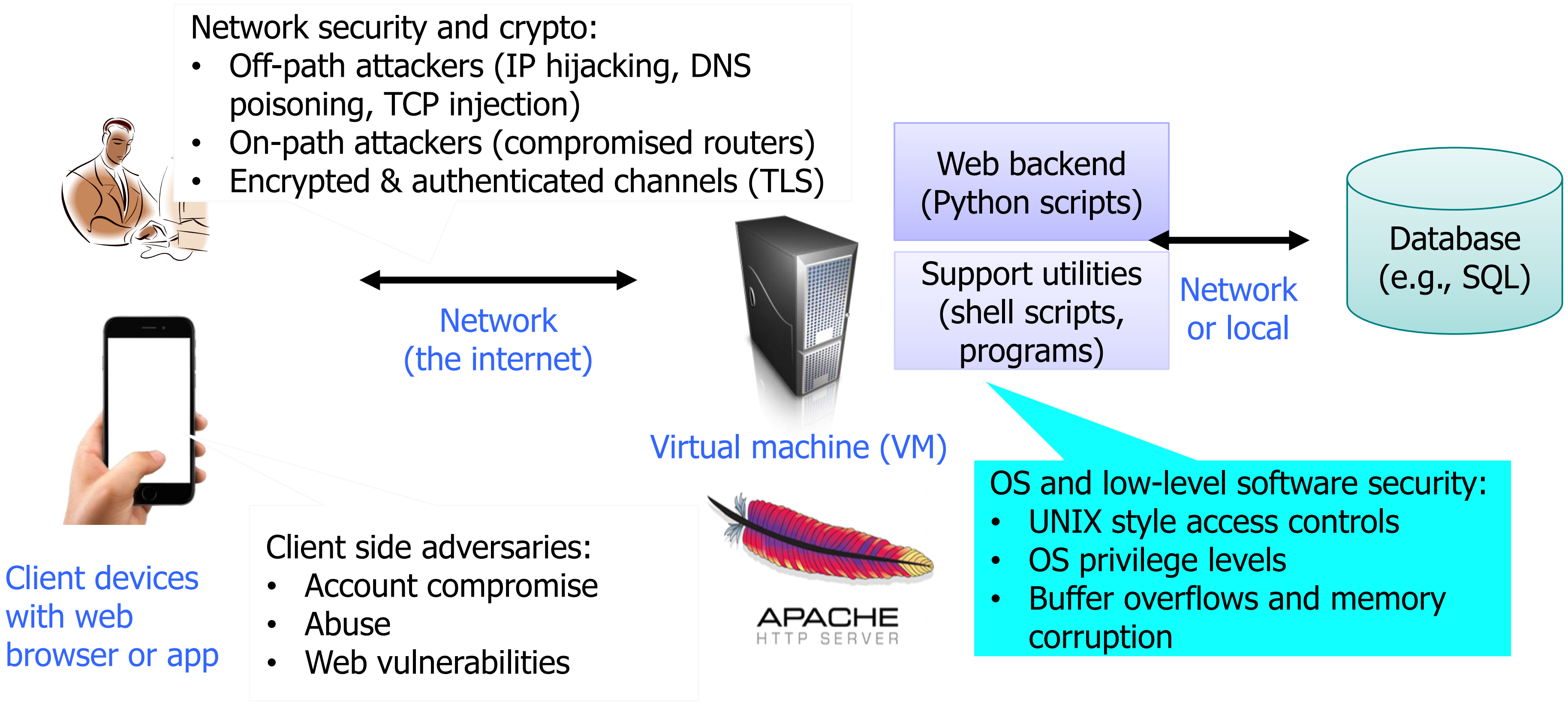
CS 5435

OS Security

Vitaly Shmatikov

Landscape of Threats


Client devices with web browser or app
Network (the internet)
Virtual machine (VM)
APACHE HTTP SERVER
Web backend (Python scripts)
Support utilities (shell scripts, programs)
Network or local
Database (e.g., SQL)

Network security and crypto:
- Off-path attackers (IP hijacking, DNS poisoning, TCP injection)
- On-path attackers (compromised routers)
- Encrypted & authenticated channels (TLS)

Client side adversaries:
- Account compromise
- Abuse
- Web vulnerabilities

OS and low-level software security:
- UNIX style access controls
- OS privilege levels
- Buffer overflows and memory corruption

Defense in Depth

Any piece of code can be buggy or compromised

Systems need multiple layers of protection

- 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)
  - Hardware should prevent permanent malware installation in device firmware
  - Network should prevent malware from infecting nearby computers

Principle of Least Privilege

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

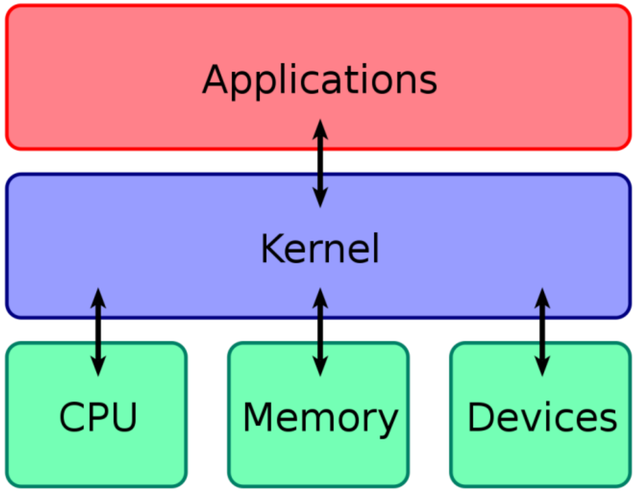
- "Faculty can only change grades for classes they teach"
- "Only employees with background checks have access to classified documents"

Requires privilege separation: dividing system into components to which we can limit access

- Compartmentalization is key!

Operating System Basics

- Multi-tasking, multi-user OS are now the norm
- **Kernel** mediates between applications and resources
- **Applications** consist of one or more processes
- **Processes** have executable program, allocated memory, resource descriptors (e.g., file descriptors), processor state

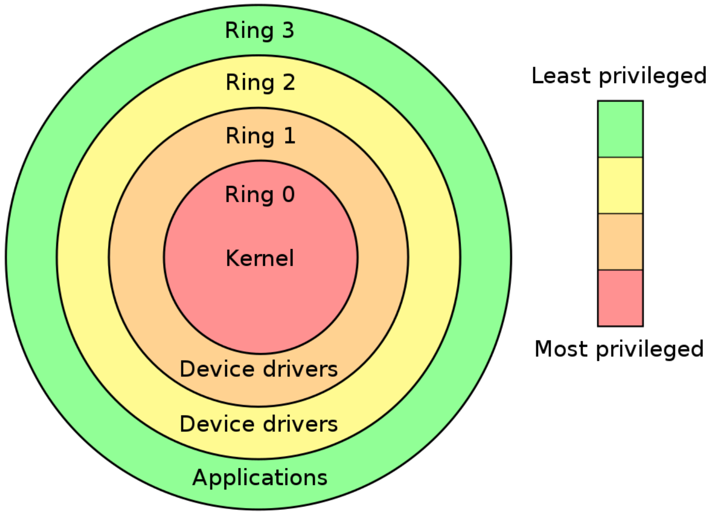


Protection Rings

Different parts of system must operate at different privilege levels

Protection rings included in all typical CPUs today and used by most operating systems

- Lower number = higher privilege
- Ring 0 is supervisor
- Inherit privileges over higher levels



Intel x86 protection rings

Principle of least privilege:
User account or process should have least privilege level required to perform their intended functions

Systems Security Ingredients

Security model

- Abstraction of system to help us express security policies

Security policies

- Specification of what parts of system should be allowed to do

Security mechanism

- How the policy is implemented

Security Policies

Principle of least privilege and privilege separation apply to any subject performing an operation on a protected object

Examples of security policies

- 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 able to edit other apps' data
- Web: A domain should only be able to read its own cookies

ACLs vs. Capabilities

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.

Access Control Matrix

due to Lampson (1971)

Objects

Subjects

	file 1	file 2	…	file n
user 1	read, write	read, write, own		read
user 2				
…				
user m	append	read, execute		read,write, own

User i has permissions for file j as indicated in cell [i,j]

ACLs vs. Capabilities

	file 1	**file 2**	**…**	**file n**
user 1	read, write	read, write, own		read
user 2				
…				
user m	append	read, execute		read,write, own

(1) Access control lists

Column stored with file

(2) Capabilities

Row stored for each user

Unforgeable tickets given to user

Discretionary access control: users can set some controls (e.g., Unix file system)
Mandatory access controls: controls set in one centralized location (e.g., SElinux)

UNIX Security Model

	Unix
Subjects (Who)	Users, Processes
Objects (What)	Memory, Files, Hardware devices …
Operations	Read, write, execute

UNIX 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 (UID)

Many system operations can only run as root

Users and Superusers

A user has username, group name, password

shmat, UID 13630

prof, GID 30

“WouldntchaLikeToKnow”

Root is an administrator / superuser (UID 0)

- Can read and write any file or system resource (network, etc.)
- Can modify the operating system
- Can become any other user
  - Execute commands under any other user’s ID
- Can the superuser read passwords?

Access Control in UNIX

Everything is a file

- Files and also sockets, pipes, hardware devices….
- Files are laid out in a tree
- Each file with associated with an inode data structure

inode records OS management information about the file

- UID and GID of the file owner
- Type, size, location on disk
- Time of last access (atime), last inode modification (ctime), last file contents modification (mtime)
- Discretionary ACL via permission bits

UNIX Permission Bits

-rw-r--r-- 1 shmat prof 116 Sep 5 11:05 midterm.tex

File type

Access rights of file owner

Access rights of group members

Access rights of everybody else

File type

-	regular file
d	directory
b	block file
c	character file
l	symbolic link
p	pipe
s	socket

Permission bits

r	read
w	write
x	execute (if directory, traverse it)
s	setuid, setgid (if directory, files have gid of dir owner)
t	sticky bit (if directory, append-only)

Each file has 12 ACL bits:
rwx for each of owner, group, all; set user ID; set group ID; sticky bit

Roles (Groups)

Group is a set of users

Administrator User Guest

Simplifies assignment of permissions at scale

Concept sometimes referred to as role-based access control (RBAC)

$$\begin{tikzcd}
\text{User 1} \arrow[dr] & \text{Administrator} \arrow[r] \arrow[dr] \arrow[ddr] & \text{/etc/passwd} \\
\text{User 2} \arrow[ur] \arrow[r] & \text{User} \arrow[r] \arrow[dr] & \text{/usr/local/} \\
\text{User 3} \arrow[r] & \text{Guest} \arrow[r] & \text{/tmp/}
\end{tikzcd}$$

UNIX Process Permissions

Process (normally) runs with permissions of user that invoked process

Suppose user wants to change password…

- Need to modify /etc/shadow password file
- /etc/shadow is owned by root
- Can user’s process modify /etc/shadow?
- How does passwd program change user’s password?

Process IDs in UNIX

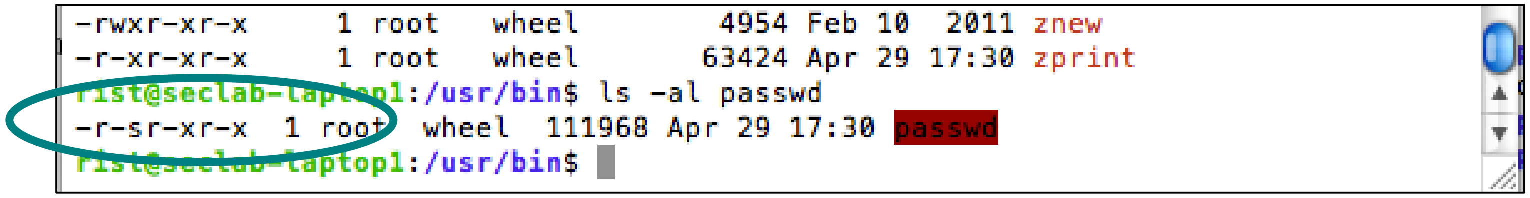
Each process has a real UID (ruid), effective UID (euid), saved UID (suid); similar for GIDs

- Real: ID of the user who started the process
- Effective: ID that determines effective access rights of the process
- Saved: used to swap IDs, gaining or losing privileges

If an executable's setuid bit is set, it will run with effective privileges of its owner, not the user who started it

- E.g., when I run lpr, real UID is shmat (13630), effective UID is root (0), saved UID is shmat (13630)

Setuid Programs



```
-rwxr-xr-x     1 root   wheel        4954 Feb 10  2011 znew
-r-xr-xr-x     1 root   wheel       63424 Apr 29 17:30 zprint
rist@seclab-laptop1:/usr/bin$ ls -al passwd
-r-sr-xr-x  1 root  wheel  111968 Apr 29 17:30 passwd
rist@seclab-laptop1:/usr/bin$
```

- setuid bit – execute with privileges of file's owner
- setgid bit – execute with privileges of file's group

- So passwd is a **setuid program**

runs at permission level of owner, not user who invoked it

Least privilege at process granularity:
passwd runs as root to access /etc/shadow
Can we do better?

Dropping and Acquiring Privilege

To acquire privilege, assign privileged UID to effective ID

To drop privilege temporarily, remove privileged UID from effective ID and store it in saved ID

- Can restore it later from saved ID

To drop privilege permanently, remove privileged UID from both effective and saved ID

Why Privilege Reduction?

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

Without privilege reduction, any Apache bug would give attacker root access to server

Instead, Apache creates children like this:

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

seteuid System Call

```
uid = getuid();
eid = geteuid();
seteuid(uid);       // Drop privileges
…
seteuid(eid);       // Raise privileges
file = fopen( “/etc/shadow”, “w” );
…
seteuid(uid);      // drop privileges
```

Setting UIDs Inside Processes

setuid(newuid)

- If process has "appropriate privileges", set effective, real, and saved ids to newuid
- Otherwise, if newuid is the same as real or saved id, set effective id to newuid (Solaris and Linux) or set effective, real, and saved ids to newuid (BSD)

What does "appropriate privileges" mean?

- Solaris: euid=0 (i.e., process is running as root)
- Linux: process has special SETUID capability
  - Note that setuid(geteuid()) will fail if euid$\neq${0,ruid,suid}
- BSD: euid=0 OR newuid=geteuid()

Example of Bad Use

Suppose SSH runs as **root** and executes this code:

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

EUID := uid, RUID and SUID unchanged

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

Better Use

Suppose SSH runs as **root** and executes this code:

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

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

User cannot change UID

More setuid Magic

seteuid(neweuid)

- Allowed if euid=0 OR if neweuid is ruid or suid OR if neweuid is euid (Solaris and Linux only)
- Sets effective ID, leaves real and saved IDs unchanged

setreuid(newruid, neweuid)

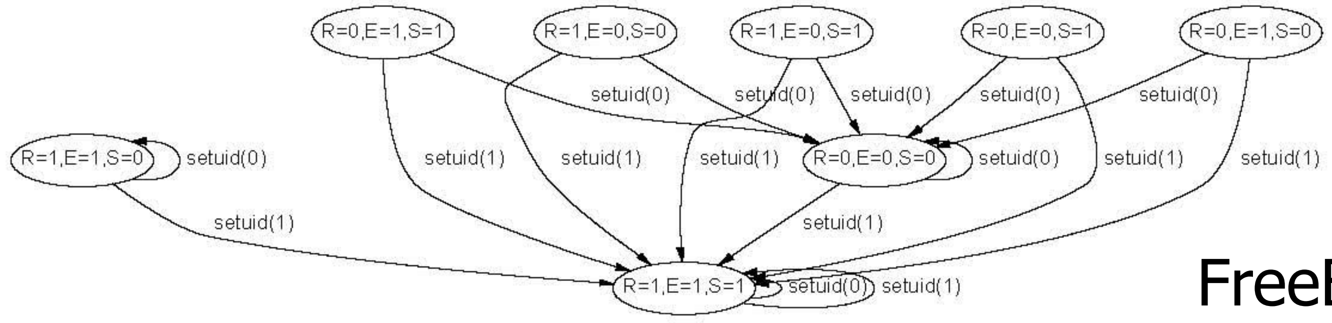
- Sets real and effective IDs
- Can also set saved ID under some circumstances
  - Linux: if real ID is set OR effective ID is not equal to previous real ID, then store new effective ID in saved ID

setresuid(newruid, neweuid, newsuid)

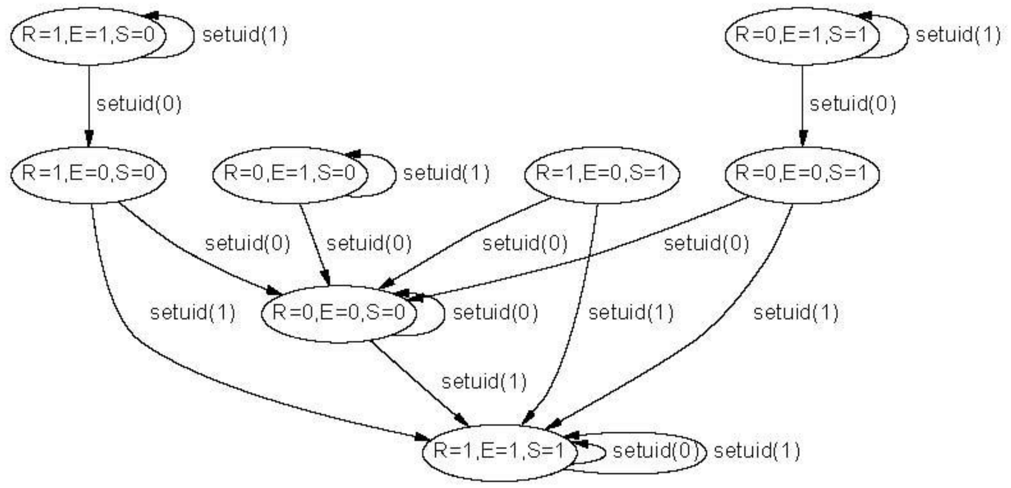
- Sets real, effective, and saved IDs

Finite-State setuid Models

[Chen, Wagner, Dean. "Setuid Demystified"]



FreeBSD



Linux

Privilege Escalation

Privilege escalation: bug that allows lower-privilege user to perform actions as higher-privilege user (typically, root)

- Control-flow hijacking vulnerabilities in local setuid program can be used for privilege escalation
- 99% of local vulnerabilities in UNIX systems exploit setuid-root programs to obtain root privileges
  - The other 1% target the OS itself

Also, race conditions

Checking Access Rights

Access control: user should only be able to access a file if he has the permission to do so

But what if user is running as setuid-root?

- E.g., a printing program is usually setuid-root in order to access the printer device
  - Runs "as if" the user had root privileges
- But a root user can access any file!
- How does the printing program know that the user has the right to read (and print) a given file?

UNIX has a special access() system call

Typical Setuid-Root File Access

```
// Assume this is running inside some setuid-root program
void foo(char *filename) {
   int fd;
   if (access(filename, R_OK) != 0)
      exit(1);
   fd=open(filename, O_RDONLY);
   … do something with fd …
}
```

Check if user has the permission to read this file

What if the file to which filename points changed **right here**?

Open file for reading

This is known as a TOCTTOU attack
("Time of Check To Time of Use")

TOCTTOU Attack

```
if( access("/tmp/myfile", R_OK) != 0 ) {
        exit(-1);
}
file = open( "/tmp/myfile", "r" );
read( file, buf, 100 );
close( file );
print( "%s\n", buf );
```

access checks RUID,
but open only checks EUID

Time

```
access("/tmp/myfile", R_OK)
```

```
ln –s /etc/shadow /tmp/myfile
```

```
open( "/tmp/myfile", "r" );
```

```
print( "%s\n", buf );
```

Prints out shadow file
(including password hashes)

Better Code

```
euid = geteuid();
ruid = getuid();
seteuid(ruid);                    // drop privileges
file = open( “/tmp/myfile”, “r” );
read( file, buf, 100 );
close( file );
print( “%s\n”, buf );
```

Summary of UNIX Security

- Simple model provides protection for most situations
- Flexible enough to make most simple systems possible in practice

- Coarse-grained ACLs don’t account for enterprise complexity
- ACLs don’t handle different applications within a single user account
- Nearly all system operations require root access — bugs give attacker full access

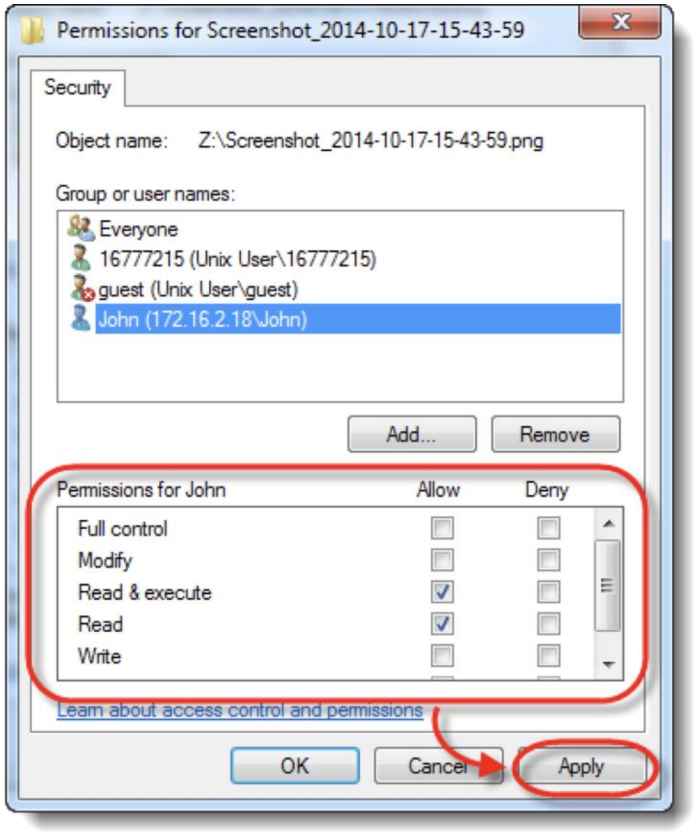
Flexible ACLs in Windows

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



Permissions for Screenshot_2014-10-17-15-43-59

Security

Object name: Z:\Screenshot_2014-10-17-15-43-59.png

Group or user names:

- Everyone
- 16777215 (Unix User\16777215)
- guest (Unix User\guest)
- John (172.16.2.18\John)

Add... Remove

Permissions for John	Allow	Deny
Full control		
Modify		
Read & execute	✓	
Read	✓	
Write		

Learn about access control and permissions

OK Cancel Apply

Object Security Descriptors

Specifies who can perform what and audit rules

Security descriptor 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"

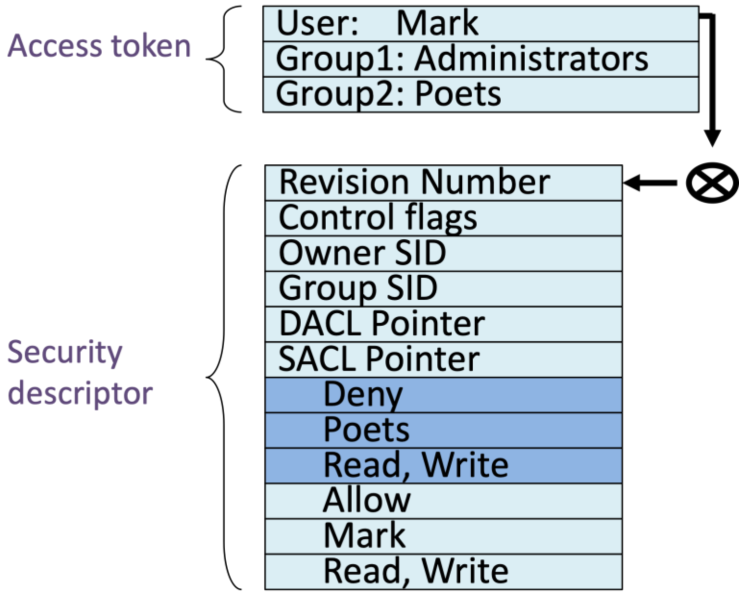
- ID of user account
- ID of groups
- ID of login session
- List of OS privileges held by user/groups
- List of restrictions

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



Windows “Job” Object

Renderer runs as a “Job” object rather than an interactive process.

Eliminates access to:
- desktop and display settings
- clipboard
- creating subprocesses
- access to global atoms table

Alternate Windows Desktop

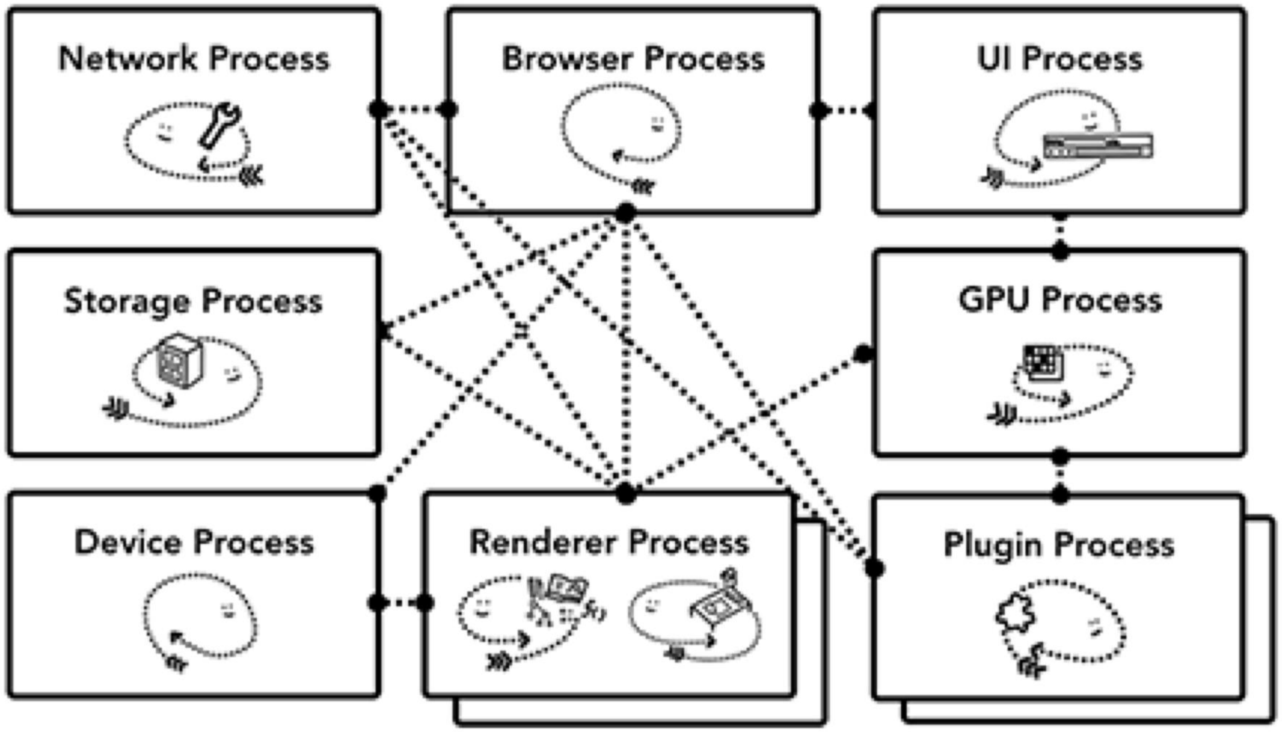
Windows on the same desktop are effectively in the same security context because the sending and receiving of window messages is not subject to any security checks.

Sending messages across desktops is not allowed.

Chrome creates an additional desktop for target processes

Isolates the sandboxed processes from snooping in the user's interactions

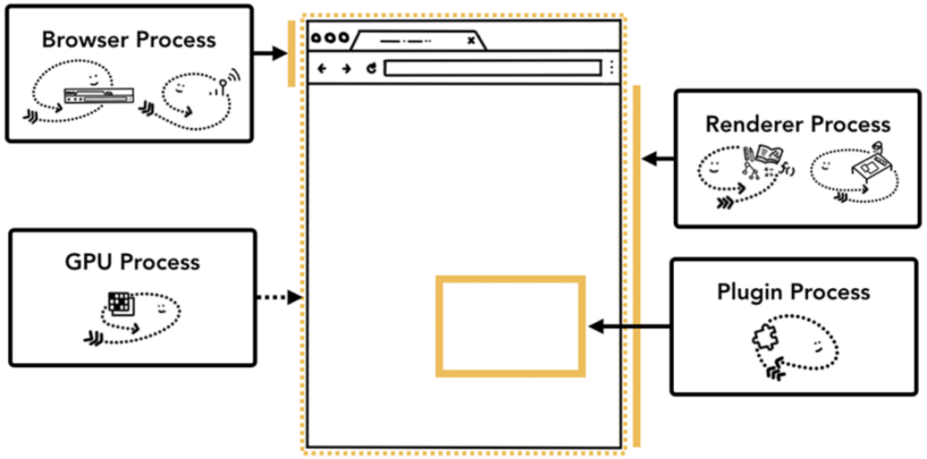
Chrome Architecture



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.

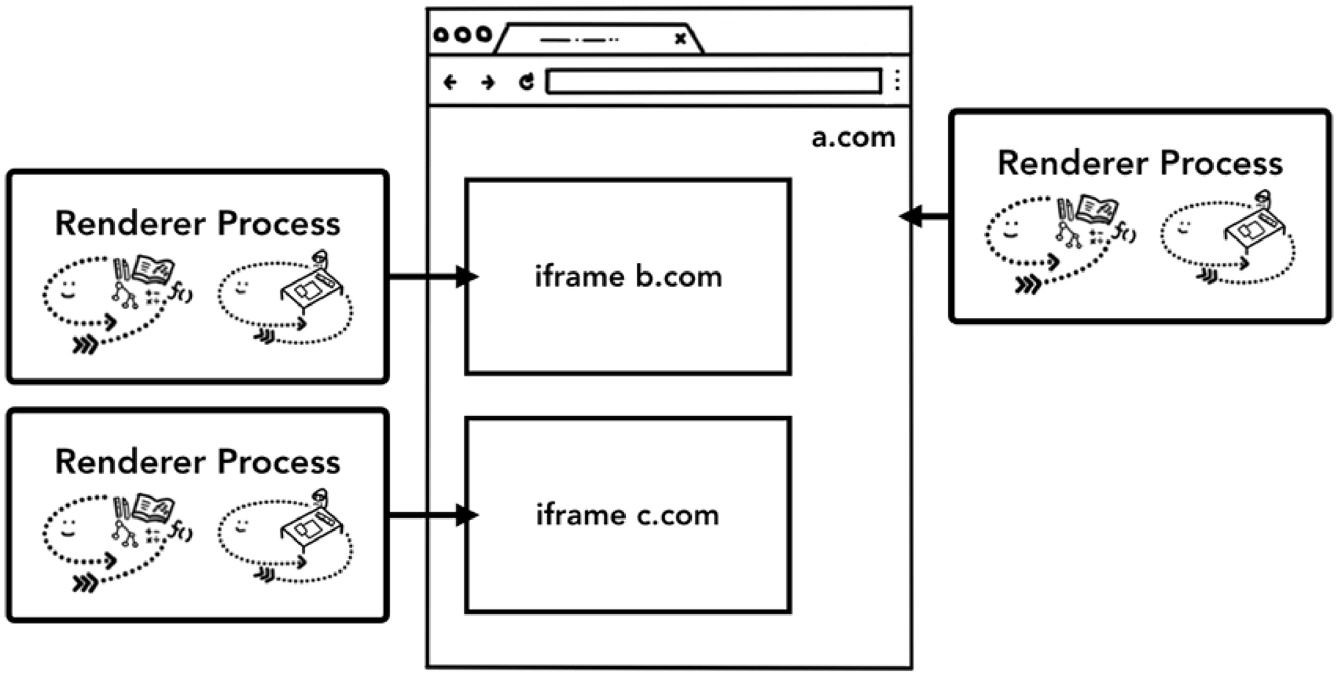
Renderer Process
Controls anything inside of the tab where a website is displayed.



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

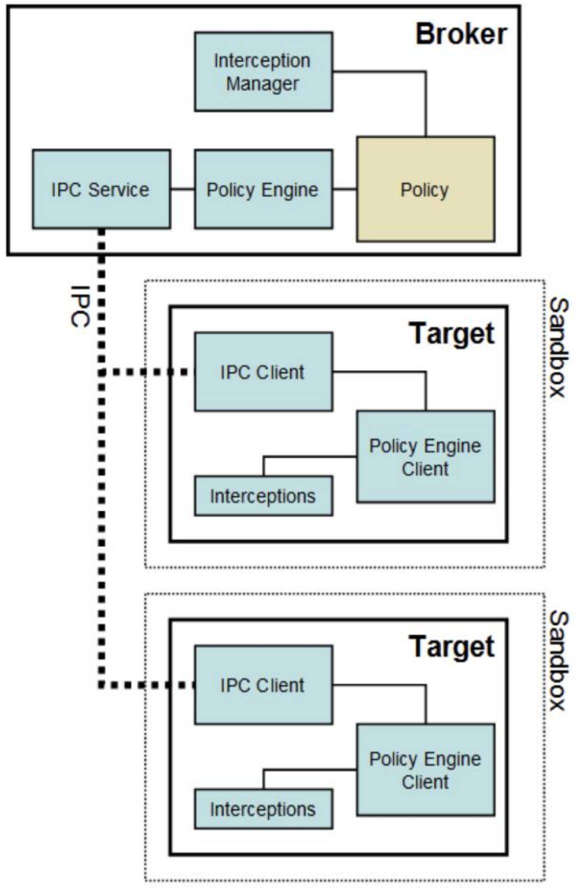
Process-Based Site Isolation



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



Restricted Security Context

Chrome calls **CreateRestrictedToken** to create a token that has a subset of the user's privileges

Assigns the token the user and group **S-1-0-0 Nobody**, removes access to nearly every system resource

As long as the disk root directories have non-null security, no files (even with null ACLs) can be accessed

No network access