CIS 4360: Computer Security Fundamentals

Software Security

Viet Tung Hoang

The slides are based on those of Prof. Stefano Tessaro, University of Washington
and the book "Computer Security: A Hands-on Approach" (Wenliang Du)

Agenda

1. **Multi-user Systems**
2. Access control in UNIX
3. Attacks on SetUID programs

Authentication $\rightarrow$ Multi-user systems

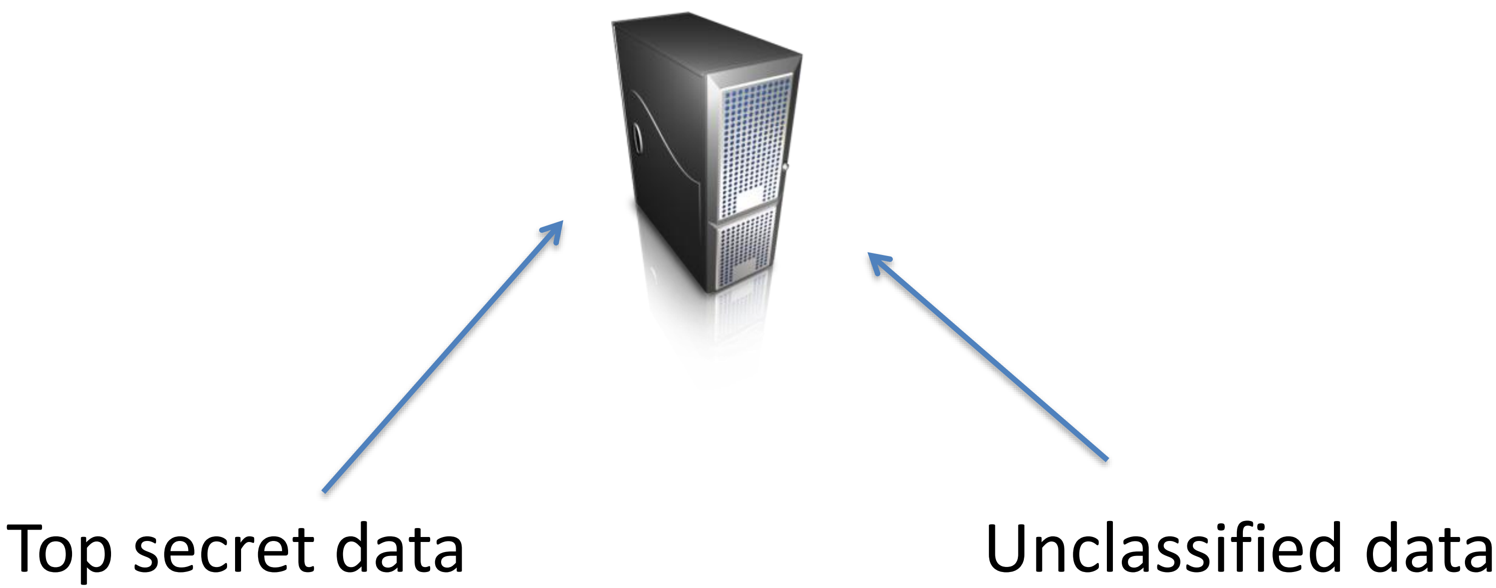
- Users authenticate to access a system
- Many users access the same system
- Users may share resources
- Access control mechanisms decide which user can access which resource

Examples:

- Gmail, Facebook, an operating system, …

Multi-level security

- Main motivation behind multi-user systems: Military and other government entities want to use time-sharing too



Security Policies

A **security policy** is a statement that partitions the states of the system into a set of authorized (or secure) states and a set of unauthorized (or non-secure) states.

A **secure system** is a system that starts in an authorized state and cannot enter an unauthorized state.

Security Policies – What do they involve?

- **Subjects**
  - People, users, employees, …
- **Objects**
  - Files, documents, physical locations, …
- **Actions**
  - Read, write, open, edit, append, …

Access control matrix

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

<u>Discretionary</u> Access Control (DAC)

- Users decide access to their own files

Mandatory Access Control (MAC)

- Security decisions are made by a central policy administrator

Examples:

- Bell-LaPadula
  - Users are assigned security clearances, general policy captures who can read a file.
- Biba
  - Dual to Bell-LaPadula, deals with integrity.

Meaning: Violating these policies would allow breaks of confidentiality / integrity

Example: Bell-LaPadula Model

Implements:

- Security clearances
- Need-to-know

Classification levels

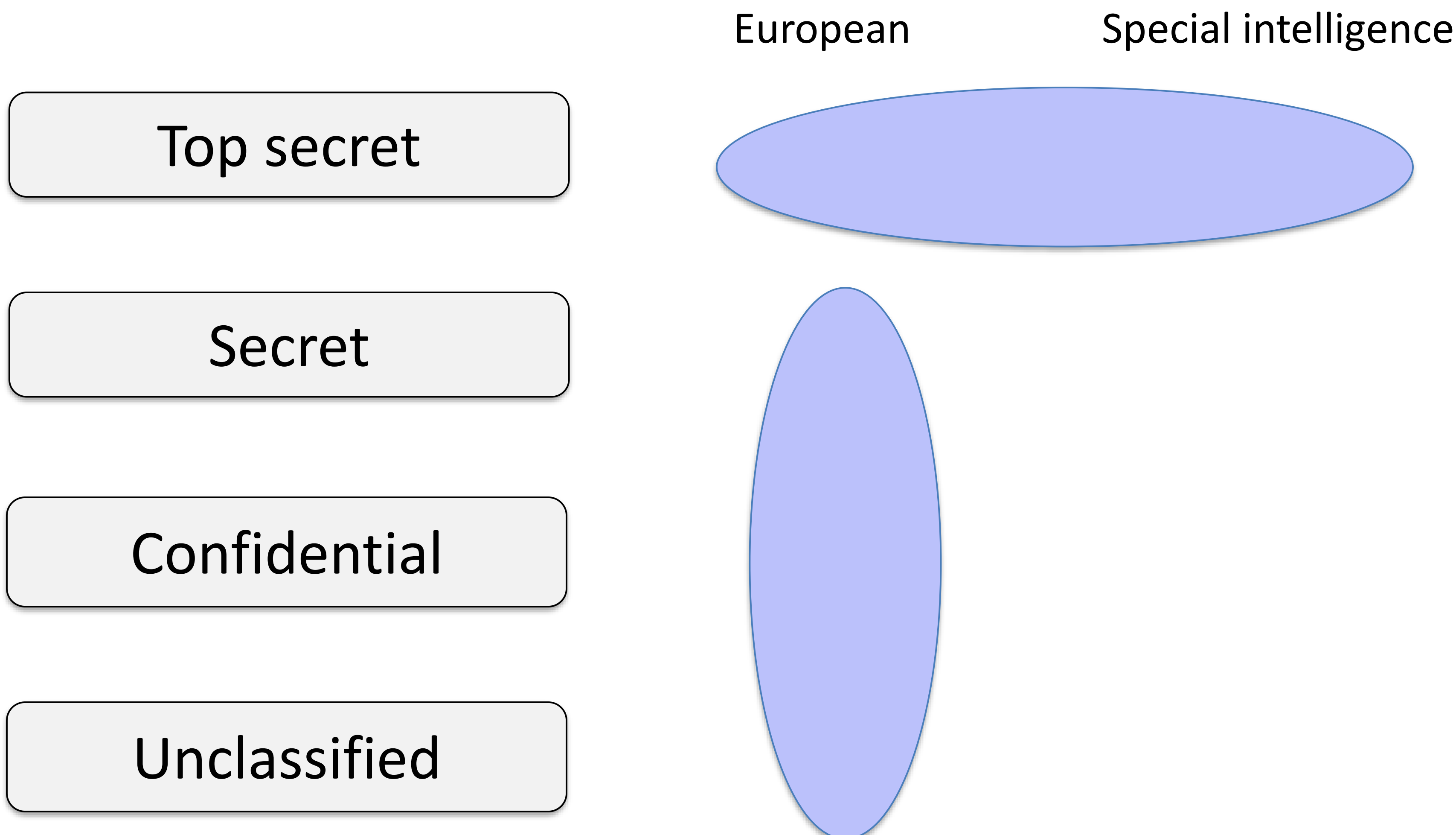
Top secret

Secret

Confidential

Unclassified

Compartmentalization



Classification levels and compartmentalization

- Security level (L,C) assigned to files and users
  - L is **classification** level (Top secret, secret, …)
  - C is **compartment** (Europe, Special intelligence…)

Dominance relationship:

(L1,C1) ≤ (L2,C2)

L1 < L2 (L1 "less secret" than L2)

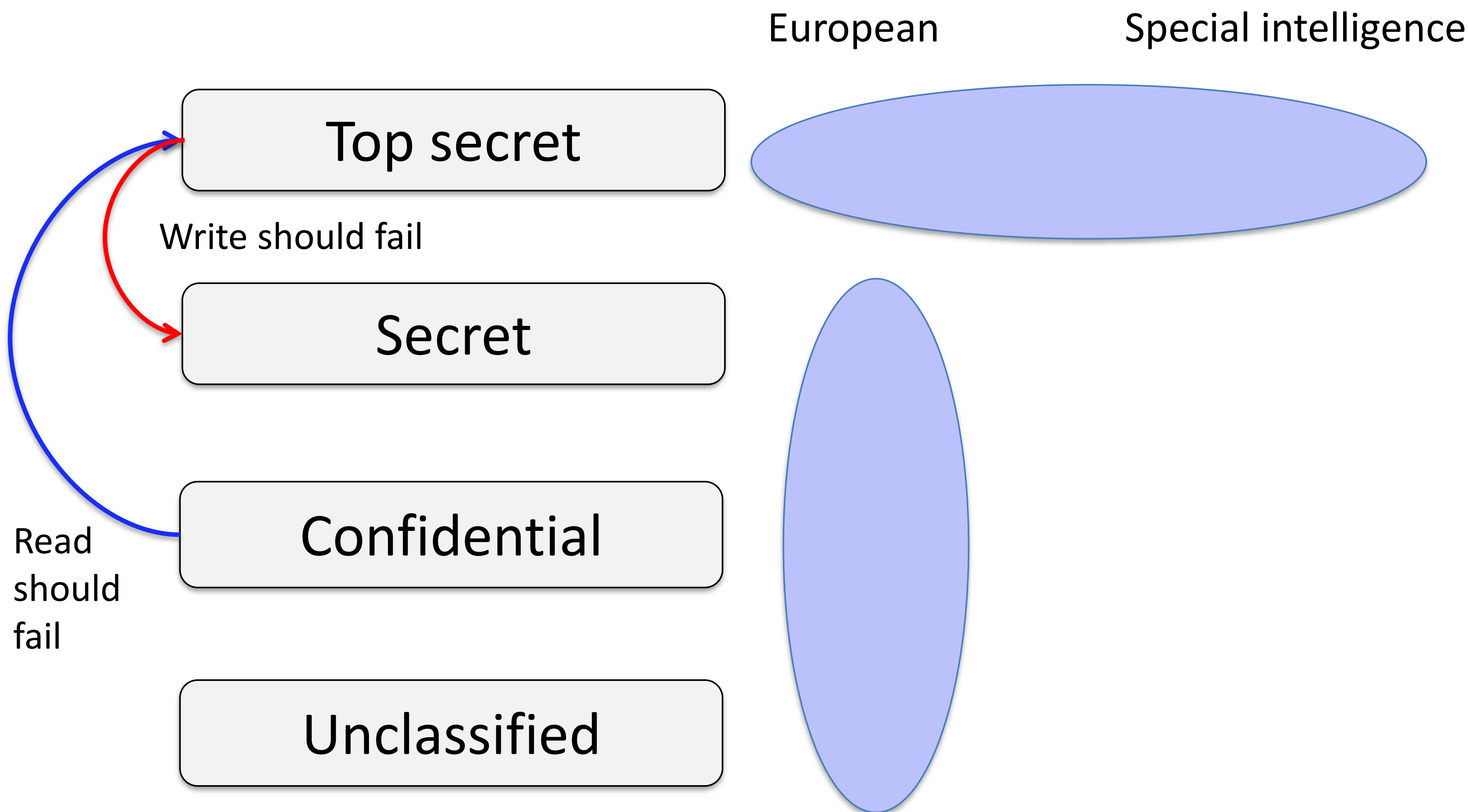
C1 subset of C2

Example:

(Secret, {European}) ≤ (Top Secret, {European,Special Intel})

Bell-LaPadula Confidentiality Model

“no reads up”, “no writes down”



Bell-LaPadula Confidentiality Model

"no reads up", "no writes down"

Simple security condition

User with (L1,C1) can read file with (L2,C2) if?

$(L1,C1) \leq (L2,C2)$ or $(L1,C1) \geq (L2,C2)$

*-property

User with (L1,C1) can write file with (L2,C2) if?

$(L1,C1) \leq (L2,C2)$ or $(L1,C1) \geq (L2,C2)$

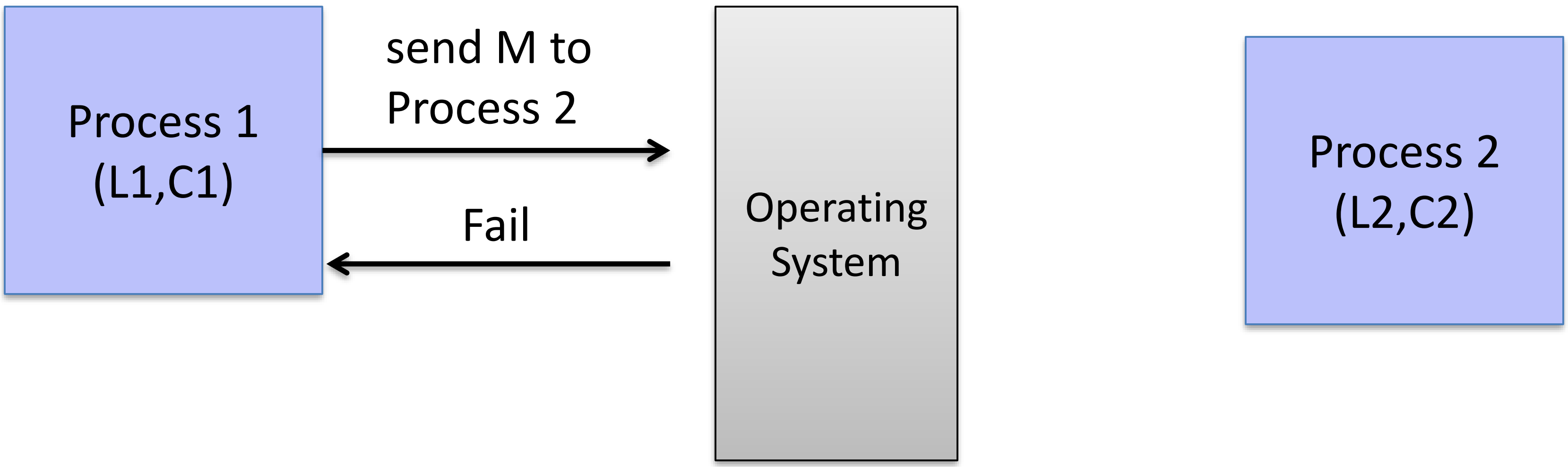
Some issues

- It may well be that someone at (“top-secret”, “Europe, Specint”) needs to write an unclassified document.
- Implementation should allow explicit lowering of security level.
- Only deals with confidentiality – what about integrity?

Circumventing access controls

Covert channels

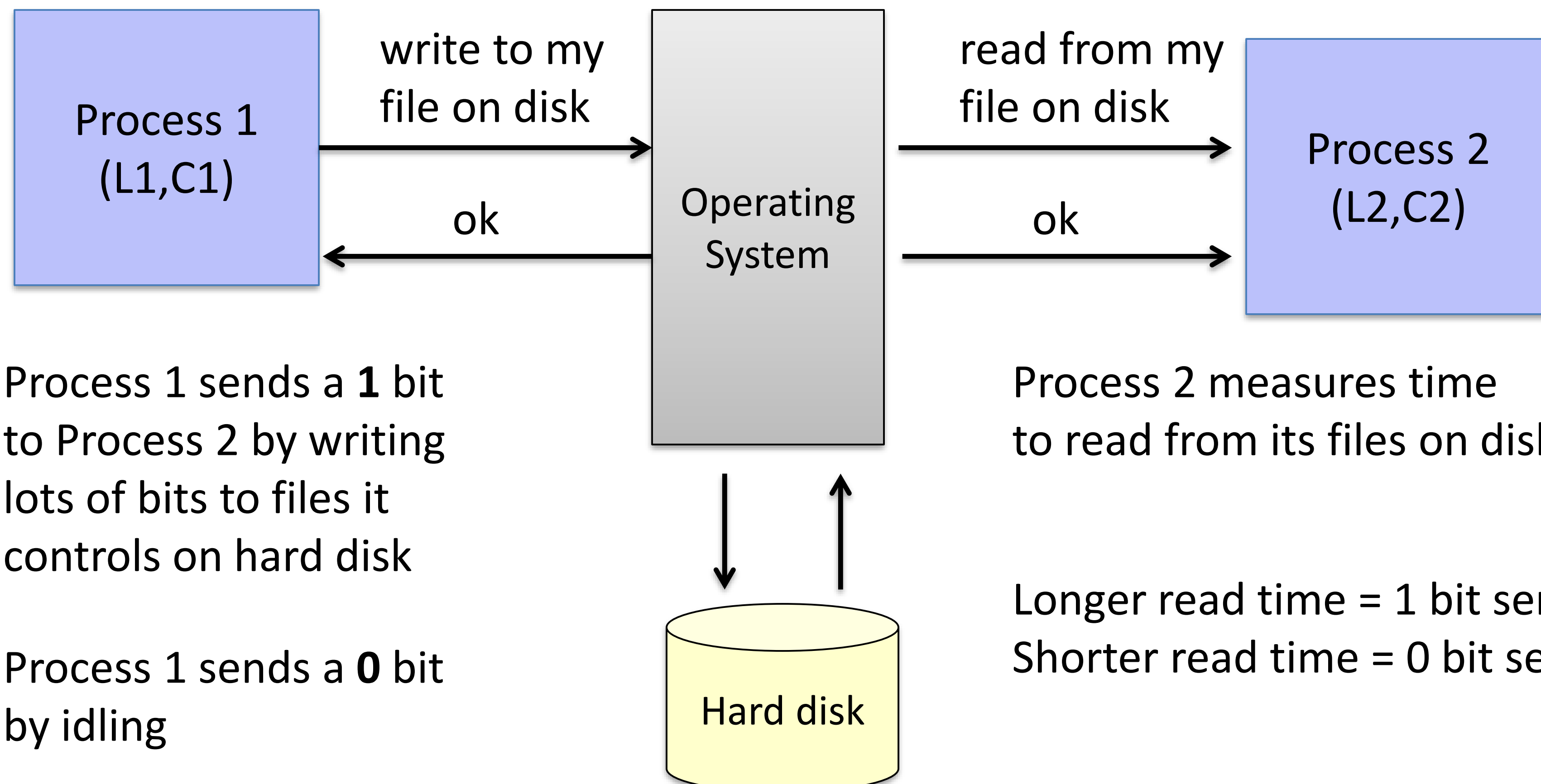
(L1,C1) ≥ (L2,C2)



Circumventing access controls

Covert channels

(L1,C1) ≥ (L2,C2)



Process 1 sends a **1** bit to Process 2 by writing lots of bits to files it controls on hard disk

Process 1 sends a **0** bit by idling

Process 2 measures time to read from its files on disk

Longer read time = 1 bit sent
Shorter read time = 0 bit sent

DAC – Two common implementation paradigms

	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

Tokens given to user

ACLs compared to Capabilities

ACLs	Capabilities
ACLs requires authenticating user	Token-based approach avoids need for auth
Processes must be given permissions	Tokens can be passed around
Operating System must protect permission setting	Operating System must manage tokens

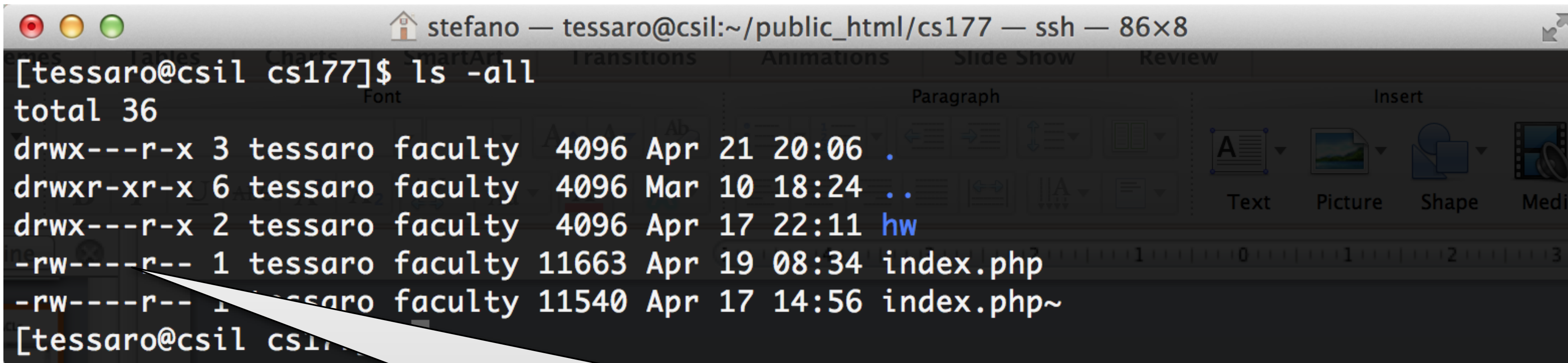
Agenda

1. Multi-user Systems

2. **Access control in UNIX**

3. Attacks on SetUID programs

UNIX-style file system ACLs



Permissions:

- Directory?
- Owner (r,w,x) , group (r,w,x), all (r, w, x)

Owner (tessaro)

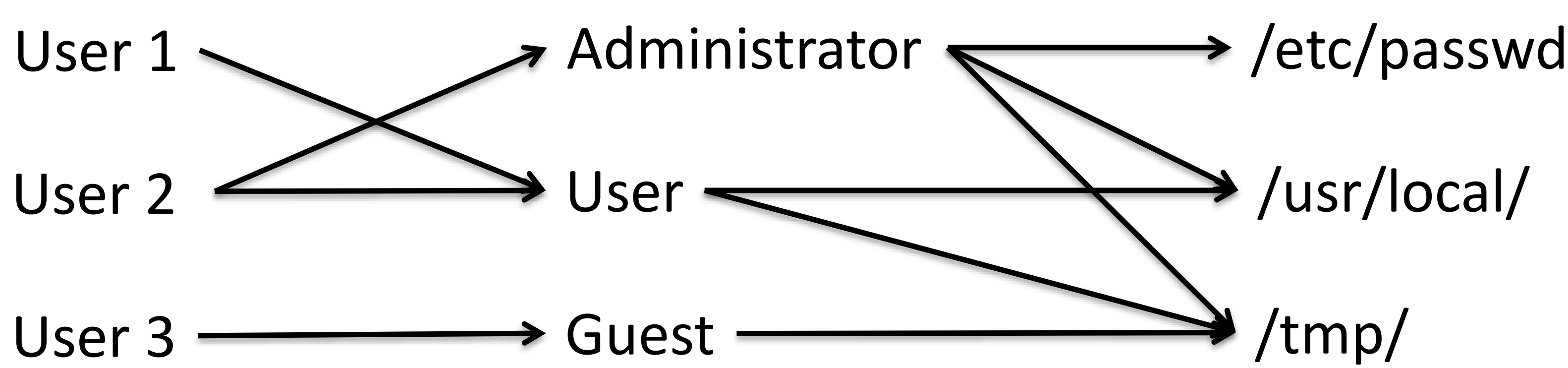
Group (faculty)

Roles (groups)

Group is a set of users

Administrator User Guest

Simplifies assignment of permissions at scale



UNIX file permissions

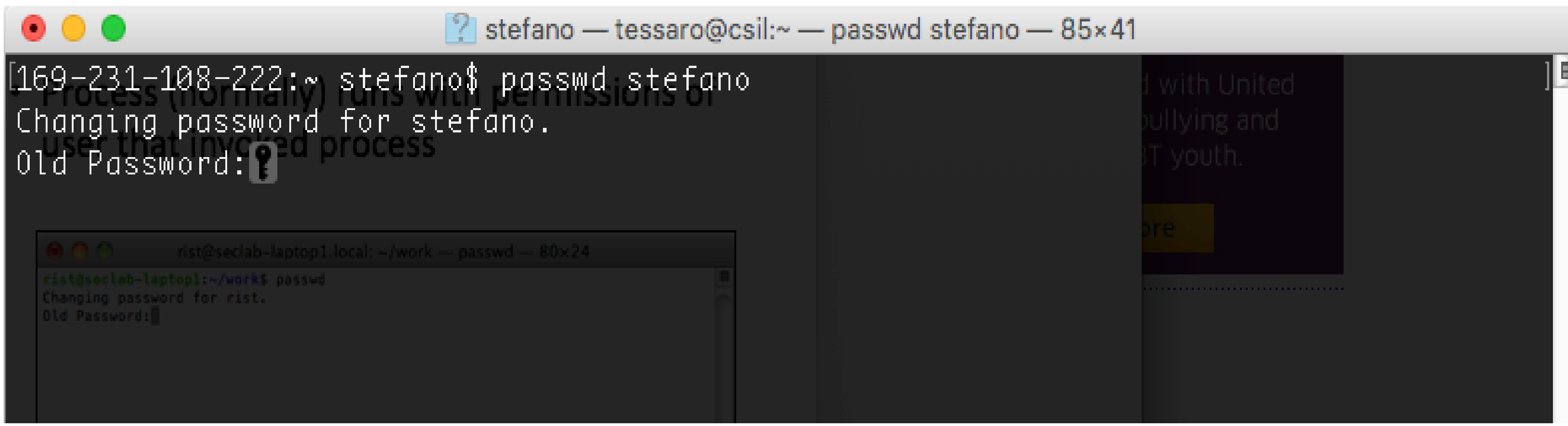
- Owner, group
- Permissions set by owner / root
- Resolving permissions:
  - If user=owner, then owner privileges
  - If user in group, then group privileges
  - Otherwise, all privileges

Processes

- So far, we have talked about permissions of files.
- **Process:** Instance of computer program being executed, generally associated with an executable file.
- Processes also have permissions
  – Which files can a process read from/write to?

UNIX Process permissions

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



/etc/shadow is owned by root

Users shouldn’t be able to write to it generally

How do you reset your password?

```
lrwxrwxrwx  1 root   root         10 Oct 18 12:43 which -> /bin/which
-rwxr-xr-x  1 root   root      46940 Mar  2  2017 who
-rwxr-xr-x  1 root   root      26364 Mar  2  2017 whoami
-rwxr-xr-x  1 root   root       1460 Apr 14  2016 wifi-status
-rwxr-xr-x  1 root   root      18040 Nov 21  2016 w.procps
lrwxrwxrwx  1 root   root         23 Oct 18 12:49 write -> /etc/alternatives/write
lrwxrwxrwx  1 root   root          1 Mar  4  2016 X11 -> .
-rwxr-xr-x  1 root   root      71156 Feb  7  2016 xargs
-rwxr-xr-x  1 root   root      39144 Mar 26  2015 xauth
-rwxr-xr-x  1 root   root        234 Apr 13  2016 xdg-user-dir
-rwxr-xr-x  1 root   root      18036 Apr 13  2016 xdg-user-dirs-update
-rwxr-xr-x  1 root   root       5126 Mar 13  2016 xsubpp
-rwxr-xr-x  1 root   root      13804 Nov 24  2016 xxd
-rwxr-xr-x  1 root   root      67516 Feb 12  2014 xz
lrwxrwxrwx  1 root   root          2 Feb 12  2014 xzcat -> xz
lrwxrwxrwx  1 root   root          6 Feb 12  2014 xzcmp -> xzdiff
-rwxr-xr-x  1 root   root       5518 Feb 12  2014 xzdiff
lrwxrwxrwx  1 root   root          6 Feb 12  2014 xzegrep -> xzgrep
lrwxrwxrwx  1 root   root          6 Feb 12  2014 xzfgrep -> xzgrep
-rwxr-xr-x  1 root   root       5421 Feb 12  2014 xzgrep
-rwxr-xr-x  1 root   root       1825 Feb 12  2014 xzless
-rwxr-xr-x  1 root   root       2168 Feb 12  2014 xzmore
-rwxr-xr-x  1 root   root      26364 Mar  2  2017 yes
-rwxr-xr-x  1 root   root      13920 Jun 16 13:37 zdump
-rwxr-xr-x  1 root   root      48459 Mar 13  2016 zipdetails
targaryend@tessaro:/usr/bin$ ls -all passwd
-rwsr-xr-x 1 root root 53128 May 16 16:38 passwd
targaryend@tessaro:/usr/bin$ passwd
Changing password for targaryend.
(current) UNIX password:
```

Process permissions continued

UID 0 is root

Real user ID (RUID) --
same as UID of parent (who started process)

Effective user ID (EUID) --
from set user ID bit of file being executed or due to sys call

Executable files have 2 setuid bits

- **Setuid** bit – set EUID of process to owner's ID
- **Setgid** bit – set EGID of process to group's ID

So passwd is a setuid program

program runs at permission level of owner, not user that runs it

How do you reset your password?

```
lrwxrwxrwx  1 root   root        10 Oct 18 12:43 which -> /bin/which
-rwxr-xr-x  1 root   root     46940 Mar  2  2017 who
-rwxr-xr-x  1 root   root     26364 Mar  2  2017 whoami
-rwxr-xr-x  1 root   root      1460 Apr 14  2016 wifi-status
-rwxr-xr-x  1 root   root     18040 Nov 21  2016 w.procps
lrwxrwxrwx  1 root   root        23 Oct 18 12:49 write -> /etc/alternatives/write
lrwxrwxrwx  1 root   root         1 Mar  4  2016 X11 -> .
-rwxr-xr-x  1 root   root     71156 Feb  7  2016 xargs
-rwxr-xr-x  1 root   root     39144 Mar 26  2015 xauth
-rwxr-xr-x  1 root   root       234 Apr 13  2016 xdg-user-dir
-rwxr-xr-x  1 root   root     18036 Apr 13  2016 xdg-user-dirs-update
-rwxr-xr-x  1 root   root      5126 Mar 13  2016 xsubpp
-rwxr-xr-x  1 root   root
-rwxr-xr-x  1 root   root
lrwxrwxrwx  1 root   root
lrwxrwxrwx  1 root   root
-rwxr-xr-x  1 root   root
lrwxrwxrwx  1 root   root
lrwxrwxrwx  1 root   root
-rwxr-xr-x  1 root   root
-rwxr-xr-x  1 root   root      1825 Feb 12  2014 xzless
-rwxr-xr-x  1 root   root      2168 Feb 12  2014 xzmore
-rwxr-xr-x  1 root   root     26364 Mar  2  2017 yes
-rwxr-xr-x  1 root   root     13920 Jun 16 13:37 zdump
-rwxr-xr-x  1 root   root     48459 Mar 13  2016 zipdetails
targaryend@tessaro:/usr/bin$ ls -all passwd
-rwsr-xr-x 1 root root 53128 May 16 16:38 passwd
targaryend@tessaro:/usr/bin$ passwd
Changing password for targaryend.
(current) UNIX password:
```

Because the setuid bit is set, passwd can run with root's privileges even if executed by any other users, and can thus operate on /etc/shadow!

seteuid system call

Idea: raise privileges only when needed within your code!

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

Agenda

1. Multi-user Systems

2. Access control in UNIX

3. **Attacks on SetUID programs**

Setuid allows privilege escalation but…

- Source of many privilege escalation vulnerabilities

Example 1:
Capability Leaking

Example 2:
Race conditions

Example 3:
Env variables

Capability leaking

- In some cases, privileged programs downgrade themselves during execution. Example: `su`

```
… // Some privileged code
setuid(getuid()); // Disable privilege
// Execute /bin/sh
v[0] = "/bin/sh", v[1] = 0
execve(v[0], v, 0)
```

- **Issue:** Program may not clean up privileged capabilities before downgrading

Capability leaking: An example

```
fd = open("/etc/shadow", O_RDWR|O_APPEND)
setuid(getuid()); // Disable privilege
// Execute /bin/sh
v[0] = "/bin/sh", v[1] = 0
execve(v[0], v, 0)
```

Forget to close the file, so the file descriptor is still valid

Exploit: Write to /etc/shadow with the content of myfile

```
cat myfile >& 3
```

File descriptor 3 is usually allocated for the first opened file

Race conditions

Time-of-check-to-time-of-use (TOCTTOU)

Say the following is run with EUID = 0

```
if( access("/tmp/myfile", R_OK) != 0 )
{
   exit(-1);   Ensures that RUID can access file. If not abort
}
file = open( "/tmp/myfile", "r" );
read( file, buf, 100 );
close( file );
print( "%s\n", buf );
```

access checks RUID, but open only checks EUID

access("/tmp/myfile", R_OK)

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

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

SetUID process

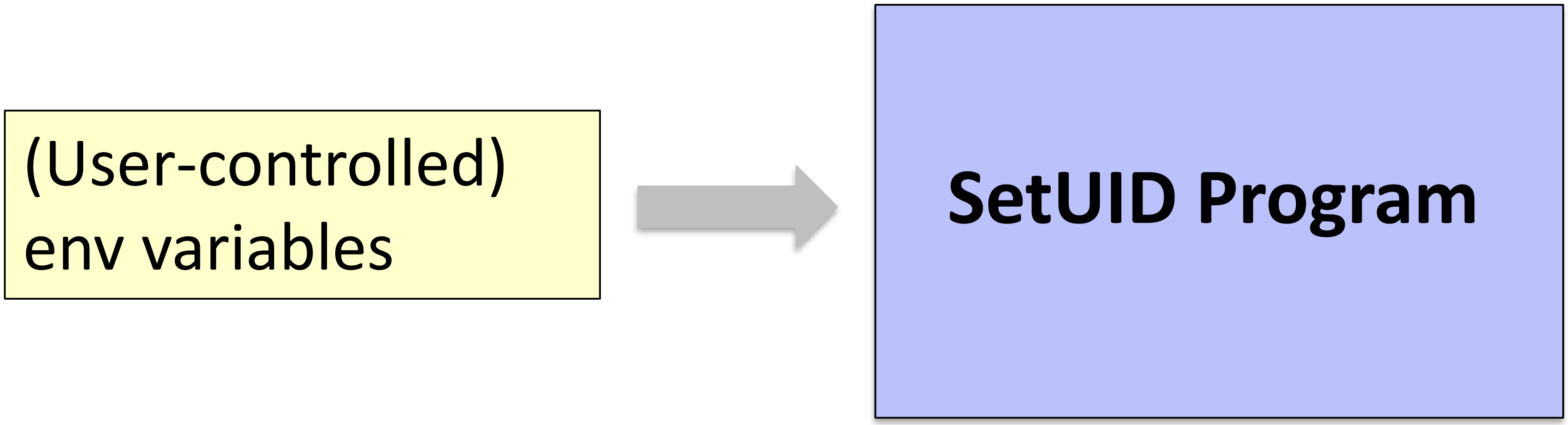
Non-privileged process

ln –s /home/root/.ssh/id_rsa   /tmp/myfile

Outcome?

Prints out root's secret key…

Environment variables



List of shared libraries that will be searched by dynamic linker

Examples: `PATH`, `LD_PRELOAD`

Location of commands that will be searched by shell if full path is not provided

Example: Attack via PATH

Say the following is run with EUID = 0

```
#include <stdlib.h>
int main()
{
system("cal"); // Run calendar
}
```

How to attack

Set up a malicious “calendar” program in the home directory

```
#include <stdlib.h>
int main()
{
system(“/bin/bash -p”); // Run shell
}
```

How to attack

Tell the shell to look up commands in the home directory first

```
$ export PATH = .:PATH
```

Run the SetUID program

```
system("cal");
```

Outcome?

Malicious "calendar" is run, and attacker gets root shell