



CS 253 Cyber Security

Security Principles and OS Security

• ShanghaiTech University •

SIST - Yuan Xiao

Admin

- Presentations begin this week!

Vulnerabilities are Inevitable

Any single buffer overflow, use-after-free, or null pointer dereference might allow an attacker to run malicious code

We're getting better at finding and preventing bugs, but vulnerabilities are still common. There will **always** be bugs.

Example: In January 2021, Qualys discovered a heap overflow in `sudo` that allows users to run programs with the security privileges of another user. The bug was introduced in 2011 (CVE-2021-3156) and affected Linux, Mac OS, and BSD.

Even Safe Languages have Bugs!

Python language is written in C and has itself had vulnerabilities

CVE-2016-5636: Integer overflow in the `get_data` function allows attackers to trigger a heap-based buffer overflow in `zipimport.c` by specifying a negative data size

Bug could be triggered inside of interpreted Python scripts

**Systems must be designed to be
resilient in the face of both software
vulnerabilities and malicious users**

Threat Modeling

Threat model: A model of who your attacker is, what their motivations are, and what resources they have available to them

Why do people attack systems?

- Money
- Intellectual Property Theft
- Politics or Retaliation
- Fun, watching the world burn

What resources do those attackers have?

Need to be asking questions like:

- What are the most relevant threats?
- What do I need to do to safeguard against these threats?

01

PART ONE

Security Principles

Defense in Depth

Systems should be built with security protections at multiple layers

Example: What if there's a vulnerability in Chrome's Javascript interpreter?

- Chrome should prevent malicious website from accessing other tabs
- OS should prevent access to other processes (e.g., Password Manager)
- HW should prevent permanent malware installation in device firmware
- Network should prevent malware from infecting nearby computers

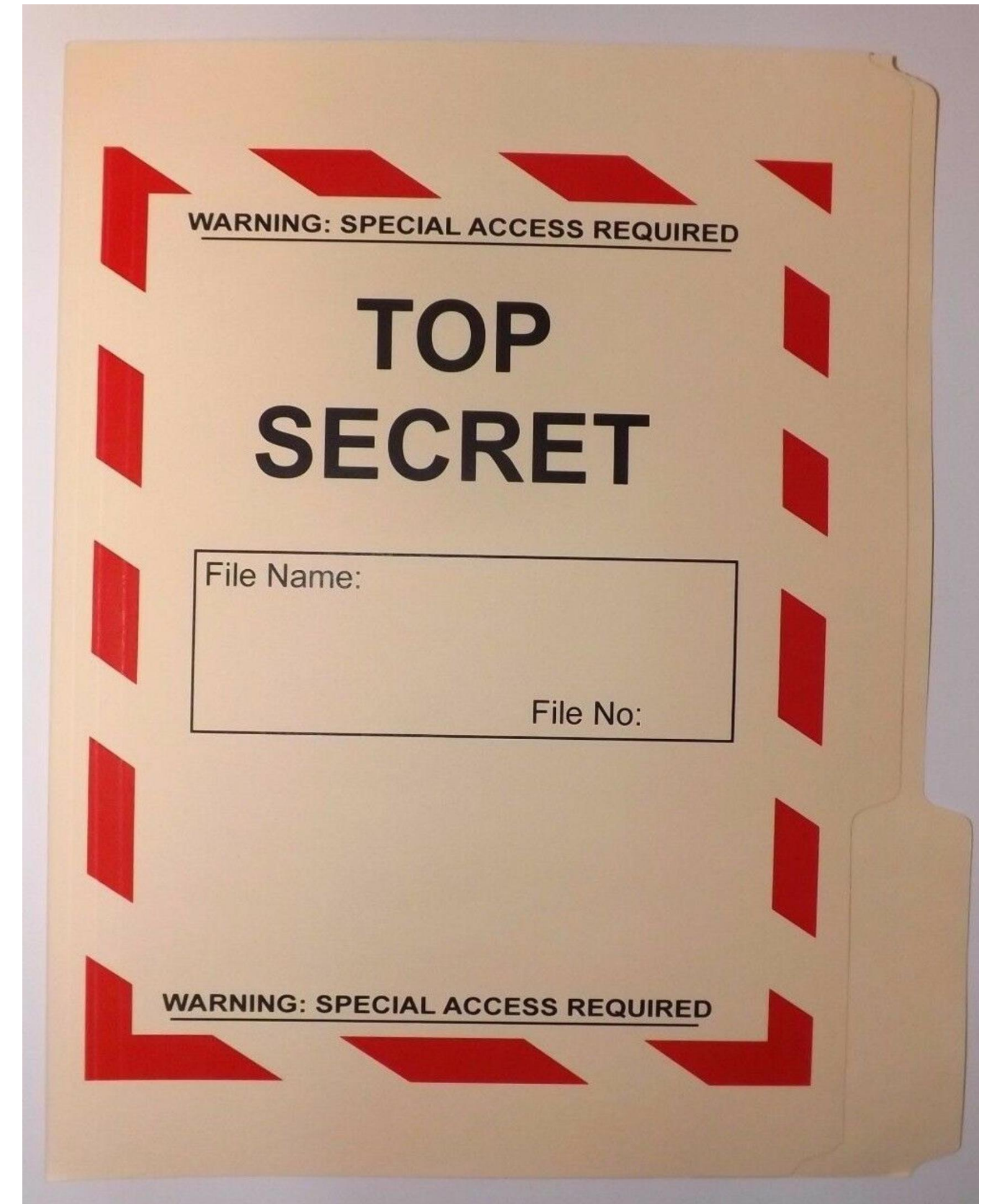


Least Privilege

Users and system components should only have access to the data and resources needed to perform routine, authorized tasks

Real World Examples:

- Faculty can only change grades for classes they teach
- Only employees with background checks have access to classified documents

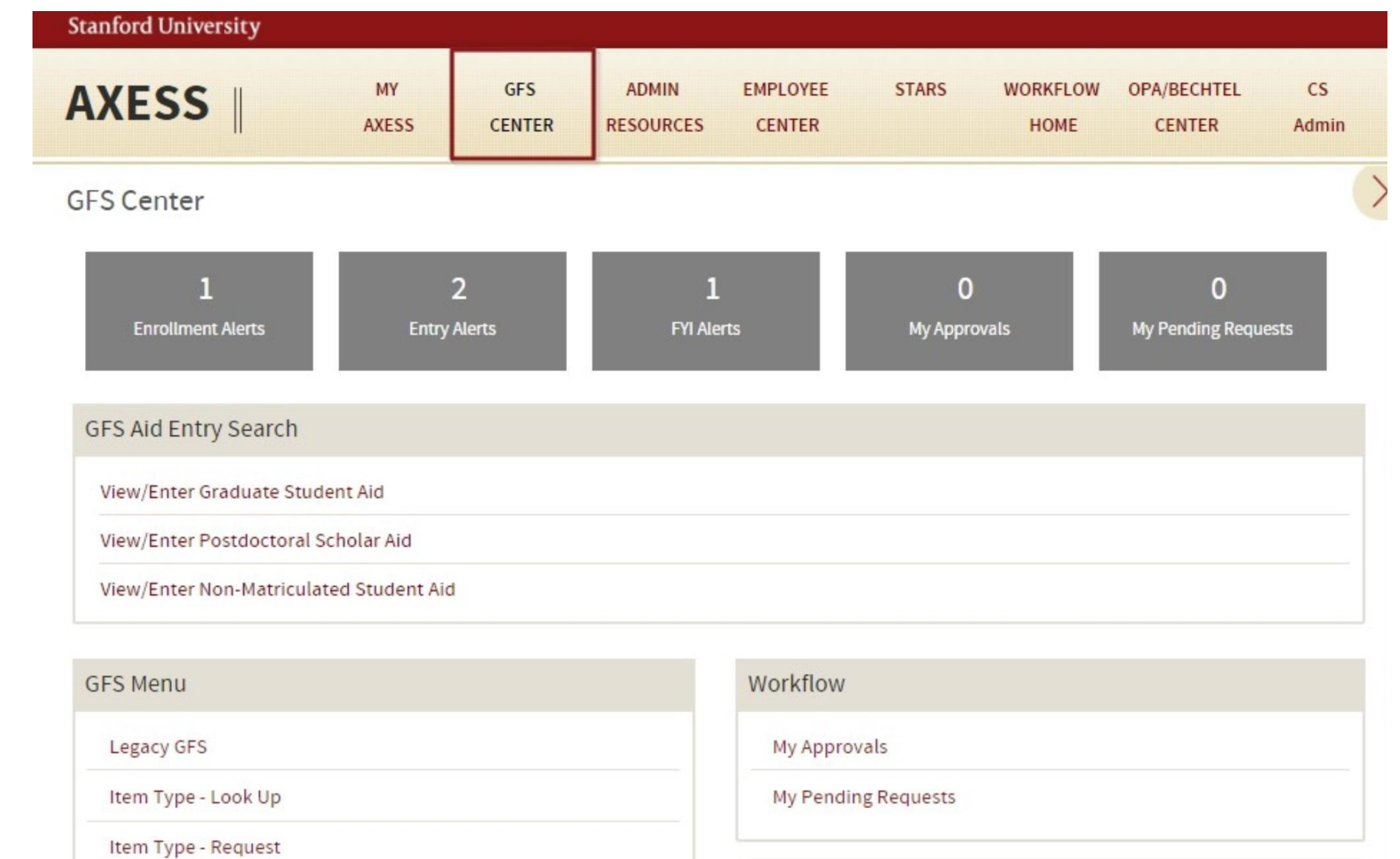


Least Privilege (2)

Faculty can only change grades for classes they teach.

Who are we really protecting against?

- Faculty themselves — curious or even malicious — could cause widespread damage
- External attackers — a student would need to own only the single least secure faculty member on campus — huge attack surface



Privilege Separation

Least Privilege requires dividing a system into parts to which we can limit access

Known as *Privilege Separation*

Segmenting a system into components with the least privilege needed can prevent an attacker from taking over the entire system



**This doesn't work
unless...**

Complete Mediation

All accesses to objects be checked to ensure that they are allowed

Generally means that there should be a single method of access that's well hardened

Ideally check immediately before access is granted



Fail Closed

What happens in an error condition?

What do you do if the power goes out?

Fail closed: No one can get in if the power is out

Fail open: Anyone can get in if the power goes out



How do we model and talk about access policies?

Security Subjects

Least privilege and privilege separation apply to more than just users!

- UNIX: A **User** should only be able to read their own files
- UNIX: A **Process** should not be able to read another process's memory
- Mobile: An **App** should not be able to edit another app's data
- Web: A **Domain** should only be able to read its own cookies
- Networking: Only a trusted **Host** should be able to access file server

Least Privilege: ~~Users~~ **Subjects** should only have access to access the data and resources needed to perform routine, authorized tasks

Security Policies

Subject (Who?): acting system principals (e.g., user, app, process)

Object (What?): protected resources (e.g., memory, files, HW devices)

Operation (How?): how subjects operate on objects (e.g., read, delete)

Example Security Policies:

- UNIX: A **User** should not be able to **delete** other users' **files**
- UNIX: A **Process** should not be able to **read** another process's **memory**
- Mobile: An **App** should only be able to **edit** its own **data**
- Web: A **Domain** should not be able to **read** another domain's **cookies**

02

PART TWO

UNIX Security Model

UNIX Security Model

Subjects (Who?)

- Users, processes

Objects (What?)

- Files, directories
- Files: sockets, pipes, hardware devices, kernel objects, process data

Access Operations (How?)

- Read, Write, Execute

Users

UNIX systems have many accounts

- Service accounts
 - Used to run background processes (e.g., web server)
- User accounts
 - Typically tied to a specific human

Every user has a unique integer ID — User ID — UID

UID 0 is reserved for special user `root` that has access to *everything*

- Many system operations can *only* run as root

Example Users

You can view the users on your system by looking at `/etc/passwd`:

```
root:x:0:0:root:/root:/bin/bash
```

```
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin  
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin  
systemd-resolve:x:101:103:,,,:/run/systemd/resolve:/usr/sbin/nologin
```

```
yuan:x:1001:1001:Yuan Xiao,,,:/home/littlefish:/bin/bash  
oldwang:x:1009:1009:Chundong Wang,,,:/home/oldwang:/usr/sbin/nologin
```

Groups

UNIX has also groups — collections of users who can share files and other system resources

Every group has a group ID (GID) and name

```
zakhir@scratch-03:~$ id
uid=1001(zakhir) gid=1001(zakhir) groups=1001(zakhir),27(sudo),2000(esrg)
zakhir@scratch-03:~$ cat /etc/group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog
tty:x:5:
```

File Ownership

All Linux resources — sockets, devices, files — are managed as files

All files and directories have a single **user owner** and **group owner**

```
yuan@scratch-01:~$ ls -l
total 8
d rwx rwx --- 5 yuan cs253-tas 4096 Apr 2 15:56 homework
d rwx rwx --- 5 yuan cs253-instr 4096 Apr 2 15:56 grades
d rwx rwx r-x 11 yuan cs253-tas 4096 Dec 28 21:09 lectures
- rwx r-x r-- 1 yuan oldwang 0 Apr 11 04:15 test.py
```

User Owner

Group Owner

File Ownership

Three **subjects** have access to a file: **user owner**, **group owner**, **other**

Subjects can have three **operations**: **read**, **write**, **execute**

Owner can change permissions and group. **Root** can change user ownership.

```
yuan@scratch-01:~$ ls -l
total 8
d rwx rwx --- 5 yuan cs253-tas      4096 Apr  2 15:56 homework
d rwx rwx --- 5 yuan cs253-instr    4096 Apr  2 15:56 grades
d rwx rwx r-x 11 yuan cs253-tas     4096 Dec 28 21:09 lectures
- rwx r-x r-- 1 yuan oldwang        Apr 11 04:15 test.py
```

↑
User

↑
Group

↑
Other

↑
User

↑
Group

File Ownership

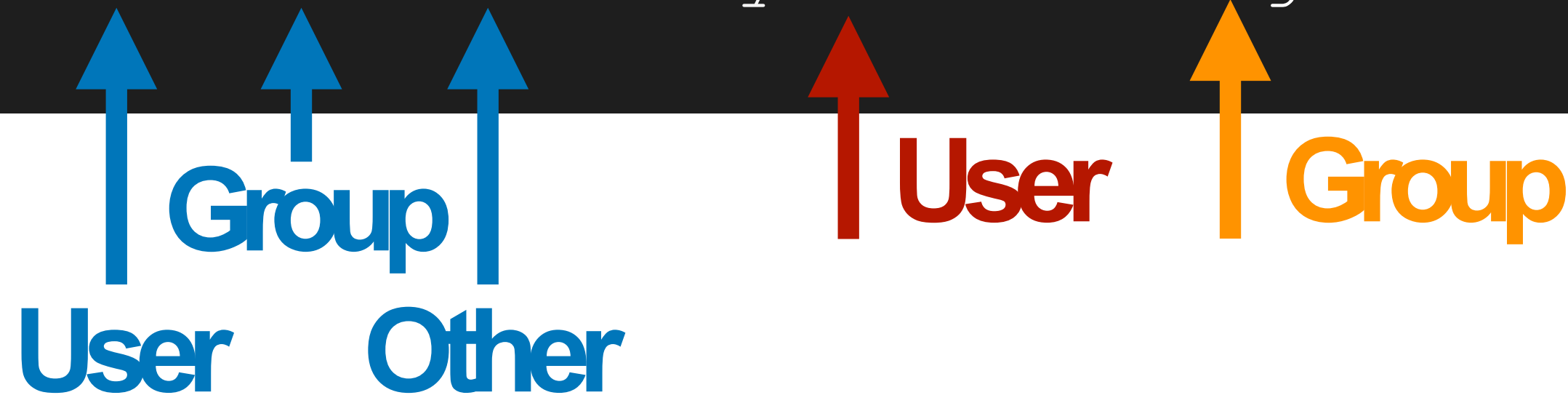
```
yuan@scratch-01:~$ ls -l
total 8
d rwx rwx --- 5 yuan cs253-tas 4096 Apr 2 15:56 homework
d rwx rwx --- 5 yuan cs253-instr 4096 Apr 2 15:56 grades
d rwx rwx r-x 11 yuan cs253-tas 4096 Dec 28 21:09 lectures
- rwx r-x r-- 1 yuan oldwang0 Apr 11 04:15 test.py
```

↑ ↑ ↑ ↑ ↑
User Group Other User Group

Q: What can yiting (member of cs253-tas) do to homework?

File Ownership

```
yuan@scratch-01:~$ ls -l
total 8
d rwx rwx --- 5 yuan cs253-tas 4096 Apr 2 15:56 homework
d rwx rwx --- 5 yuan cs253-instr 4096 Apr 2 15:56 grades
d rwx rwx r-x 11 yuan cs253-tas 4096 Dec 28 21:09 lectures
- rwx r-x r-- 1 yuan oldwang0 Apr 11 04:15 test.py
```



**Q: If a student has access to this server,
which files can they access?**

Access Control Lists (ACLs)

UNIX's permission model is a simple implementation of a generic access control strategy known as **Access Control Lists (ACLs)**

Every object has an ACL that identifies what operations subjects can perform.

Each access to an object is checked against the object's ACL.

hw/	
yuan	read/write
oldwang	read/write
yiting	read

03

PART THREE

UNIX Processes

Processes

Processes are isolated

- Processes cannot access each other's memory

Every process runs as a specific user on the system

- When you run a process, it runs with your UID's permissions
- Process can access any files that the UID has access to
 - Processes run by the same UID have the same permissions

Processes started by root can reduce their privileges by changing their UID to a less privileged UID

Process Example

```
yuan@scratch-01:~$ ls -l
total 8
d rwx rwx --- 5 yuan cs155-tas 4096 Apr 2 15:56 homework
d rwx rwx --- 5 yuan cs155-instr 4096 Apr 2 15:56 grades
```

When you run a command, it runs with all of your privileges because your shell runs as your user account and **forks** to start the command

When any process **forks**, it inherits its parent's UID

Process User IDs

Every process has three different User IDs:

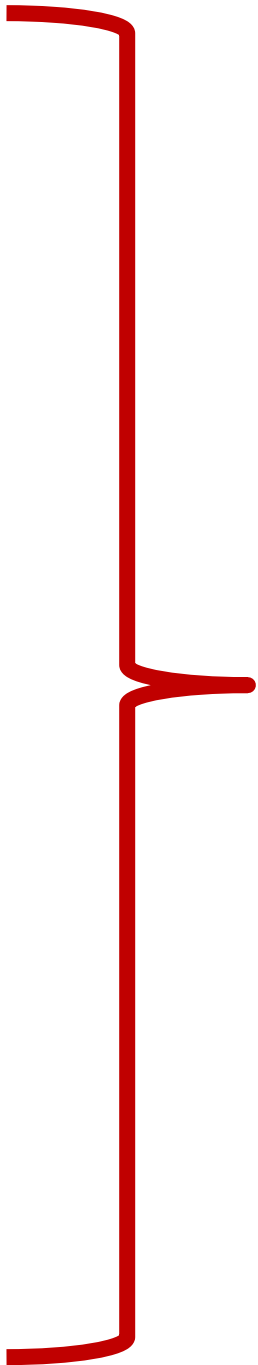
Effective User ID (EUID)

- Determines the permissions for process
- Determines the user that started the process

Real User ID (RUID)

- EUID prior to change

Saved User ID (SUID)



Typically same value
(User who started process)

Changing User IDs

root can change EUID/RUID/SUID to arbitrary values

Unprivileged users can change EUID to **only RUID or SUID**

setuid(x) :

Effective User ID (EUID) => x

Real User ID (RUID) => x

Saved User ID (SUID) => x

Reducing Privilege through setuid

Apache Web Server must start as `root` because only `root` can create a socket that listens on port 80 (a privileged port)

Without any privilege reduction, any Apache bug would result in the attacker having unrestricted server access

Instead, Apache creates children using the following scheme:

```
if (fork() == 0) {  
    int sock = socket(":80");  
    setuid(getuid("www-data"));  
}
```

Temporarily Changing UID

Remember: unprivileged users can change EUID back to the RUID or SUID

`setuid(x) :`

Effective UID => x

Real UID => x

Saved UID => x

```
# EUID = RUID = SUID = 0
```

```
seteuid(100);
```

```
# EUID=100; RUID/SUID=0;
```

```
<perform dangerous operation>
```

`seteuid(x) :`

Effective UID => x

Real UID (no change)

Saved UID (no change)

```
setuid(0)
```

```
# EUID = RUID = SUID = 0
```

SSH Example

Suppose SSH runs as **root** and runs the following code:

```
if (authenticate(uid, pwd) == S_SUCCESS) {  
    if (fork() == 0) {  
        setuid(uid);  
        exec("/bin/bash");  
    }  
}
```

SSH Example — Vulnerable

Suppose SSH runs as **root** and runs the following code:

```
if (authenticate(uid, pwd) == S_SUCCESS) {  
    if (fork() == 0) {  
        setuid(uid);  
        exec("/bin/bash");  
    }  
}
```

EUID := uid, RUID and SUID unchanged

Attack: user can call **setuid(0)**
to become root because `SUID == 0`

SSH Example — Correct Syscall

Suppose SSH runs as **root** and runs the following code:

```
if (authenticate(uid, pwd) == S_SUCCESS) {  
    if (fork() == 0) {  
        setuid(uid);  
        setuid(uid);  
        exec("/bin/bash");  
    }  
}
```

EUID := uid, RUID := uid, SUID := uid

User cannot change UID

SETUID Bit — Elevating Privileges

The `passwd` utility allows you to change your password by updating `/etc/shadow` — a file that only root can read/write

Normally, this would not be possible. Remember: executables run with the privilege of the executing user — and your account can't access

UNIX allows you to set EUID of an executable to be the file owner rather than the executing user.

SETUID on passwd

```
zakir@scratch-03:~$ ls -ali /usr/bin/passwd
2235 -rwsr-xr-x 1 root root 59640 Mar 22 2019 /usr/bin/passwd
zakir@scratch-03:~$
```



setuid

Q: How does `passwd` know which user it should allow the caller change the password for?

setuid vs. setuid 🐱

setuid syscall (in code):

Allows caller to change
User IDs of the process

setuid(x) :

Effective UID => x

Real UID => x

Saved UID => x

setuid bit on Executable

Execution runs as owner
and group of executable
rather than the calling user

```
zakir@scratch-03:~$ ls -ali /usr
2235 -rwsr-xr-x 1 root root 5964
zakir@scratch-03:~$
```

Becoming Root User

System configuration files are owneded by root

Important system processes run as root

Sometimes, you as a user, need to "become" root to fix problems

sudo: run a single command as root (requires you to be blessed)

su: allows you to become root by knowing its password

sudo su: become root without their password

04

PART FOUR

Windows Security Model

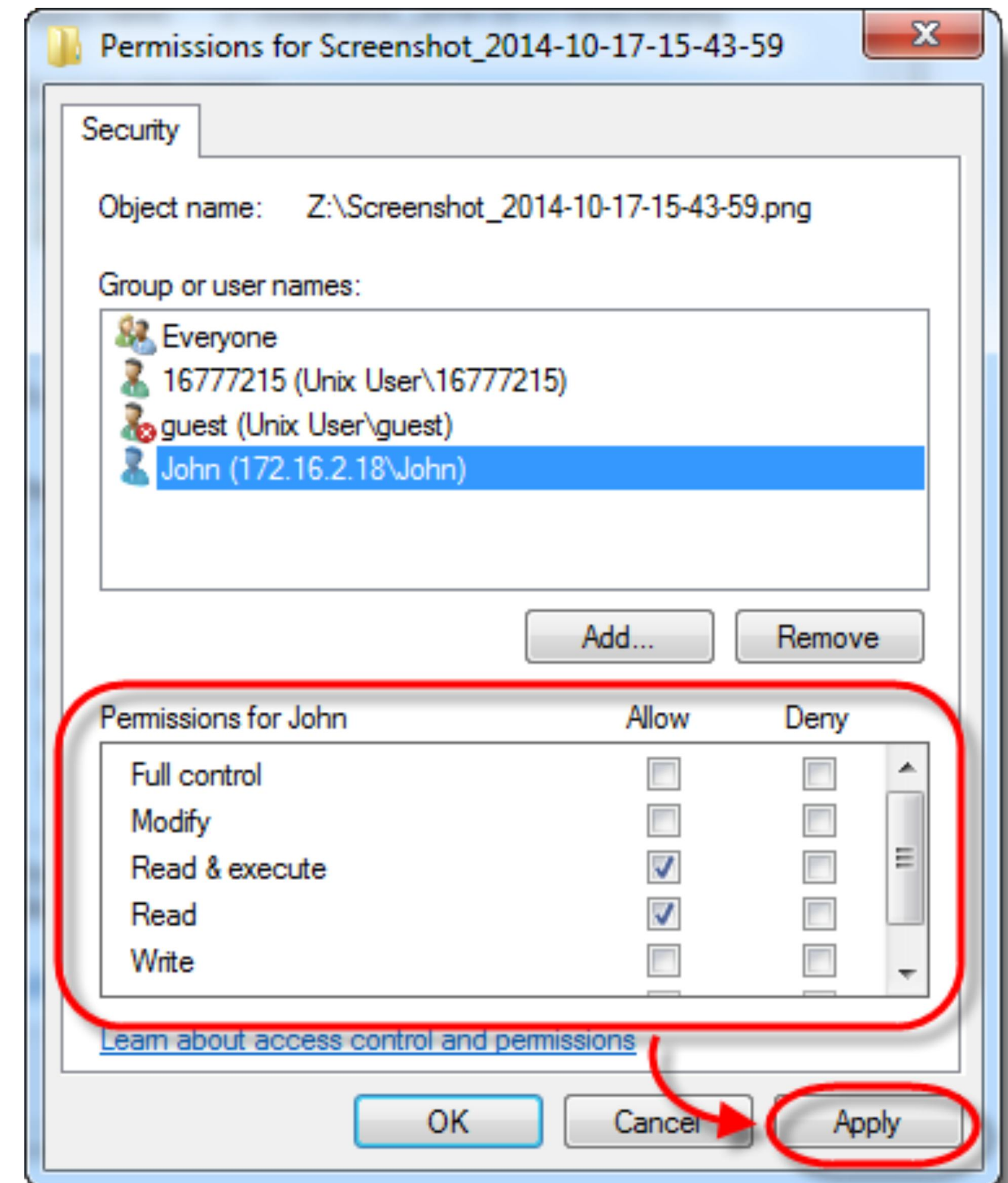
Flexible ACLs

Windows has complex access control options

Objects have full ACLs — possibility for fine grained permissions

Users can be member of multiple groups, groups can be nested

ACLs support Allow and Deny rules



Object Security Descriptors

Every object has a security descriptor

- Specifies who can perform what and audit rules

Contains

- Security identifiers (SIDs) for the owner and primary group of an object.
- Discretionary ACL (DACL): access rights allowed users or groups.
- System ACL (SACL): types of attempts that generate audit records

Tokens

Every process has a set of tokens — its “security context”

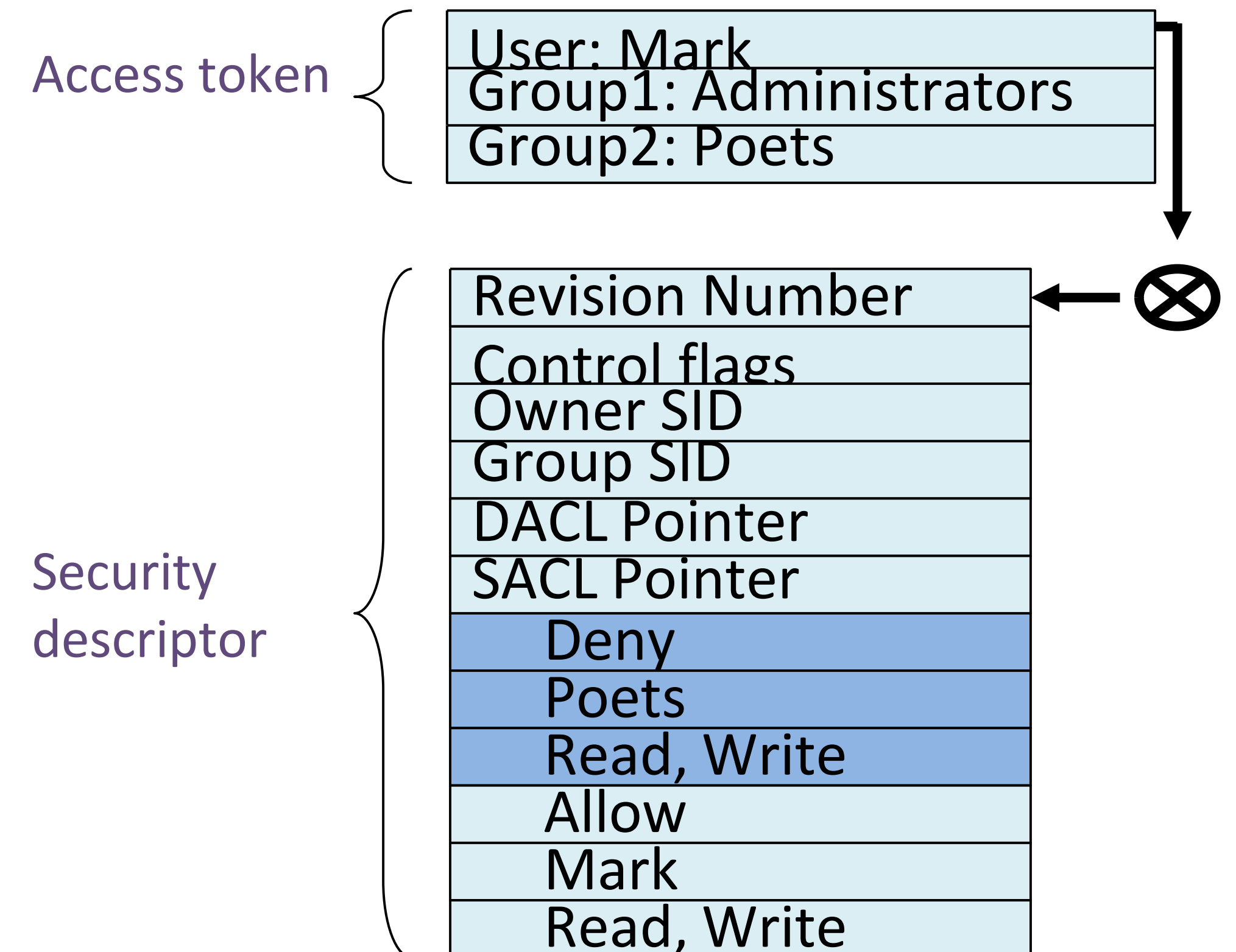
- ID of user account
- ID of groups
- ID of login session
- List of OS privileges held by user/groups
- List of restrictions

Impersonation token can be used temporarily to adopt a different context

Access Request

When a process wants to access an object, it presents its set of security tokens (security context)

Windows checks whether the security context has access to the object based on the object's security descriptor



Capabilities vs. ACLs

Capabilities: subject presents an unforgeable ticket that grants access to an object. System doesn't care who subject is, just that they have access



ACL: system checks where subject is on list of users with access to the object



Weak Protection on Desktops

Relying on user permission provides user with little protection against malicious applications

Malicious application running as you has access to all of your files

Adobe Acrobat can edit, delete, and encrypt/ransom all of your data

Mac OS App Sandbox

Mac OS now sandboxes many applications and mediates access to:

- Hardware (Camera, Microphone, USB, Printer)
- Network Connections (Inbound or Outbound)
- App Data (Calendar, Location, Contacts)
- User Files (Downloads, Pictures, Music, Movies, User Selected Files)

Access to any resource not explicitly requested in the project definition is rejected by the system at run time.

Android Process Isolation

Android uses Linux and its own kernel application sandbox for isolation

Each application runs with its own UID in its own VM

- Apps cannot interact with one another
- Limit access to system resources (decided at installation time)

Reference monitor checks permissions on intercomponent communication

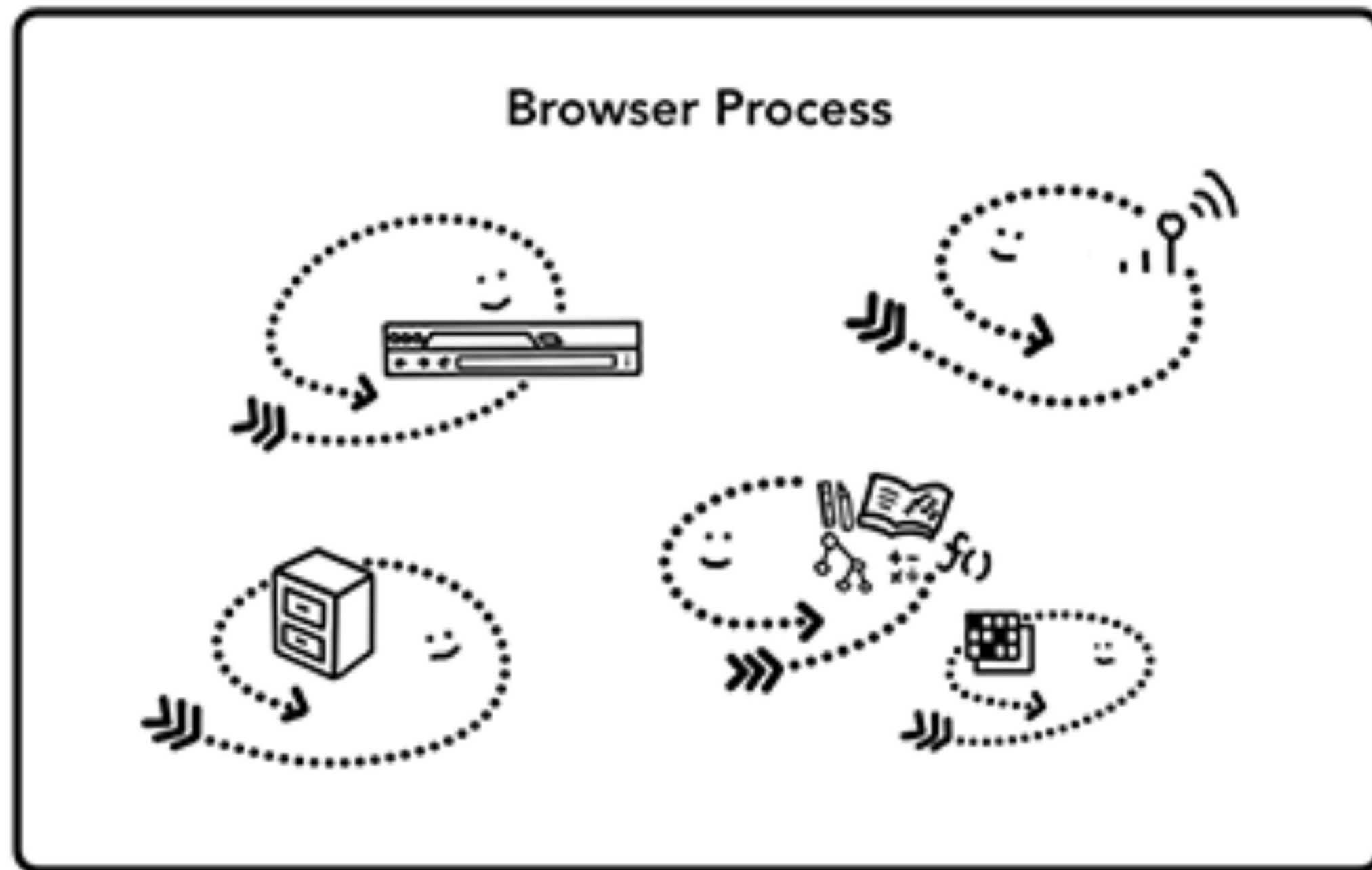


05

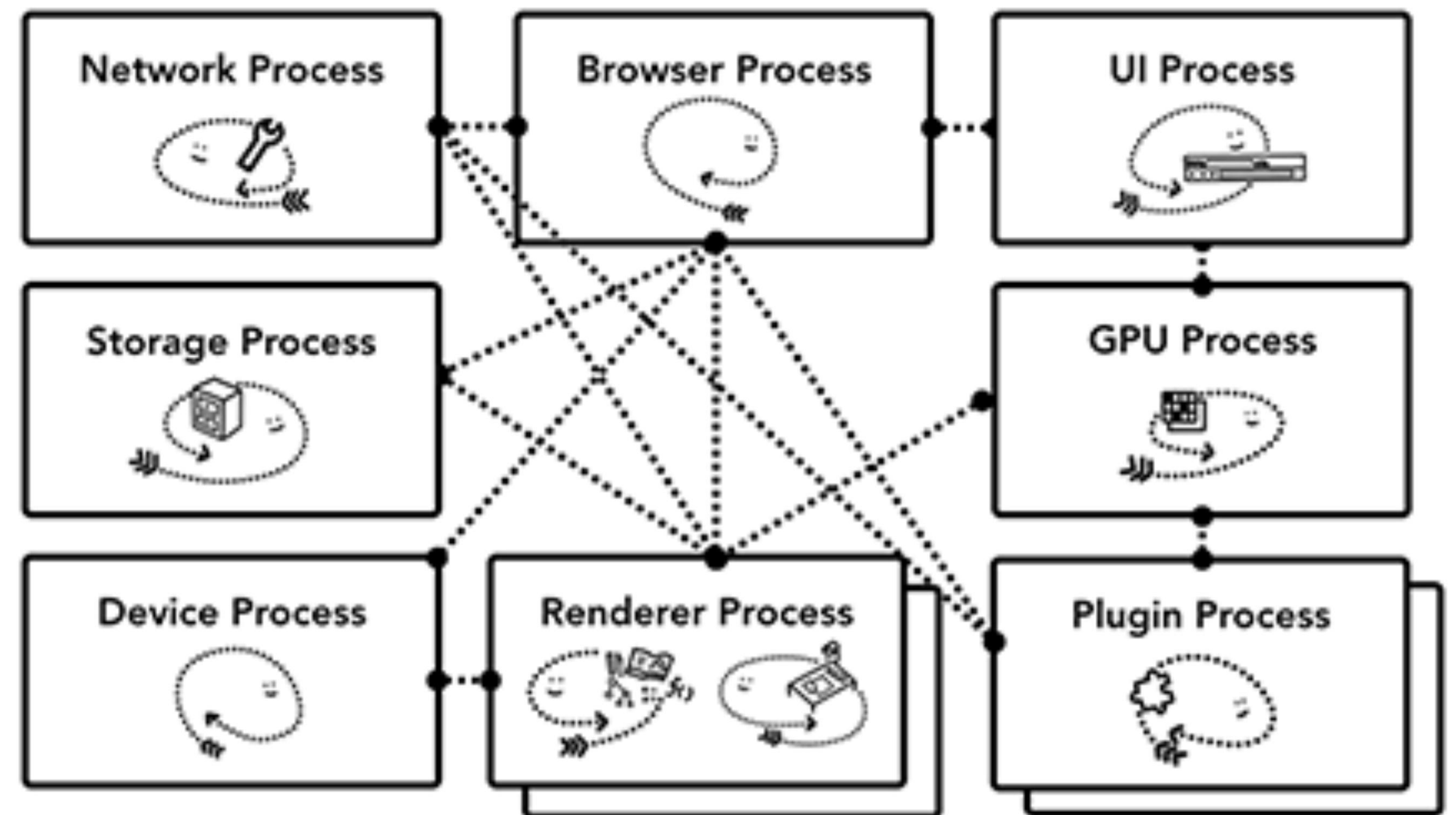
PART FIVE

Chrome Security Architecture

Modern Chrome Architecture



Pre 2006



Modern

Chrome Processes

Browser Process

Controls "chrome" part of the application like address bar and, bookmarks. Also handles the invisible, privileged parts of a web browser like network requests.

Renderer Process

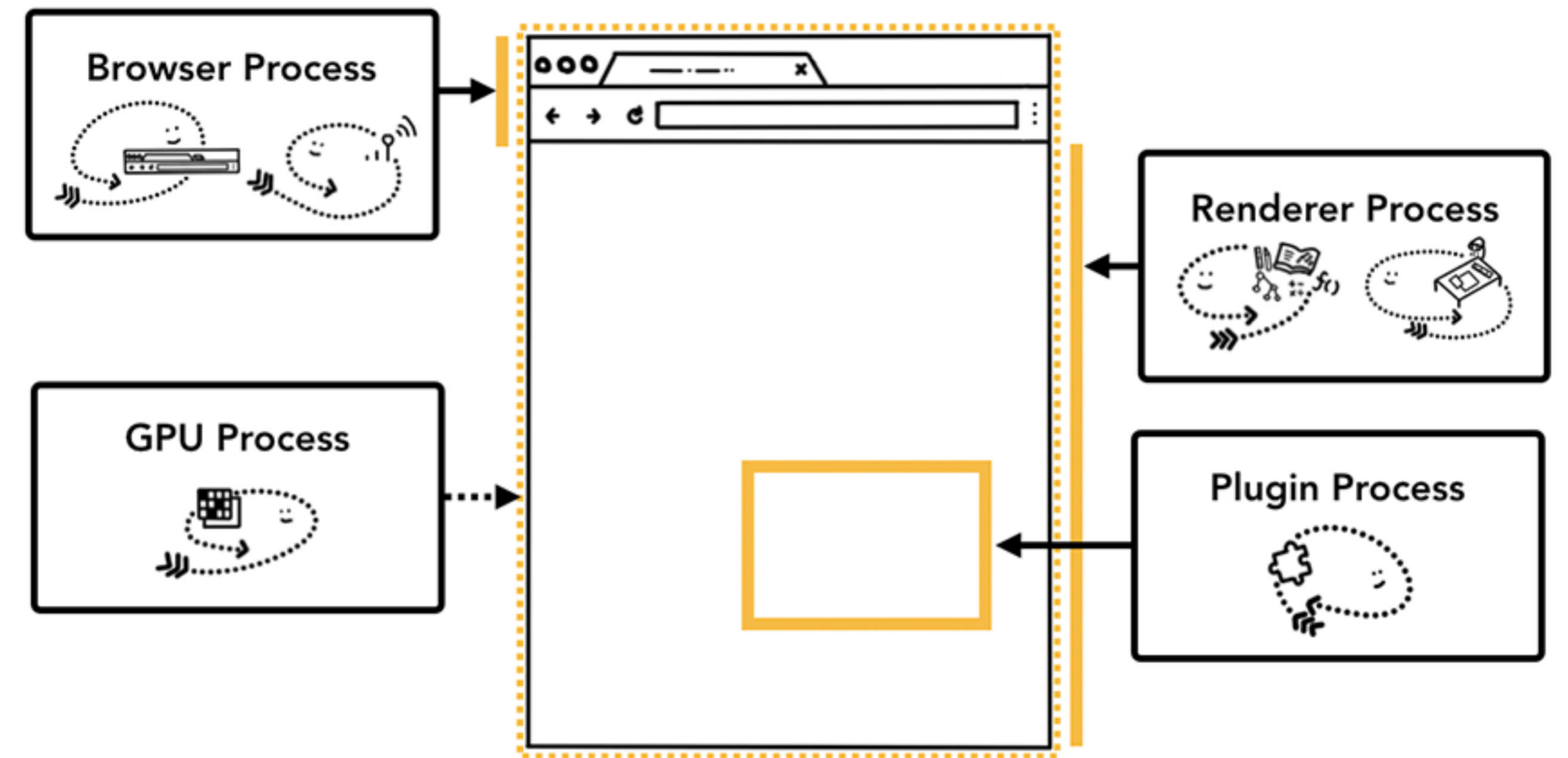
Controls anything inside of the tab where a website is displayed.

Plugin Process

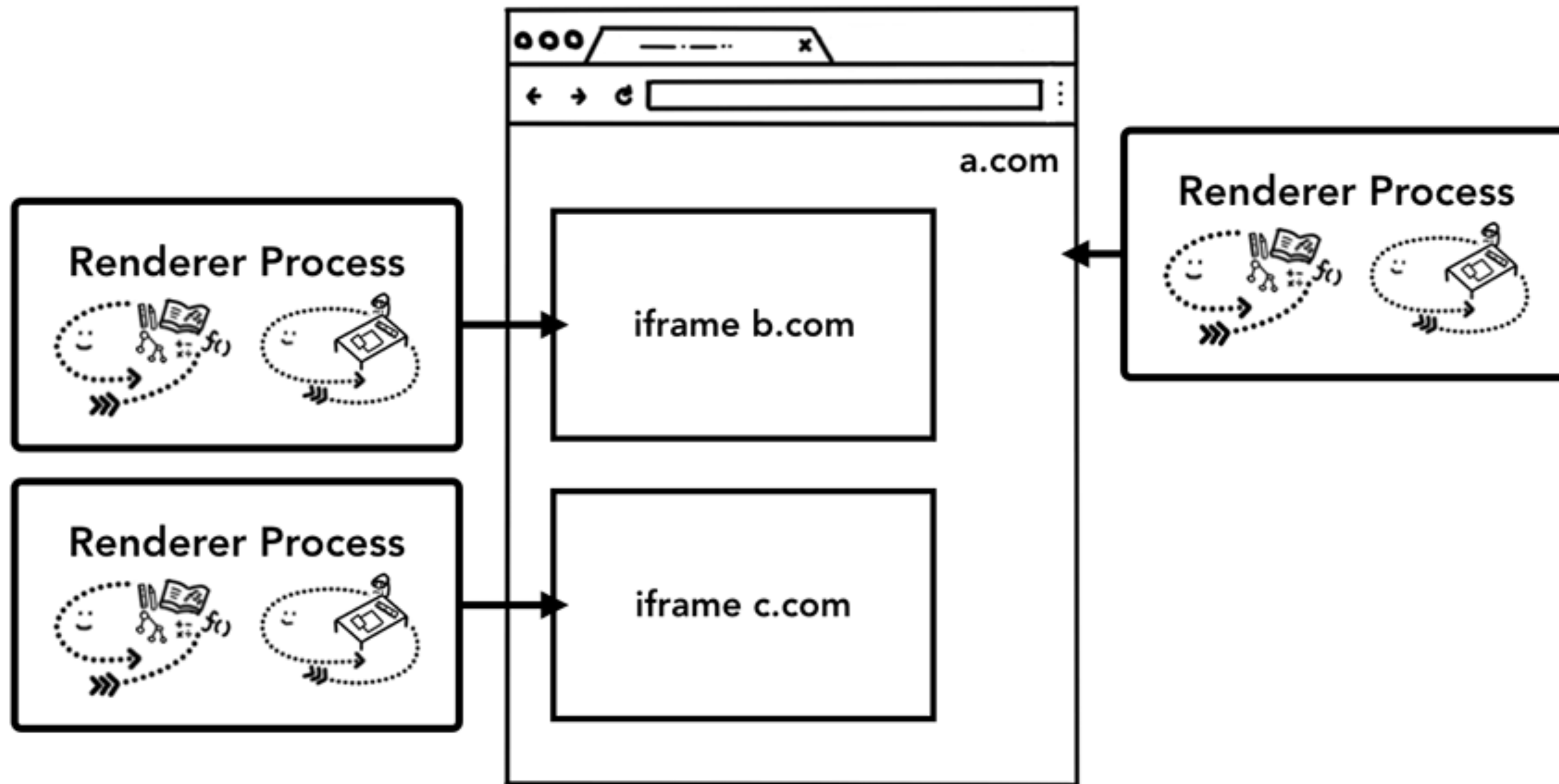
Controls any plugins used by the website, for example, flash.

GPU Process

Handles GPU tasks in isolation from other processes. It is separated into different process because GPUs handles requests from multiple apps and draw them in the same surface



Process-Based Site Isolation

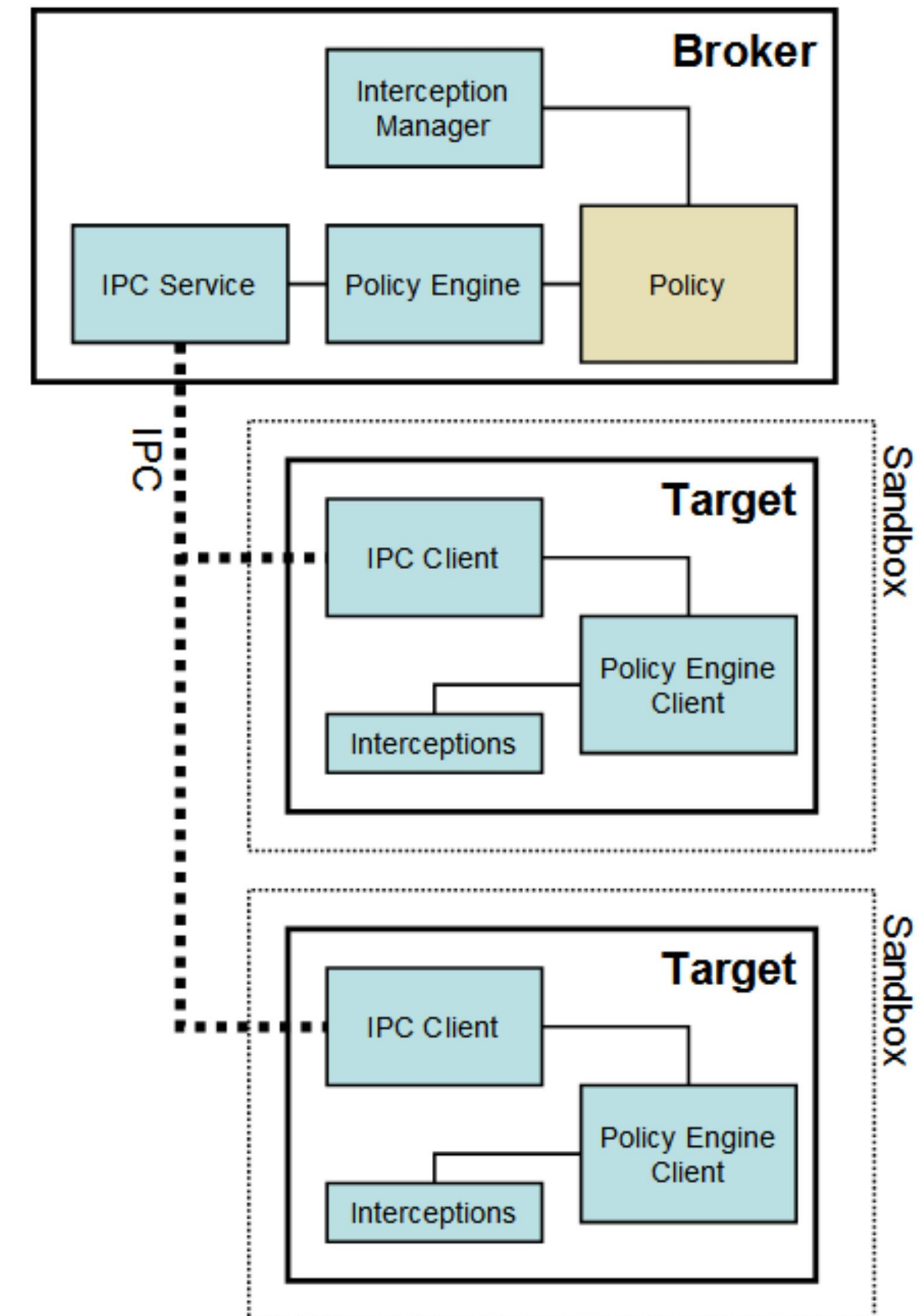


Chrome Architecture

Broker (Main Browser)

Privileged controller/supervisor of the activities of the sandboxed processes

Renderer's only access to the network is via its parent browser process and file system access can be restricted



Principles of Secure Systems

- ✓ Defense in depth
- ✓ Principle of least privilege
- ✓ Privilege separation

Open design

Keep it simple

Open Design

“The security of a mechanism should not depend on the secrecy of its design or implementation.”

If the details of the mechanism leaks (through reverse engineering, dumpster diving or social engineering), then it is a catastrophic failure for all the users at once.

If the secrets are abstracted from the mechanism, e.g., inside a key, then leakage of a key only affects one user.

Kerckhoff's Principle

“a crypto system should be secure even if everything about the system, except the key, is public knowledge.”

- Auguste Kerckhoff

Principles of Secure Systems

- ✓ Defense in depth
- ✓ Principle of least privilege
- ✓ Privilege separation
- ✓ Open design

Keep it simple