layout: post title: "Security Design Principles" date: 2019-09-01 categories: [security] tags: [summary, design]

About

This post will summarize security design principles from three papers (see references).

Defining Initial Security Architecture

Security architecure should be synthesized from security requirements. A *good* architecture is one that abstractly describes the relationship between key elements of the system in a way that satisfies the security requirements. It is also distinct from the functional specs of the system and recommended to be a ***design overview***.

Good security architecture is not all technical, an important feature is establishing development standards to adhere to the policies set in place by the architecture.

A high-level security architecture should be in place as early as possible in the software project. An example draft can be high level and ***evolve*** over time. The following points could make up a draft for starters:

- System security policies.
- How much assurance is desired.
- How said security requirements impact the development process.

Evolving Security Requirements

It's ideal to have the security architecture specified early on, but how do we anticipate or prepare for future security requirements? For one, ***don't be overly specific about anticipating security enhancements*** (sic). E.g.:

- Maybe we've added hooks to handle security checks, but such checks may introduce new possibilities of failures that the existing software can't handle. I.e., ***new kind of failures***.
- A classic example of such failure is a mechanism that ***knows*** a file exists but can't read it.

Gasser claims that the **keys** to incorporating security hooks for future security enhancements are:

- Understanding computer security requirements in general.
- Include such requirements ***as explicitly as possible*** as future security needs.
- Work out the details of how to handle such future requirements.

Stunting System Security Evolution

Evolving security requirements for a system could be impossible in some cases. For example, what if we have a system feature that allows anonymous, shared-access to a directory for temporary files.

From a feature standpoint, it sounds reasonable and efficient use of resources, however it is also ***fundamentally insecure*** since multiple users may read other users files in said directory.

Economics of Security

How does one determine how much to spend on security? There's a few variables to consider to determine this:

- How much the defender values asset *x*.
- How much the attacker values asset *x*.
- How much the defender spends to protect *x*.
- How much the attacker spends to protect *x*.

Work Factor

> Compare the cost of circumventing the mechanism with the resources of the potential attacker.

This *cost* is known as the work factor, and can easily be calculated in some cases. For example, if our passwords can only be two alphabetic characters, that is `26^2` possible variations. An attacker with only a terminal and keyboard as resources has a *much higher work factor*, than an attacker with a computer capable of generating and automatically entering millions of passwords per second.

Some troubles with this principle causing unreliable or difficult cost estimates:

- Many protection mechanisms are **not** conducive to work factor calculations.
- Defeating such systems systematically may be *logically impossible*.
- Defeat is accomplished *indirectly*, such as waiting for hardware faults, or exploiting implementation errors.

Operating Region

Given the variables above, the ideal operating region for the defender is such:

> The cost to defend asset *x* is less than how much *x* is worth and the cost to attack is greater than how much the attack values asset *x*.

I.e., security cost **must be** commensurate with threat level **and** asset level.

Inherent Design Challenges

Negative Requirements and Preventing Unauthorized Actions

In practice, building a system that prevents *all unauthorized* acts is extremely difficult. The exception being *level-one* systems, which are essentially completely unprotected systems. There's on-going research in systematically excluding security flaws from system implementations, but no general solution is available. It's inherently difficult to achieve because preventing **all** unauthorized actions is a **negative requirement**.

Principles in Absence of Methodical Techniques

Since there's no answer to systematically preventing unauthorized access, here are some design principles recommended to mitigate it.

Fail-safe Defaults

> The default situation is lack of access, and the protection scheme identifies conditions under which access is permitted.

> A default of lack of permission is safer.

I.e., strive to base access decisions on permissions rather than exclusion. It is better to ask why an object ***should be*** accessible, rather than why they ***shouldn't*** be. This is especially needed in large systems where it's difficult to manage access for every single resource.

An instance where this principal of ***requiring permission rather than exclusion*** is a system with a faulty access-exclusion mechanism that could fail by ***allowing access***.

Complete Mediation

> All object accesses must be checked for authority.

This forces a system-wide view of access control, including system:

- Initialization
- Recovery
- Shutdown
- Maintenance

This principal implies:

- The method of identifying the ***source*** of each request must be foolproof.
- Performance enhancements by remembering the result of a previous security check must be examined skeptically. If the authority changes, the remembered results could be stale.

Economy of Mechanisms

To achieve higher assurance in the architecture security, design the security-relevant modules such that ***size and complexity*** is minimized.

Gasser says that even if we stuck to this design principal, there may be diminishing returns of adding new security mechanisms. I.e., if the effort, size, and or complexity required to implement the new security feature requires ***as much*** mechanisms already in place, the assurance and reliability is likely to not be commensurate.

The goal to combat this is ***economy of mechanism***, which is minimizing the variety security mechanisms. This forces security-relevant actions to be taken in a ***few*** isolated sections.

Complexity versus Security

Observe the two extreme systems, A and B. A is all features, minimal security, and B is all security while sacrificing most usability. We want to build a system closer to C in most cases.

Feasibility

It largely depends on how flexible the system was designed. Historically, economy of mechanism is hard to attain due to how much security mechanisms ***permeate*** the system. E.g., take the mechanism of ***access of control***:

- DBMS worries about access at the row level.
- Message-handling systems worry about access at the message level.
- Document-processing worries about whole file access.

How would one isolate this mechanism when it's sprinkled throughout the system?

Going Overboard

It's possible to take this principal to the extreme. For example, historically it made sense to store security attributes for files alongside other file attributes. If we instead ***isolated*** these security attributes in a database, we'd potentially degrade security by adding on a synchronization mechanism.

Implementation

There are a few design ideas for economy of mechanism, however they are not infallible (more on this in the following sections):

- Open design- the more users see and vet your security architecture design, the more suggestions and patches can be made. However not in all cases, e.g. Heartbleed.
- Less code, less complexity- it's recommended that the TCB belongs to the hypervisor (~50 KLOC) versus the operating system (~50 MLOC).

Separation of Privilege

> Where feasible, a protection mechanism requiring two keys to unlock it is more robust and flexible than one that allows access with just one key.

Once the mechanism is locked, no ***single accident or breach*** can access the protected mechanism since the keys are separated among different entities.

This principle is already used effectively in physical systems, e.g.:

- Bank safe deposits, where there is a separate key into the vault, and another key to get into the deposit box.
- Defense systems, where two or more people need to give the correct command to fire a nuclear weapon.

Least Privilege

> Each entity of the system should operate using the least set of privileges needed to complete the job. In the military, this is known as need-to-know basis.

This has practical benefits:

- Reduces potential interactions among privileged programs to the minimum for correct operation.
- Reduces unintentional, unwanted, or improper uses of privilege.
- Minimizes the number of programs to audit when someone asks how a privilege was misused.

- Provides a rational of where to install firewalls should a mechanism provides one.

> Subjects should have just enough privilege to do their jobs.

I.e., prefer to **fail-safe** by default rather than allow potentially insecure access.

The idea is very familiar- the valet can use the car keys since they need to park the car, but the car washer does *not* need the car keys since they only wash it.

Least privilege is used in computing, one example being kernel mode versus user mode. This example exhibits *coarse-grained* privilege and suffers from an all-or-nothing state of possible privileges. Throwing in more privileges gives a more robust hierarchy of possible privileges to assign, but also requires more complex hardware or software support.

An example of least privilege used in software system goes as follows: a file back-up daemon may have ***read*** privileges on the files, but it shouldn't need ***write*** privileges on said files to perform its job successfully.

Least Common Mechanism

> Avoid having multiple subjects sharing mechanisms to grant access to a resource.

E.g., consider we share our application on the internet and there are two known subjects:

1. Valid users
2. Hackers

Our only mechanism for access is one- simply having the sight open and accessible to the Internet. Now sensitive info can be used by real users and attackers to gain application access.

If we have a different mechanism for each type of subject or class of subjects we have more access control flexibility and prevent security violations that'd occur if we only have one mechanism.

Defense in Depth

I.e., having multiple layers of security. This can be demonstrated with the following thought exercise, referencing the reflections on trusting trust paper.

Lets say we're detecting potential compiler bugs or trojans. We'll accept the following assumptions as valid:

> Two distinct binaries compiling the ***same source*** do not need to be binary equivalent, but should be functionally equivalent.

We have two compilers from different vendors, A and B. Assume a trojan was planted in A. There's corresponding executable for A and B as well.

1. Compile the source code of A with A's executable
2. Compile the source code of A with B's executable.
3. Repeat this process once more, which produces two more executables.
4. If these new executables don't match, we have a bug (or trojan) on our hands.

Defense in depth principal emphasizes that *functionally equivalent* programs should produce the same output.

Open Design

> The safest assumption is that the penetrator knows everything.

Secrecy of design is not a requirement for even the most highly secure systems. No system will ever be free of covert channels, whether the design is privatized or open. Disclosing the design of the system's security mechanism **can improve security** because its internals are scrutinized by a much larger audience.

Favor hiding and protecting a small set of keys and passwords over keeping the system security design a secret. By doing this, we *separate mechanism from protection keys*, allowing the mechanisms to be scrutinized and improved by the community.

User Acceptability

User or psychological acceptability design principle. Users are usually the weakest link when measuring system security.

Gasser recommends we keep these goals in mind as well:

- Security shouldn't affect users obeying the rules.
- It should be easy for users to *give* access.
- It should be easy for users to *restrict* access.

To routinely apply the security mechanisms correctly, the interface must be simple to use. If the user's *mental image* of the protection goals matches the required mechanisms, mistakes will be minimized.

If the security mechanism is very difficult or awkward to have a *mental image* or abstraction of, it's easier for the user to misuse the mechanism, leading to potential security violations or attacks.

Compromise Recording (not recommended)

Some arguments say that *recording* a potential attack could be as strong as the mechanism preventing the attack. E.g., some systems log the date and entity that last accessed a record. If such log was tamper-proof, it could provide valuable data on the potential attacker. The reason this is *imperfect* is that it's difficult to guarantee discovery once the security has been broken, e.g. attacker can undo the log.

References

- Seltzer paper
- Gasser paper
- Examples of these principles in use
- us-cert least common mechanism