CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science

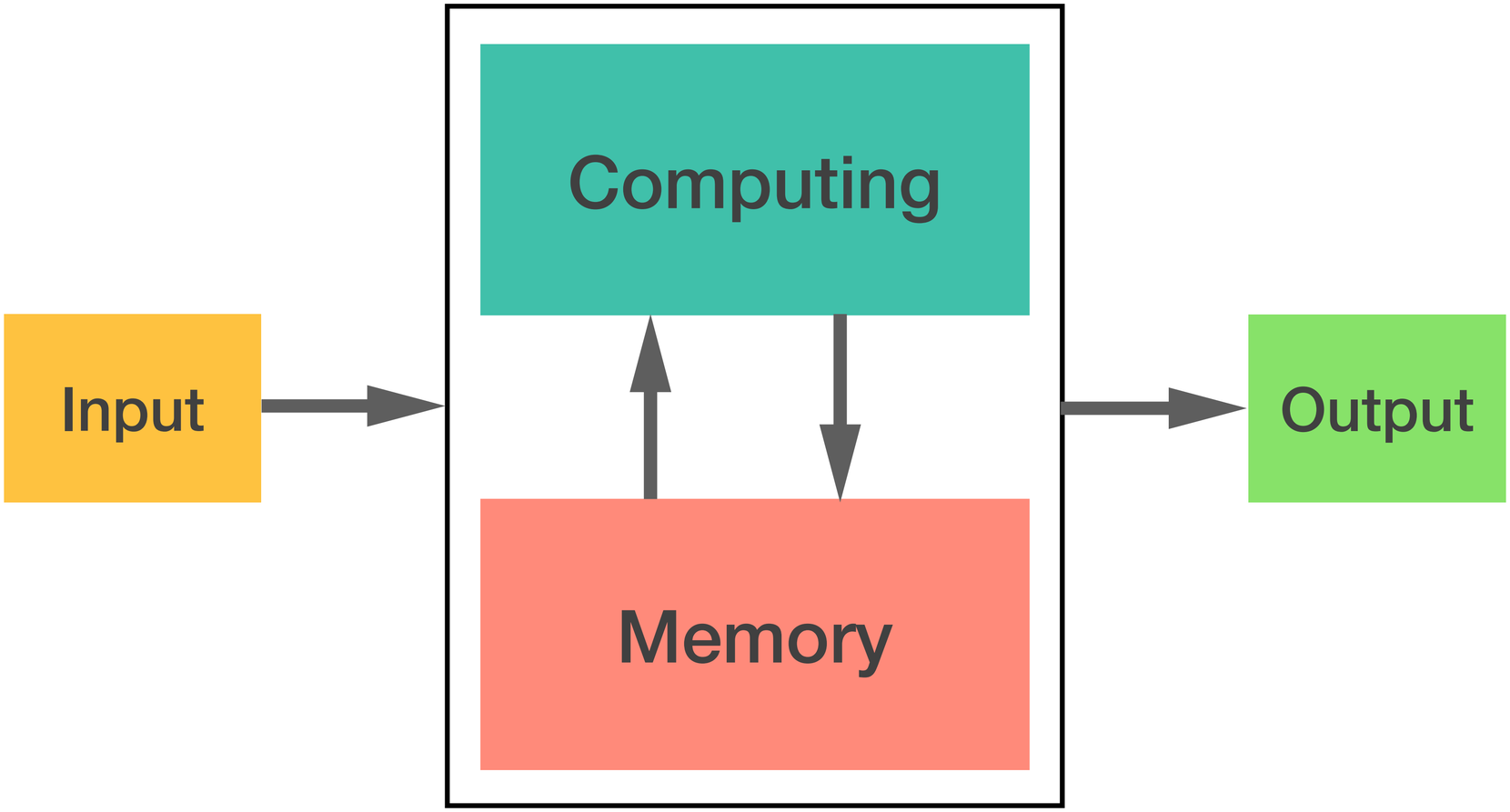
George Washington University

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

Outline

- Review: Isolation
- Least Privilege Principle
- Software Compartmentalization

Architecture of Modern Computers



How to ensure safety when sharing memory with untrusted programs?

How to ensure safety when sharing memory with untrusted components?

Principle: Isolation

Isolate two components from each other

- One component cannot access data/code of the other component except through a well-defined API

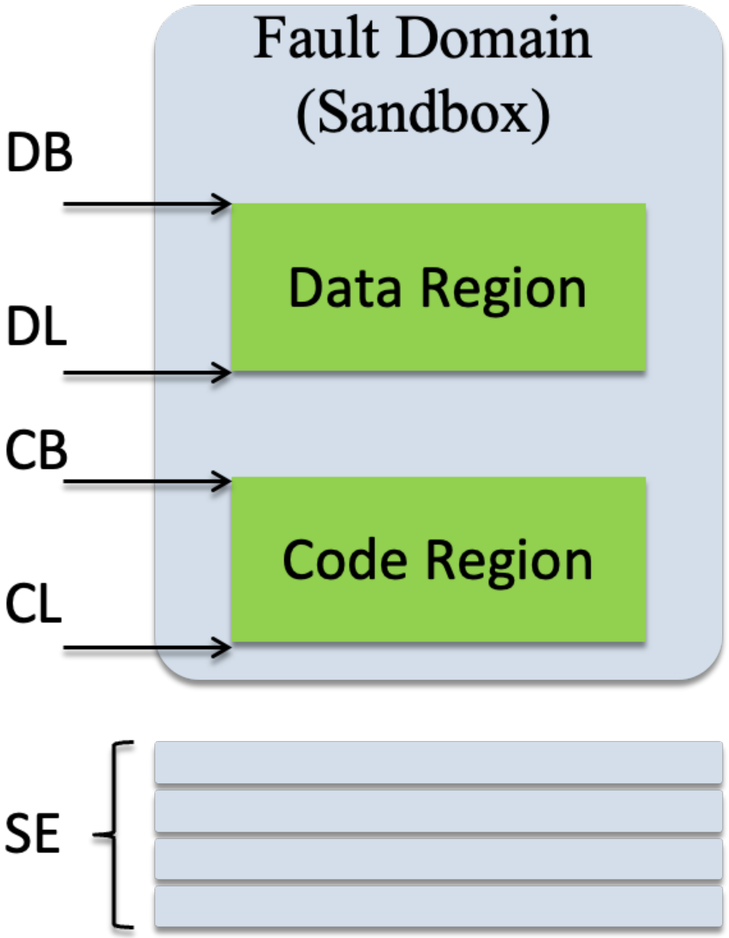
User-space application may only access the disk through the filesystem API (i.e., the OS prohibits direct block access to raw data). The OS isolates the user-space process from the disk.

Isolation incurs overhead due to switching cost between components.

Software-based Fault Isolation (SFI)

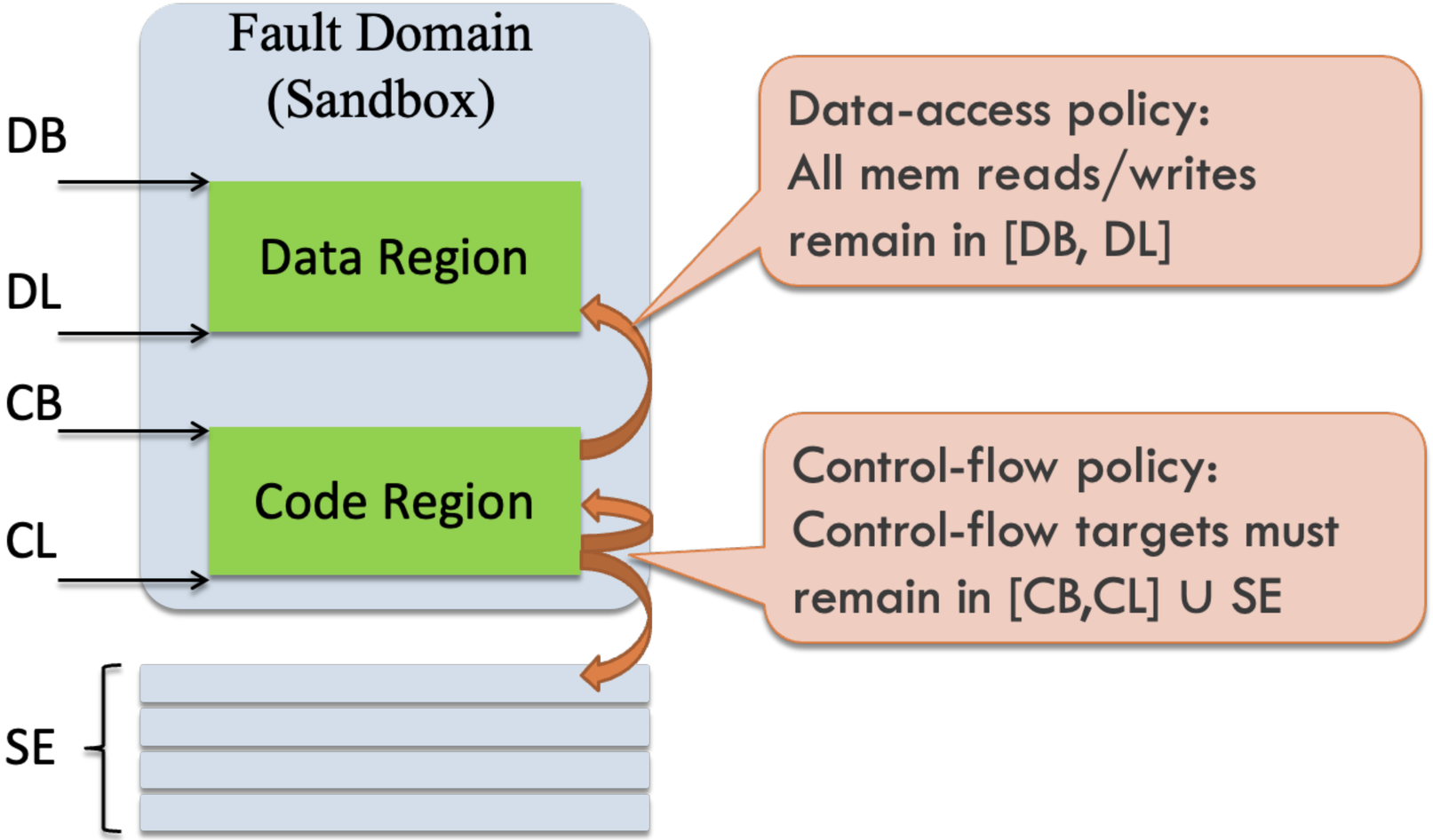
- Introduced by [Wahbe et al. 93] for MIPS
  - [McCamant & Morrisett 06] extended it to x86
  - [PNaCl] Google implemented SFI for ARM, ADM64, & MIPS for Chrome
- SFI is within the the same process address space
  - One type of *intra-address space* isolation
  - Each protection domain has a designated memory region.
  - Same process: avoiding costly context switches
- Implementation by inserting software checks before critical instructions
  - E.g., memory reads/writes, indirect branches
- Pros: Fine-grained, flexible, low context-switch overhead
- Cons: May require compiler support and software engineering effort

SFI Sandbox Setup



- Data region (DR): [DB, DL]
  - Hold data: stack, heap, global
- Code region (DR): [CB, CL]
  - Hold code
- Safe external (SE) addresses
  - Host trusted services that require higher privileges
  - Code can jump to them for accessing resources.
  - Code can safely transition out of the current domain.
- DR, CR, and SE are disjoint.

SFI Policy


Fault Domain
(Sandbox)
DB
DL
CB
CL
SE
Data Region
Code Region
Data-access policy:
All mem reads/writes
remain in [DB, DL]
Control-flow policy:
Control-flow targets must
remain in [CB,CL] U SE

SFI Enforcement Overview

- Dangerous instructions: memory reads/writes, control-transfer instructions
  - They have the potential of violating the SFI policy.
- SFI enforcement
  - Check every dangerous instruction to ensure it obeys the policy
- Two general enforcement strategies
  - Dynamic binary translation
  - Inlined reference monitors

Example

```
  r3 := r1
  r4 := r2 * 4
  r4 := r1 + r4
  r5 := 0
loop:
  if r3 ≥ r4 goto end
  r6 := mem(r3)
  r5 := r5 + r6
  r3 := r3 + 4
  jmp loop
end:
```

- r1 is a pointer to the beginning of an array
- r2 holds the array's length
- The program computes in r5 the sum of the array items.

```
int *end = arr + len * 4;
int sum = 0;
while (arr < end) {
    sum += *arr;
    arr++;
}
```

Optimization: Integrity-only Isolation

- A program performs many more reads than writes.
  - In SPEC2006, 50% instructions perform some memory reads or writes; only 10% perform memory writes [Jaleel 2010]
- For integrity, check only memory writes
- Sufficient when confidentiality is not needed or less of a concern.
- Much more efficient
  - [Wahbe et al. 1993] on MIPS using typical C benchmarks
    - 22% execution overhead when checking both reads and writes; 4% when checking only writes
  - PittSFIeld on x32 using SPECint2K
    - 21% execution overhead when checking both reads and writes; 13% when checking only writes

Jaleel 2010: https://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf

Optimization: Data Region Specialization

- Example: `DB = 0x12340000 ; DL = 0x1234FFFF`
  - The data region ID is `0x1234`
- `r6 = mem(r3)` becomes

```
r10 = r3 >> 16 // right shift 16 bits to get the region id
if r10 != 0x1234 goto error
r6 = mem(r3)
```

Optimization: Address Masking

- Address checking stops the program when the check fails
  - Strictly speaking, unnecessary for isolating faults
- A more efficient way: force the address of a memory operation to be a DR address and continue execution
  - Called **address masking**
  - "Ensure, but don't check."
    - When using data region specialization, just modify the upper bits in the address to be the region ID
    - PittSFIeld reported 12% performance gain when using address masking instead of checking for SPECint2000

Optimization: Address Masking

- Example: `DB = 0x12340000`; `DL = 0x1234FFFF`
  - The data region ID is `0x1234`
- Instead of

```
r10 = r3 >> 16 // right shift 16 bits to get the region id
if r10 != 0x1234 goto error
r6 = mem(r3)
```

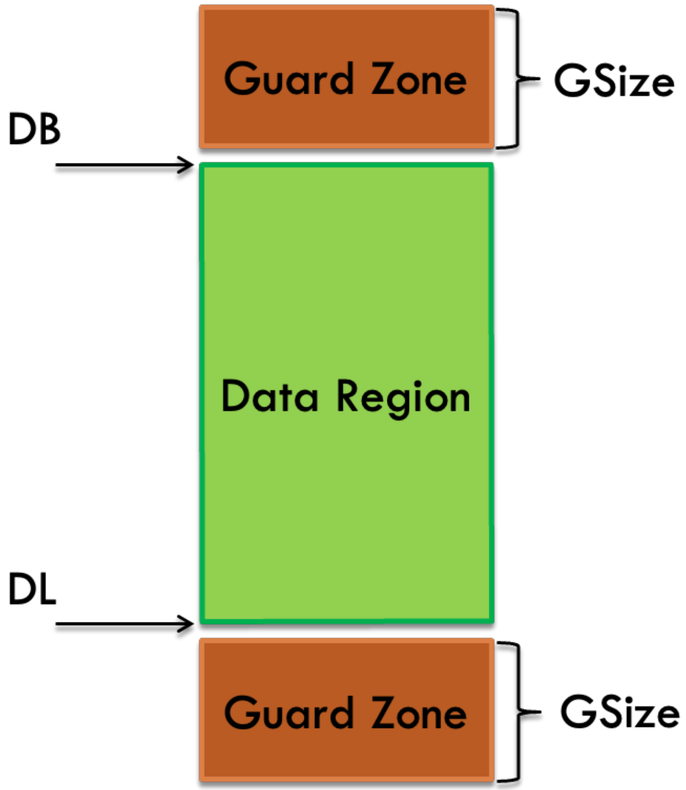
- `r6 = mem(r3)` becomes

```
r3 = r3 & 0x0000FFFF  // bit-mask to clear the first 16 bits
r3 = r3 | 0x12340000 // bit-mask to set the first 16 bits to 0x1234
r6 = mem(r3)
```

Data Guards

- A data guard refers to either address checking or address masking.
  - When which one is used is irrelevant.
- Introduce a pseudo-instruction “`r'=dGuard(r)`”
  - To hide implementation details
- An implementation should satisfy the following properties of `r'=dGuard(r)`”
  - If r is in DR, then r’ should equal r
  - If r is outside DR, then
    - For address checking, an error state is reached.
    - For address masking, `r'` gets an address within the safe range
    - The safe range is implementation specific; it’s often DR.

Guard Zones Enable More Optimizations



- In-place sandboxing
- Redundant check elimination
- Loop check hoisting

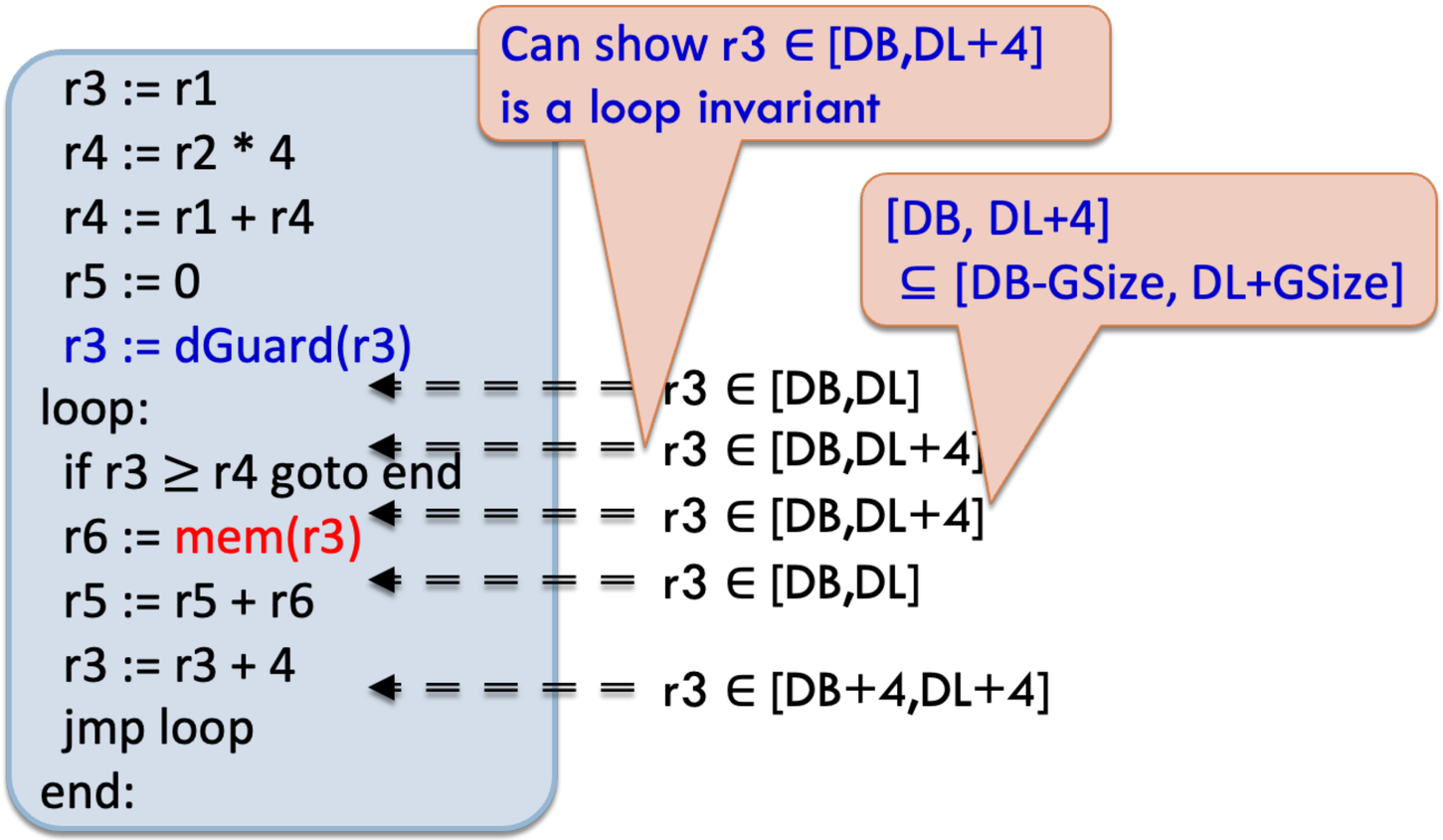
Optimization: In-place Sandboxing

- Example: `r6 = mem(r3 + 12)` becomes

```
r3 = dGuard(r3)
r6 = mem(r3 + 12)
```

- Why is the above safe?
  - "`r3 := dGuard(r3)`" constrains r3 to be in DR and then `r3+12` must be in [DB-GSize, DL+GSize], assuming GSize ≥ 12.

Why is the Optimized Code Safe?



Optimization: Guard Changes Instead of Uses

- Some registers are changed rarely but used often.
  - E.g., in 32-bit code, ebp is usually set in the function prologue and used often in the function body.
- Sandbox the changes to those special registers, instead of uses
  - E.g., ebp = esp becomes

```
ebp = esp
ebp = dGuard(ebp)
```

  - Later uses of %ebp plus a small constant do not need to be guarded, if used together with guard zones.

Anything Vulnerable about This Program?

```
  r3 := r1
  r4 := r2 * 4
  r4 := r1 + r4
  r5 := 0
  r3 := dGuard(r3)
loop:
  if r3 ≥ r4 goto end
  r6 := mem(r3)
  r5 := r5 + r6
  r3 := r3 + 4
  jmp loop
end:
```

What if a control flow hijacking (e.g., by corrupting an return address) causes the control flow to jump over dGuard and directly go to the memory access?

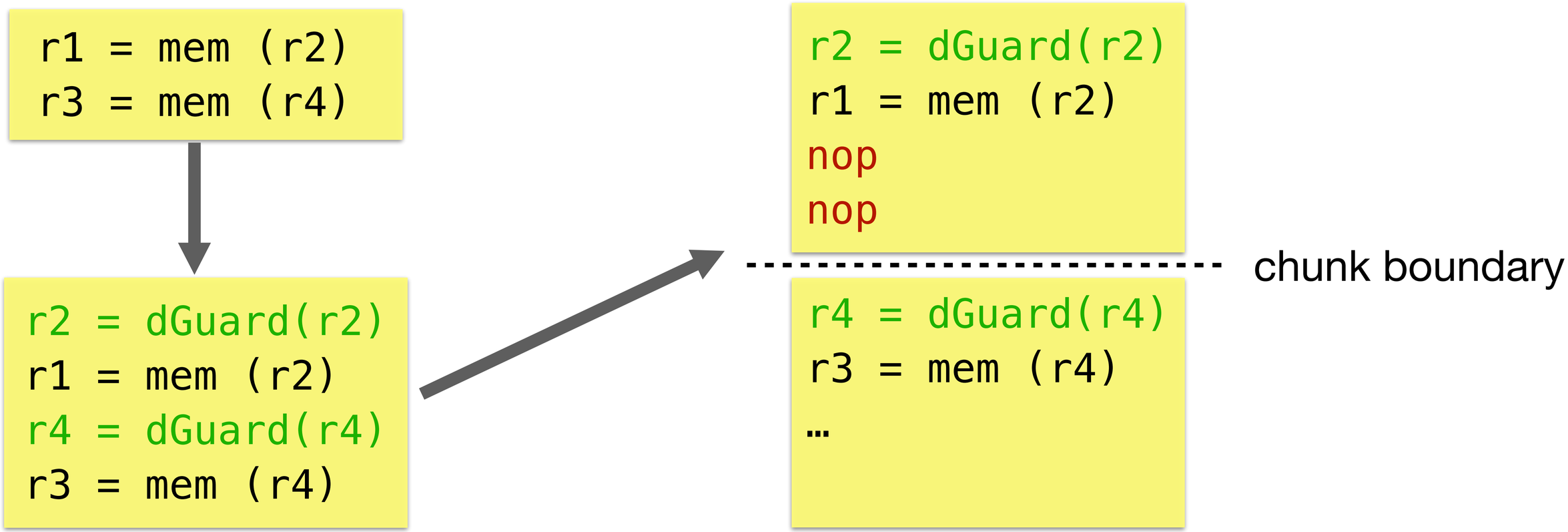
CFI is often needed for other security policies such as SFI.

Align-chunk Enforcement

- Divide the code into chunks of some size
  - E.g., 16 or 32 bytes
- Each chunk starts at an aligned address
  - So we can force an address to align by chunkSize with "`addr / chunkSize`"
- Make dangerous instructions and their guards stay within one chunk.
  - E.g., "`r10 := dGuard(r10); mem(r10) := r2`" stay within one chunk
- Insert guards before indirect branches so that they target only aligned addresses (chunk beginnings)

Downside of Align-chunk Enforcement

- All legitimate jump targets have to be aligned.
  - No-ops have to be inserted for that.
- E.g., assuming a 16-byte chunk, and each instruction is 4-byte long.



Hardware-enforced Memory Access Configuration

- Memory access permissions, e.g., read/write/executable, checked and enforced by hardware
- Two primary types:
  - Memory Management Unit (MMU)
    - Supports virtual address
    - Mainly for general-purpose computing systems, such as desktops/smartphones
  - Memory Protection Unit (MPU)
    - Flat address space—no virtual address
    - Mainly for low-end embedded systems

Example: Intel x64 PTE That Maps a 4-KB Page

63	62…59	58…52	51…M	M-1…12	11…9	8	7	6	5	4	3	2	1	0
XD	PK	Reserved	Reserved	PFN	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

- P: Present
- R/W: Read/Write
- PWT: Page-level write-through
- PCD: Page-level cache disable
- A: Accessed
- D: Dirty
- G: Global
- AVL: Available for software to define
- PFN: Physical frame number (physical address)
- PK: Protection key (if supported)
- XD: execute-disable

Configure Page Table Entry to Isolate Memory

- With data guards, memory access "`mem(r1) = r2`" becomes

```
r1 = dGuard(r1)
mem(r1) = r2
```

- Alternative: Before `mem(r1)`, set protected memory domains outside of r1 to unwritable, and resume the permission afterwards.
  - In *nix systems, use `mprotect()` syscall.

```
mprotect(protected_mem, PROT_NONE);
mem(r1) = r2
mprotect(protected_mem, PROT_READ | PROT_WRITE);
```

  - Pros: Strong protection
  - Cons: High performance penalty; introducing security hazards

Example: Intel x64 PTE That Maps a 4-KB Page

63	62…59	58…52	51…M	M-1…12	11…9	8	7	6	5	4	3	2	1	0
XD	PK	Reserved	Reserved	PFN	AVL	G	PAT	D	A	PCD	PWT	U/S	R/X	P

- P: Present
- R/W: Read/Write
- PWT: Page-level write-through
- PCD: Page-level cache disable
- A: Accessed
- D: Dirty
- G: Global
- AVL: Available for software to define
- PFN: Physical frame number (physical address)
- **PK: Protection key (if supported)**
- XD: execute-disable

Intel Memory Protection Key (MPK)

- A *protection key* represents an access permission configuration.
  - E.g., PK2 set to read-only, and PK5 set to read + write
- Memory pages are divided into different groups.
- A group of pages are associated with a protection key.
- Supports up to 16 protections keys (bits 59–62 in PTE)
  - I.e., 16 different protection domains
- Register `pkru` (Protection Key Rights for User Pages) for memory access checks
  - 32-bit register
  - Every two bits represents the memory access permission of one PK.
    - first bit: Access Disabled (AD) when set to 1
    - second bit: Write Disabled (WD) when set to 1

pkru Register

- E.g., a PTE's has PK13, and PK13 is 10
  - Meaning this page of memory is set to be read-only
- E.g., a PTE's has PK5, and PK5 is 00
  - Meaning this page of memory is set to be readable and writable

How to Manage MPK

- Use syscall `pkey_mprotect()`

```
int pkey_mprotect(void *addr, size_t len, int prot, int pkey);
```

  - Convenient, but slow (~1,100 CPU cycles)
- Directly manipulating `pkru` using the `wrpkru` instruction
  - Allows a program to change the memory access permissions for selected PK
    - `eax` contains the new PK value to be set
    - `ecx` and edx must be 0
  - Fast (~23 CPU cycles)
- `rdpkru` is used to read PKs into `eax`.

Configure Page Table Entry to Isolate Memory

- With data guards, memory access "`mem(r1) = r2`" becomes

```
r1 = dGuard(r1)
mem(r1) = r2
```

- Alternative: Before `mem(r1)`, set protected memory domains outside of r1 to unwritable, and resume the permission afterwards.
  - In *nix systems, use `mprotect()` syscall.

```
mprotect(protected_mem, PROT_NONE);
mem(r1) = r2
mprotect(protected_mem, PROT_READ | PROT_WRITE);
```

  - Pros: Strong protection
  - Cons: High performance penalty; introducing security hazards

Example of Protecting Memory Domain with MPK

- Assume protected memory domain is associated with `pk1`.
- For dangerous instruction "`mem(r1) = r2`", it becomes

```
xor %ecx, %ecx
xor %edx, %edx
rdpkru
or %eax, 0x00000004
wrpkru
mem(r1) = r2
… // recover original PKs
```

Anything vulnerable about this solution?

How to make sure your MPK gate instructions (those transitioning to/from a specific PK configuration) are respected?

Check the optional readings for this lecture.

Least Privilege Principle

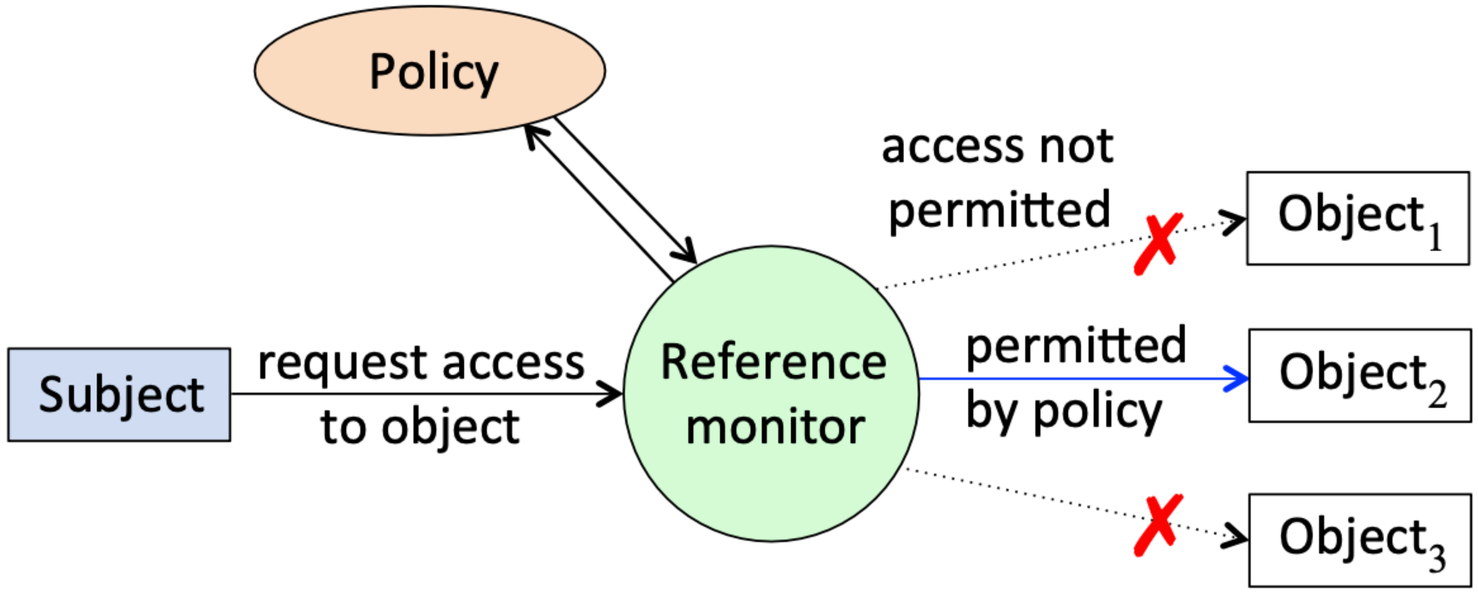
Outline: Principle of Least Privilege

- 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*?

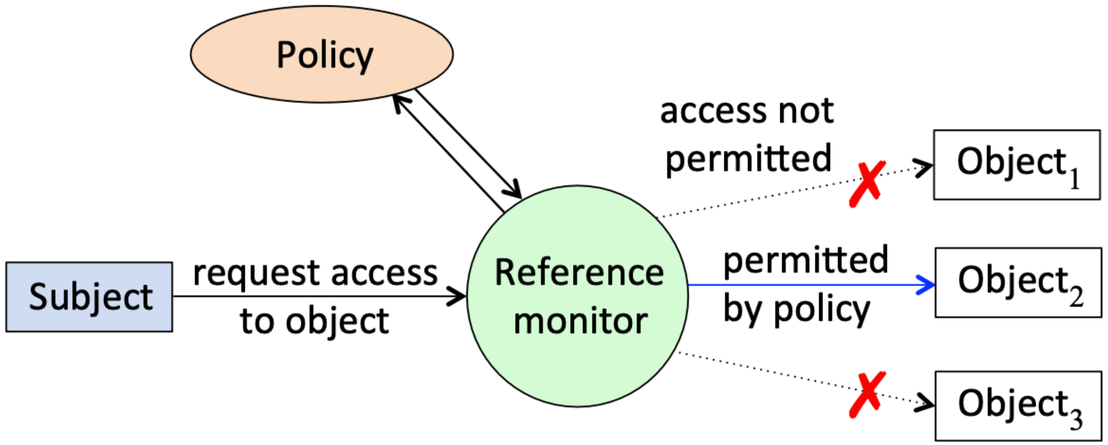
Reference Monitor

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



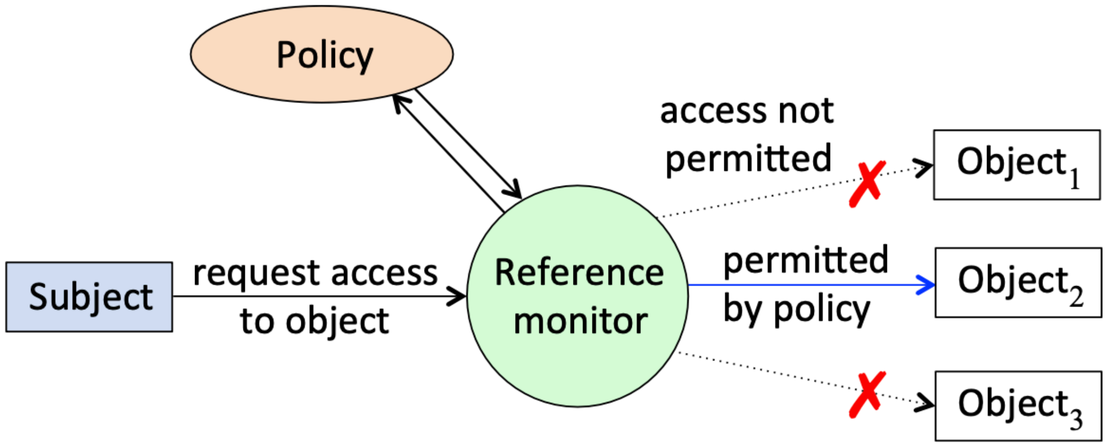
Terminology

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



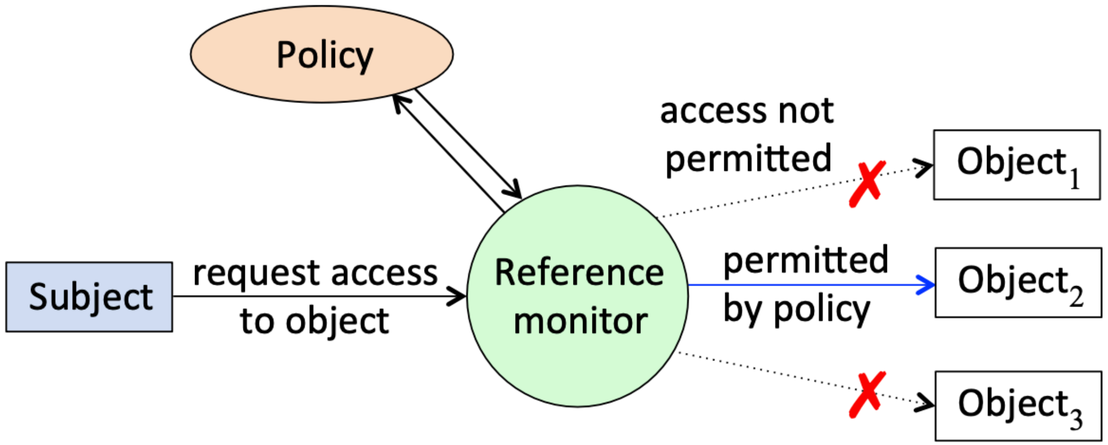
Terminology

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



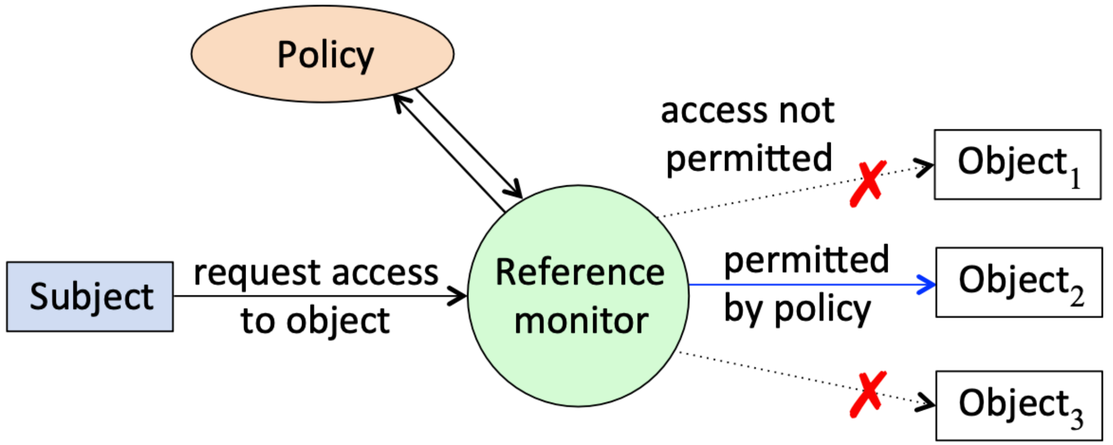
Terminology

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



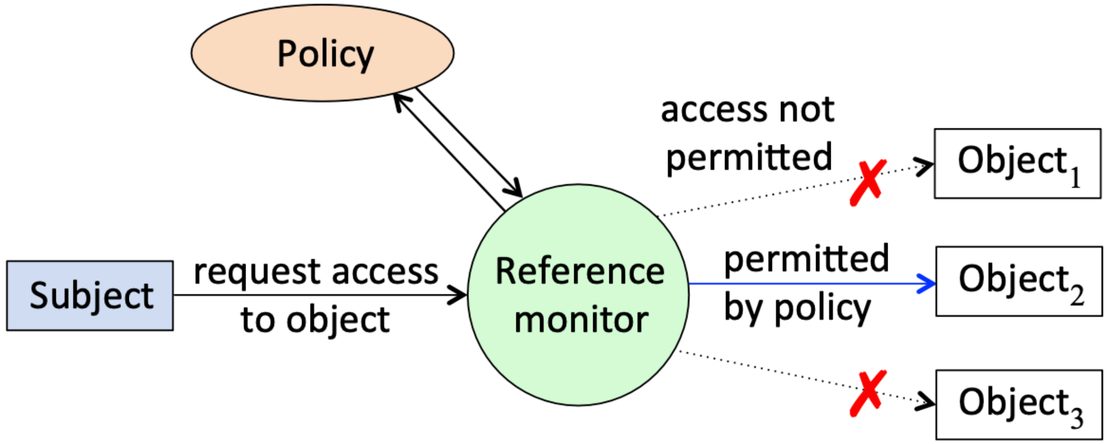
Terminology

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



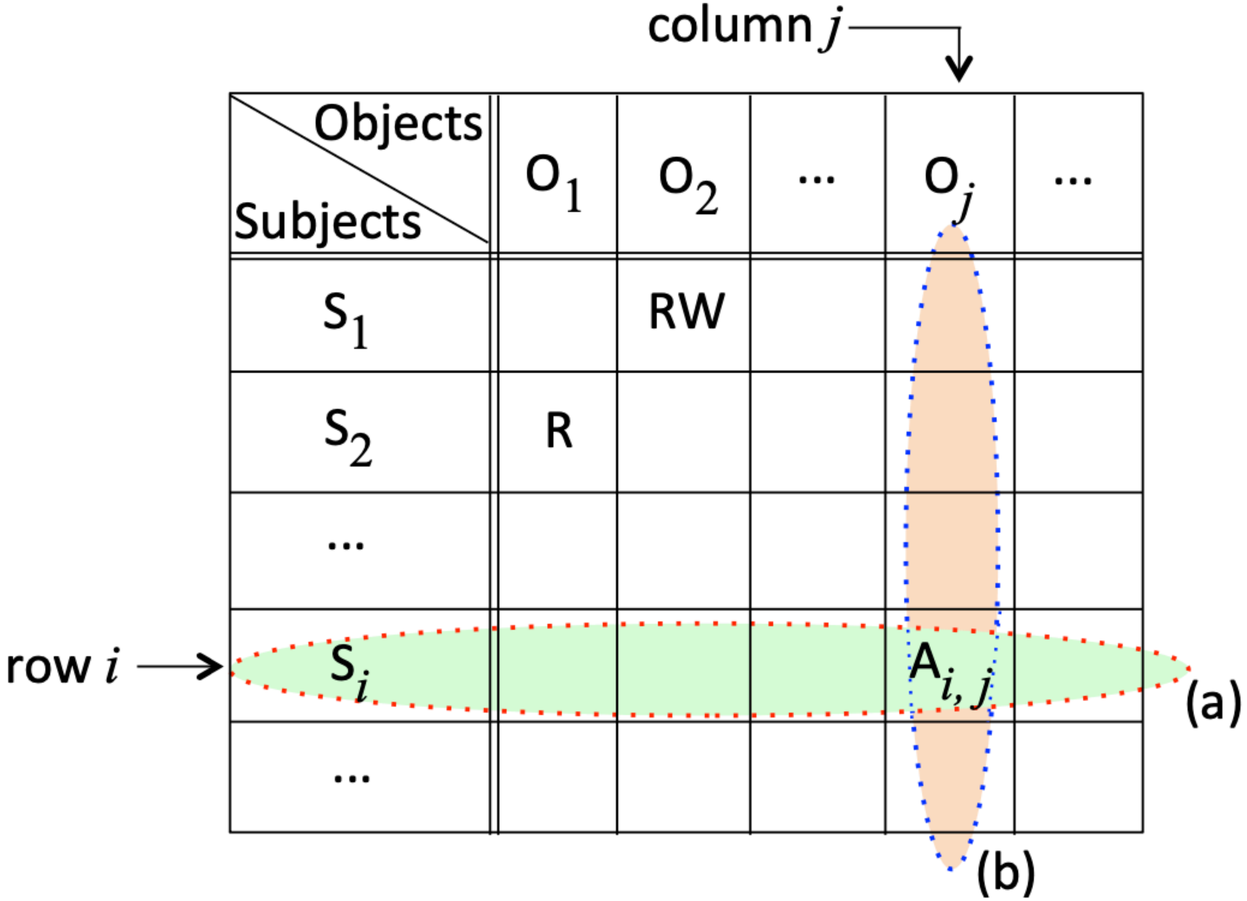
Terminology

- 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



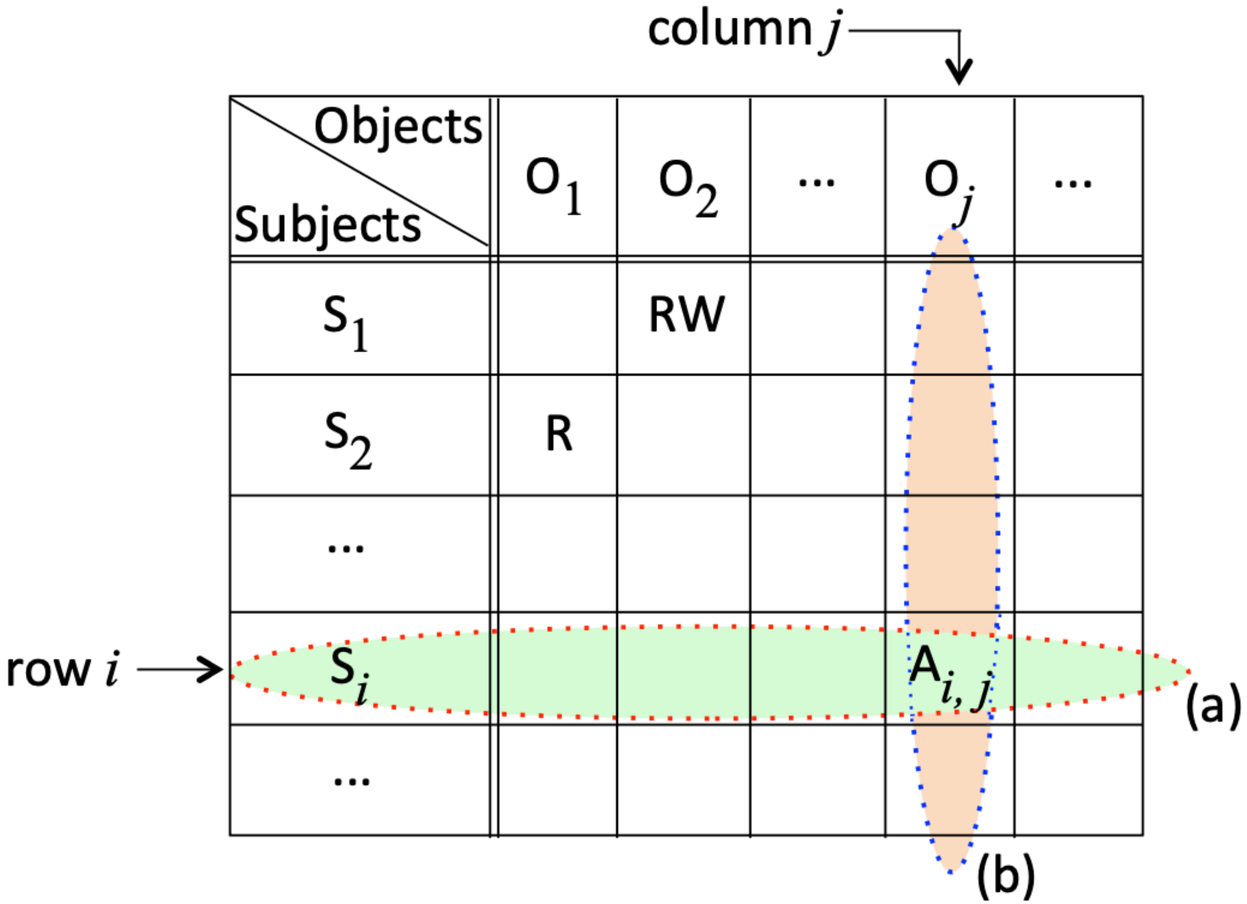
Access Matrix

- 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 | ˈpriv(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

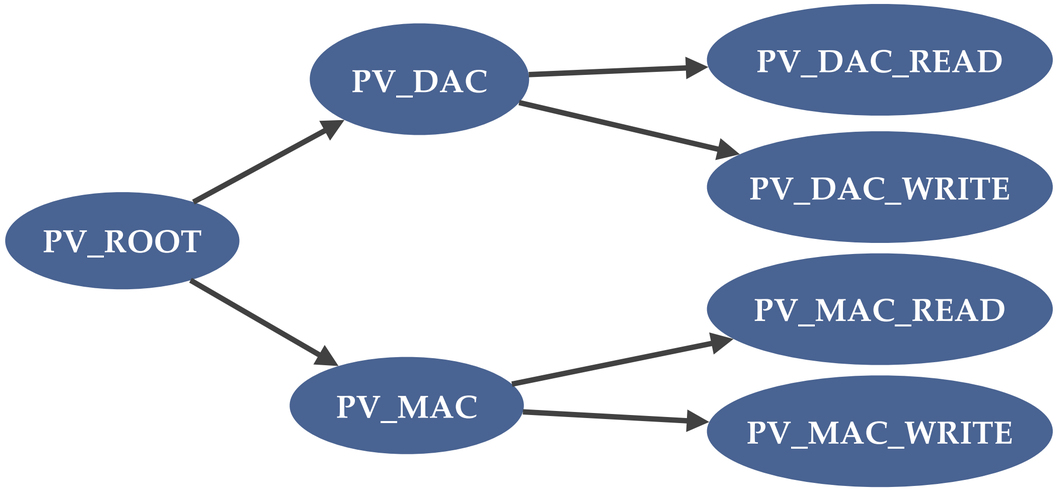
Medium-grained Privileges (Linux)

- 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.

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
    - Don't 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”

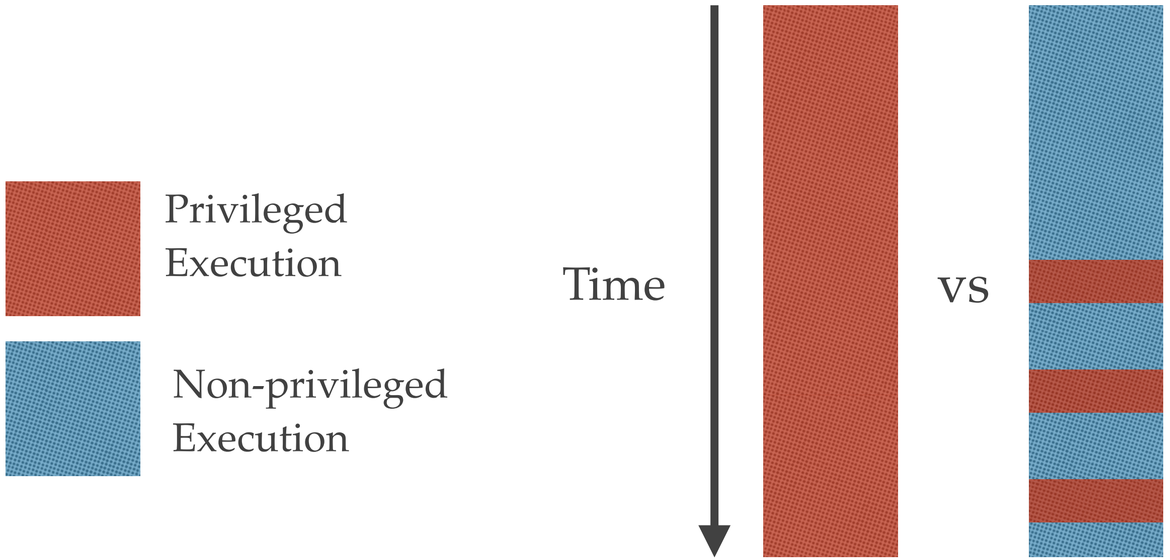
- To disable root, swap EUID and SUID
- To enable root, swap EUID and SUID

Enabling and Disabling Unix “Privilege”

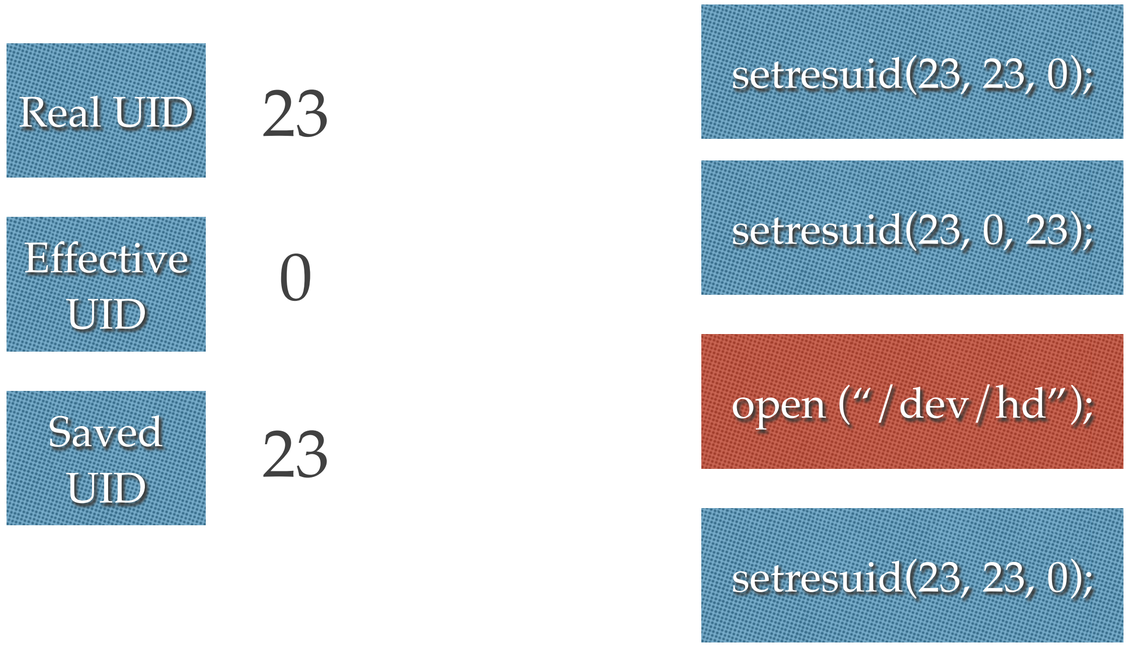
- To disable root, swap EUID and SUID
- To enable root, swap EUID and SUID

Privilege Bracketing

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

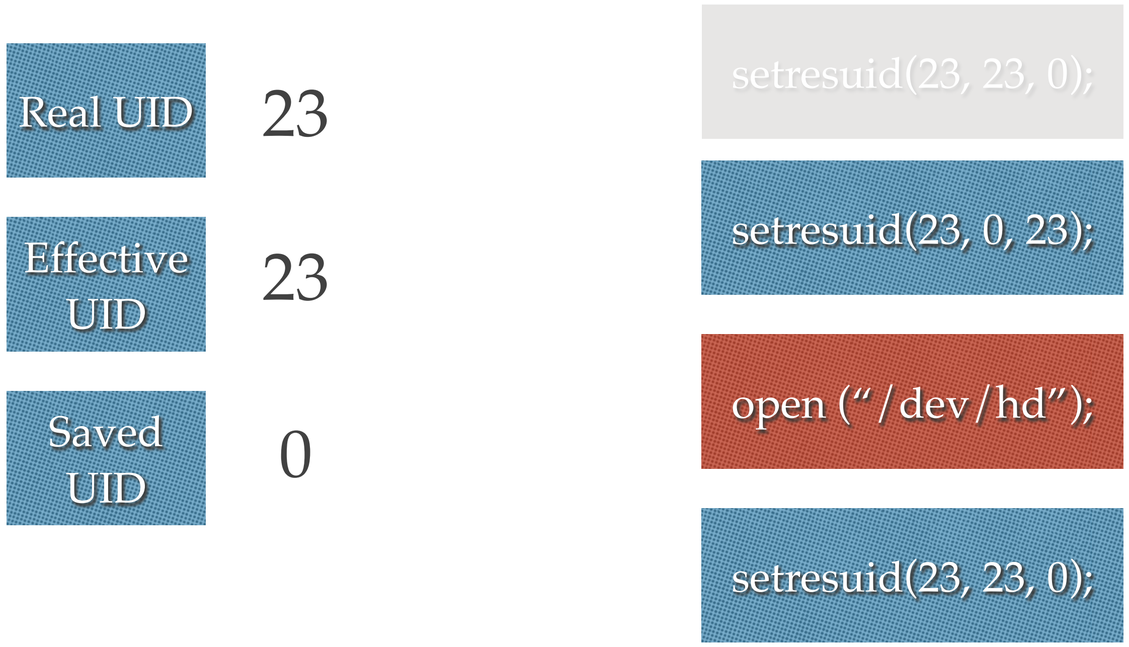


Unix Privilege Bracketing: 1



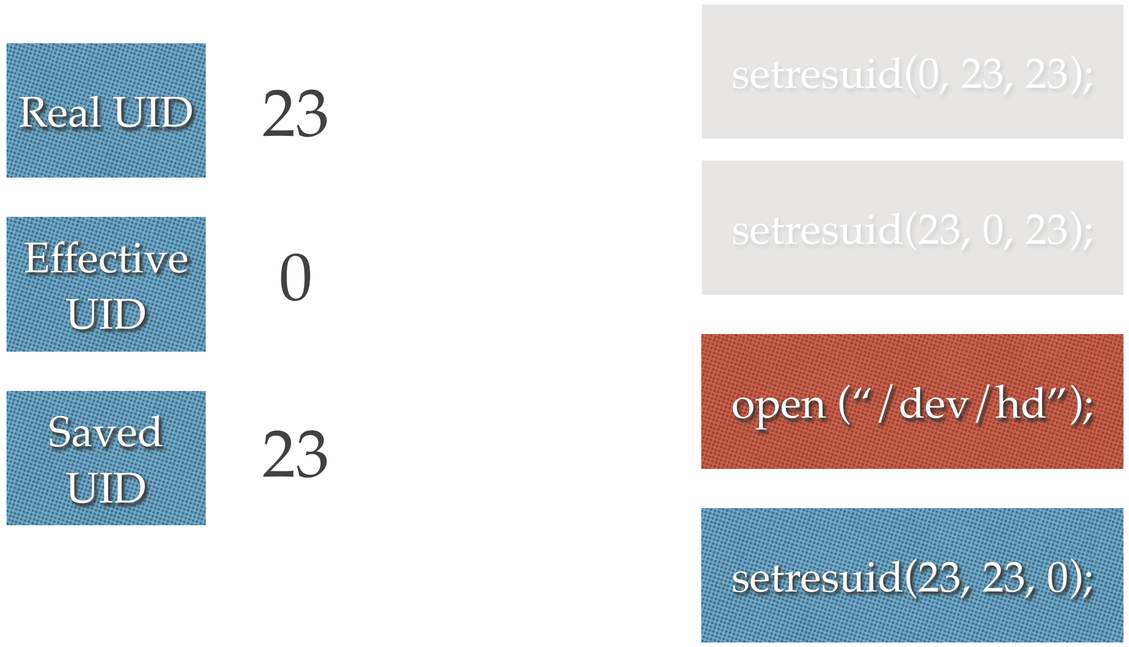
- `setresuid()` sets the real user ID, the effective user ID, and the saved user ID of the calling process.

Unix Privilege Bracketing: 2



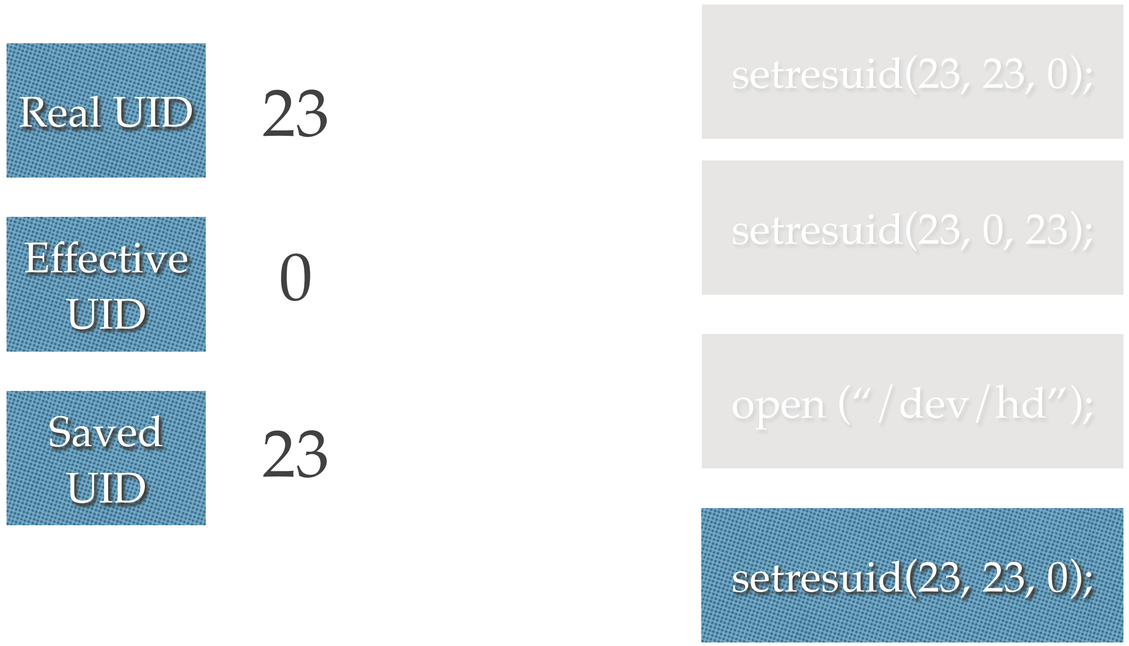
- `setresuid()` sets the real user ID, the effective user ID, and the saved set-user-ID of the calling process.

Unix Privilege Bracketing: 3



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

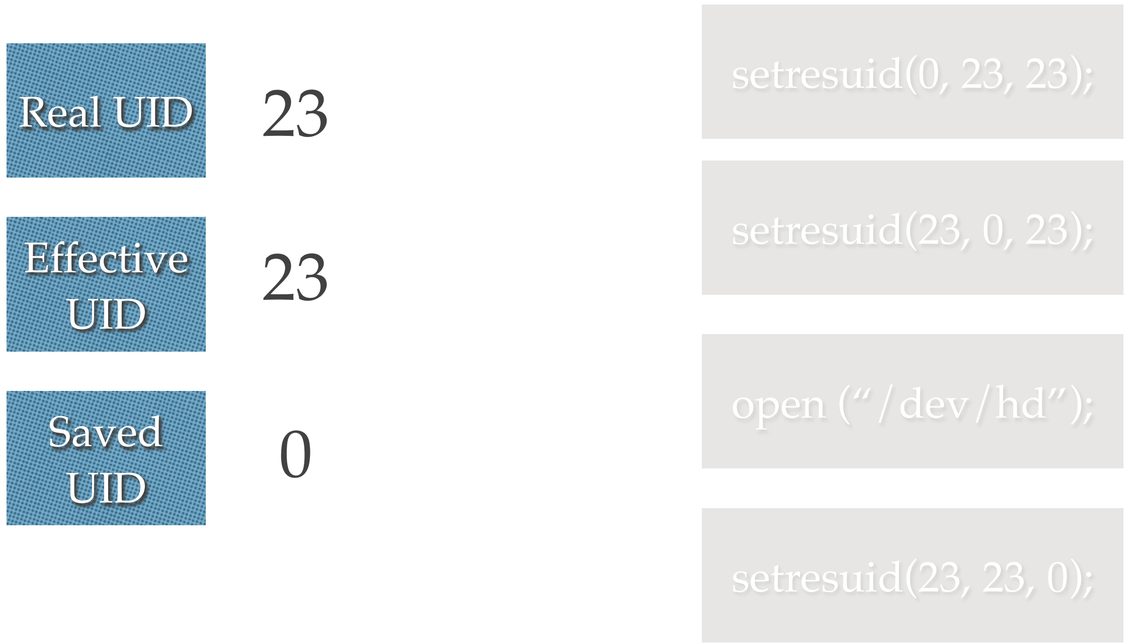
Unix Privilege Bracketing: 4



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

Unix Privilege Bracketing: 5

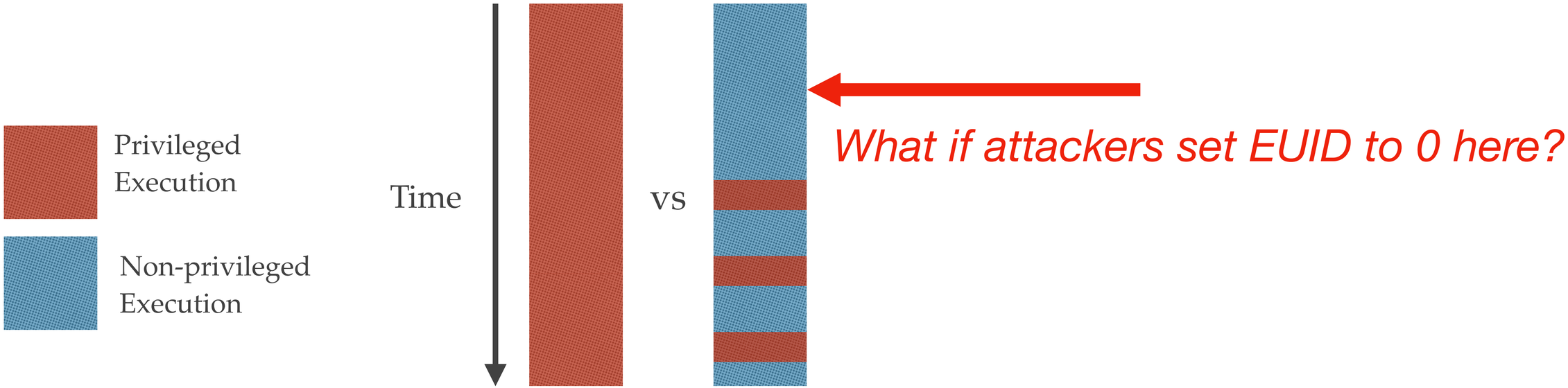
What if attackers turn on the root UID at unexpected points?



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

Privilege Bracketing

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



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?**

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

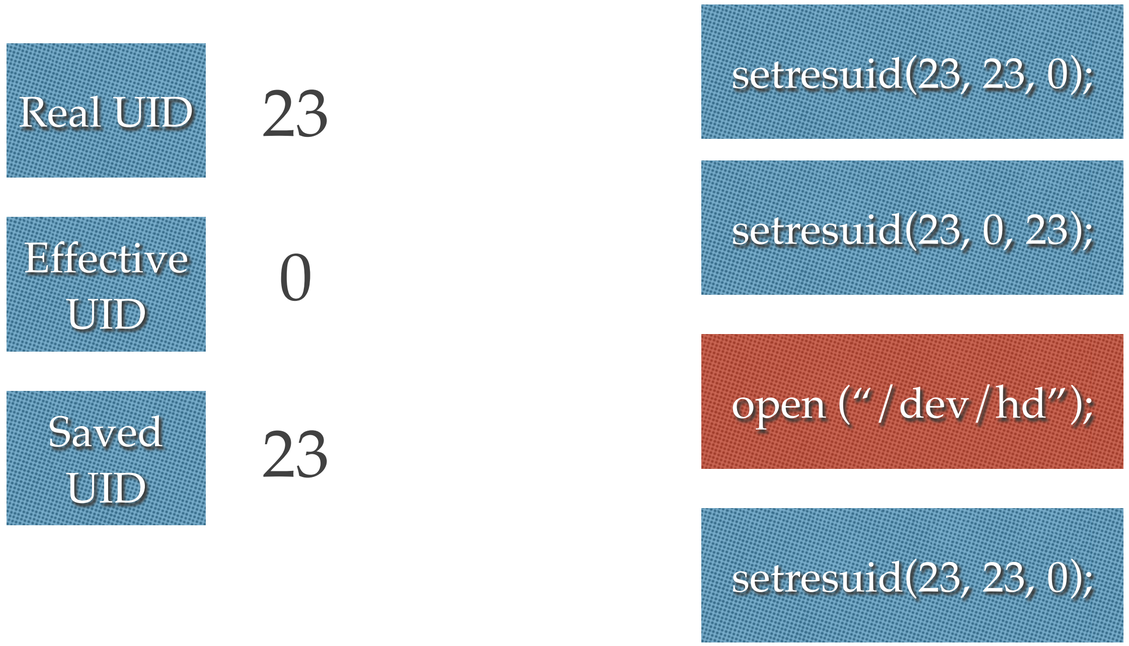
Operations on Privilege Sets

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);

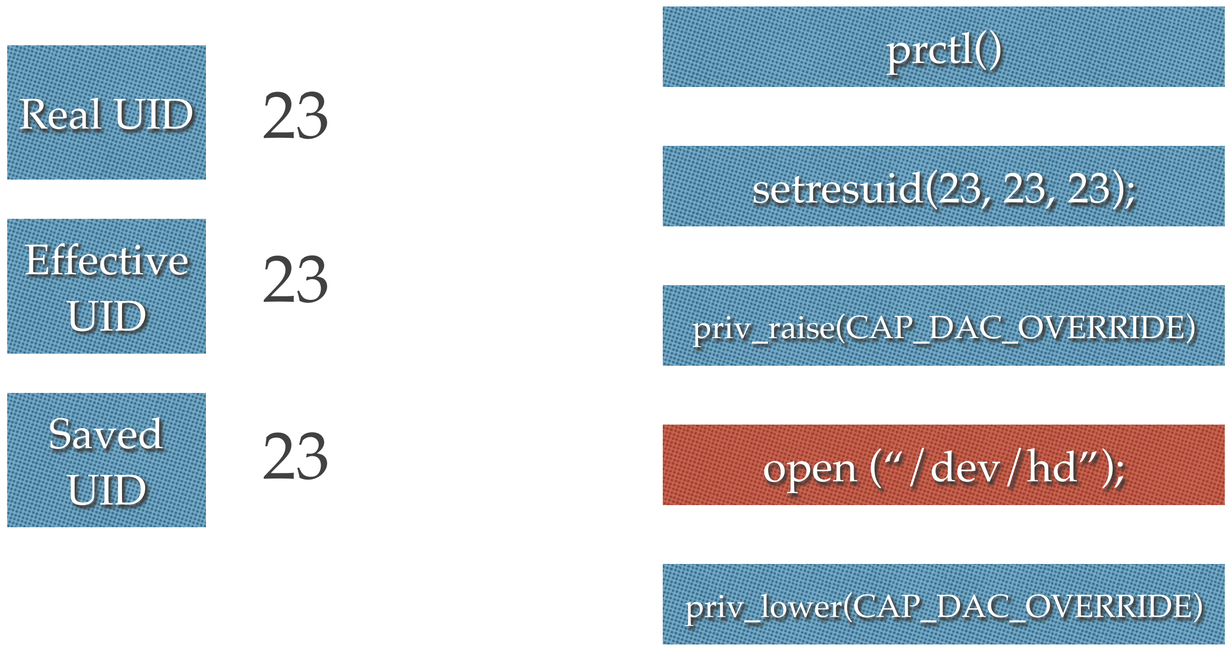
priv_raise (CAP_DAC_READSEARCH);
open (“/etc/passwd”, O_RDONLY);
priv_lower(CAP_DAC_READSEARCH);

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 vulnerable about 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

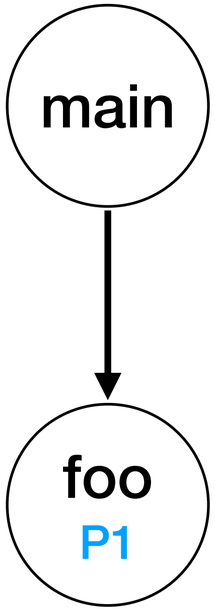
- Unix
  - Set EUID, RUID, and SUID to non-zero values
- Linux
  - Remove privilege from effective and maximum set

Remove Privileges

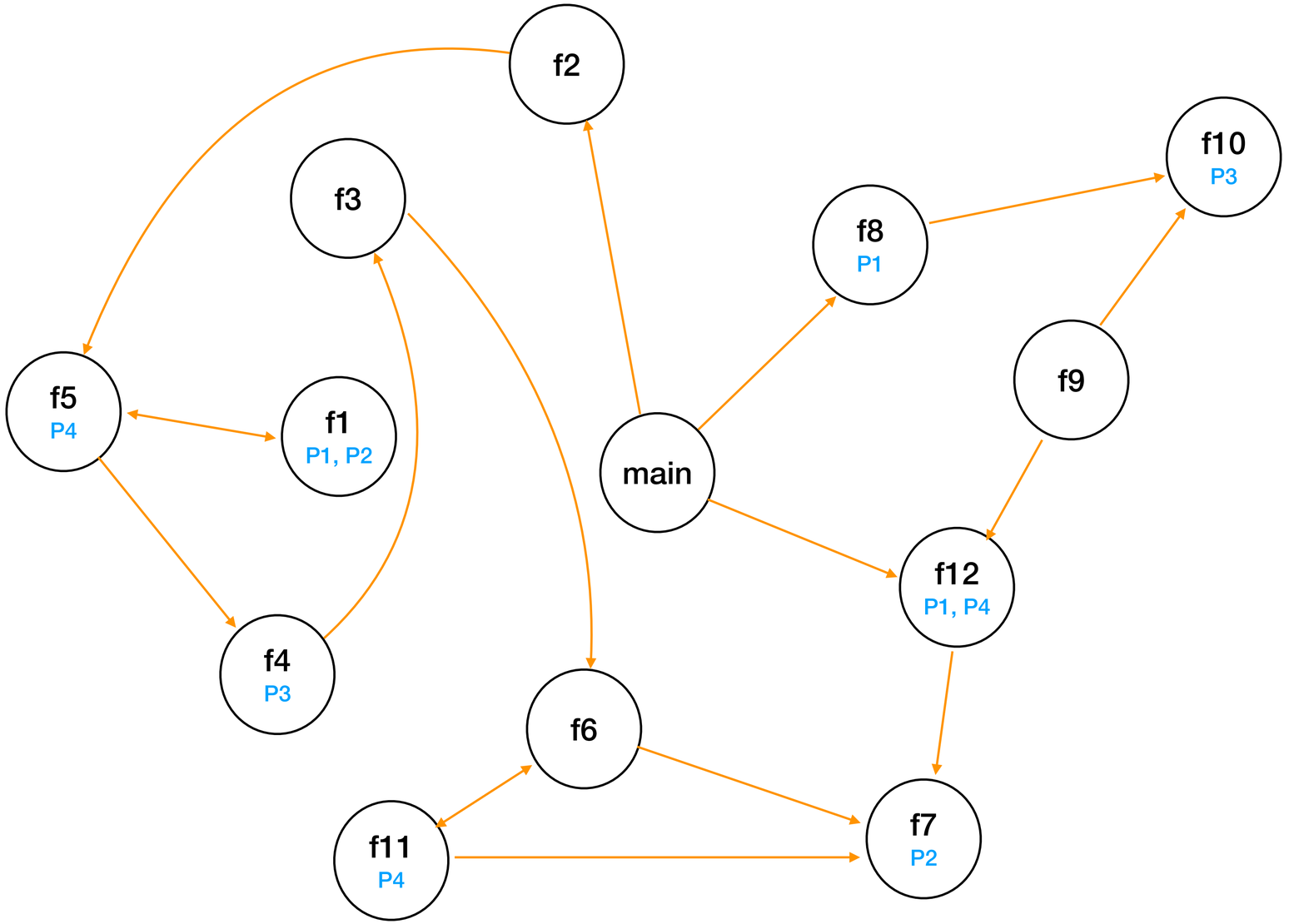
```
int main() {
    ......
    foo();
    ......
}



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

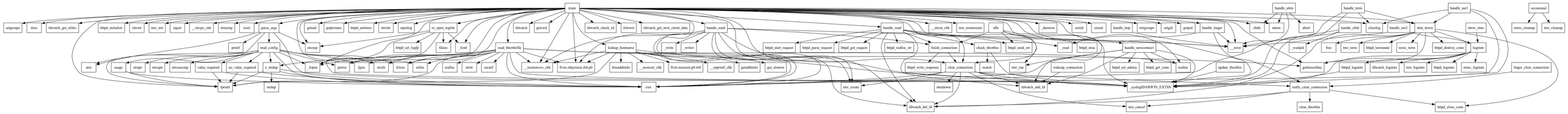


Insert *priv_remove(P1)* after P1 is no longer needed.


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`

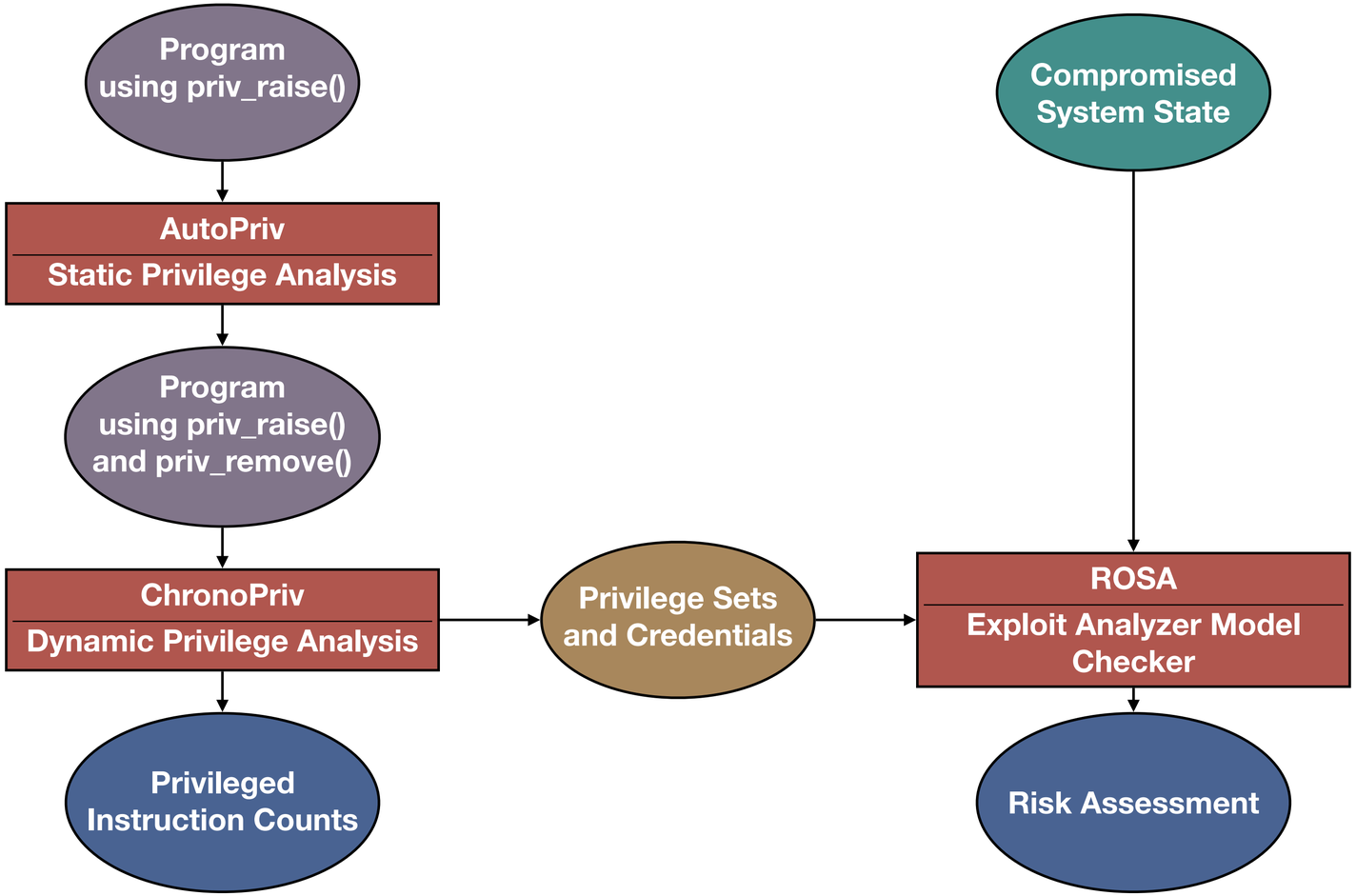
- It's extremely difficult to manually figure out when we can remove which privileges permanently.
- We need an *automated* tool.

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
  - Measured 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 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 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 needed late in execution

Strategy 1: Leverage Saved UID

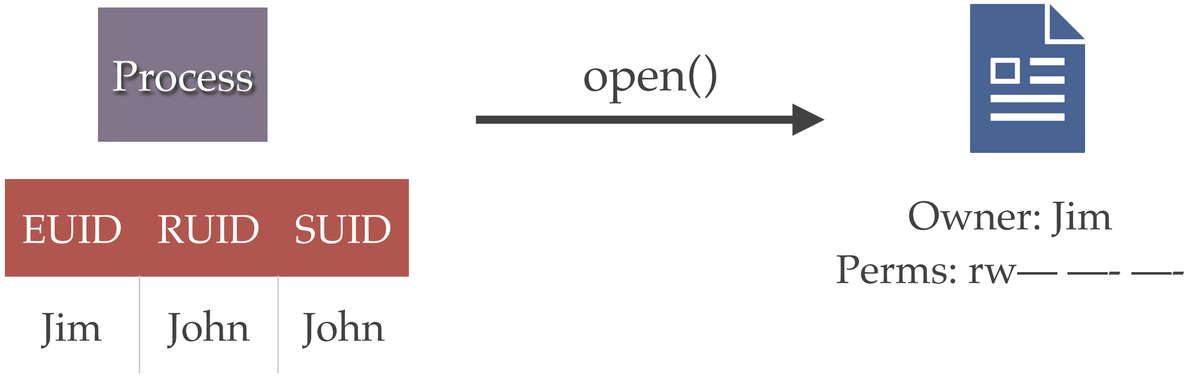
- Use privilege to place file owner UID in saved UID
- Remove `CAP_SETUID` from maximum privilege set
- Switch to saved UID to open file using *no privilege*

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*

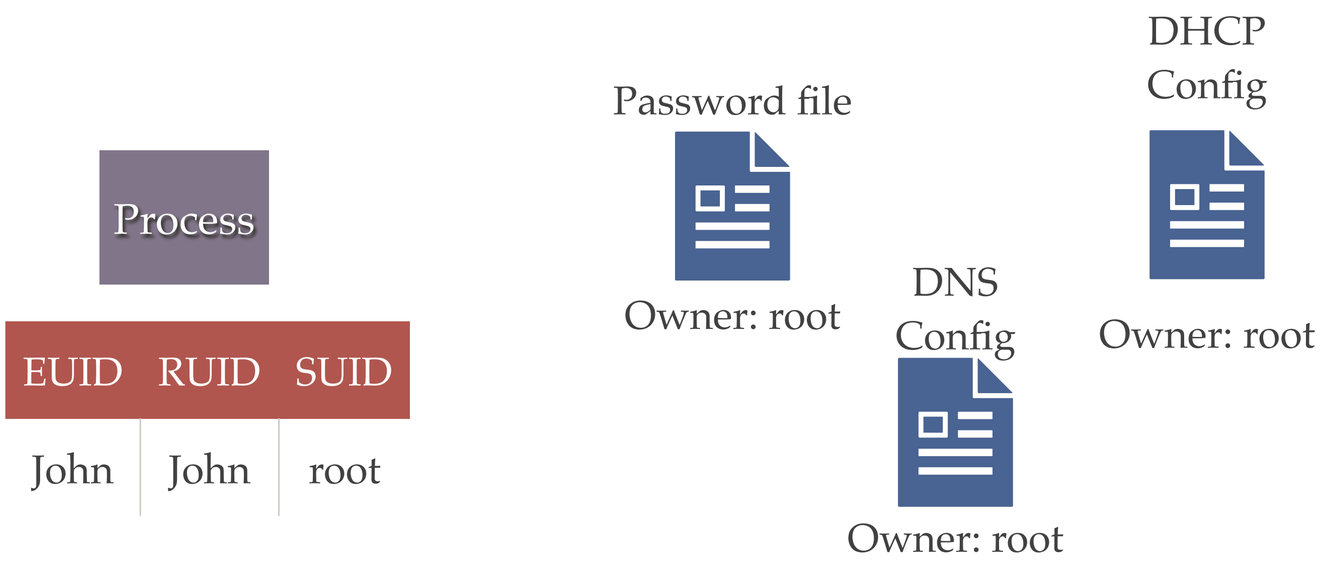


Strategy 1: Leverage Saved UID

- 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*

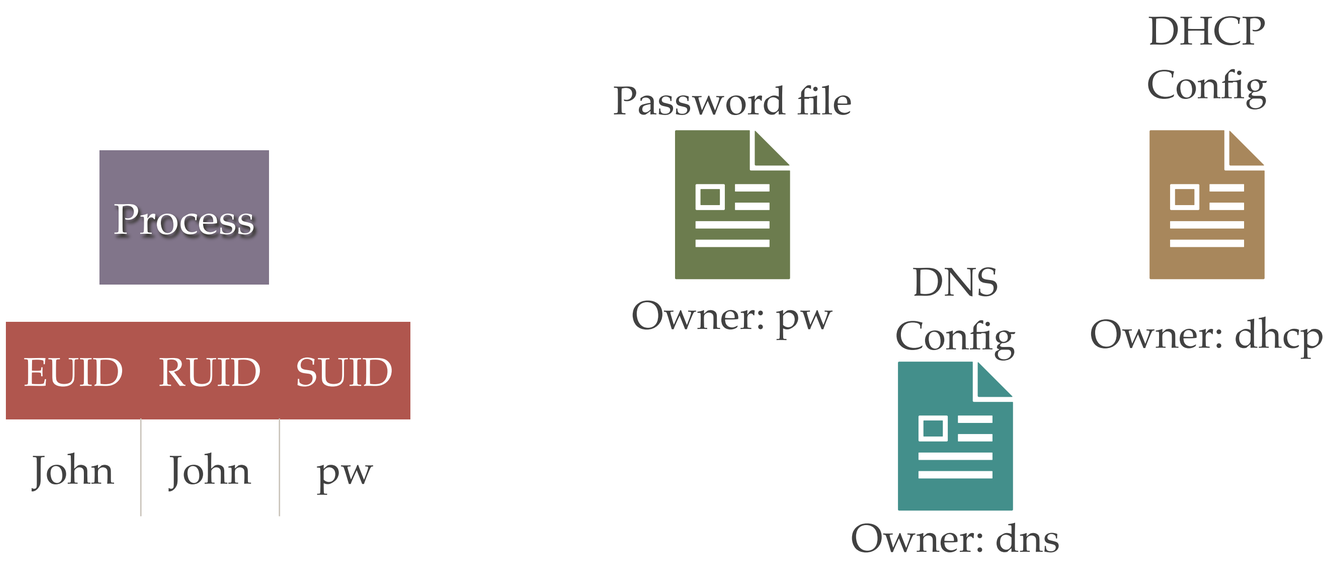
Strategy 2: Use Different File Owners

- 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 unit of privilege enforcement
  - E.g., a byte of memory, a file, a network socket
- Permissions: actions that a subject may perform on objects
  - E.g., read, write
- Protection domain: maximal set of subjects sharing the same permissions

Policy Definition Method (PDM)

PDMs identify subjects, objects, and permissions to enforce.

- Automation
- Policy languages
- Separation granularities
- Analysis techniques
- Subject selections
- Generality

Automation for Compartmentalization

- 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
- Policy-refinement methods
  - Developers write policies (e.g., isolating certain libraries) in a policy language.
  - User-provided policies are refined into concrete, low-level rules.
- Fully automated methods
  - Automatically analyze and understand programs and enforce policies.
  - Difficult to pinpoint boundaries

Policy Languages

- Allows developers to *describe* high-level policies.
  - Guide manual methods and policy-refinement methods
- Two types: annotations and placement rules
  - Annotations provide fine-grained semantics on subjects and objects
    - E.g., annotating a variable as confidential, a function as sensitive
    - Tightly coupled with program code
  - Placement rules provide corse-grained, high-level description of component trusted relationships and building rules.
    - E.g., place libraries X and Y in separate domains
    - Less dependent on program code
    - Expressed in many ways, e.g., JSON, XML
- In general, annotations express local, low-level semantics, whereas placement rules express full-system properties.

Separation Granularities

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

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

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 syscalls
- 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

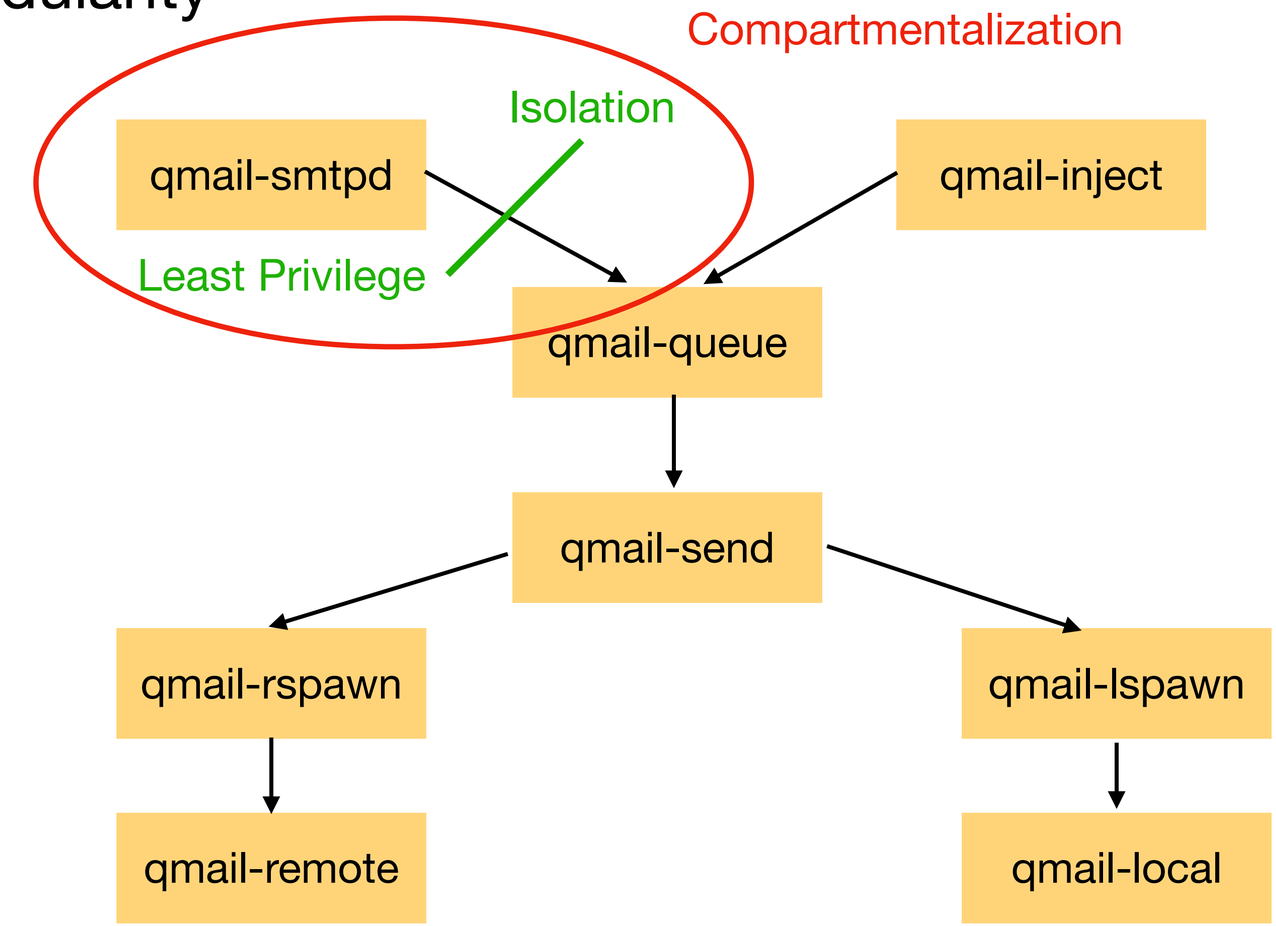
`Sendmail` uses a typical Unix approach with a large monolithic server and is known for the high complexity and previous security vulnerabilities.

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.