Next

DAC (today)

MAC (Thursday)

Capabilities (later)

There are lots of "AC"s that get tossed around these days (DAC, MAC, ABAC, RBAC, etc.), but we'll concentrate on three fundamental forms of authorization:

- discretionary access control (DAC)
- mandatory access control (MAC)
- capabilities

Other schemes can often typically implemented in terms of the above. For example, role-based access control (RBAC) can be implemented using MAC primitives.

DAC

Discretionary access control

Organizing principle:

Files and directories have **owners** who have the **discretion** to say who gets to access them.

Major implementations:

Unix permissions
Access control lists (ACLs)

We saw an example of Unix permissions in Lab 0, when we had to use the `chmod` command to make the binary executable `game`, well, executable!

Unix DAC

Users:

User-readable names, user IDs in `/etc/passwd`*

... or elsewhere

Groups:

Numeric *group ID* with names in `/etc/group`

Users can be members of multiple groups

This file doesn't contain what you might think it does... stay tuned for password hashing in later lectures!

We often think of users as being identified by _____: short, human-readable names that are unique to _____ _____. If I'm using a computer, I can see my current username by running `whoami(1)`. Something a bit more meaningful to the computer, however, is not a _____ but a _____. User IDs are still short and _____ _____, but instead of strings, they're _____ _____. You can see information about your user ID (and group IDs!) by running `id(1)` (or just `whoami` on Windows).

Most Unix-like computers have a *Name Service Switch* configuration file in `/etc/nsswitch.conf` that tells the host where to find names for users, groups, networks, hosts, RPCs...

In addition to a user ID, every user can be a member of multiple *groups* that are identified by integer *group ID*.

Unix file permissions

Each file has **read**, **write** and **execute** permission for each of **owner**, **group** and **other** users:

```
[jon website]$ ls
drwxr-xr-x   4 jon  jon     8B Mar 26  2017 assets
-rw-r--r--   1 jon  jon   948B Jan 26 15:37 config.yaml
drwxr-xr-x   8 jon  jon    10B Feb 13 23:19 content
-rwxr-xr-x   1 jon  jon   271B Jan 13  2017 deploy
drwxr-xr-x   7 jon  jon     9B Jan 22 23:14 layouts
drwxr-xr-x  12 jon  jon    13B Jan 24 16:18 static
```

File owner can set permissions with `chmod` command

These permissions sound very much like virtual memory permissions, and they do indeed have the same meanings. However, their enforcement is very different!

Unix permissions

For each of (owner, group, anyone):

Value	Meaning
4	Readable
2	Writable
1	Executable

Octal example: `0644` (writable by owner, readable by anyone).

```
$ chmod 644 file.txt
$ chmod g+rx game
```


These power-of-two values can be XOR'ed together.

This is one of the very few instances of an octal representation that you're likely to see anywhere!

Changing file owner

Owner has discretion to set file access permissions... but how do we set the owner?

Answer: `chown(1)`

But:

```
$ chown alice foo.txt
chown: foo.txt: Operation not permitted
```


Show man page for `chown(2)`

Superuser
a.k.a., `root` user

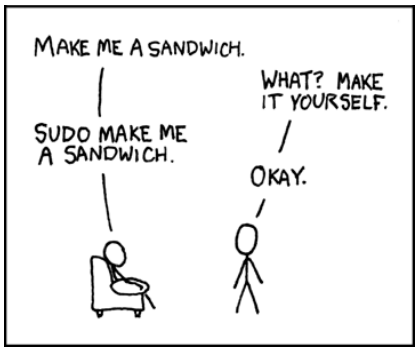
- UID 0
- can change file owner, `chmod` other users' files
- second-level objective for many attacks

The `root` user is allowed to violate the DAC policy, overriding the access control decisions made by a file's owner (and even _____!). To "get root" is to gain administrative control over a computer, whether legitimately becoming a system administrator ("yeah, I've got root on that box") or otherwise.

Many, many attacks against systems start by gaining _____ (running whatever the attacker wants within a process, with that process' credentials) and then a _____ attack against a service that allows the attacker to _____ to administrative access.

Root-only programs

- lots of tools require *root privilege*:
  - filesystem management
  - package managers
  - service management
  - often via `sudo(8)`



Exercise: Consider how a user who can control all software installation on a computer could violate another user's security policy

We don't want just any user being able to, e.g., control a mounted filesystem or install a package. Why not?

For all of these examples, being in control of such a subsystem would allow a user to be able to _____.

Root-only programs

- some programs require `root` privilege
- some programs must be runnable by anyone
- some are both!
- e.g., `ping(8)`, even `intel_backlight(1)`!

```
$ ls -l `which intel_backlight`
-r-sr-xr-x  1 root  wheel    16K Feb 26 17:03 /usr/local/bin/intel_back
```


Since we don't want just anybody controlling critical subsystems, some programs require `root` privilege in order to do their work. For example, I can _____ on my machine from an ordinary user account, but I can only _____ to system locations (e.g., `/usr/local/bin`) as `root`.

Some programs, however, require privilege to do their job and *also* need to be run by ordinary users! We can implement such functionality, overriding the normal DAC policy, using `setuid` and `setgid` software.

setuid/setgid programs

`setuid`: set *effective UID* to file owner's UID on run
`setgid`: set *effective GID* to file group's GID on run

Can query *real* or *effective* UID/GID:

```c
#include <unistd.h>

uid_t   getuid(void);
uid_t   geteuid(void);
gid_t   getgid(void);
gid_t   getegid(void);
```


Example: getuid.c

Authorization

DAC (today)

MAC (Thursday)

Capabilities (later)