SECURITY (COMP0141): UNIX PROCESSES

(MANY OF THESE SLIDES ARE TAKEN FROM ZAKIR DURUMERIC. THANKS ZAKIR!)

PROCESSES

Processes are isolated (cannot access each others' memory)

Processes run with the user ID (`uid`) of a specific user

- When you run a process, it's with the permissions of your `uid`
- Processes can access any files that you have access to

Processes started by `root` (`uid` 0) can reduce their privileges by changing to a less privileged `uid`

PROCESS USER IDS

Every process has three different user IDs:

Effective User ID (EUID): determines permissions for the process

Real User ID (RUID): determines the user that started the process

Saved User ID (SUID): EUID prior to any changes

CHANGING USER IDS

`root` can change EUID / RUID / SUID to arbitrary values

Unprivileged users can change EUID to RUID or SUID

`setuid(x)` changes all of EUID / RUID / SUID to `x`

`seteuid(x)` changes just EUID to `x`

SSH EXAMPLE

What if SSH runs as `root` and ran the following code?

```
if (authenticate(uid, passwd) == SUCCESS) {
      seteuid(uid);
      exec("/bin/bash");
}
```

```
euid = 0
ruid = 0
suid = 0
```

SSH EXAMPLE

What if SSH runs as `root` and ran the following code?

```
if (authenticate(uid, passwd) == SUCCESS) {
      seteuid(uid);
      exec("/bin/bash");
}
```

euid = ~~0~~ uid

ruid = 0

suid = 0

SSH EXAMPLE

What if SSH runs as `root` and ran the following code?

```
if (authenticate(uid, passwd) == SUCCESS) {
      seteuid(uid);
      exec("/bin/bash");
}
```

euid = ~~0~~ uid
ruid = 0
suid = 0

Unprivileged users can change EUID to RUID or SUID

`setuid(0)` ← get root privileges

SSH EXAMPLE

What if SSH runs as `root` and ran the following code?

```
if (authenticate(uid, passwd) == SUCCESS) {
      setuid(uid);
      exec(“/bin/bash”);
}
```

euid = ~~0~~ uid
ruid = ~~0~~ uid
suid = ~~0~~ uid

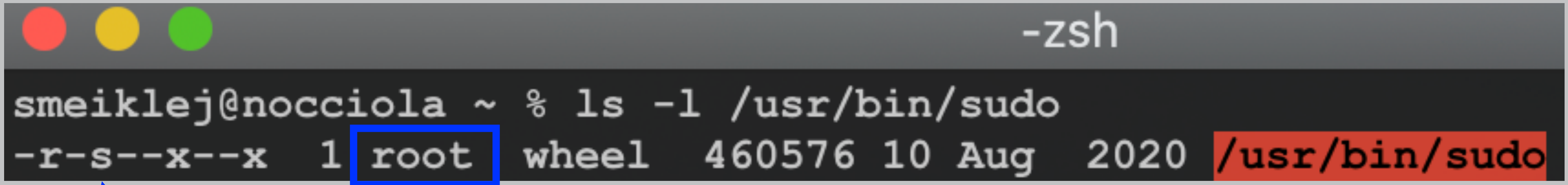
ELEVATING PRIVILEGES

Sometimes we need to elevate our own privileges

Example: Running `passwd` modifies `/etc/shadow`, which only `root` can read/write

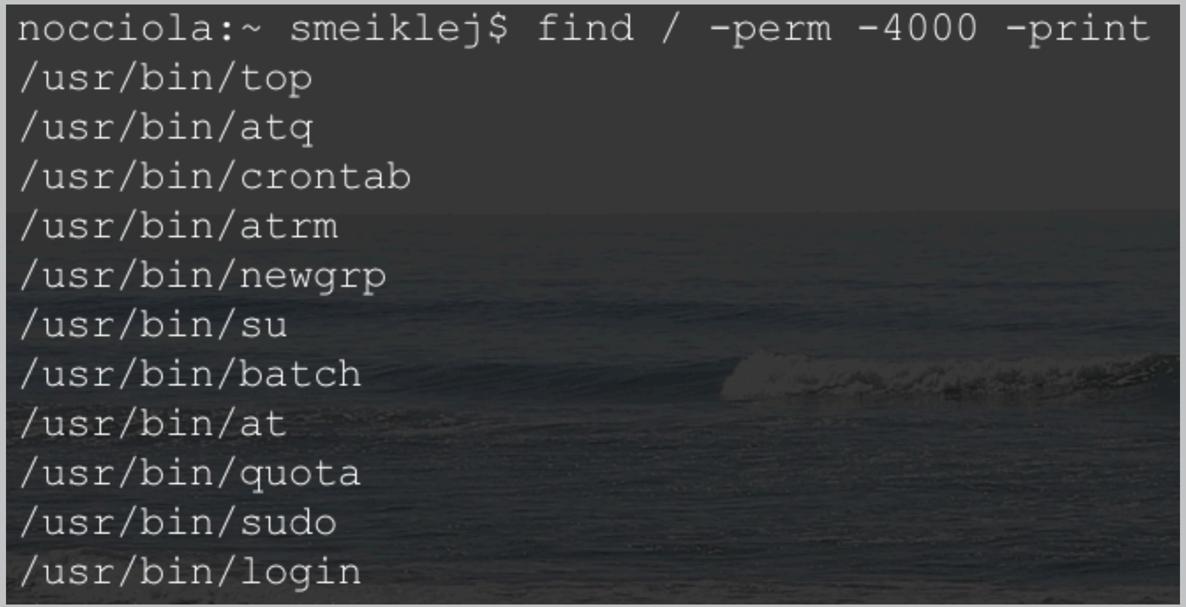
UNIX allows you to set EUID of an executable to be the file owner rather than the executing user using the **`setuid` bit**

SETUID BIT



```
smeiklej@nocciola ~ % ls -l /usr/bin/sudo
-r-s--x--x  1 root  wheel  460576 10 Aug  2020 /usr/bin/sudo
```

says that executing user can run with the permissions of the file owner (euid=root)



```
nocciola:~ smeiklej$ find / -perm -4000 -print
/usr/bin/top
/usr/bin/atq
/usr/bin/crontab
/usr/bin/atrm
/usr/bin/newgrp
/usr/bin/su
/usr/bin/batch
/usr/bin/at
/usr/bin/quota
/usr/bin/sudo
/usr/bin/login
```

Question: When running `passwd`, how do we know which user's password can be modified?

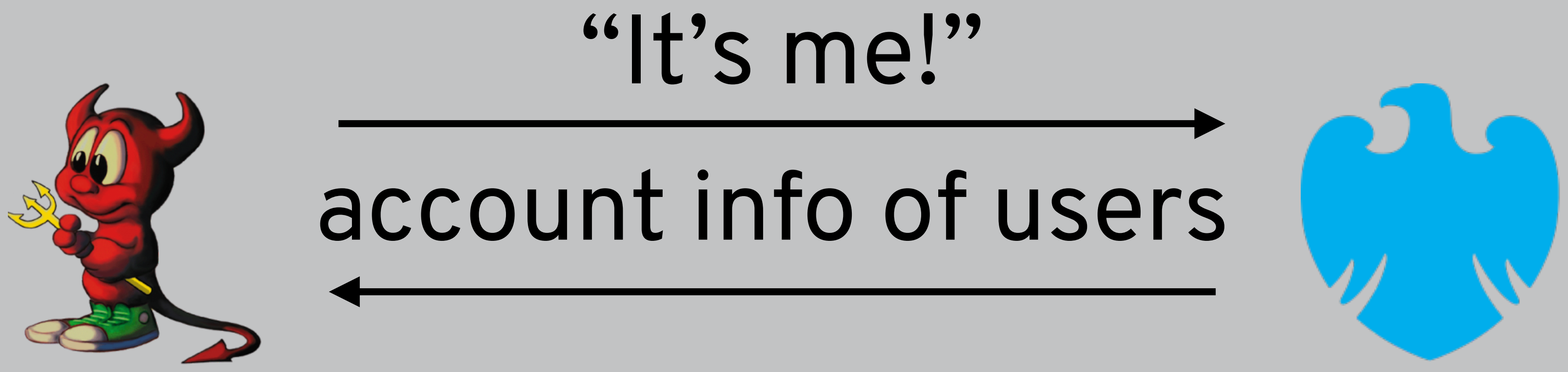
Answer: The SUID (Saved User ID)

Question: What if `setuid` has a vulnerability?

STRIDE

S T R I D E

Elevation of privilege



“It’s me!”

account info of users

CHANGING PRIVILEGES

When a user connects to a system, it runs `login` process as root

- Authenticates user with their username and password
- Changes userid and groupid to be those of the user
- Executes the user’s shell
- So system **drops** privileges from root to regular user

Does a user ever need to **elevate** privilege? Yes!

- One example: changing their password (edits master password file for the system)
- This needs some authorised way to elevate privileges
- Achieved using the `setuid` bit

PRIVILEGES

Other architectures (like Windows) have differences but the themes are the same

Pros?

- Simple model provides protection for most situations
- Flexible enough to make most access control policies possible

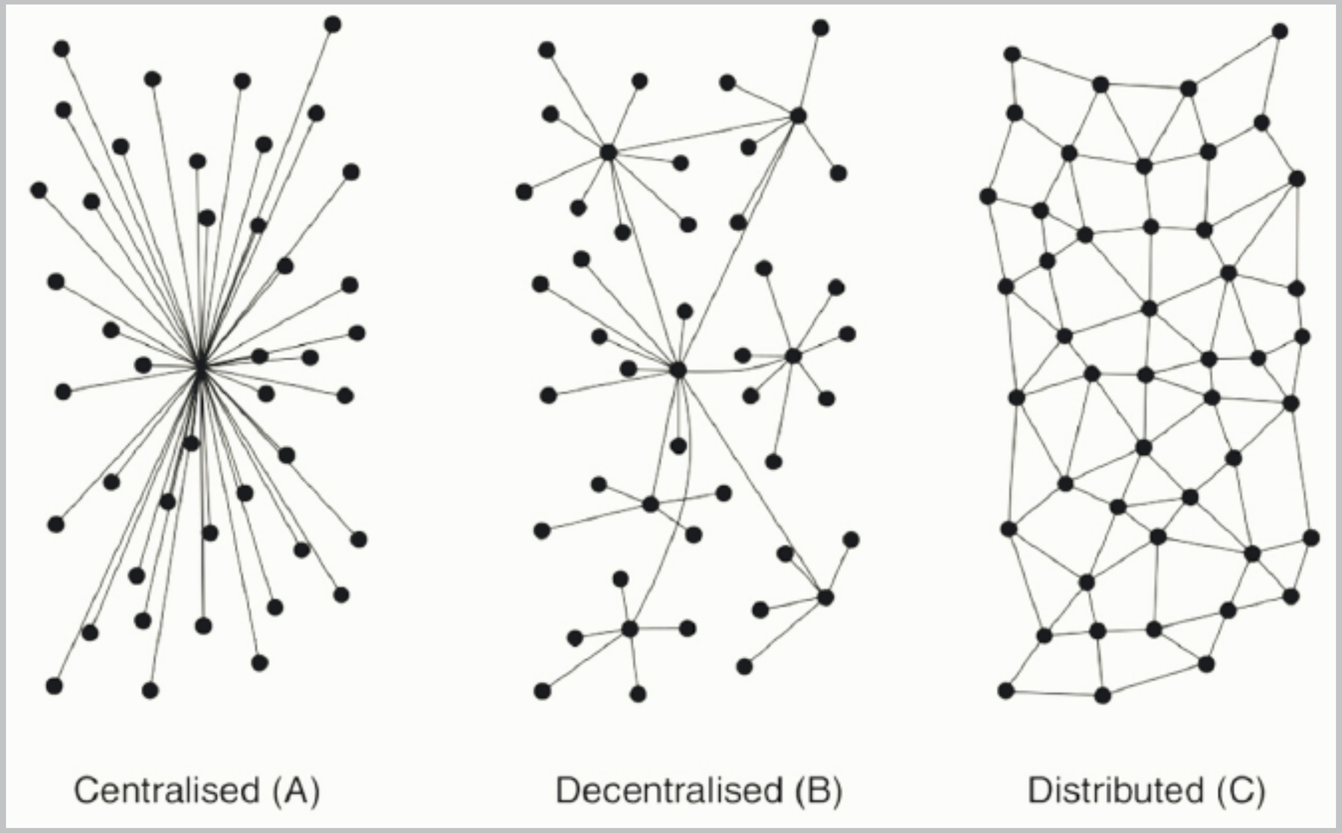
Cons?

- ACLs are coarse-grained
- Can’t differentiate processes run by a single user
- Nearly all systems operations require root access

PERMISSIONS

The past (and present!): one mainframe computer with many users

- Still highly relevant in large organisations
- Also the model we follow in platforms like Moodle



The present: many distributed personal devices

- Users need to make more decisions for themselves