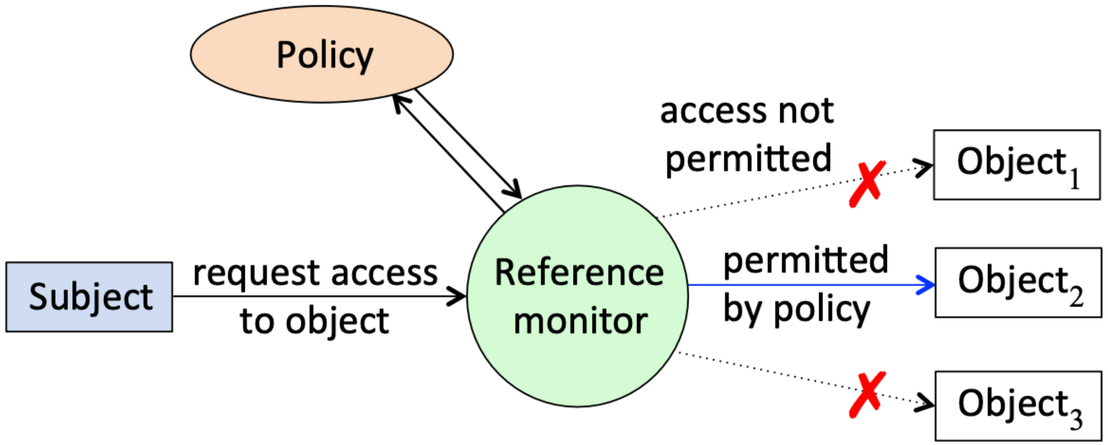
Terminology

- *Subject*
  - Entity that wants to take an action
  - Usually, the subject has been authenticated
  - May be an unknown subject



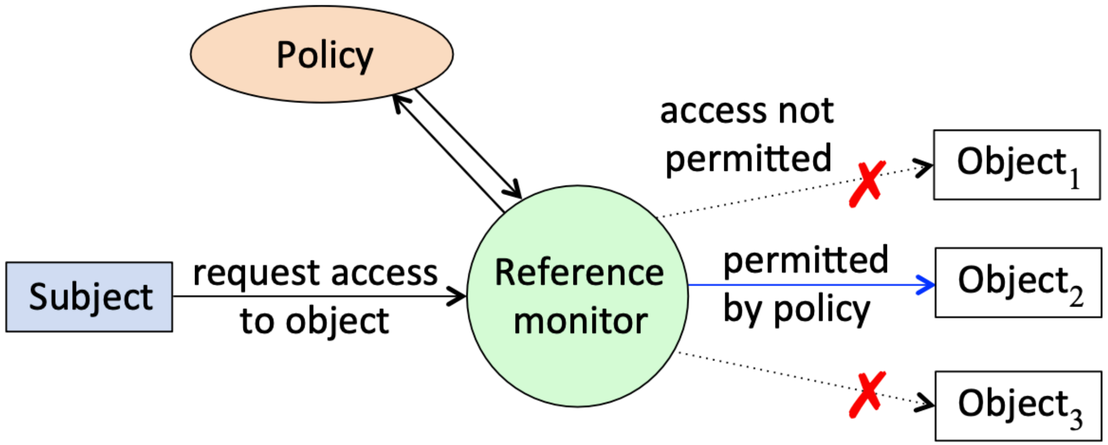
Terminology

- Subject
- *Action*
  - What the subject wants to do
    - Read, write, execute
    - Start, shutdown
    - Debug (one process monitoring another)



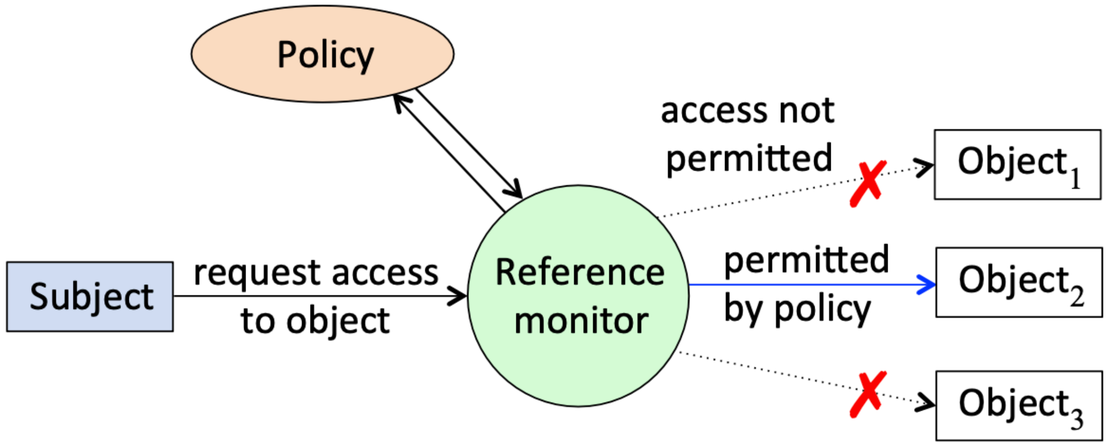
Terminology

- Subject
- Action
- *Object*
  - What item the subject wants to take the action on
    - Memory
    - Files and directories
    - Services
    - Devices



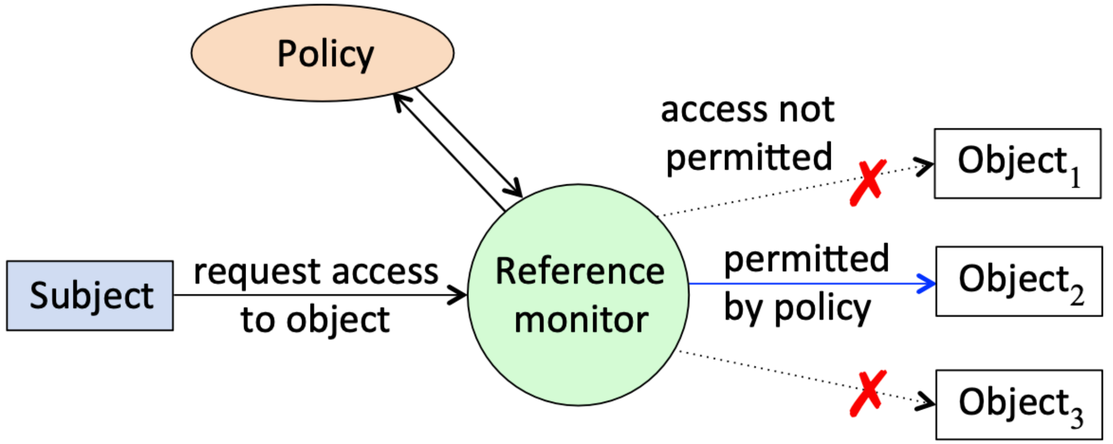
Terminology

- Subject
- Action
- Object
- *Policy*
  - Defines what is allowed and what is not allowed
  - Usually parameterized by (subject, action, object) triple



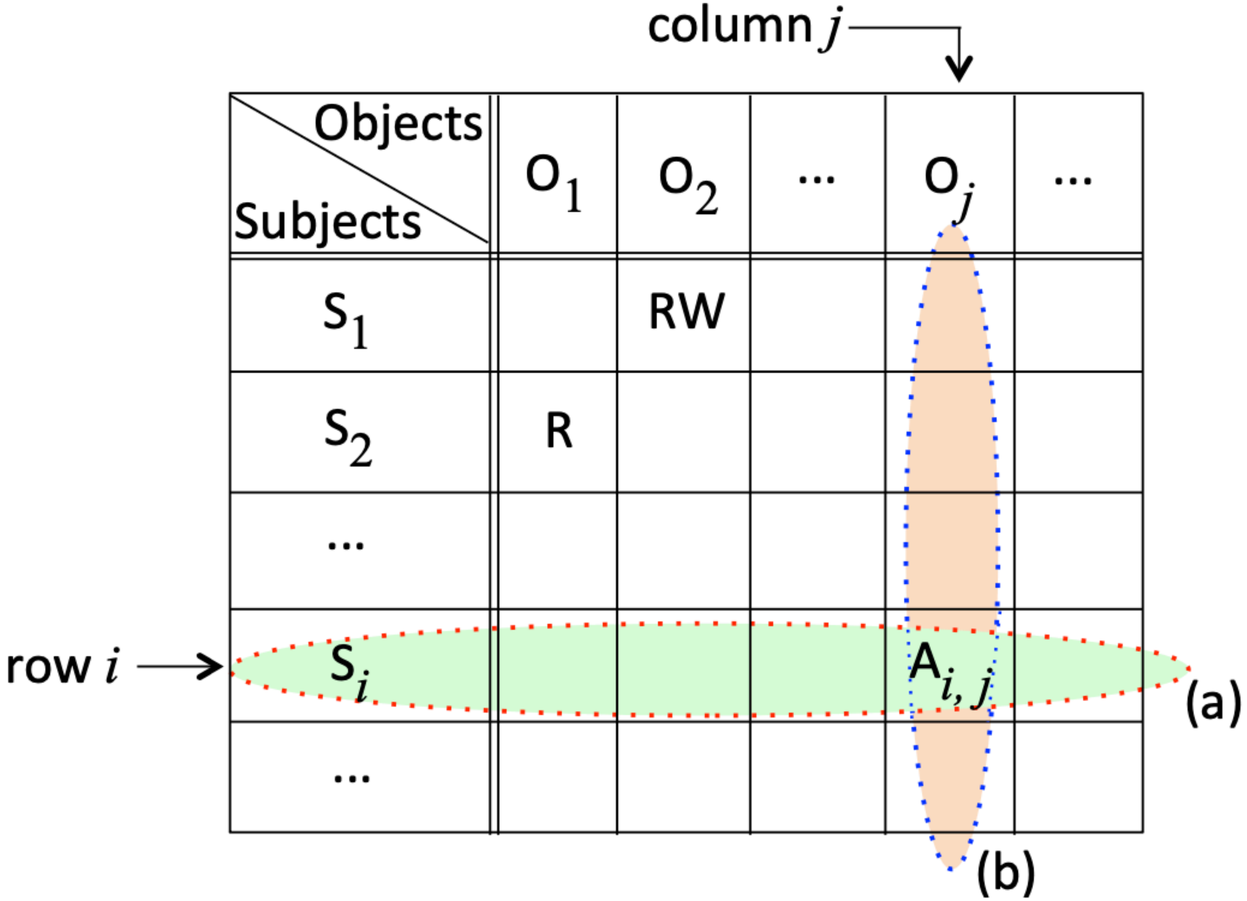
Terminology

- Subject
- Action
- Object
- Policy
- *Reference Monitor*
  - Controls access to an object
  - Only allows a subject to execute an action on an object if that action conforms with the policy



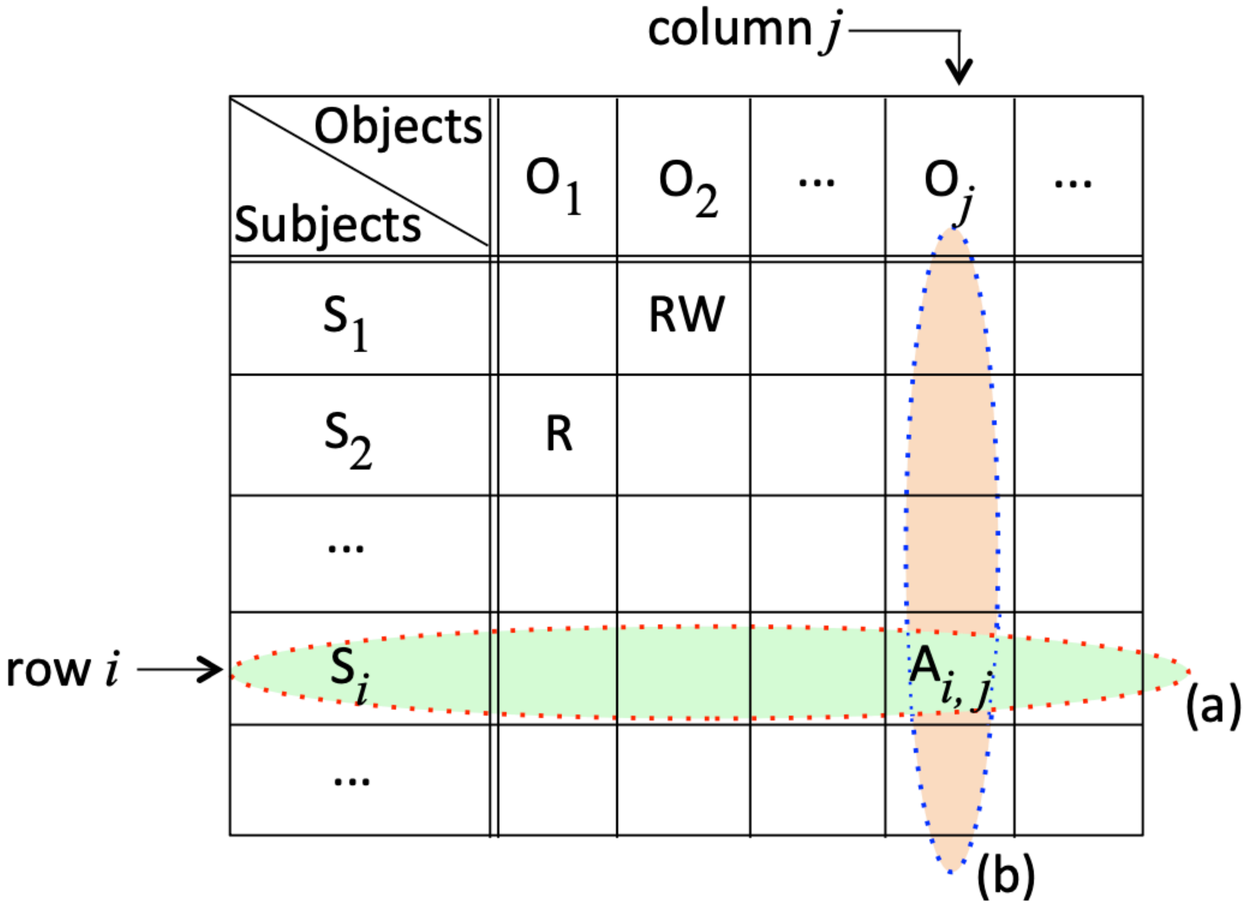
Access Matrix

- Access policy organized as a matrix
  - Rows indexed by subjects
  - Columns indexed by objects
- Access Control Entry (ACE)
  - Specifies access of Subject on a given Object



Access Control List / Capability List

- Capability List
  - For a given subject, what objects and what permissions over those objects
  - A row of the Access Matrix
- Access Control List
  - For a given object, which subjects have access and what permissions to that object
  - A column of the Access Matrix



Principle of Least Privilege



A component has the least privileges needed to function

- Any further removed privilege reduces functionality
- Any added privilege will not increase functionality (according to the specifications)
- This property constraints an attacker in the obtainable privileges.



Rendering in Chromium executes in an encapsulated sandbox where only minimal system calls are allowed.

Privileges

priv·i·lege | 'priv(ə)lij |

noun

a special right, advantage, or immunity granted or available only to a particular person or group: *education is a right, not a privilege* | *[mass noun] : he has been accustomed all his life to wealth and privilege.*

- Override (i.e., make exceptions to) access control rules
- Usually a thread or process attribute

Rules are made to be broken.

Why Do We Need Privileges?

- Real systems need “exceptions” to access control rules.
  - Installing new software
  - Change of policy
  - Change of ownership
  - Fix incorrect configurations
  - Help users solve problems

Privilege Granularity



Coarse-grained Privileges

- All or nothing
- Effective UID of 0 (root) overrides all access controls.

What Can the Root User Do (Unix)?

Depends on the OS. But in general, (almost) omnipotent.

- Open any file for reading
- Open any file for writing
- Write to any directory
- Change file's owner
- Change file's permission bits
- Change process UIDs to arbitrary values
- Send signals to any process
- Change the system time
- Bind socket to a privileged port
- Reboot the system

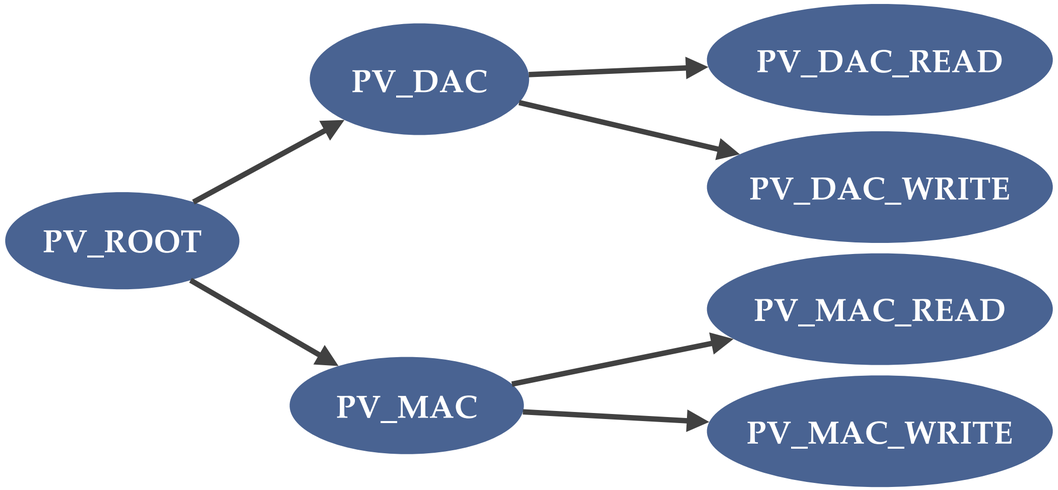
Medium-grained Privileges (Linux)

- Named “capabilities” in the Linux documentation
- Kernel checks for privileges in process’s effective privilege set
- Kernel provides “hack” to mimic Unix access control
  - Turns all privileges on when effective UID is root
  - Turns all privileges off when effective UID is not root
  - A process can disable this behavior using `prctl()`
- Linux kernel 6.17 uses 41 capabilities.
  - “cat /proc/sys/kernel/cap_last_cap” prints the last capability ID (start from 0)

Medium-grained Privileges (Linux)

Privilege	Description
CAP_DAC_READSEARCH	Override read permissions on files Override search permissions on directories
CAP_DAC_OVERRIDE	Override read, search, and write access on files and directories
CAP_CHOWN	Change owner of files
CAP_SETUID	Change real, effective, and saved UIDs to any value

Fine-grained Privileges (Argus PitBull)



- Separate privileges for overriding read, write, execute
- Separate privilege classes for overriding each type of access control (MAC and DAC)
- Hierarchical tree to make privileges easier to manage: Top privilege is a superset of sub-tree.
- Root user is no longer a special user.

Outline: Principle of Least Privilege

- What are privileges?
- **What problems do current systems have with privileges?**
- What can we do to more safely use privileges?

Unneeded privileges may be exploited to launch attacks.

Outline: Principle of Least Privilege

- What are privileges?
- What problems do current systems have with privileges?
- **What can we do to more safely use privileges?**

Turning Privileges On and Off

- Programs do not need all operations to be privileged.
  - The program's functionality may not need privileges.
    - Use privilege to open password file
    - Not use privilege to open user preferences file
- Follow Saltzer and Schroeder Principle of Least Privilege
  - Programs using fewer privileges tend to have fewer vulnerabilities.

Enabling and Disabling Unix “Privilege”



- To disable root, swap EUID and SUID
- To enable root, swap EUID and SUID

Enabling and Disabling Unix “Privilege”



- To disable root, swap EUID and SUID
- To enable root, swap EUID and SUID

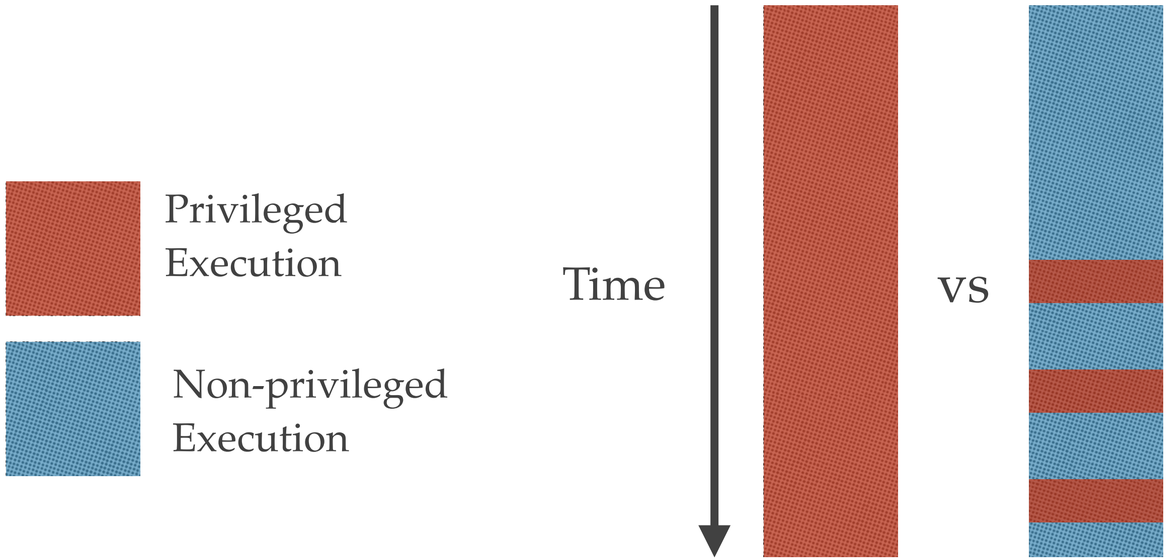
Enabling and Disabling Unix “Privilege”



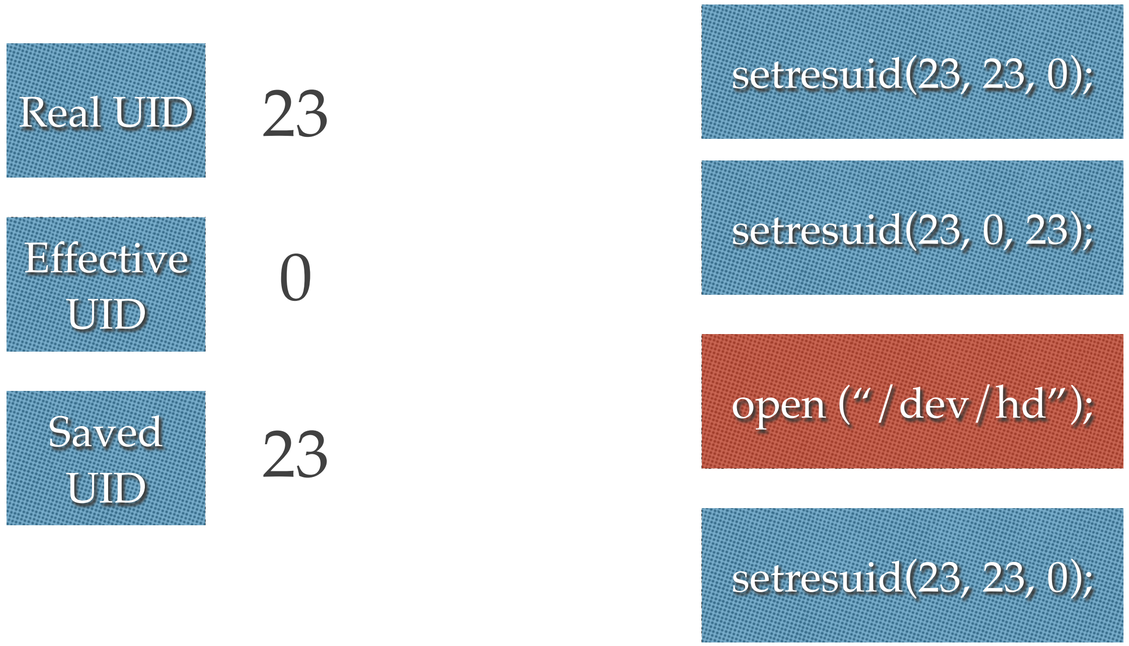
- To disable root, swap EUID and SUID
- To enable root, swap EUID and SUID

Privilege Bracketing

- Enable privileges before an privileged operation
- Disable privileges after the operation

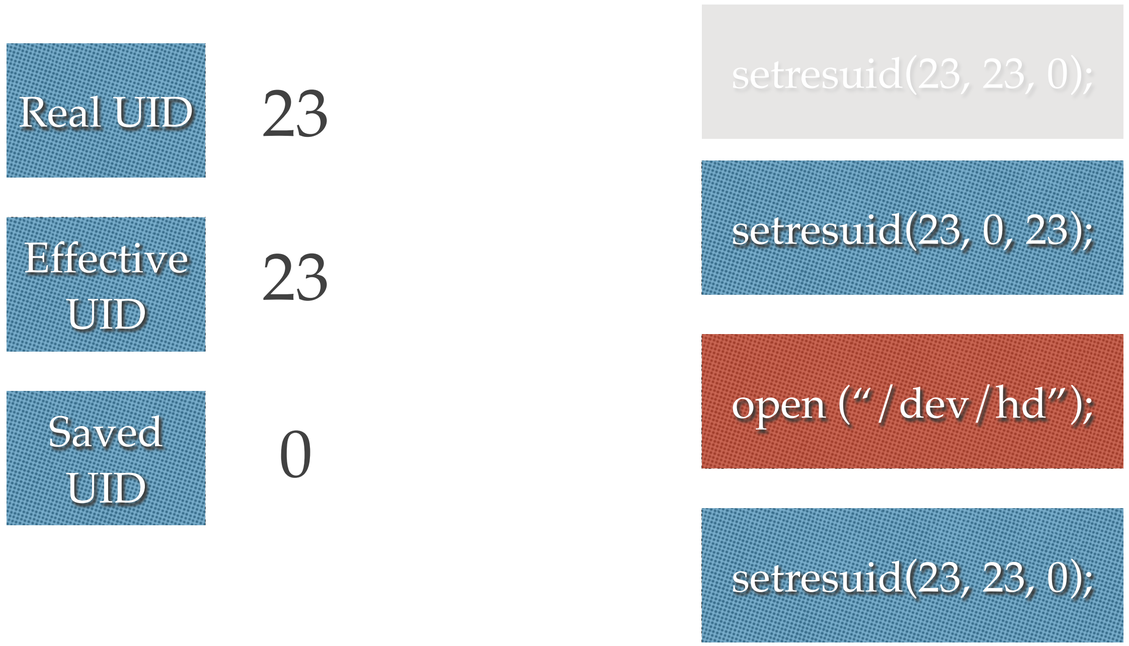


Unix Privilege Bracketing: 1



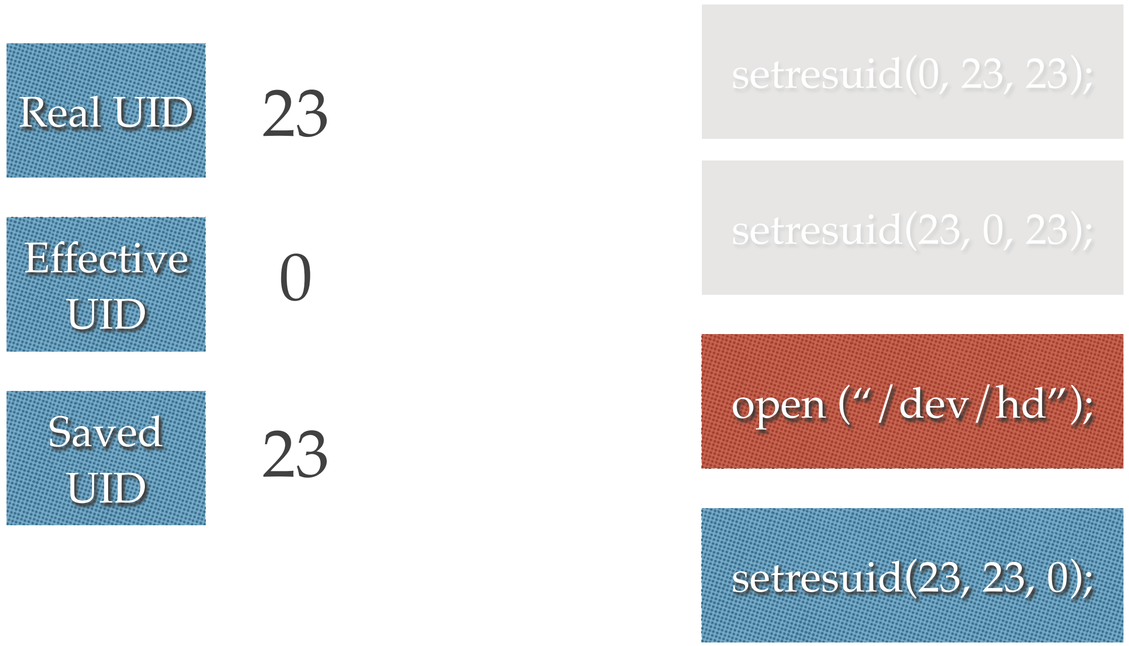
- `setresuid()` sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.

Unix Privilege Bracketing: 2



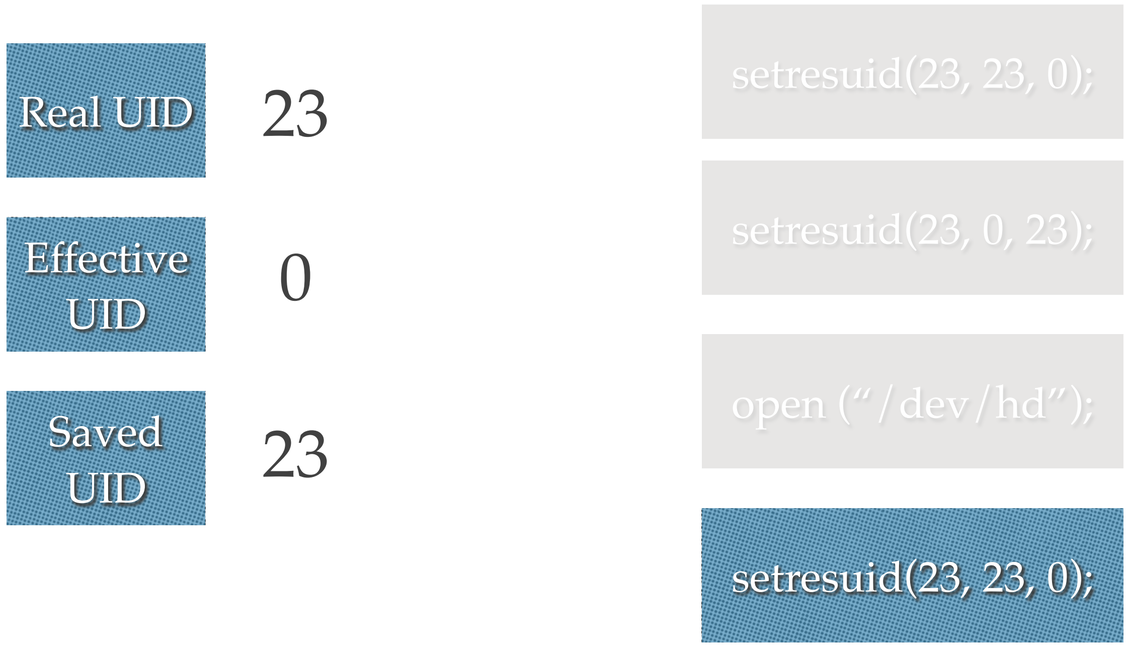
- `setresuid()` sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.

Unix Privilege Bracketing: 3



- `setresuid()` sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.

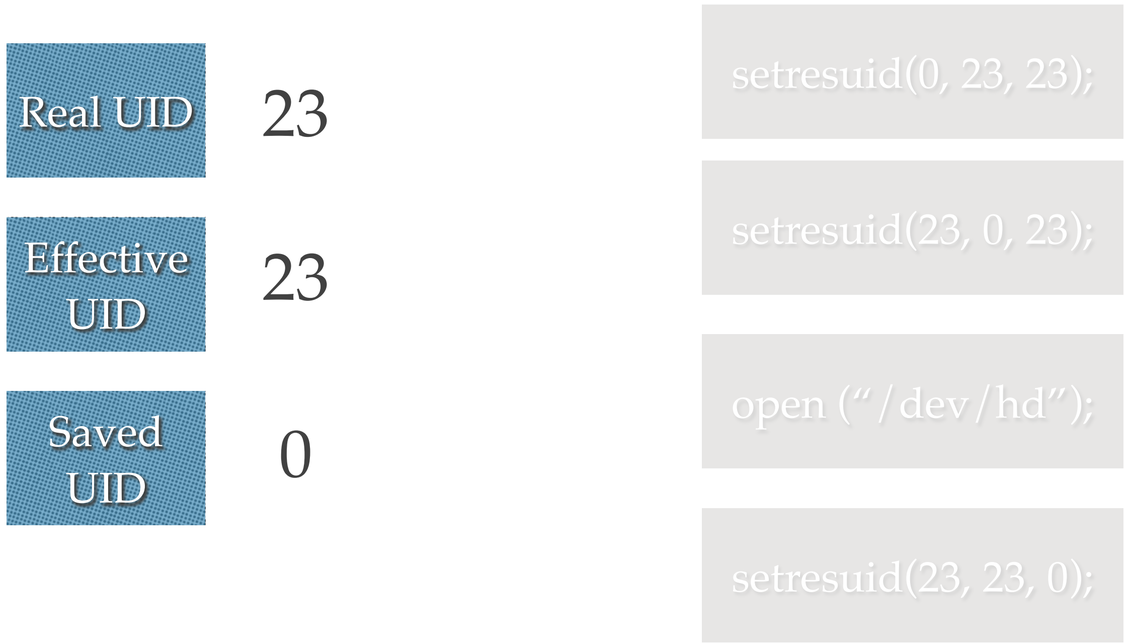
Unix Privilege Bracketing: 4



- `setresuid()` sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.

Unix Privilege Bracketing: 5

What if attackers turn on the root UID at unexpected points?



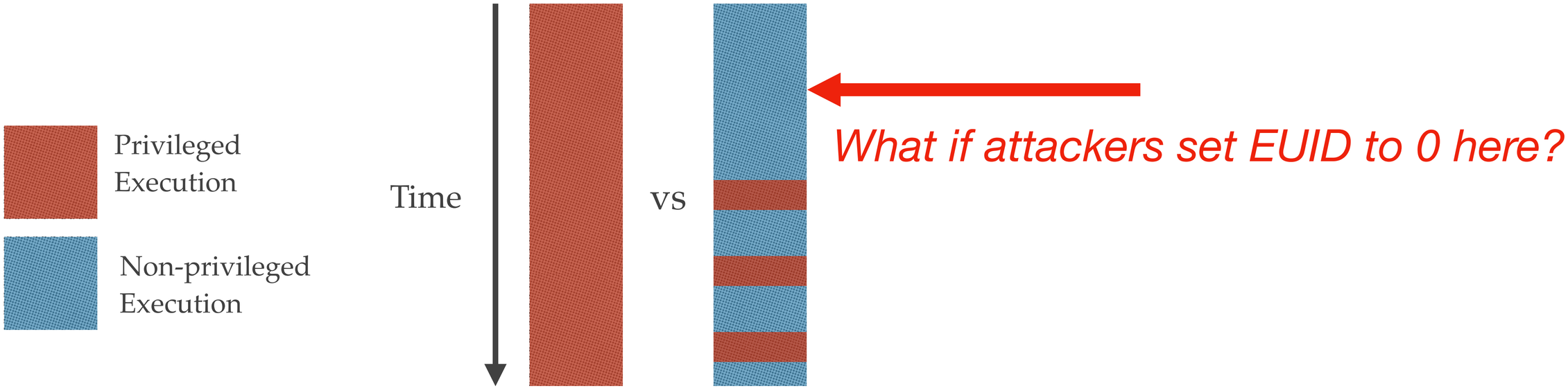
- `setresuid()` sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.



What weakness does this approach have?

Privilege Bracketing

- Enable privileges before an privileged operation
- Disable privileges after the operation



How does it work on systems with more than one privilege?

Medium-grained Privileges (Linux)

- Named “capabilities” in the Linux documentation
- Kernel checks for privileges in process’s effective privilege set
- Kernel provides “hack” to mimic Unix access control
  - Turns all privileges on when effective UID is root
  - Turns all privileges off when effective UID is not root
  - A process can disable this behavior using `prctl()`
- Linux kernel 6.17 uses **41 capabilities**.
  - “cat /proc/sys/kernel/cap_last_cap” prints the last capability ID (start from 0)

Process Privilege Sets

- Maximum privilege set: All allowed privileges
  - E.g., `CAP_DAC_READSEARCH`, `CAP_DAC_OVERRIDE`
- Effective privilege set: Currently active privileges
  - E.g., `CAP_DAC_READSEARCH`
- Process can add effective privilege if it is in maximum set.
- Process can remove privileges in maximum and effective.

Fine-grained Privilege Bracketing

Operations on Privilege Sets

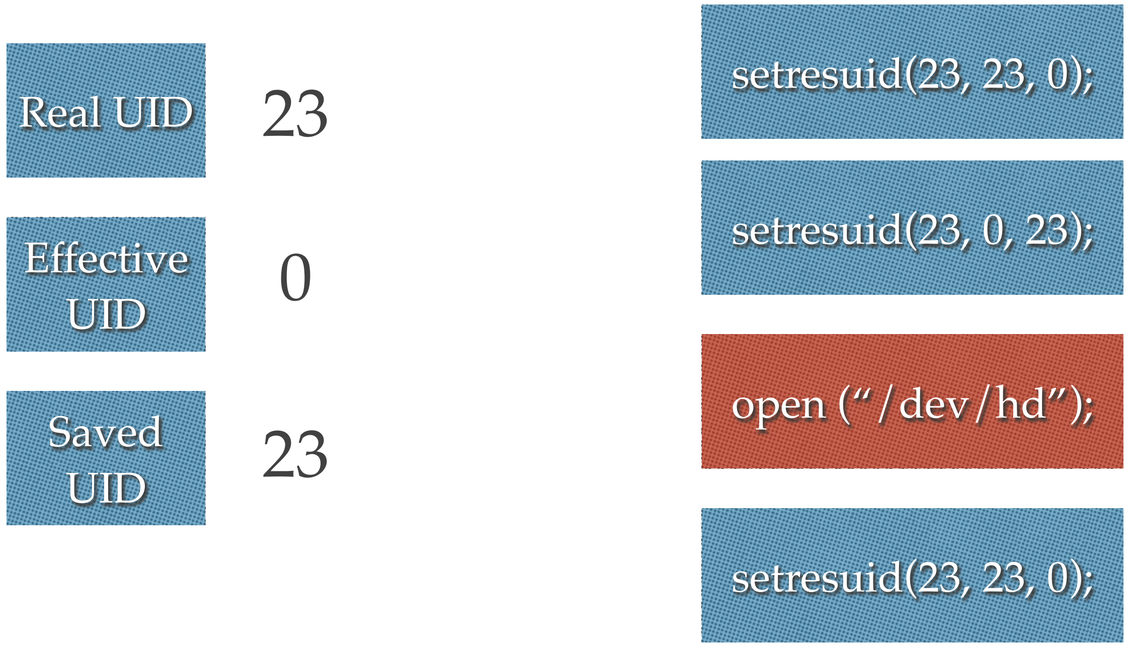
Operation	Description
`priv_raise()`	Enable privileges in effective set
`priv_lower()`	Disable privileges in effective set
`priv_remove()`	Remove privileges in effective and permitted set

open (“/etc/passwd”, O_RDONLY);



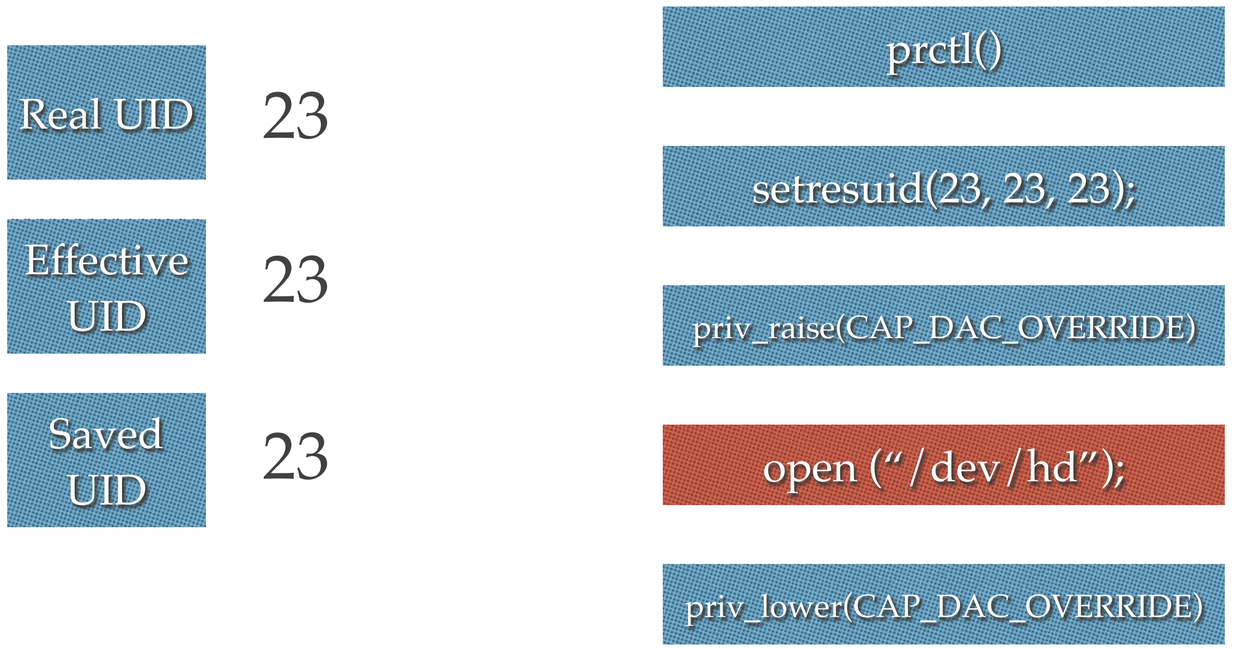
priv_raise (CAP_DAC_READSEARCH);
open (“/etc/passwd”, O_RDONLY);
priv_lower(CAP_DAC_READSEARCH);

Unix Privilege Bracketing: 1



- `setresuid()` sets the real user ID, the effective user ID, and the saved user ID of the calling process.

Linux Privilege Bracketing



Anything weakness in this strategy?

What if `priv_raise()` is exploited by attackers?

How to remove privileges in Unix/Linux?

Process Privilege Sets

- Maximum privilege set: All allowed privileges
  - E.g., CAP_DAC_READSEARCH, CAP_DAC_OVERRIDE
- Effective privilege set: Currently active privileges
  - E.g., CAP_DAC_READSEARCH
- Process can add effective privilege if it is in maximum set.
- Process can remove privileges in maximum and effective.
- Challenge: Removing unneeded privileges at the *earliest* point.

Remove Privileges

- Unix
  - Set EUID, RUID, and SUID to non-zero values
- Linux
  - Remove privilege from effective and maximum set

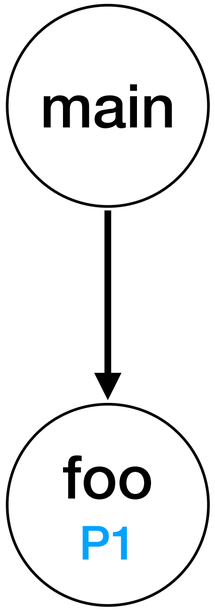
Remove Privileges

```
int main() {
    ......
    foo();
    ......
}

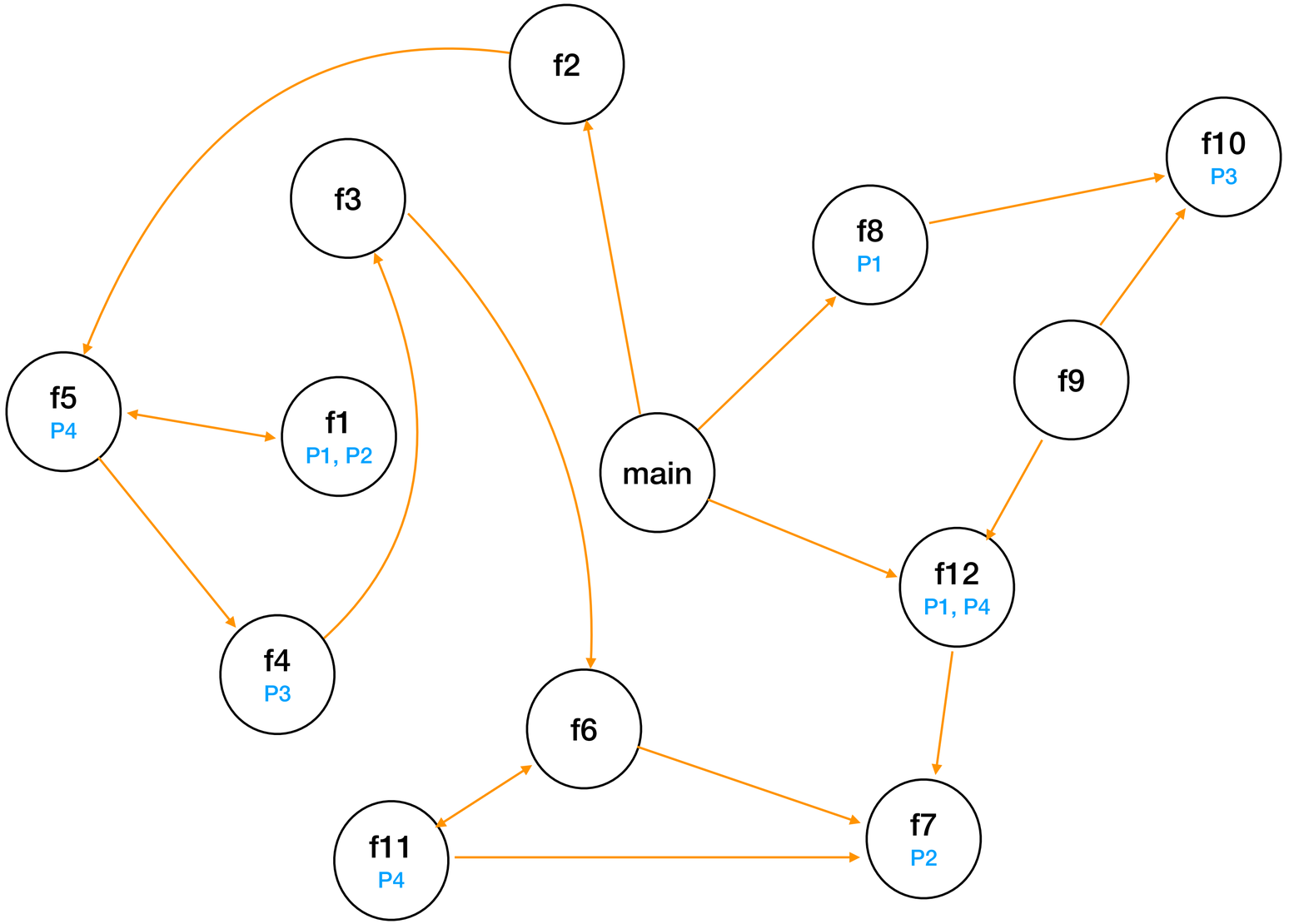


void foo() {
    ...
    ... // use privilege P1
}
```



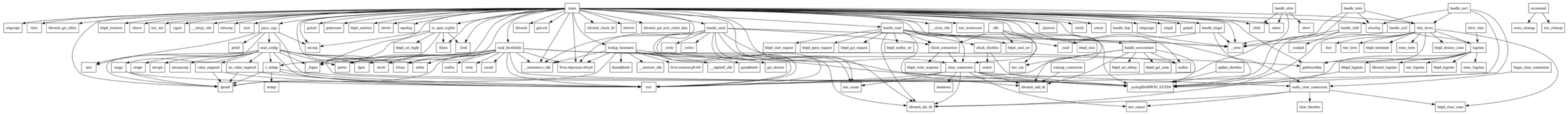


Insert *priv_remove(P1)* after P1 is no longer needed.


f2
f3
f10
P3
f8
P1
f5
P4
f1
P1, P2
f9
main
f12
P1, P4
f4
P3
f6
f11
P4
f7
P2

`thttpd`: A Lightweight HTTP Server Written in C

SLOC: 8,360



Call Graph of `thttpd`

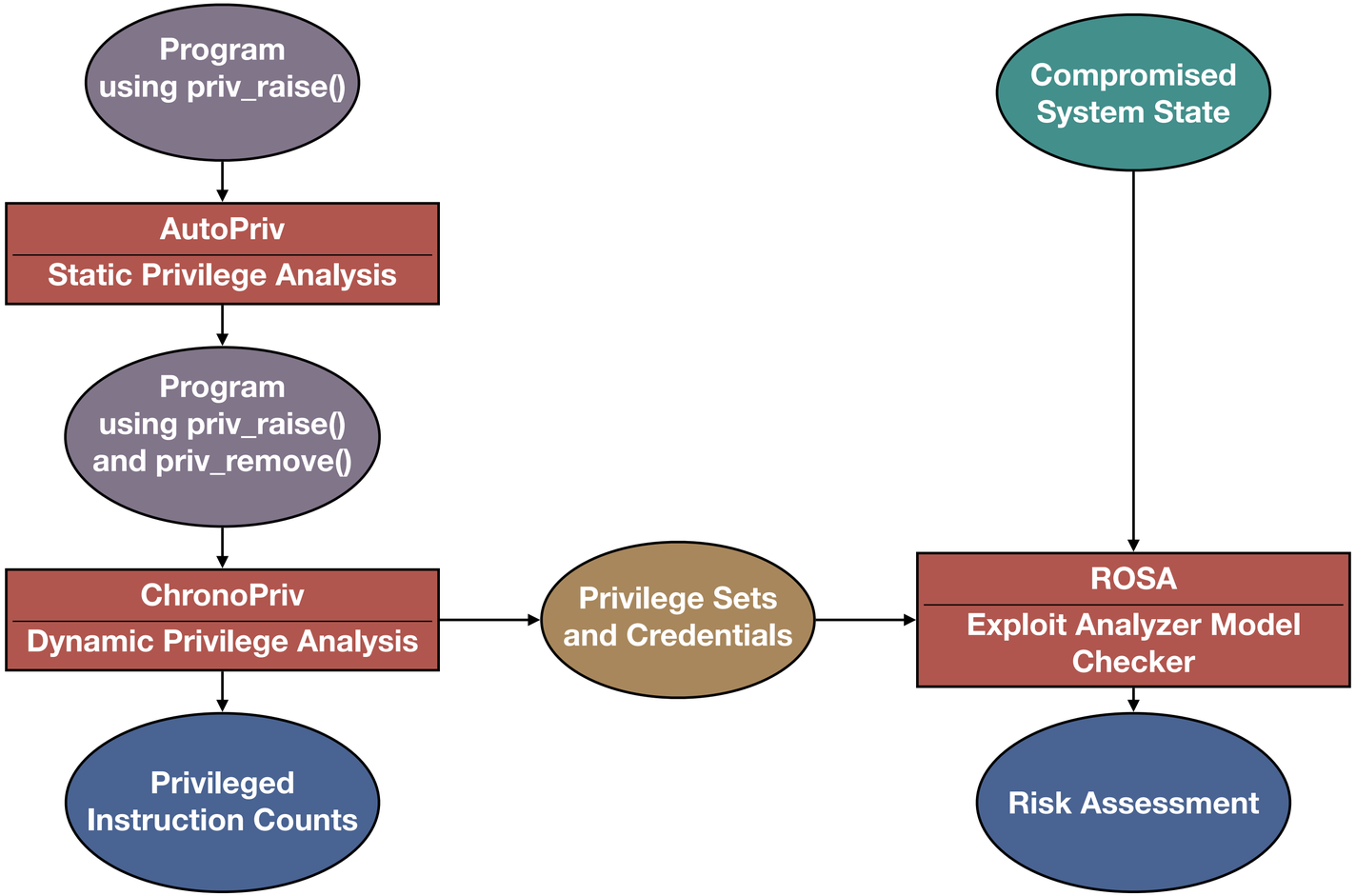
- It’s extremely difficult to manually figure out when we can remove which privileges permanently.
- We need an *automated* tool.

Compiler Comes to The Rescue (Again)!

Using Compiler to Enforce Least Privilege

- Programmers priv-bracket operations needing privileges.
- Compiler analyzes and inserts calls to `priv_remove()`
- Model checker determines if system is capable of entering unsafe state.
  - Assume attacker can exploit memory safety errors, e.g. buffer overflows
  - Assume attacker can use system calls in program in any order
  - Measure amount of execution spent with each privilege set

PrivAnalyzer Architecture



PrivAnalyzer Evaluation

- Tested 4 attacks i.e., unsafe system states
  - Open `/dev/mem` for reading
  - Open `/dev/mem` for writing
  - Bind to a privileged network port
  - Send `SIGKILL` signal to kill the `sshd` server

Security Analysis Results

Program	Unsafe State			
	Read /dev/mem	Write /dev/mem	Privileged Port	Kill Process
passwd	100%	100%	0%	63%
ping	0%	0%	0%	0%
sshd	100%	100%	~0%	100%
su	88%	88%	0%	88%
thttpd	10%	~0%	10%	~0%

Why Do Programs Use Privileges Ineffectively?

- Programs were originally designed for root user.
  - No reason to design program with least privilege
- System files are owned by root.
  - Privileged processes access subset of system files.
  - Root processes can access all system files even with no privileges!
- Privileges are needed late in execution

Strategy 1: Leverage Saved UID

- Use privilege to place file owner UID in saved UID
- Remove `CAP_SETUID` from maximum privilege set
- Switch to saved UID to open file using *no privilege*



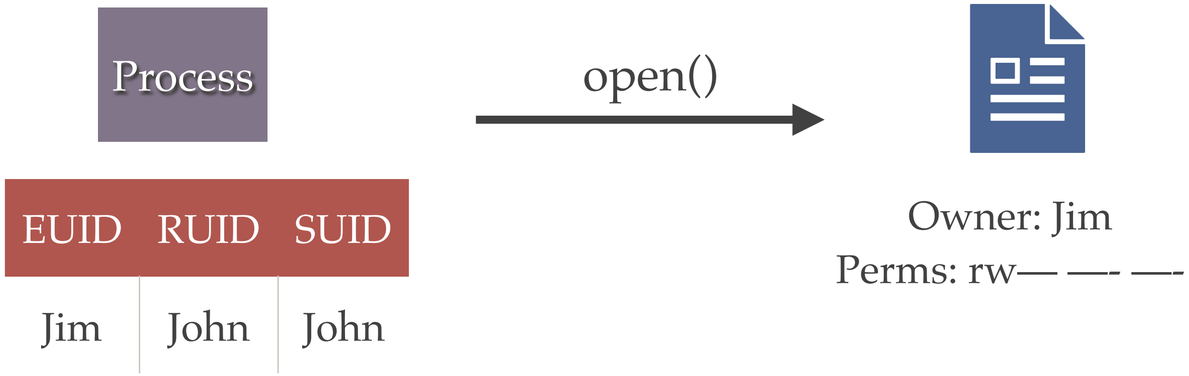
Strategy 1: Leverage Saved UID

- Use privilege to place file owner UID in saved UID
- Remove `CAP_SETUID` from maximum privilege set
- Switch to saved UID to open file using *no privilege*



Strategy 1: Leverage Saved UID

- Use privilege to place file owner UID in saved UID
- Remove `CAP_SETUID` from maximum privilege set
- Switch to saved UID to open file using *no privilege*



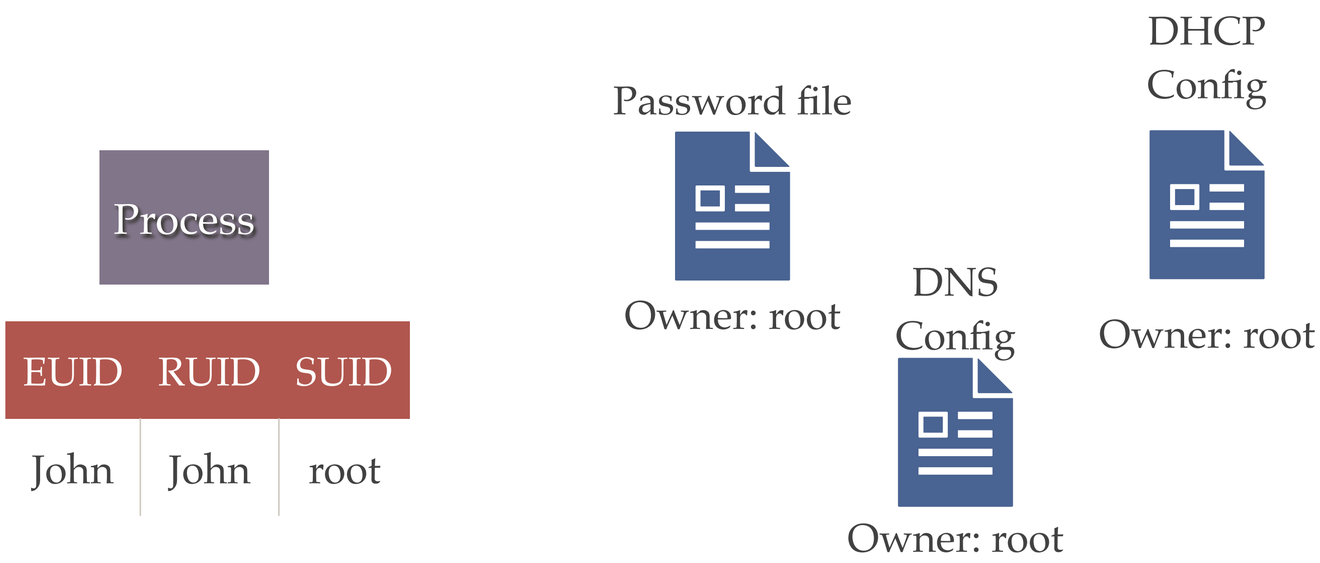
Strategy 1: Leverage Saved UID

- Use privilege to place file owner UID in saved UID
- Remove `CAP_SETUID` from maximum privilege set
- Switch to saved UID to open file using *no privilege*



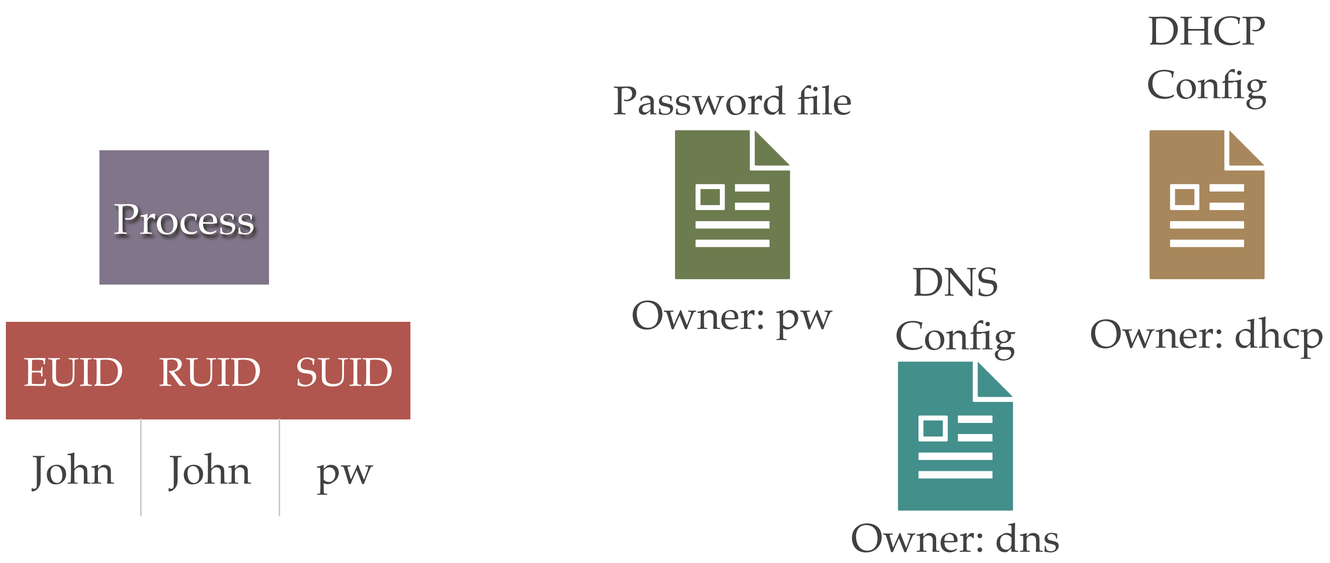
Strategy 2: Use Different File Owners

- Have each set of files owned by different user
- Unprivileged process can now open only needed files



Strategy 2: Use Different File Owners

- Have each set of files owned by different user
- Unprivileged process can now open only needed files



Summary of Refactoring Strategies

- Switch UID early in execution to owner of needed files
  - Use privilege to place file owner UID in saved UID
  - Remove `CAP_SETUID` from maximum privilege set
  - Can switch to file owner using `setresuid()` with no privilege
- Change owner of files to a unique unused UID
  - Unprivileged process can now open only needed files
  - Other system files inaccessible

Refactored Security Analysis Results

Program	Unsafe State			
	Read /dev/mem	Write /dev/mem	Privileged Port	Kill Process
passwd	100%	100%	0%	63%
Refactored passwd	**4%**	**4%**	**0%**	**4%**
su	88%	88%	0%	88%
Refactored su	**1%**	**1%**	**0%**	**1%**

Software Compartmentalization

Principle: Fault Compartmentalization



Separate individual components into smallest functional entity possible.

- These units contain faults to individual components.
- Allows abstraction and permission checks at boundaries.
- In practice, smallest functional entity can be too expensive.



A chatting app’s image processing module and audio processing module are compartmentalized so that bugs in one module will not affect another.

Isolation vs. Compartmentalization

- Isolation is a fundamental *mechanism*.
  - Emphasizes barriers/walls between components; often no intended interactions
  - E.g., between processes, VMs, untrusted libs
- Compartmentalization is a design strategy/policy that often uses isolation.

Compartmentalization builds on least privilege and isolation. Both properties are most effective in combination: many small components that are isolated and running and interacting with least privileges.

Key Design Questions

- How to determine the right policy to enforce?
- How to express the policy in software?
- How to enforce the policy at runtime?

Policy Definition Method (PDM)

PDMs identify subjects, objects, and permissions to enforce.

Essential Elements

- Subject: a unit of computation; also called *principle*
  - E.g., a sequence of assembly instructions, a thread of execution
- Object: a resource that a subject can access or manipulate
  - E.g., a byte of memory, a file, a network socket
- Permissions: actions that a subject may perform on objects
  - E.g., read, write, open a socket
- Protection domain: maximal set of subjects sharing the same permissions

Automation for Policy Specification

- Manual methods
  - Developers must specify *which* object is given *what* permissions for *what* object.
  - Pros: accurate
  - Cons: prone to human errors, leading to over- or under-privileged components
- Guided manual methods
  - Often provide a feedback loop to guide users to specify and refine boundaries
    - E.g., developers propose isolating an image decoder; tools point out which memory accesses potentially respect/violate the goal, and developers iterate based on that.
- Fully automated methods
  - Automatically analyze and understand programs and enforce policies.
  - Difficult to pinpoint boundaries

Analysis Techniques

- Static analysis: analyzing a program without running it
  - Usually conservative but guarantees functionality
  - Over-privileged compartments
- Dynamic analysis: analyzing a program at execution time
  - Enforce policies based on dynamic behavior
- Example: Sandboxing all unsafe code and its accessed memory in Rust
  - Static analysis: Identifying all unsafe pointers **and** their aliases, and their accessed memory
    - Extremely challenging to do *precisely*!
  - Profile the executed unsafe code, memory it accesses, and code in the safe region that accesses this memory.
    - Reliability depends on the coverage of profiling.
    - Suffer from availability issues

Separation Granularities

- Functions
- Libraries
- Source files
- Software packages
- Others

Subject Selections

- Code-centric
  - Subjects are defined as program code.
  - Protection domains constitute code regions.
  - E.g., the `libjpeg` library
  - Most popular selection
- Data-centric
  - Subjects are temporal units of executes.
  - Protection domains may contain one or more of these subjects.
  - E.g., each worker process in a web server runs in isolation
- Hybrid
  - Data-centric subjects bounded within code regions
  - E.g., a thread bounded to a specific library

Key Design Questions

- How to determine the right policy to enforce?
- **How to express the policy in software?**
- How to enforce the policy at runtime?

Compartmentalization Abstractions

Key factors to consider

- A model of actions
- Trust models
- Target properties to enforce
- Composing with other abstractions

Model of Actions: Five Primitives

- Create
  - What to do during compartment initialization
  - E.g., reserve a dedicated memory region, eliminate unnecessary privileges
- Destroy
  - What to do after a compartment finishes its task
  - E.g., erasing memory that may contain sensitive data
- Enter (or call)
  - What to do when entering into a compartment
- Return
  - What to do when exiting a compartment
- Assign
  - How to communicate between compartments
    - message passing, e.g., via sockets or pipes
    - shared memory

Trust Models

- Sandbox: restrain the untrusted compartments
  - Support arbitrary number of compartments
  - Most commonly adopted, e.g., browser plugins, device drivers
- Safebox: restrain everything else
  - Dual-world model: only two compartments (trusted and untrusted)
  - Suitable for situations where the trusted is small and well-specified

Properties to Enforce

- Integrity
  - Targeted by all compartmentalization mechanisms
- Confidentiality
  - Lack of it may benefit performance and simplify implementation.
  - Whether to support it depends on the application scenarios.
- Availability
  - Prevent and recover from resource exhaustion
  - Usually require whole-system consideration
  - Relatively less studied

Key Design Questions

- How to determine the right policy to enforce?
- How to express the policy in software?
- **How to enforce the policy at runtime?**

Enforcing Compartmentalization

- Software-based
  - E.g., SFI, AutoPriv
- Hardware-based
  - E.g., page-based, MPK
- Hybrid
- Other considerations
  - Granularity: byte-level to entire physical memory
  - Supported number of domains
  - Performance

Evaluating Compartmentalization

- Security benefits
- Performance compared to a monolithic design
- Compatibility with existing software and programming idioms
- Usability of separated software

Case Study: Mail Server

- Mail Transfer Agents (MTA) need to do a plethora of tasks:
  - Send/receive data from the network
  - Manage a pool of received/unsent messages
  - Provide access to stored messages for each user

`Sendmail` uses a typical Unix approach with a large monolithic server and is known for the high complexity and previous security vulnerabilities.

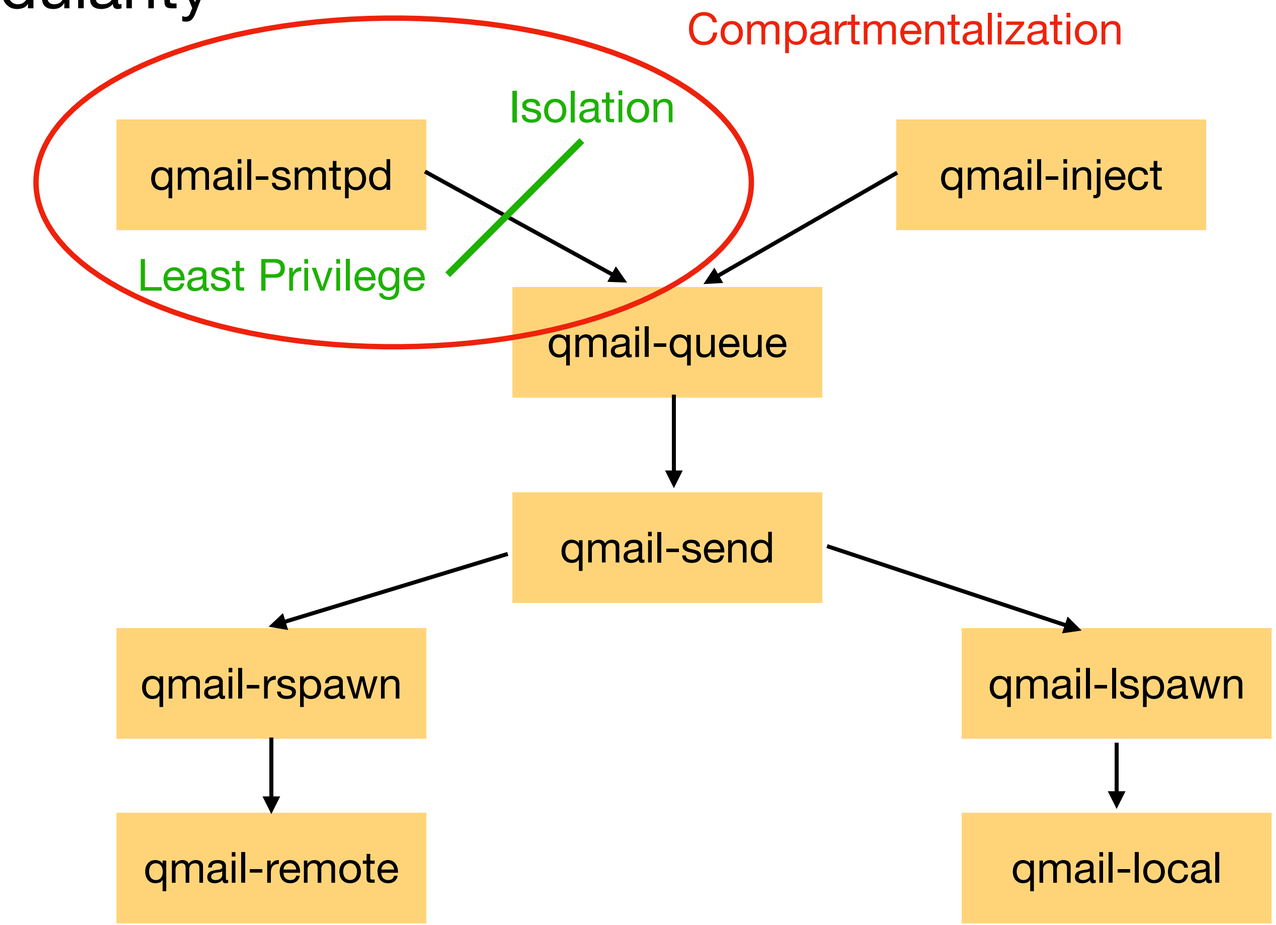


How would you compartmentalize a mail server?

Case Study: Mail Server

`qmail:` An mail MTA designed with security in mind.

- Key enabler: modularity



Case Study: Mail Server



What can we do to further reduce potential exploits?

- Separate modules run under separate user IDs.

