CS 458 — Module 3

1 Outline

- Allows different entities to access different resources in a shared manner.
- The OS needs to control this sharing and provide an interface for allowing this access.
- Identification and authentication are required for access control.
- Focus on memory protection for now.

2 Protection in General Purpose Operating Systems

2.1 History

- Worth looking at how OSes have evolved until now.
- They now allow multiple users to use the same hardware, and added sequential abilities to run multiple *executives*.
- The OS should make resources available to users if required and permitted to by some policy.
- OSes also protect users from each other; attacks, mistakes, malware, and resource overconsumption are things to protect against, even if it is a single-user OS.

2.2 Protection Basics

- Some protected objects:
  - Memory
  - Data
  - CPU
  - Programs
  - I/O devices
  - Networks
  - The OS itself

2.3 Memory Protection

- We talk about memory protection first — a program should not be able to read the memory of another program unless permitted.
- This is important as a *lot* of things are stored in memory!
- One way to invoke security is separation — keep a user's objects separate from another's!
- We can do this:
  - Physically (easy but expensive and inefficient)
  - Temporally (execute different users' programs at different times)
  - Logically (users don't see other users)

  - Cryptographically (make users unable to see other user's data unencrypted)

- But sometimes, users do want to share resources.
- For example, library routines or files/DB records.
- OSes should allow for flexible sharing — but this creates new questions:
  - Which files to share? What part?
  - Which users?
  - Which users can share objects further?
  - What uses are permitted? R/W/E perms? Maybe only specific bits of information (ie: only aggregate info, not individual entries of a DB).
  - How long?
- Often, for memory protection, the OS can exploit hardware support (virtual memory from CS 350) for a cheap solution.
- Memory addresses used are virtual and the MMU will manage this and translate to physical addresses.
- the OS maintains the mapping tables the MMU will use and deals with raised exceptions from the MMU.
- What kinds of hardware support is used for memory protection?
- The simplest is a fence register — exception if the memory access below addresses is in the fence register.
  - This protects the OS from user programs.
  - This only works for a single-user OS.
- Another technique is base/bound register pairs.
  - This throws an exception if memory access is below/above an address in the base/bound register.
  - Each user program has different values.
  - When a ctx switch occurs, the OS is responsible for moving the current base/bond registers to the currently executing user program.
  - We can extend this to have 2 base-bound pairs, one for code, one for data.
  - This approach is more flexible.
- The tagged architecture takes this to the extreme; each memory word has one or more extra bits that identify access rights to the word.
  - This is very flexible, but it incurs a large overhead and is not portable at all.
- Segmentation is a common approach (remember CS 350?).
  - Each program has multiple address spaces, or segments.
  - We have different segments for code, data, stack, though this can be split up even further.
  - The virtual address contains of two parts — the segment name, and the offset within the segment.
  - The OS will keeps mappings from the segment name to its base physical address in the segment table; one is made for each process.
  - The OS can relocate/resize segments and share them between processes.
  - The segment table can also keep protection attributes.
  - However, because of the resizing/relocation capabilities, this means every access requires a bounds check.

- In paging, we divide the virtual and physical address spaces into pages and frames respectively. Frame size is equal to page size.
  - The virtual address consists of the page number and offset within page.
  - The number of bits for the offset is $\log_2(\text{page size})$.
  - The OS keeps the mapping from page number to its base physical address in the page table.
  - Similarly to segmentation, protection attributes are kept in the page table.
  - Typically used; advantages are:
    - Each address reference is checked for protection by hardware
    - Prevent users from accessing unpermitted pages
    - Less used pages can be moved to disk
    - Users can share access to a page
  - Disadvantages:
    - Internal fragmentation still is a problem
    - Assigning different levels of protection of different classes of data items is not feasible
- The x86 architecture supports both segmentation and paging, though paging is more common.
- Memory protection bits can indicate no access, R/W access, or RO access.
- Processors now also typically include a no execute bit, or NX bit. This forbids execution of instructions stored in a page, which would make the stack/heap non-executable.
- Note that this does not stop buffer overflow attacks; see module 2 (see return oriented programming, where we don't inject new code but change existing code to do what we want).

2.4 Access Control

- Access control has 3 goals:
  1. Check every access
  2. Enforce the least privilege
  3. Verify acceptable use
- To describe the access control matrix, we first define some sets:
  - Set of protected objects: O
  - Set of protected subjects: S
  - Set of rights: R

  The access control matrix consists of entries $a[s, o]$, where $s \in S$, $o \in O$, and $a[s, o] \subseteq R$.
- For example, we could make a table like:

	File 1	File 2	File 3
Alice	orw	rx	o
Bob	r	orx	
Carol			

- Often, access control matrices are not actually implemented as a matrix... the size and time would be awful to deal with.

- It's too sparse!
- Instead, typically they use access control lists, a set of capabilities, or some combination.
- An access control list, or ACL, is such that each object has a list of subjects and their access rights.
- We can quickly determine sets of allowed users per object, or revoking access to one specific object, but determining which objects a user can access or revoking a user's access to all objects is slow.
- For example, digital signatures.
- We can make tokens transferable.
- An example of combining usage of ACLs and capabilities:
  - In Unix, each file has an ACL, which is consulted when executing an `open()` call.
  - If approved, then the caller is given a capability listing type of access allowed in ACL.
  - Upon `read()` or `write()` call, the OS looks at capability to determine whether type of access is allowed.
  - This doesn't completely mediate; it does not check *every* access.
- Another way to do access control is role-based access control, or RBAC.
- Administrators assign users to roles and grant access rights to roles.
- This is similar to groups but groups are less flexible. A group is a set of subjects, while a role is as set of privileges to objects.
- When a user takes over a new role, they need to update only their role assignment, not all their access rights.
- Many commercial DBs do this.
- Also supports more complex access control scenarios:
  - Hierarchical roles — a manager is also an employee; this reduces the number of role/access rights assignments
  - Users can have multiple roles and assume/give up roles as required
  - Separation of duty — maybe a task needs two people of different roles that cannot be the same person

3 User Authentication

- Computers often have to identify and authenticate users before authorizing them.
- Difficult for computers to do for people both locally and remotely.
- Four classes of authentication factors:
  - Something the user knows
  - Something the user has
  - Something the user is (biometrics)
  - Something about the user's context (location, time, devices in proximity, etc.)
- Authentication factors may be combined for multi-factor authentication (ie: 2FA).
- Using multiple factors from the same class may not be a good idea; for example 2 passwords isn't really useful.
- Also, must make sure the factor cannot be changed — for example, a factor the user has can become a factor the user knows, like tokens that can be easily duplicated (mag strips) or SMS messages (ie: SIM-jacking).

3.1 Passwords

- A familiar and old authentication method.
- Enter an ID and password, potentially multiple attempts are allowed.
- Many usability problems:
  - Entering passwords may be inconvenient.
  - Password composition/change rules.
  - Forgotten passwords may not be recoverable; people tend to forget them.
  - If a password is shared then updating passwords is difficult.
- And security problems:
  - If the password is leaked, then an individual can immediately access the protected resource unless something like 2FA is used.
  - Shoulder surfing.
  - Keylogging.
  - Interface illusions/phishing.
  - Password re-use.
  - Password guessing (brute-forcing, for example).
- Some guessing attacks are exhaustive searching, dictionary attacks, etc.
- In online attacks, we can try to stop this by rate-limiting. This can still be defeated by social engineering mechanisms, though. For example, bypassing via secret questions/recovery methods.
- Password hygiene can be helped by things like password managers (though this now keeps all eggs in one basket), or passphrases that are easier to remember but still hard to guess.
- Don't reveal your passwords, don't use passwords on public computers, and don't re-use passwords.
- Advice for developers:
  - Don't use password composition rules.
  - At least 8 characters minimum length.
  - At least 64 characters max length.
  - Allow any characters.
  - Blacklist frequently used/compromised passwords.
  - Avoid password hints or secret questions.
  - Don't force users to change passwords.
  - Allow passwords to be copy-pasted.
  - Use 2FA (but avoid SMS-based).
- Attacks on password files — websites need to store passwords (or something) in a file to validate password inputs.
- Passwords in plaintext is a stupid idea as access to it from an intruder or… anyone really, could immediately screw over a lot of people.
- Cryptographic tools:

  - Hash — compute a fixed-length, one-way output. Deterministic.
  - MAC — takes a secret key as an input value; otherwise a hash. Deterministic.

- Now, store a hash of the password in the password file.
- Compare the *hash of the input* with the stored hash (fingerprint).
- But this would still allow offline guessing attacks…
- Add a user-specific *salt* to the password fingerprint.
- So two users with the same password have different fingerprints.
- This makes pre-computed tables useless.
- Using an iterated hash function which is slow to compute is better than using a standard hash function that would be fast to compute.
- This would not be noticed when a user inputs a password, but much more noticeable during a brute-force attack.
- Then, to make it better, throw in a MAC instead of a cryptographic hash.
- This adds a secret key required to compute the password fingerprint.
- This makes guessing attacks based on the fingerprints alone useless, but would require some kind of secret key generation/storage, using software/hardware.
- If the key does leak, then the scheme remains as secure as a scheme based on solely a cryptographic hash.
- How do we do password recovery?
- Well, that would require storing the password in some way…that's a no-go.
- So, use password-resetting mechanisms instead!
- Interception attacks — intercept the password as it is being sent to the server.
- OTP (one-time passwords) make this attack useless; for example fobs, auth. apps, challenge-response protocols.
- A challenge-response protocol is using a random challenge to the client and the completion of the client + password computes a one-time password.
- Also, passwords (or their hashes) are usually encrypted via TLS to protect from interception attacks.
- Though sometimes, they're still transmitted via PT!
- There are also cryptographic protocols, like SRP, that make intercepted info useless to an attacker.
- Unlock patterns are an alternative.
- May have an issue; fingerprints/smudges may reveal the combination!
- Another graphical password is image selection; this has some issues with randomness and shoulder-surfing.
- Systems should authenticate the user, but the user should also authenticate the server is real!
- Biometrics are hailed as a way to get rid of problems with password and token-based authentication.
- Problems:
  - Observed fingerprint just has to be *close enough*, otherwise it would be too hard to use reliably.

  - Could potentially be tricked if not done correctly (ie: face identification).
  - Not secret!
  - Cannot change!

- Authentication vs. identification — former is checking whether a captured trait corresponds to a particular stored trait, while the latter is if it corresponds to *any of the stored traits*.

- False positives can make biometrics-based identification useless (ie: Alice is accepted as Bob).

- Another potential example of the base rate fallacy — if the base rate is low and there are some false positives, it will seem like a huge failure.

- More issues with biometrics: privacy, accuracy.

4 Security policies and models

- Trusting something means that if that entity misbehaves, the security of the system fails.

- We trust an OS if we have confidence it provides security services (memory/file production, access control and user auth.).

- Typically a trusted OS will build on four factors:

  1. Policy (a set of rules outlining what is secured and why)
  2. Model (a model that implements the policy and that can be used for reasoning about the policy)
  3. Design (a spec. of how the OS implements the model)
  4. Trust (assurance that the OS is implemented according to design)

- We expect trusted software to do what it says it is going to do, and nothing more.

- We also expect:

  - Functional correctness — does it work correctly?
  - Enforcement of integrity — wrong input won't impact the correctness of data (test via fuzzing).
  - Limited privilege — access rights are minimized and not passed to others if not intended.
  - Appropriate confidence level — software has been rated as required by the environment.

- Trust can change over time.

4.1 Security Policies

- Security policies had their roots in military security policies.

- Each object/subject had a sensitivity/clearance level — for example, "Top Secret" $>_c$ "Secret" $>_c$ "Confidential" $>_c$ "Unclassified".

- Each object/subject might also be assigned to one or more compartments; this means that people would know only what they needed to know.

- A subject s could only access object o iff their level was greater than that of the object, and their allowed compartments were a superset (or equal) of the compartments that o used.

- If s dominates o, then we could denote it as $s \geq_{dom} o$.

- So if someone had clearance “Top Secret” and access to compartment “East Germany”, then they could not access a document that had the same clearance but a compartment of “East Germany” *and* “Soviet Union”!
- The integrity of information could be more important than confidentiality — Clark-Wilson Security Policy.
- Uses transactions; remember DBs!
- Another issue is potential conflicts of interests — Chinese Wall Security Policy; hard to go to the other side of the wall once you’re on one side.
- Once you have been given access to some specific information, you cannot access information of other categories in a similar manner, for example.
- Think of being able to access info about a company, and being restricted from accessing similar data from competing companies, or the like.
- The *ss-property* means that subject s can access object o iff each object previously accessed by s either belongs to the same company as o or a different kind of company than o.
- The **-property* means that for a write access to o from s, we also need to ensure that all objects readable by s either belong to the same company as o or have been sanitized.
  - The *-property prevents writing such that there is indirect information flow.
- Many security models have been defined with may properties; however their relevance to practically used security models may not be clear.
- We focus on looking at two prominent models:
  - Bell-La Padula Confidentiality
  - Biba Integrity
- These are targeted at multilevel security (MLS) policies, where subjects/objects have clearance/classification levels.

4.2 Lattices

- The dominance relationship $\geq_{dom}$ is transitive and anti-symmetric.
- It defines a partial order (you cannot have $a \geq b$ and $b \geq a$) (subscript omitted for brevity).
- In a *lattice*, for every a and b, there is a unique lowest upper bound u for which $u \geq a$ and $u \geq b$ and a unique greatest lower bound l for $a \geq l$ and $b \geq l$.
- There are also two elements U and L that dominate/are dominated by all levels.

4.3 BLP

- Regulates information flow in MLS policies, like lattice-based ones.
- Users should get info only according to their clearance!
- The underlying principle is information can only flow up.
- The ss-property is “no read up” — s should have read access to o only if $C(s) \geq C(o)$.
- The *-property is “no write down” — s should have write access to o only if $C(o) \geq C(s)$.
- Typically though, strict enforcement is too restrictive.

4.4 Biba

- To prevent inappropriate modification of data.
- Subjects/objects are ordered by an integrity classification scheme, $l(s)$ and $l(o)$.
- Write access: s can only modify o if $l(s) \geq l(o)$ (unreliable person cannot modify file with high integrity info).
- Read access: s can read o only if $l(o) \geq l(s)$ (unreliable information cannot contaminate subject).
- In practice, Biba's rules are *too* restrictive, as a subject can never read a lower integrity object.
- We can instead use a dynamic integrity level.
- The *Subject Low Watermark Property* is where if s reads o then $l(s) = min(l(s, l(o)))$.
- The *Object Low Watermark Property* is where if s modifies o, then $l(o) = min(l(s), l(o))$.
- The integrity of subject/object can only go down — that is, info flows down.

4.5 Review of BLP and Biba

- Very simple which makes proving properties about them easy.
- For example, we can prove that if a system starts secure, it remains secure.
- But they are *too* simple.
- For example, we need declassification, we need confidentiality *and* integrity, we need object creation.
- And information leaks can still be possible through covert channels!
- So what do?

4.6 Information Flow Control

- Describes authorized paths along which information can flow.
- For example, BLP describes a lattice-based information flow policy.
- For a compiler-based information flow control, a compiler would check whether the information flow in a program would violate the policy.
- How could information flow from a variable to another?
- It could be explicit (`y = x;`) or implicit (`if (x = 1) y = 0; else y = 1;`).
- The input parameters of a program could have a lattice-based security classification associated with them.
- The compiler would then go through the program and update the security classification of each variable depending on the individual statements that update the variable.
- Therefore, a security classification for each variable that is output by the program are computed.
- A user/program is allowed to see the output only if allowed by its security classification!

5 Trusted OS Design

- Eight design principles for security:
  - Least privilege: give the least privileges possible
  - Economy of mechanism: use simple and straightforward protection mechanisms
  - Open design: avoid security by obscurity (Kerckhoff's principle/Shannon's maxim)
  - Complete mediation: every access attempt must be checked
  - Permission based/fail-safe defaults: default should be denial of access
  - Separation of privileges: avoid a single point of failure; perhaps use two or more conditions to get access (ie: 2fa, dual-custody)
  - Least common mechanism: avoid one failure in a mechanism from cascading; don't share mechanisms if possible
  - Ease of use: if the protection mechanism is difficult to use, people will not want to use it correctly or at all
- Some security features of a trusted OS:
  - Access control:
    - Mandatory access control (central authority establishes permissions).
    - Discretionary access control (owners of an object have some control over who can access what).
  - Object reuse protection
    - Suppose memory is used to store a password.
    - If someone else uses the same memory, the OS should have erased that memory so that other users can't peek into the memory to find old, sensitive data.
    - This is also the case for data. For example, a hidden file, or data within a file, may actually still exist despite being deleted/hidden.
  - Trusted path
    - Give assurance to a user that their inputs are being sent to the correct receiver application.
    - This is difficult.
  - Accountability and auditing
    - Keep an audit log of security-related events.
    - This provides accountability if something goes wrong.
    - Something something Murphy.
    - The audit log does not give accountability though if an attacker can *modify* said log.
    - The granularity of event logging may be important — if it is too fine-grained then finding an attack or storing it may become a problem. But if it is too coarse, then we might miss the attack entirely!
  - Intrusion detection
    - Alarm if behaviour looks suspicious.

5.1 Trusted Computing Base (TCB)

- The TCB consists of the part of a trusted OS that is necessary to enforce OS security policy.
- The security kernel runs below the operating system level; this makes it harder to attack.
- In the ring structure, it is between OS and hardware.
- It should be minimal and only have core functionality.

- Many OSes don't use the 4 rings of an x86 architecture (Linux and Windows only use user and supervisor mode).
- The reference monitor is a collection of access controls for devices, files, memory, etc.
  - Important part of TCB.
  - Must be tamperproof, un-bypassable, and analyzable.

5.2 Virtualization

- A way to provide logical separation/isolation.
- Virtual memory, for example, uses a page mapping to give processes the impression of having separate memory spaces.
- We can go further with virtual *machines*. These virtualize, well, a machine, with virtual I/O devices, files, etc.
- This means that attacks would be limited to the virtual environment.

5.3 Application Insulation

- A similar approach; shield applications from other apps.
- The application is partitioned into trusted and untrusted code.
- The former is encrypted in memory with the key in secure hardware, while the latter talks to trusted code via an API.
- Examples are Intel SGX and AMD memory encryption.

5.4 Security in Popular OSes

- Windows has UAC for temporary access rights, and integrity levels.
- In UNIX, root has full access to anything.
- SELinux/AppArmor provide MAC to Linux.
- Other approaches like Chroot, compartmentalization, SUID also exist for UNIX.
- And smartphones usually restrict apps in terms of what they can access.
- `chroot` sandboxes/jails a command by changing its root directory.
  - Thus, the process cannot access files outside of the jail.
  - This is not perfect though, there are ways to break out of it and some commands/programs get difficult to use in a jail.
- Containers can run processes in a set of namespaces (process IDs, user IDs, filesystem mounts, etc.), which means that the process cannot see many things from the client machine. Examples are Docker.
  - This also means that a program can get elevated privileges *only* within the container, and not the actual client machine.
- Compartmentalization splits applications into parts and apply the least privilege on each part.
  - For example, OpenSSH splits SSH into a privileged monitor and a jailed, unprivileged child.

  - The child must contact the monitor to get protected information.
  - The child will be shut down by the monitor if it finds the child suspicious.

- setuid/suid bit
  - UNIX ACLs contain a suid bit.
  - If it is set, then the executable will run under the identity of its owner, *not* the caller.
  - So, `passwd` belongs to root and has suid bit set.
  - This means the program will run as root.
  - There is a possibility of a "confused deputy" attack, where an attacker can fool the program into thinking they are another user which could potentially cause some harm (ie: passwd changing password for another user than the one calling it).

5.5 Assurance

- How to convince people to trust our software?
- Testing and formal verification are two aforementioned ways.
- Validation is another.
- These are things that a trusted entity can evaluate for software, certifying that it satisfies some criteria.
- Two well-known setso f criteria are the Orange Book (US) and the Common Criteria.
- The OB lists several ratings based on evaluation and security.
- Ranges from D to A1.
- For example, Windows NT has C2, UNIXes are usually C1.
- The international version of this is the Common Criteria. Products are rated against profiles, where there is EAL 1 (worst) to EAL 7 (best).
- RHEL and Windows 7 have EAL 4+, for example.