Secure Coding Typed Notes

Working class notes for CNT 4419: Secure Coding the the University of South Florida.

Trevor Flahardy

目录

Part 1: Mechanisms

Types (categories) of mechanisms:

- Prevent
- Detect
- Contain
- Recover

The professor notes that these categories are not “crystal clear”; they can be subjective depending on the context. So, for example, a **dynamic type checker** would fall as one of these things. It’s either preventative or detective, depending on how you look at it (eager or lazily checking types).

1.1. Preventative Mechanisms

Done before something bad has happened. Examples of this are such as a firewall (FW) and passwords.

A static type checker would be a preventative mechanism (an eager one that checks types at compile time).

1.2. Detective Mechanisms

Detective mechanisms are **done after something bad has happened**.

An example of this would be performing audits and monitoring. This is done to detect if something bad has happened (so taking logs and analyzing them).

Some aspects of anti-virus software are detective mechanisms, such as the signature-based detection. This is because it detects if a virus is present on the system.

1.3. Containment Mechanisms

The idea here is that: let’s assume an attack has been successful and we want to contain it such that it has the smallest effect possible on our system. So how we do that?

The trick is **replication**, so we can kind of “forget about” one machine and continue with our operations on another machine because we have replicated the capabilities of the first machine on the second machine. So if the first machine gets compromised, we can just “throw it away” and continue with the second machine.

This is the big idea: **replication** with containment mechanisms.

The main idea here is that you are trying to **isolate** a problem. Ideally, the machine that has been compromised is isolated and **simply can be shut off**. Additionally, instead of powering the system off you can isolate it.

1.4. Recovery Mechanisms

Revert to backup. This may be backup data, or a backup system(s) or machine(s); but somehow you are going to revert to a known good state.

So a common way to recover something is to cut and restore power (turn off and back on again); using backed up data on another machine, etc.

When you look at this for devastating attacks (when the attacker has been very successful), performing a restart may not be enough. So you may have to re-flash things or switch machines entirely.

The professor notes that another way to recover is to "file an insurance claim".

Aside Note

So how we do we know if some software is malicious (aka malware)? It turns out that attackers have a lot of sophisticated tricks too make their software look "legit" (or safe). So once some malware is discovered and people want to start preventing it, malware writers circumvent this by adapting how the malware looks or behaves. This is known as **polymorphic**. So how we do we know if some software is malicious (aka malware)? It turns out that attackers have a lot of sophisticated tricks too make their software look "legit" (or safe). So once some malware is discovered and people want to start preventing it, malware writers circumvent this by adapting how the malware looks or behaves. This is known as **polymorphic**.

🧠 **Definition**

Polymorphic malware: Malware that can take many shapes or forms.

- There are also some interesting contexts in which we can associate these goals with "medical" terminology:
  - Preventing some disease
  - Detecting some disease
  - Containing some disease (someone who has a disease)
  - Recovering from some disease (getting better from a disease)

Often people like to make these analogies between disease and medical contexts and security contexts. So for example, we can think of a virus as a disease, and then we can think of anti-virus software as a way to prevent or detect that disease. We can also think of a firewall as a way to prevent unauthorized access to our system, similar to how we might use a vaccine to prevent a disease.

❓ **Question**

But why is that a difficult analogy to make?

As you get into the deep aspects of computer security, measuring "system health" and similar metrics becomes very difficult. How do you know a security mechanism is good? Or that it is preventing the attacks we want it to prevent?

So associate this idea to medicine. Say 1,000 people have some new disease and you give some new drug to all of them and everyone is cured. This would be a resounding success and the disease is resolved. Switch over to computer security and take 1,000 machines with some malware on it. Some mechanism has been put in place, and the mechanism detects or cleans up after all 1,000 instances of that attack. Can we claim that we have "solved" this attack? Maybe this specific attack, **but what about polymorphic versions of it?** So a very intelligent adversary may change their program very quickly, or automatically, to go around your mechanism. So as fast as you can change a mechanism is as fast as an attacker can change it.

1.4.1. Keeping "Good" Backups

Ransomware: is an example of an attack that is devastating and can be prevented by having good backups. So the idea is that you have some malware that encrypts your data and then demands a ransom to give you the decryption key.

- The defense against them are replication, revering to your backup, getting a new machine, reverting to factory settings, etc. So the idea is that you have a backup of your data and you can restore it if something bad happens.
- There is often a *trust among thieves*: If you pay the ransom, the attacker may give you the decryption key. However, there is no guarantee that they will actually do so, and they may even demand more money after you pay the initial ransom. So it's a risky situation to be in. This is the attacker's "business" model, and it's within the attacker's best interest to do what they say (providing the key upon payment) to continue to make money from other victims.

There exists a more modern variation on ransomware, though. The ransomware may not just encrypt your data ("scramble" your data) - now often the ransomware will **leak your data**. This is called **exfiltrate** your data or sensitive information.

- Commonly, all the data on the machine is exfiltrated (including PII information).

> 🧠 **Definition**
> **Exfiltration**: The unauthorized transfer of data from a computer or network. This is often done by attackers to steal sensitive information, such as personal identifiable information (PII), financial data, or intellectual property.

> 🧠 **Definition**
> **Personal Identifiable Information (PII)**. This is information that can be used to identify an individual, such as their name, address, social security number, etc. This is often the type of data that is exfiltrated in a ransomware attack.

Part 2: Secure Software Design Principles

2.1. 1. Sanitize Inputs

2.2. 2. Try to handle errors securely

2.3. 3. Use layers of heterogeneous mechanisms

2.4. 4. Adhere to Principle of Least Privilege (PoLP)

2.5. 5. "Avoid Security by Obscurity"

This is the idea that you should not rely on secrecy of your design or implementation as the main method of providing security. So for example, if you have some software that has a vulnerability, and you try to hide that vulnerability by not disclosing it, this is not a good security practice. This is because attackers can still find the vulnerability through other means, such as reverse engineering or fuzzing.

Typically, **we want to try and encapsulate any secrets being relied on into cryptographic keys**.

2.6. 6. Be careful about when and where/how keys are stored

- So we ant to isolate these secure things into "keys", and then we want to be really careful about where we store these keys. We want to make sure that they are not stored in a way that is easily accessible to attackers, such as in plain text on a hard drive or in a configuration file.
- Most hardware, now, has **special hardware that helps us protect these keys**. Processors or modules are specifically created for this purpose
  - Called the **TTM Trusted Platform Module** (at least, on Windows)
  - The OS often has APIs for using this part of the machine. Your code can call the APIs and make use of this hardware.

So making these truly random keys is very difficult. One very popular one is the Cloudflare lava lamp room.

This is a room full of lava lamps that are randomly moving around, and the patterns of the lava lamps are used to generate random numbers. This is a very creative way to generate random numbers, and it is also very secure because it is based on physical randomness.

Historically, it's very difficult to write your own "random number generator" – the professor recommends using a well-known library for this.

🧠 **DEFINITION**

Cryptographically Secure Pseudo Random Number Generator (CSPRNG): A CSPRNG is a type of random number generator that is designed to be secure against certain types of attacks. It is often used in cryptographic applications to generate keys, nonces, and other random values that need to be unpredictable and resistant to reverse engineering.

Notes hereon taken on Mon Feb 23, 2026

Professor started with summary of last class session and what we discussed.

- Principle of least privilege (PoLP)
- Avoid security by obscurity
- Be careful about when and where/how keys are stored
  - IE avoid hardcoding keys in your code, and instead use a secure key management system.
  - Prof notes that it's hard to clean version history (git, commit messages, etc) and how careful you have to be with secrets in general.
  - Examples of leaking secrets could be: leaked DB name/password, committing ENV variables with secrets in them, **hardcoding secrets in your code** (which is a big no-no), etc.
  - Prof discussed automated crawlers that search for these kinds of secrets in public repositories (like GitHub) and how they can be used by attackers to find vulnerabilities.

❓ **QUESTION**

So how do you manage secrets in your code?

You put these keys into trusted hardware called the TPM (Trusted Platform Module) or some secured area of memory, or an encrypted file. But, you want to store it somewhere NOT in plain text of the secret.

This *could* be in memory but you, at least, would want to encrypt it.

When you are using the keys, **minimize their duration of use so they are not in memory for a long time**. This ties back into "memory remanence" (see below), where often developers will overwrite their keys (sometimes **multiple times!**) to try and prevent this issue. However, this is not a perfect solution, and there are still ways for attackers to recover the keys from memory even after they have been overwritten.

🧠 **Definition**

Memory Remanence

The phenomenon where data that has been stored in memory can still be accessed after it has been deleted or overwritten. This is because the data may still exist in the physical memory cells, even if it is no longer accessible through normal means. This can be a security risk if sensitive information is stored in memory and not properly cleared.

The professor notes that the above in this section are things to be thinking about as a "Secure Coder".

❓ **Question**

So how effective are `gitignore` files?

The professor does not know this, but qualifies it as a good question.

In actuality, though: `gitignore` files are not a security mechanism. They are a convenience mechanism for developers to avoid accidentally committing certain files to the repository. However, they do not prevent someone from manually adding those files to the repository or from accessing them through other means. So while they can be helpful in preventing accidental commits, they should not be relied upon as a security measure.

The professor is going to continue secure software design principles now…

2.7. 7. Simplicity (The "keep it simple" idea)

So, like, if you can keep your designs simpler then they are easier to think about and easier to reason about in terms of "potential attacks". So more complex code is harder to analyze (by automated tools and developers).

Ideally, **you keep your code and its design as simple as you can**.

- **Design**: You want to keep your design as simple as possible. This is because complex designs are more likely to have vulnerabilities, and vulnerabilities can lead to attacks. Additionally, complex designs are harder to maintain and understand, which can lead to more mistakes being made by developers.
- **Implementation**: You want to keep your code as simple as possible. This is because complex code is more likely to have bugs, and bugs can lead to vulnerabilities. Additionally, complex code is harder to maintain and understand, which can lead to more mistakes being made by developers.
- **User involvement (or interaction)**: Users may make mistakes (or are more likely to make mistakes).

  - The professor notes here that you want to minimize as much user involvement as possible, because users can make mistakes that lead to security vulnerabilities.
  - This ties into **secure defaults** (see below), where you want to have secure defaults that do not require users to make decisions that could lead to vulnerabilities.
- **Default Settings**: You want to have secure defaults that do not require users to make decisions that could lead to vulnerabilities. For example, you should not have a default password that is the same for all users, because this is a major security vulnerability in practice. Instead, you should require users to create their own unique passwords, or use some other secure authentication mechanism.

2.8. Secure the "weakest link"

In many cases, this is the users (or the professor notes that "we" are the weakest links).

So, what does this mean? What kinds of mistakes do we make? This ties into **user involvement** from the previous principle. In short, EVERYTHING. If we let users choose their password arbitrarily, for example, they choose weak passwords. So, in this example, you want to do some basic checks on passwords and ensure some arbitrary strength value.

So this is a clear "usability" vs "security" tradeoff. If you make your security mechanisms too strict, users will find ways to circumvent them (like writing down their passwords on a sticky note). If you make them too lenient, users will choose weak passwords that are easy to guess.

Formally,

- **Passwords** (see above)
- **Insider attacks**
- **Social engineering**
  - Ties into spearfishing (high likelihood of success)
  - IE "flagging emails" as part of an organization or not allowing users to open attached files right away, etc.

2.9. Understand the limits of our security mechanisms

We can't just run one security mechanism and it solves all our problems, of course. This ties into the "Defense in Depth" idea.

For example, recall the textbook example of the Java app vulnerabilities. That example was using HTTP connections, but what if we modified the code to use HTTPS (HTTP secure) instead? If you go through your notes and look at the vulnerabilities, you will see that none of them are mitigated by using HTTPS instead of HTTP. **Adding HTTPS to the simple webserver mitigates none of the attacks (the 4 attacks total)**.

2.10. Make judicious use of existing/AI-generated code or algorithms

Don't reinvent the wheel! In practice, usually, you can't do everything yourself. So if you are building a "real system" you are using someone else's code there. it's important to think through what you are using and if that code you are using has vulneraviliies.

On Stack Overflow for example, there are many code snippets that are not secure. Sure, you can trust answers that have more "up-votes", but you should take this with a grain of salt. The professor notes a paper that came out last year that tested copilot. They asked it to generate some security relevant code and

40% of the modules that copilot created had security vulnerabilities in it. So the TLDR here is that you have to be careful about using AI generated code as well.

You, and other people, should be reviewing the code you are producing.

2.11. Bonus: Time to Check to Time to Use (TOCTOU) Vulnerabilities

So let's say that the user wants to open some file. We do a check to see igf they have permissions to do this, but what do we do about the "potential gap". "Gap" here being the time between when we check the permissions and when we actually open the file. This is known as a **Time of Check to Time of Use (TOCTOU)** vulnerability.

🧠 **Definition**

Security State

The state of the system in terms of its security. This can include things like whether the system is currently under attack, whether it has been compromised, etc. The idea is that there is a "gap" between when we check the security state and when we actually use that information, and **this gap can be exploited by attackers to bypass security mechanisms**.

You want to minimize the gap between when you check the security state and when you use that information. This can be done by using **atomic operations**, or by **using locks** to prevent other processes from modifying the security state while you are using it. You don't want your code to get "sidetracked" on other things.

🧠 **Definition**

Atomic Operations

Atomic operations are operations that are indivisible, meaning that they either happen completely or not at all. This is important for preventing TOCTOU vulnerabilities because it ensures that the security state cannot be modified by another process while you are using it. For example, if you are checking permissions for a file, you want to make sure that the check and the use of that information (opening the file) happen in an atomic operation so that there is no gap for an attacker to exploit.

🧠 **Definition**

TOTP: Time-based One-Time Password

TOTP is a type of two-factor authentication (2FA) that generates a one-time password based on the current time. This is often used in conjunction with a username and password to provide an additional layer of security. The idea is that even if an attacker manages to steal your username and password, they would still need access to the TOTP code, which is generated on your device and changes every 30 seconds or so, making it much harder for attackers to gain unauthorized access to your accounts.

🧠 **Definition**

Complete Mediation

Complete mediation is the principle that **permissions should be checked every time a resource is accessed**, rather than just once upfront. The core motivation is preventing TOCTOU vulnerabilities — if you only check once and assume permissions won't change, **an attacker can exploit the window** between that check and the actual use of the resource.

This is especially important for dynamic mechanisms, where the security state can change over time. **Your mechanism should fully mediate** — that is, "interpose" and run security checks — before every sensitive operation.

There should be no holes or gaps where a security-sensitive operation happens without a preceding check. You're not just checking at initialization; you're stepping in every single time.

The key principle: check all operations, every time, without exception.

Part 3: Securing Software Strategies

The professor notes that all the above security design principles are not exhaustive but cover all the ones he can think of. Next, he wishes to focus on "how are we going to protect certain resources against attacks?"

We are moving towards more and more concrete examples of attacks on SW (software). We talked about the compiler with the confused deuputy attack.

? Question

What kinds of resources do attackers like to go after? What kinds of standard resources on a machine would the attacker have a chance of hitting?

- Processor time is a good example. This would be a late stage in an attack; you can do a lot of things with processor time, such as creating botnets and mining cryptocurrency. So this is a very valuable resource for attackers.
- Memory usage. Memory is usually one of the first things an attacker can get. This is a super important resource for attackers. So many of these atatcks below are something that attackers will go for.
  - The professor notes we have to understand memory very well for this.

3.1. Access Control and Access Control Mechanisms

3.1.1. Authentication

Access control mechanisms try to enforce **access control (AC)** policies (pols). There are two subset controls to access control:

3.1.1.1. Authenticate Users

This is where you have users "establish their identities".

Determine subjects identities. This is often done through some form of authentication mechanism, such as a username and password, biometric authentication, or multi-factor authentication (MFA). The goal of authentication is to ensure that only authorized users can access sensitive resources, and to prevent unauthorized access that could lead to security breaches or data leaks.

3.1.2. Authorization

This is where you determine if a subject (user) has permission to access a resource. Ie, this is where you authorize the access from the user for a resource.

Formally, determine whether (or how) an identified subject may access an object (resource).

Expanded, authorization is the process of determining whether a user has permission to access a resource. This is often done by checking the user's permissions against an access control list (ACL) or by using some other form of access control mechanism. The goal of authorization is to ensure that only authorized users can access sensitive resources, and to prevent unauthorized access that could lead to security breaches or data leaks.

Authentication happens first, and then authorization happens after that. So you first need to establish the identity of the user (authentication) before you can determine what they are allowed to do (authorization).

3.2. Authentication Factors

We are going to start comparing these two things (authentication and authorization) to understand them better.

IE, like how hard is it to create a new password compared to obtaining a new fingerprint? Or face shape? The point is you can't, right, so this is a good example of how authentication can be more or less difficult depending on the method used. For example, using a password is generally easier to implement and use than biometric authentication, but it is also less secure because passwords can be easily guessed or stolen. On the other hand, biometric authentication is more secure because it relies on unique physical characteristics, but it can be more difficult to implement and use.

These are all based on **three factors**:

1. What you **know**: Such as a password, a pin number, etc
2. What you **have**: Such as a smart card, physical key, credit card, phones, etc
3. What you **are**: Such as a fingerprint, a face shape, voice etc (⟵ this relates to biometrics)

Some people like to say that a fourth factor exists: **location**, but the professor argues that the location factor fits into the "what you are" category, because it is based on something about you (your location) that can be used for authentication.

The professor went on to discuss **multi-factor authentication** (**MFA**), which is the use of two or more authentication factors to increase security. For example, using a password (what you know) and a fingerprint (what you are) together would be an example of MFA. The idea is that even if one factor is compromised (like a password being stolen), the other factor (like a fingerprint) would still provide protection against unauthorized access. The main common types are:

- **1-factor** (or **single-factor**): This is where you only use one authentication factor, such as a password. This is generally considered less secure because if that one factor is compromised, there is no additional layer of security to protect against unauthorized access.

- **2-factor**: This is where you use two authentication factors, such as a password and a fingerprint. This is generally considered more secure than single-factor authentication because even if one factor is compromised, the other factor would still provide protection against unauthorized access.

> ❓ **Question**
> **How would you classify an email code?**
>
> It really depends. Some email services allow you to login on any device, so in that case, the email code would be considered a "what you know" because it is something you have access to (your email account). However, if the email service is tied to a specific device (like a phone), then it could be considered a "what you have" because it is something you physically possess (the phone). So it really depends on the context and how the email code is being used for authentication.

3.3. Authorization Factors

So now we're talking about checking permissions. So there has to be some **data structure** that knows if some user has permission to access some resource. What are these data structures that the mechanism can use to remember permissions?

- **Access Control Lists (ACLs)**: This is a list of permissions attached to an object (resource) that specifies which users or groups have access to that object and what kind of access they have (read, write, execute, etc). So for example, a file on a computer might have an ACL that specifies which users can read the file, which users can write to the file, and which users can execute the file.
  - Example: some `root` may have read, write, execute permissions (RWE), some `user5` may have read and write permissions (RW), but all other users have nothing ($\emptyset$).
- **Capabilities**: Converse to ACLs, capabilities are attached to the subject (user) rather than the object (resource). So a capability is a token or key that grants a user permission to access a resource. For example, a user might have a capability that allows them to read a file, but not write to it. The user would then need to present that capability in order to access the file.
  - Every user is **assigned permissions** (or capabilities) that specify what they can do with certain resources. So for example, `root` may have a capability that allows them to read, write, and execute a file, while `user5` may have a capability that only allows them to read the file.
  - **Capability List**: A list of capabilities that a user has, which specifies what resources they can access and what kind of access they have (read, write, execute, etc). So for example, a user might have a capability list that includes a capability to read a file, but not write to it.
  - Example: for a given user, let's say that for `file1` this user is allowed read it (R), for `file2` this user is allowed to write to it (W), and for all others they have access to nothing ($\emptyset$).
- **Access Control Matrix** (**ACM**): This is a matrix that combines both ACLs and capabilities. It is a two-dimensional array where the rows represent subjects (users) and the columns represent objects (resources). Each cell in the matrix contains the permissions that the corresponding user has for the corresponding resource. So for example, if we have a user `user5` and a file `file1`, the cell at the intersection of `user5` and `file1` would contain the permissions that `user5` has for `file1` (such as read, write, execute, etc).
  - Example: we have users `root`, `user5`, and `user6`, and files `file1`, `file2`, and `file3`. The ACM would be a matrix where the rows are the users and the columns are the files, and each cell contains the permissions that each user has for each file.

🧠 **Definition**
Access Control List (ACL): A data structure that specifies which users or groups have access to a resource and what kind of access they have (read, write, execute, etc). ACLs are attached to the resource (object) and are used to enforce access control policies.

🧠 **Definition**
Capability: A token or key that grants a user permission to access a resource. Capabilities are attached to the user (subject) and are used to enforce access control policies. A user must present the appropriate capability in order to access a resource.

🧠 **Definition**
Capability List: A list of capabilities that a user has, which specifies what resources they can access and what kind of access they have (read, write, execute, etc). Capability lists are used to enforce access control policies by specifying the permissions that a user has for various resources.

🧠 **Definition**
Access Control Matrix (ACM): A two-dimensional array where the rows represent subjects (users) and the columns represent objects (resources). Each cell in the matrix contains the permissions that the corresponding user has for the corresponding resource. The ACM combines both ACLs and capabilities to enforce access control policies.

	File 1	**File 2**	**File 3**	**File 4**	**File 5**
root	RWX	RWX	RWX	RWX	RWX
user1	RW	R	RWX	∅	∅
user2	R	RW	∅	R	∅
user3	∅	∅	R	R	RW
user4	∅	R	∅	∅	R

表 3.1 Example Access Control Matrix (ACM). R = Read, W = Write, X = Execute, ∅ = No Access. These are very inefficient for large systems, but they are a useful theoretical model for understanding access control policies.

There are engineering tradeoffs between ACLs and capabilities. In theory they are the same, but one may be more efficient than the other in certain contexts. For example, ACLs may be more efficient for resources that have a small number of users with access, while capabilities may be more efficient for resources that have a large number of users with access. Additionally, ACLs can be easier to manage and understand for administrators, while capabilities can be more flexible and scalable in certain situations. Ultimately, the choice between ACLs and capabilities depends on the specific requirements and constraints of the system being designed.

3.4. RBAC: Role Based Access Control

This is where the role determines permissions. So instead of assigning permissions to individual users, you assign permissions to roles, and then you assign users to those roles. For example, you might have a role called “admin” that has permissions to read, write, and execute all files, and then you would assign users to that role based on their job function or responsibilities. This can make it easier to manage permissions in large systems because you can simply assign users to roles rather than having to manage permissions for each individual user.