[02] PROTECTION

OUTLINE

- Protection
  - Motivation, Requirements, Subjects & Objects
  - Design of Protection Systems
  - Covert Channels
- Low-level Mechanisms
  - IO, Memory, CPU
- OS Structures
  - Dual-mode Operation, Kernels & Microkernels
  - Mandatory Access Control, `pledge(2)`
- Authentication
  - User to System, System to User
  - Mutual Suspicion
- Access Matrix
  - Access Control Lists (ACLs) vs Capabilities

PROTECTION

- **Protection**
    - **Motivation, Requirements, Subjects & Objects**
    - **Design of Protection Systems**
    - **Covert Channels**
- Low-level Mechanisms
- OS Structures
- Authentication
- Access Matrix

WHAT ARE WE PROTECTING AGAINST?

Unauthorised release of information

- Reading or leaking data
- Violating privacy legislation
- Covert channels, traffic analysis

Unauthorised modification of information

- Changing access rights
- Can do sabotage without reading information

(Unauthorised) denial of service

- Causing a crash
- Causing high load (e.g. processes or packets)

Also protection against effects of errors: e.g., isolate for debugging, damage control

Impose controls on access by **subjects** (e.g. users) to **objects** (e.g. files)

COVERT CHANNELS

Information leakage by side-effects: lots of fun! At the hardware level:

- Wire tapping
- Monitor signals in machine
- Modification to hardware
- Electromagnetic radiation of devices

By software:

- File exists or not
- Page fault or not
- Compute or sleep
- 1 or 0
- System provided statistics

E.g., lowest layer of recent OCaml TLS library in C to avoid side channel through garbage collector

ASPECTS OF PROTECTION SYSTEM

Physical, e.g.,

- Lock the computer room
- Restrict access to system software

Social, e.g.,

- De-skill systems operating staff
- Keep designers away from final system!
- Legislate

Technical, e.g.,

- Use passwords (in general challenge/response)
- Use encryption

DESIGN OF PROTECTION SYSTEMS

From [Saltzer & Schroeder, Proc. IEEE, September 1975]:

- Design should be public
- Default should be no access
- Check for current authority
- Give each process minimum possible authority
- Mechanisms should be simple, uniform and built in to lowest layers
- Should be psychologically acceptable
- Cost of circumvention should be high
- Minimize shared access

LOW-LEVEL PROTECTION

- Protection
- **Low-level Mechanisms**
  - **IO, Memory, CPU**
- OS Structures
- Authentication
- Access Matrix

PROTECTING IO & MEMORY

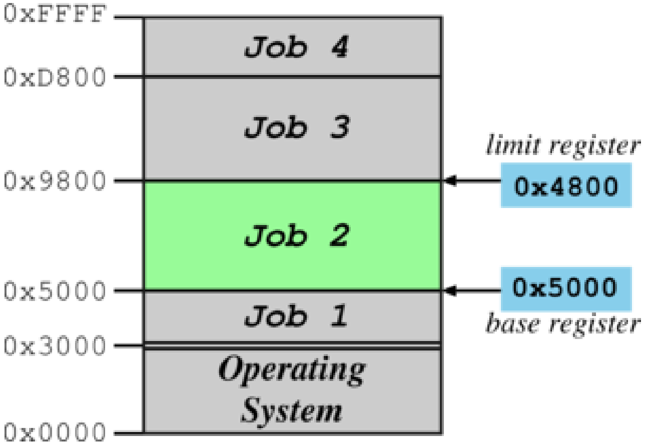
Initially, try to make IO instructions privileged:

- Applications can't mask interrupts (that is, turn one or many off)
- Applications can't control IO devices

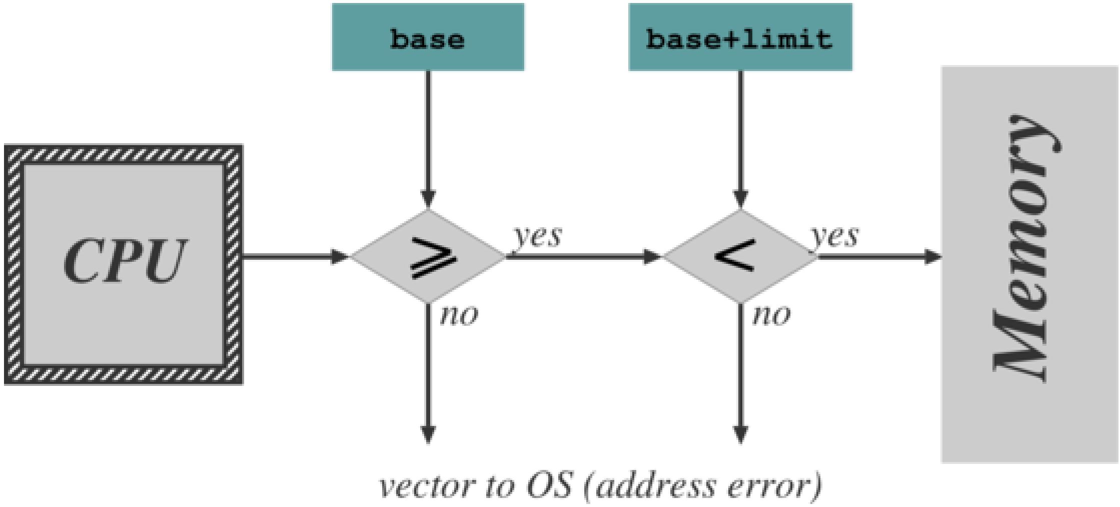
But!

- Some devices are accessed via memory, not special instructions
- Applications can rewrite interrupt vectors

Hence protecting IO means also protecting memory, e.g. define a base and a limit for each program, and protect access outside allowed range



IMPLEMENTING MEMORY PROTECTION



Have hardware check every memory reference:

- Access out of range causes vector into OS (as for an interrupt)
- Only allow update of base and limit registers when in kernel mode
- May disable memory protection in kernel mode (although a bad idea)

In reality, more complex protection hardware is used (see *Paging* and *Segmentation*)

PROTECTING THE CPU

Need to ensure that the OS stays in control:

- I.e., must prevent any application from "hogging" the CPU the whole time
- Means using a timer, usually a countdown timer, e.g.,
  - Set timer to initial value (e.g. `0xFFFF`)
  - Every tick (e.g. $1 \mu s$ or, nowadays, programmable), timer decrements value
  - When value hits zero, interrupt
- Ensures the OS runs periodically

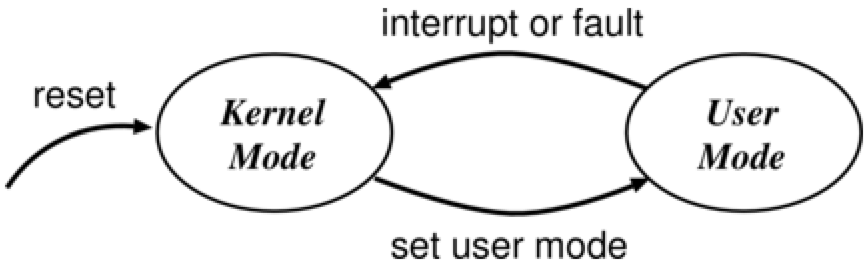
Requires that only OS can load timer, and that interrupt cannot be masked:

- Use same scheme as for other devices
- Re-use to implement time-sharing (later)

OS STRUCTURES

- Protection
- Low-level Mechanisms
- **OS Structures**
  - **Dual-mode Operation, Kernels & Microkernels**
  - **Mandatory Access Control, `pledge(2)`**
- Authentication
- Access Matrix

DUAL-MODE OPERATION



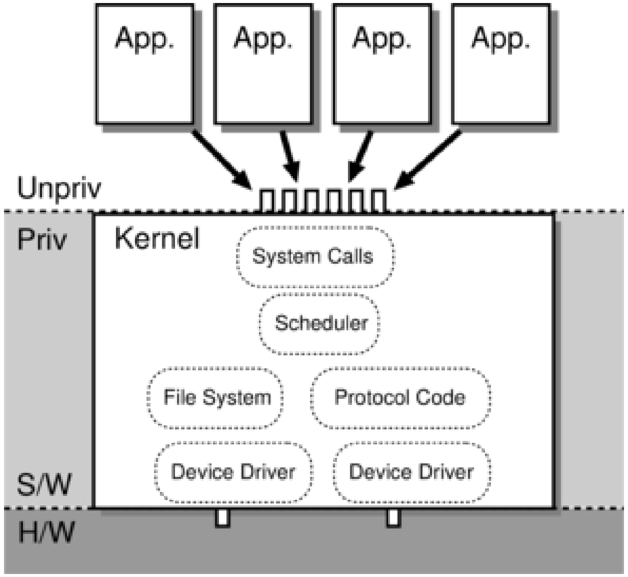
Simply want to stop buggy (or malicious) program from doing bad things

- Trust boundary between user **application** and the **OS**
- Use hardware support to differentiate between (at least) two modes of operation
  1. User Mode : when executing on behalf of a user (i.e. application programs).
  2. Kernel Mode : when executing on behalf of the OS
- Make certain instructions only possible in kernel mode, indicated by **mode bit**

E.g., x86: Rings 0—3, ARM has seven modes including `IRQ` and `FIQ`

Often "nested" (per x86 rings): further inside can do strictly more. Not ideal — e.g., stop kernel messing with applications — but disjoint/overlapping permissions hard

KERNEL-BASED OPERATING SYSTEMS



Applications can't do IO due to protection so the OS does it on their behalf

This means we need a secure way for application to invoke OS: a special (unprivileged) instruction to transition from user to kernel mode

Generally called a **trap** or a **software interrupt** since operates similarly to (hardware) interrupt...

OS services accessible via software interrupt mechanism called **system calls**

OS has vectors to handle traps, preventing application from leaping to kernel mode and then just doing whatever it likes

Alternative is for OS to emulate for application, and check every instruction, as used in some virtualization systems, e.g., QEMU

MICROKERNEL OPERATING SYSTEMS

We've protected "privileged instructions" via dual-mode operation, memory via special hardware, and the CPU via use of a timer. But now applications can't do much directly and must use OS to do it on their behalf

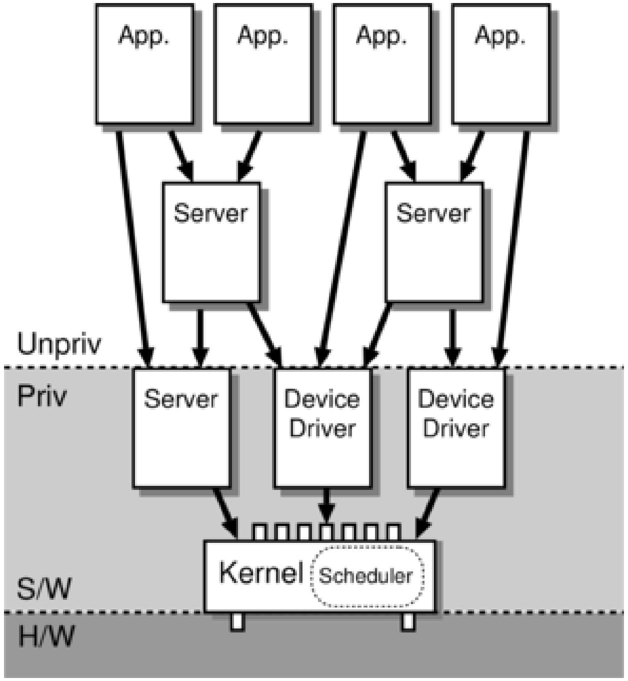
OS must be very stable to support apps, so becomes hard to extend

Alternative is **microkernels**: move OS services into (local) servers, which may be privileged

Increases both modularity and extensibility

Still access kernel via system calls, but need new ways to access servers: **Inter-Process Communication** (IPC) schemes

Given talking to servers (largely) replaces trapping, need IPC schemes to be extremely efficient



KERNELS VS MICROKERNELS

So why isn't everything a microkernel?

- Lots of IPC adds overhead, so microkernels (perceived as) usually performing less well
- Microkernel implementation sometimes tricky: need to worry about synchronisation
- Microkernels often end up with redundant copies of OS data structures

Thus many common OSs blur the distinction between kernel and microkernel.

- E.g. Linux is "kernel", but has kernel modules and certain servers.
- E.g. Windows NT was originally microkernel (3.5), but now (4.0 onwards) pushed lots back into kernel for performance
- Unclear what the best OS structure is, or how much it really matters...

VIRTUAL MACHINES AND CONTAINERS

More recently, trend towards encapsulating applications differently. Roughly aimed towards making applications appear as if they're the only application running on the system. Particularly relevant when building systems using **microservices**. Protection, or **isolation** at a different level

- **Virtual Machines** encapsulate an entire running system, including the OS, and then boot the VM over a hypervisor

  E.g., Xen, VMWare ESX, Hyper-V

- **Containers** expose functionality in the OS so that each container acts as a separate entity even though they all share the same underlying OS functionality

  E.g., Linux Containers, FreeBSD Jails, Solaris Zones

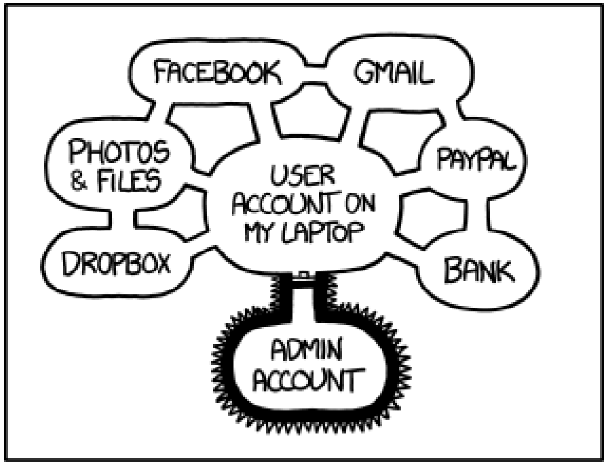
MANDATORY ACCESS CONTROL

From a user point of view, nowadays one often wants to protect applications from each other, all owned by a single user. Indeed, with personal single-user machines now common (phones, tablets, laptops), arguable that protection model is mismatched

Mandatory Access Control (MAC) mandates expression of policies constraining interaction of system users

E.g., OSX and iOS *Sandbox* uses subject/object labelling to implement access-control for privileges and various resources (filesystem, communication, APIs, etc)

https://xkcd.com/1200/



IF SOMEONE STEALS MY LAPTOP WHILE I'M LOGGED IN, THEY CAN READ MY EMAIL, TAKE MY MONEY, AND IMPERSONATE ME TO MY FRIENDS,

BUT AT LEAST THEY CAN'T INSTALL DRIVERS WITHOUT MY PERMISSION.

PLEDGE(2)

One way to reduce the ability of a compromised program to do Bad Things™ is to remove access to unnecessary system calls

Several attempts in different systems, with varying (limited) degrees of success:

- Hard to use correctly (e.g., Capsicum), or
- Introduce another component that needs to be watched (e.g., `seccomp`)

Observation:

- Most programs follow a pattern of `initialization()` then `main_loop()`, and
- The `main_loop()` typically uses a much narrower class of system calls than `initialization()`

Result? `pledge(2)` — ask the programmer to indicate explicitly which classes of system call they wish to use at any point, e.g., `stdio`, `route`, `inet`

AUTHENTICATION

- Protection
- Low-level Mechanisms
- OS Structures
- **Authentication**
  - **User to System, System to User**
  - **Mutual Suspicion**
- Access Matrix

AUTHENTICATING USER TO SYSTEM

Current practice: passwords

- But people pick badly
- And what about security of password file?
  - Restrict access to `login` programme (CAP, TITAN)
  - Store scrambled (Unix) using one-way function
- Often now prefer key-based systems (e.g., SSH)

E.g., in Unix:

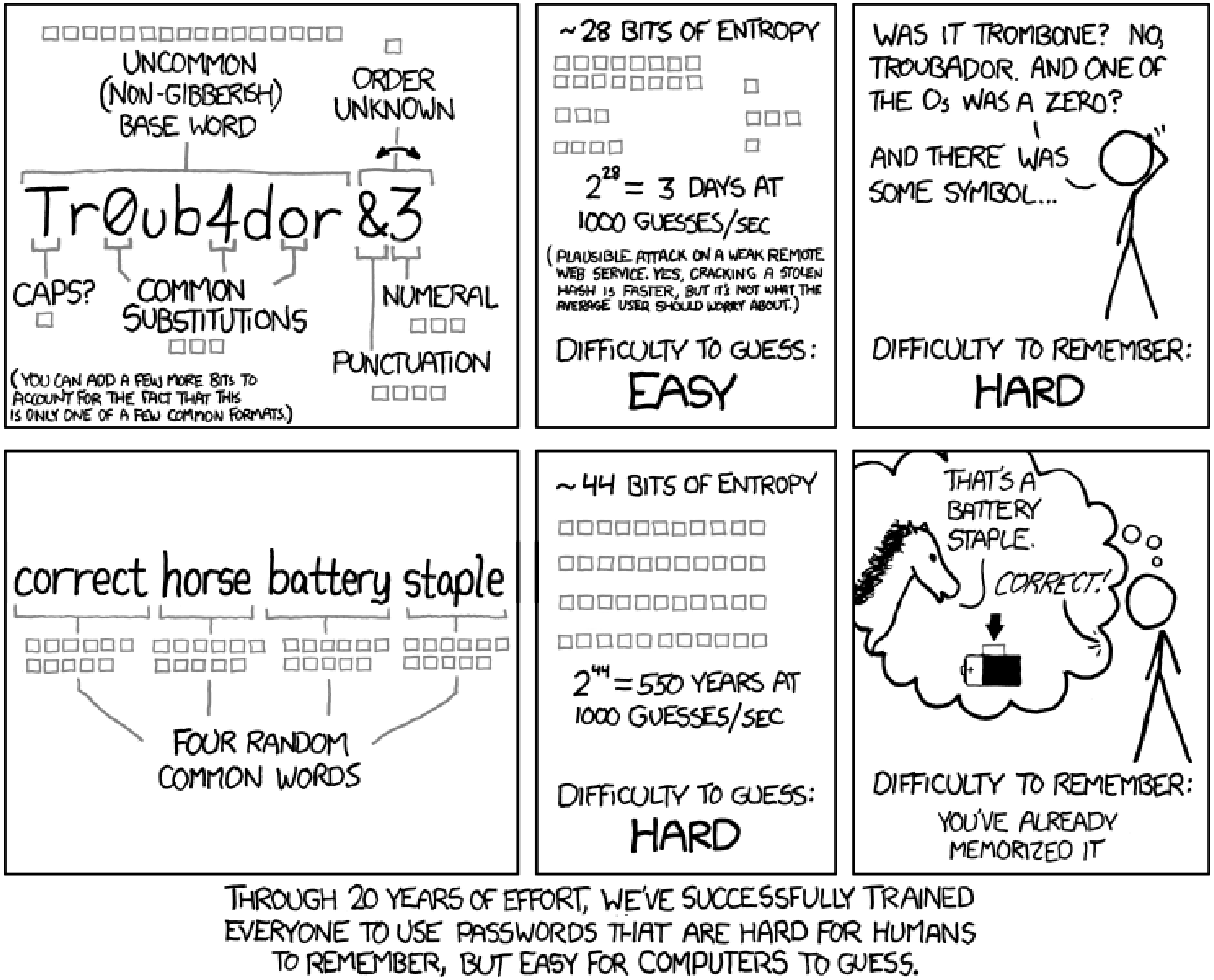
- Password is DES-encrypted 25 times using a 2-byte per-user "salt" to produce a 11 byte string
- Salt followed by these 11 bytes are then stored

Enhance with: biometrics, smart cards, etc.

- ...though most of these can be stolen

AUTHENTICATING USER TO SYSTEM

https://xkcd.com/936/



UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor&3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

AUTHENTICATION OF SYSTEM TO USER

Want to avoid user talking to:

- Wrong computer
- Right computer, but not the login program

Partial solution in old days for directly wired terminals:

- Make login character same as terminal attention, or
- Always do a terminal attention before trying login
- E.g., Windows NT's `Ctrl-Alt-Del` to login — no-one else can trap it

But, today micros used as terminals

- Local software may have been changed — so carry your own copy of the terminal program
- ...but hardware / firmware in public machine may also have been modified
- Wiretapping is easy

(When your bank phones, how do you know it's them?)

MUTUAL SUSPICION

Encourage lots and lots of suspicion:

- System of user
- Users of each other
- User of system

Also, called programs should be suspicious of caller

- E.g., OS calls always need to check parameters

And caller should be suspicious of called program

- E.g., Trojan horse: a "useful" looking program, a game perhaps, inherits user privileges when called
- Can then copy files, modify files, change password, send mail, etc...
- E.g. Multics editor trojan horse, copied files as well as edited.

ACCESS MATRIX

- Protection
- Low-level Mechanisms
- OS Structures
- Authentication
- **Access Matrix**
  - **Access Control Lists (ACLs) vs Capabilities**

ACCESS MATRIX

A matrix of **subjects** against **objects**.

Subject (or **principal**) might be:

- Users e.g. by UID, or sets thereof
- Executing process in a protection domain, or sets thereof

Objects are things like:

- Files, devices
- Domains, processes
- Message ports (in microkernels)

Matrix is large and sparse so don't store it all. Two common representations:

1. By *object* (**Access Control List**): store list of subjects and rights with each object
2. By *subject* (**Capabilities**): store list of objects and rights with each subject

ACCESS CONTROL LISTS

Often used in storage systems:

- System naming scheme provides for ACL to be inserted in naming path, e.g. files
- If ACLs stored on disk, check is made in software, so use only on low duty cycle
- For higher duty cycle must cache results of check
- E.g. Multics: open file is a memory "segment" (see later) — on first reference, causes a fault which raises an interrupt which allows OS to check against ACL

ACL is checked when file opened for read or write, or when code file is to be executed

In (e.g.) Unix, access control is by program, allowing arbitrary policies

CAPABILITIES

Associated with active subjects, so:

- Store in address space of subject
- Must make sure subject can't forge capabilities
- Easily accessible to hardware
- Can be used with high duty cycle e.g. as part of addressing hardware

Hardware capabilities:

- Have special machine instructions to modify (restrict) capabilities
- Support passing of capabilities on procedure (program) call

Software capabilities:

- Checked by encryption
- Nice for distributed systems

SUMMARY

- Protection
  - Motivation, Requirements, Subjects & Objects
  - Design of Protection Systems
  - Covert Channels
- Low-level Mechanisms
  - IO, Memory, CPU
- OS Structures
  - Dual-mode Operation, Kernels & Microkernels
  - Mandatory Access Control, `pledge(2)`
- Authentication
  - User to System, System to User
  - Mutual Suspicion
- Access Matrix
  - Access Control Lists (ACLs) vs Capabilities