So far

~~Introduction~~

~~Software security~~

Host security

Network security

Web security

Today

Processes

Users

Next time:

Authorization

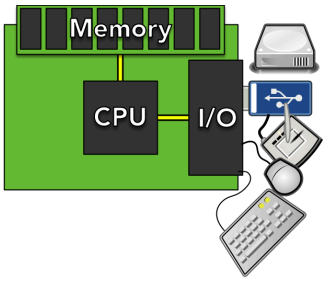
Processes

A process is a *running program*

Processes have memory

Processes execute

Processes have indirect access to I/O



We've been working with processes already, but we haven't really defined them. A process is just a _____, and it runs on a _____ computer. To see the processes that are currently running on your computer, you can use a GUI like Activity Monitor or Task Manager, or you can use the `ps` command at the command line.

Run `ps aux | wc -l` on macOS, FreeBSD.

The addresses used directly by your software are _____. This is why, when the same code is executed in ten processes in parallel, they can all refer to the _____ _____, but *without interference*.

The OS must *schedule* processes for execution, deciding which process will get to run on which processor at which time. This is a topic for ECE 8400 / ENGI 9875.

External resources such as data on a disk or a network connection are generally/typically/almost always represented as _____.

Files

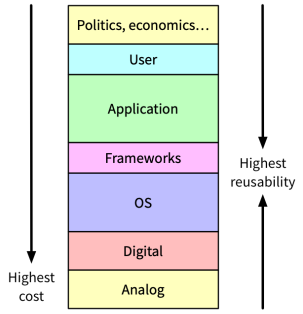
Multiple abstraction layers:

```
new PrintWriter(new BufferedWriter(new Fil
```

```
std::cout << "Hello, world!" << std::endl;
```

```
fwrite(stdout, "Hello, world!\n");
```

```
char message[] = "Hello, world!\n";
int fd = STDOUT_FILENO;
write(fd, message, sizeof(message));
```



There are lots of ways to look at files from lots of levels of abstraction. Each layer tends to add something nice and/or helpful, but at bottom, all of these layers' concepts of files are rooted in an _____. This is because no library can control a disk: only _____ _____ can.

This is another example of how some interfaces are more "real" than others. Interfaces within a single address space — such as library APIs — can wrap up the functionality of lower-level libraries, but when it comes down to the level of _____ (as in a _____ programming language), there is no "real" separation among them. However, interfaces between different _____ of things (users and computers, processes and the OS, software and hardware) tend to be much clearer, "harder" interfaces.

Process file abstractions

Each process has a set of integer *file descriptors*

Can use *system calls* to open, close, read, write, etc.

```
int fd = open("/home/jon/hello.txt", O_RDONLY);
/* ... */
write(fd, some_data_bytes, data_length);
/* ... */
```


A file descriptor can then get wrapped up, together with some buffering for performance, into a _____. That can then get wrapped up into a C++ _____, which might be used to implement a Java `java.io.FileWriter`.

File I/O system calls

From a process' perspective:

- system calls are C functions
- files are named by small integer indices (e.g., FD 3)
- each process has its **own** array of files
- ... and that's enough detail for now

There's a lot more detail to be dug into about file descriptors (e.g., how `libc` communicates with the OS kernel), and we'll go into that detail in ECE 8400 / ENGI 9875. In fact, our first lab in that course will have you invoking system calls via nothing but native instructions in assembly code! For now, however, this is all that we need to know in order to start talking about host security.

Processes

Processes have *memory* — virtual memory

Processes *execute* — threads

Processes have (indirect) access to *resources* — files

Processes execute on behalf of... ???

Users

Usernames

User IDs

User authentication

User authorization

What's a user? A human being, sure, but how does the computer see a user? How do we think of users?

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).

How does the computer know that the person sitting in front of it is *actually* the person they claim to be? That's called user *authentication*, and we'll get into it when we start talking about passwords (and their problems, and alternatives).

Next time, we'll start getting into user *authorization*: saying _____ is allowed to do _____ to _____.

User databases

Where is user information stored?

- Active Directory
- Binary databases (e.g., Berkeley DB caches)
- NIS[plus] (though not really any more)
- OpenLDAP
- Text files (e.g., `/etc/passwd`)

We'll examine user details specific to discretionary access control next time, but for now we will look at the contents of files like `/etc/nsswitch.conf`. This *Name Service Switch* config file tells a Unix machine where to find information about lots of kinds of names: users, groups, protocols, shells, etc. It can point us at `files`, at binary `db` files (typically used as a cache), at `ldap` (which might actually be Active Direcftory) or at the basically-defunct `nisplus`.

User database: files

```
# $FreeBSD$
#
root:*:0:0:Charlie &:/root:/bin/sh
toor:*:0:0:Bourne-again Superuser:/root:
daemon:*:1:1:Owner of many system processes:/root:/usr/sbin/nologin
operator:*:2:5:System &:/:/usr/sbin/nologin
bin:*:3:7:Binaries Commands and Source:/:/usr/sbin/nologin
tty:*:4:65533:Tty Sandbox:/:/usr/sbin/nologin
kmem:*:5:65533:KMem Sandbox:/:/usr/sbin/nologin
games:*:7:13:Games pseudo-user:/:/usr/sbin/nologin
news:*:8:8:News Subsystem:/:/usr/sbin/nologin
man:*:9:9:Mister Man Pages:/usr/share/man:/usr/sbin/nologin
sshd:*:22:22:Secure Shell Daemon:/var/empty:/usr/sbin/nologin
# ...
```


We can examine local user details via:

`cat /etc/passwd`

If we look at `/etc/passwd` on a local Unix-like machine, we should see users like the user who set up the box. On a LabNet machine, however, _____...

User database: LDAP

Lightweight Directory Access Protocol

Queries *directory servers*:

- Active Directory
- ApacheDS
- FreeIPA / 389 directory server
- OpenLDAP

In practice, Microsoft Active Directory is absolutely dominant, as most large networks support large numbers of Windows PCs.

In a LabNet environment, we can query lots of interesting details from LDAP using commands like:

```
ldapsearch –H ldaps://dogbert.cs.mun.ca "(uid=p15jra)"
```

Next

DAC (today)

MAC (Tuesday)

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.

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

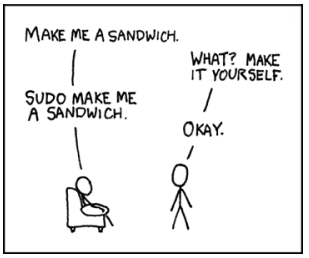
- UID 0
- can change file owner, `chmod` other users' files
- second-level objective for many attacks

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_bac
```


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

DAC

Organizing principle:

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

Major implementations:

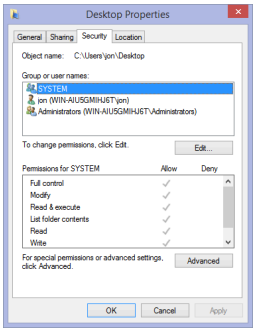
Unix permissions
Access control lists (ACLs)

So... how about that other implementation, ACLs?

ACLs: access control lists

(the other way of doing discretionary access control)

- explicit list of users, groups
- independent permissions for each
- useful for complex authorization on multi-user shared systems* implemented in NFSv4, POSIX 1e, Windows/SMB...



ACLs are useful, but it's also very easy to write an ACL that you yourself don't understand! Just look at some of the literature around trying to make ACLs understandable by users:

Intentional access management: making access control usable for end-users Cao and Iverson, *SOUPS '06: Proceedings of the second symposium on Usable privacy and security, July 2006*. DOI: https://doi.org/10.1145/1143120.1143124

Relating declarative semantics and usability in access control Krishnan, Tripunitara, Chik and Bergstrom, *SOUPS '12: Proceedings of the Eighth Symposium on Usable Privacy and Security*, July 2012. DOI: https://doi.org/10.1145/2335356.2335375

The poor usability of OpenLDAP Access Control Lists, Chen, Punchhi and Tripunitara, *IET Information Security* 17(1), January 2023. DOI: https://doi.org/10.1049/ise2.12079

If people can publish "how to make ACLs usable" over three decades... maybe there's a problem with ACLs.

Summary

Processes and Users

Authorization

DAC (today)

MAC (Thursday)

Capabilities (later)