# Introduction to Cybersecurity

UT Knoxville, Spring 2023, COSC 366

Dr. Scott Ruoti, Alex Zhang

May 12, 2023

# Contents

# Security Concepts and Principles

## 1.1 Fundamental Goals of Computer Security

> **Definition 1.1.1 ▶ Computer Security**
>
> **Computer security** is the practice of protecting computer-related assets from unauthorized actions, either by preventing such actions or detecting and recovering from them.

The goal of Computer Security is to help users complete their desired task safely, without short or long term risks. To do this, we support computer-based services by providing essential security properties.

> **Definition 1.1.2 ▶ Confidentiality, Integrity, Availablity (CIA)**
>
> - **Confidentiality**: only authorized parties can access data, whether at rest or in motion (i.e. being transmitted)
> - **Integrity**: data, software, or hardware remaining unchanged, except by authorized parties
> - **Availability**: information, services, and computing resources are available for authorized use
>
> Together, these form the **CIA triad**.

> **Definition 1.1.3 ▶ Principal, Privilege**
>
> A **principal** is an entity with a given identity, such as a user, service, or system process.
> A **privilege** defines what a principal can do, such as read/write/execute permissions.

In addition to the CIA triad, we also have three security properties relating to principals.

> **Definition 1.1.4 ▶ Authentication, Authorization, Auditability (Golden Principles)**
>
> The **Golden Principles** are three security properties regarding principals:
> - **Authentication:** assurance that a principal is who they say they are

> • *Authorization:* determining whether a requested privilege or resource access should be granted to the requesting principal
> • *Auditability:* ability to identify principals responsible for past actions

Auditability (or accountability) gives away two key things: who conducted the attack and the methods by which an attack was made.

> **Definition 1.1.5 ▶ Trustworthy, Trusted**
>
> Something is **trustworthy** if it deserves our confidence. Something is **trusted** if it has our confidence.

> **Definition 1.1.6 ▶ Privacy, Confidentiality**
>
> **Privacy** is a sense of being in control of access that others have to ourselves. It deals exclusively with people. **Confidentiality** is an extension of privacy to also include personally sensitive data.

## 1.2 Computer security policies and attacks

> **Definition 1.2.1 ▶ Asset**
>
> An **asset** is a resource we want to protect, such as information, software, hardware, or computing/communications services.

Note that asset can refer to any tangible or intangible resources.

> **Definition 1.2.2 ▶ Security Policy**
>
> A **security policy** specifies the design intent of a system's rules and practices (i.e. what the system is supposed to do and not do).

> **Definition 1.2.3 ▶ Adversary**
>
> An **adversary** is an entity who wants to violate a security policy to harm an asset. Can also be called "threat agents" or "threat actors". Some attributes of an adversary include:
> • *identity*: who are they?
> • *objectives*: what assets the adversary might try to harm

- ***methods***: the potential attack techniques or types of attacks
- ***capabilities***: skills, knowledge, personnel, and opportunity

A formal security policy should precisely define each possible system state as either authorized (secure) or unauthorized (non-secure). Non-secure states raise the potential for attacks to happen.

### Definition 1.2.4 ▶ Threat

A ***threat*** is any combination of circumstances and/or entities that allow harm to assets or cause security violations.

### Definition 1.2.5 ▶ Attack, Attack Vector

An ***attack*** is a deliberate attempt to cause a security violation. An ***attack vector*** is the specific methods and steps by which an attack was executed.

### Definition 1.2.6 ▶ Mitigation

***Mitigation*** describes countermeasures to reduce the chance of a threat being actualized or lessen the cost of a successful attack.

Mitigation can include operational and management processes, software controls, and other security mechanisms.

### Example 1.2.7 ▶ House Security Policy

Consider this simple security policy for a house: no one is allowed in the house unless accompanied by a family member, and only family members are authorized to take things out of the house.

- The presence of someone who wants to steal an asset from our house is a **threat**.
- An unaccompanied stranger in the house is a **security violation**.
- An unlocked door is a **vulnerability**.
- A stranger entering through the unlocked door and stealing a television is an **attack**.
- Entry through the unlocked door is an **attack vector**.

## 1.3    Risk, risk assessment, and modeling expected losses

> **Definition 1.3.1 ▶ Risk**
>
> A **risk** is the expected loss of assets due to future attacks.

There are two ways we assess risk: **quantitative** and **qualitative** risk assessment.

- **Quantitative** risk assessment computes numerical estimate of risk

- **Qualitative** risk assessment compares risks relative to each other

Quantitative risk assessment is more suited for incidents that occur regularly, with historical data and stable statistics to generate probability estimates. However, computer security incidents occur so infrequently that any estimate of probability likely isn't precise. Thus, qualitative risk assessment is usually more practical. For each asset or asset class, their relevant threats are categorized based on probability of happening and impact if it happened.

Precise estimates of risk are rarely possible in practice, so qualitative risk assessment is usually yields more informed decisions. A popular equation for modeling risk is:

$$R = T \cdot V \cdot C$$

where:

- $T$ is the probability of an attack happening,

- $V$ is the probability such a vulnerability exists, and

- $C$ is cost of a successful attack, both tangible and intangible costs

$C$ can encompass tangible losses like money or intangible losses like reputation. Whatever model we use, we can then model expected losses as such:

In risk assessment, we ask ourselves these questions:

1. What assets are most valuable, and what are their values?

2. What system vulnerabilities exist?

3.

In answering these questions, it becomes apparant we cannot employ strictly quantitative risk assessment. A popular model for qualitative risk assessment is the DREAD model:

> **Definition 1.3.2 ▶ DREAD**
>
> **DREAD** is a method of qualitative risk assessment using a subjective scaled rating system for five attributes.
>
> | Attribute | 10 | 1 |
> |---:|---|---|
> | ***Damage Potential*** | data is extremely sensitive | data is worthless |
> | ***Reproducibility*** | works every time | works only once |
> | ***Exploitability*** | anyone can mount an attack | requires a nation state |
> | ***Affected Users*** | 91-100% of users | 0% of users |
> | ***Discoverability*** | threat is obviously apparent | threat is undetectable |
>
> The final DREAD score takes the average of all five attributes.

A common criticism of DREAD is that rating discoverability might reward security through obscurity. People will sometimes omit discoverability, or simply assign it the maximum value all the time.

$$ALE = \sum_{i=1}^{n} F_i \cdot C_i$$

where:

- $F_i$ is the estimated frequency of events of type $i$, and

- $C_i$ is the average loss expected per occurence of an event of type $i$

## 1.4   Adversary modeling and security analysis

In designing computer security mechanisms, it is important to think like an adversary. Try to enumerate what methods might they use, and design around them. Different adversaries can have wildly different objectives, methods, and capabilities.

> **Definition 1.4.1 ▶ Security Analysis**
>
> **Security analysis** aims to identify vulnerabilities and overlooked threats, as well as ways to improve defense against such threats

Types of analysis include:

- *Formal security evaluation*: standardized testing to ensure essential features and security
- *Internal vulnerability testing*: internal team trying to find vulnerabilities
- *External penetration testing*: third-party audits to find vulnerabilities; often the most effective

Security analysis is a heavily involved process that may employ a variety of methodologies. For example, manual source code review, review of design documents, and various penetration testing techniques. It's important to be aware of potential vulnerabilities as we are writing code.

## 1.5 Threat Modeling

A threat model will identify possible adversaries, threats, and attack vectors. We need to list a set of assumptions about the threats as well as clarify what is in and out of the scope of possibilities.

**Diagram-Driven Threat Model**    Threat model diagrams include assets, infrastructure, and defenses/mitigations, making apparent the possible attack vectors. Popular examples include:

- *Data Flow Diagrams:* models all possible data routes
- *User Workflow Diagram:* models how users interact with the program, both frontend and backend
- *Attack Trees:* models all possible attack vectors like a flow chart; leaf nodes are initial actions

---

**Definition 1.5.1 ▶ STRIDE**

*STRIDE* is a model for enumerating and identifying possible computer security threats.
- *Spoofing:* attacker impersonates another user, or malicious server posing as a legitimate server
- *Tampering:* altering data without proper authorization; can occur when data is stored, processed
- *Repudiation:* lying about past actions (e.g. denying/claiming that something happened)
- *Information Disclosure:* exposure of confidential information without authorization

---

- ***Denial of Service:*** render service unusable or unreliable for users
- ***Elevation of Privilege:*** an unprivileged user gains privileges

## 1.6 Threat Model Gaps

Often, wrong assumptions about risk or focus on wrong threats will lead to gaps between our threat model and what can realistically happen (e.g. if we don't account for something, but it does happen).

**Changing Times** New adversaries, technologies, software, and controls means we need to constantly adjust our threat model.

## 1.7 Design Principles

1. ***Simplicity and necessity:*** complexity increases risks (KISS: keep it simple, stupid)

2. ***Safe Defaults:*** ensure default settings are secure; users rarely change defaults

3. ***Open Design:*** don't rely on the secrecy of our code; just assume it will leak

4. ***Complete Mediation:*** never assume access is safe; always check access is allowed

5. ***Least Privilege:*** don't give principals extraneous privileges; limit impact of compromise

6. ***Defense in depth:*** multiple layers of security; don't rely on only one security control

7. ***Security by design:*** think about security throughout development, not an afterthought

8. ***Design for evolution:*** allow for change for whenever it's needed

Overall, computer security is hard! `explain why tho`

# 2 Cryptography

Whenever working with cryptography, always remember:

1. Do not design your own cryptographic protocols or algorithms. Your design will almost always be flawed.
2. **Kerckhoff's Principle:** Security should only come from the secrecy of the crytographic key, NOT the algorithm or code. In other words, just assume that any adversary can view your source code.
3. Encryption only provides **confidentiality**, not integrity. That is, encryption only keeps your data secret, but an attacker could still modify the encrypted data.

## 2.1   Introduction

> **Definition 2.1.1 ▶ Cipher, Plain Text, Cipher Text**
>
> A **cipher** is any encryption or decryption algorithm. A **plaintext** message is a message prior to cipher and is thus vulnerable. A **ciphertext** message is after the cipher is applied.

Some possible threats to encryption may be:

- **Brute force attack:** guess and check all keys
- **Algorithmic break:** some flaw in the encryption algorithm

> **Definition 2.1.2 ▶ Cryptographic Key, Key Space**
>
> A **cryptographic key** is a value which is used to decrypt cipher text. It should be relatively large and remain private. The **key space** is the set of all possible cryptographic keys in a cipher.

The key space for a cipher should be large enough such that it would be impractical for an attacker to perform a brute force attack. For example, AES-256 uses 256-bit keys. The key space of AES-256 contains $2^{256}$ distinct keys. Even a thermodynamically perfect computer with the power of the sun could not even count through $2^{256}$ keys, much less guess and check them!

**Definition 2.1.3 ▶ Passive Adversary, Active Adversary**

A *passive adversary* only observes and records communications. An *active adversary* interacts with and/or modifies communications, possibly by injecting data, altering data, or starting new interactions with legitimate parties.

**Definition 2.1.4 ▶ Information-Theoretic Security**

We say ciphertext has *information-theoretic security* if given **unlimited** computer power and time, an attacker could not recover the plaintext from the ciphertext.

**Definition 2.1.5 ▶ Computational Security**

We say ciphertext has *computational security* if given **fixed** computer power and time, an attacker could not recover the plaintext from the ciphertext.

**Definition 2.1.6 ▶ Stream Cipher, Block Cipher**

A *stream cipher* encrypts information one bit at a time. A *block cipher* encrypts information in blocks.

## 2.2   Symmetric Encryption Model

**Definition 2.2.1 ▶ Symmetric-Key Encryption**

*Symmetric-key encryption* uses the same cryptographic key to encrypt plaintext and decrypt ciphertext.

**Definition 2.2.2 ▶ One-Time Pad (OTP)**

The *one-time pad* is a symmetric stream cipher that simply XOR's a message with a key of equal length. The resulting ciphertext is information theoretic so long as the cryptographic key is:

- as long as the plaintext,
- completely randomly generated,
- never reused, and
- kept completely secret between authorized parties.

Many modern cryptographic algorithms work by taking a small cryptographic key and expand-

ing it into a sufficiently large one-time pad key.

> **Definition 2.2.3 ▶ Advanced Encryption Standard (AES)**
>
> *AES* is the most common symmetric block cipher.
> - Messages are split into 128-bit blocks (with padding)
> - Cryptographic key is 128, 196, or 256 bits long
> - Provides computational security
> - A single bit changed should change (on average) 50% of the ciphertext
>
> In addition, different block modes help to remove patterns among blocks.

## 2.3  Asymmetric Encryption Model

> **Definition 2.3.1 ▶ Asymmetric Encryption, Public Key, Private Key**
>
> *Asymmetric encryption* utilizes two cryptographic keys: a *public key* which encrypts plaintext, and a *private key* which decrypts ciphertext.

Unfortunately, public key encryption is orders of magnitude slower than symmetric key encryption. We can incorporate both using hybrid encryption, providing the best of both worlds.

> **Definition 2.3.2 ▶ Hybrid Encryption, Key Wrap**
>
> *Hybrid encryption* incorporates both symmetric key and public key encryption. We encrypt our plaintext using a symmetric key, and we send both the encrypted ciphertext and the symmetric key encrypted using the other party's public key. This encrypted key is called a *key wrap*.

> **Definition 2.3.3 ▶ Padding**
>
> Many encryption algorithms *pad* messages to:
> - ensure the message is properly formatted
> - prevent attacks by adding random noise

To see why padding is important, suppose we only need to send a single integer. While the key space is large, the message space is small. Thus, an attacker could guess and check all possible messages until it matches the encrypted message.

There are many ways to pad a message. Generally, we want to use the optimal asymmetric

encryption padding (OAEP).

> **Definition 2.3.4 ▸ RSA**
>
> **RSA** is a common and mathematically simple cryptographic algorithm that provides computational security.

> **Code Snippet 2.3.5 ▸ RSA Encryption using C#**
>
> ```csharp
> var message = "Hello World!";
> var plaintext = Encoding.ASCII.GetString(ciphertext);
> var rsa = RSA.Create(2048);
> var public_key = rsa.ExportRSAPublicKeyPem();
> var private_key = rsa.exportRSAPrivateKeyPem();
>
> var ciphertext = rsa.Encrypt(plaintext,
>   ↳ RSAEncryptionPadding.OaepSHA256);
>
> var decrypted = rsa.Decrypt(ciphertext,
>   ↳ RSAEncryptionPadding.OaepSHA256);
> var decrypted_plaintext = Encoding.ASCII.GetString(decrypted);
> ```

## 2.4 Digital Signatures

> **Definition 2.4.1 ▸ Digital Signature**
>
> **Digital signatures** are tags (bitstrings) that accompany a message, verifying that data came from the right person. The **private key** is used to sign data, and the **public key** to verify a signature.

Digital signatures provide three security properties:

1. **Data origin authentication:** the data comes from the owner of the private key
2. **Data integrity:** the data has not changed since it was sent
3. **Non-repudiation:** the sender cannot later claim to have not sent the data

In practice, digital signatures provide non-repudiation if and only if nobody besides the legitimate party has the private key for signing. They could try to deny ever signing something by

saying "someone must have stolen and used my private key."

It is **crucial** to avoid using the same key for signing and for cryptography because:

1. If our private key gets stolen, someone can then decrypt our data **and** forge our digital signatures.
2. There can be strange mathematical interactions between encryption and signing that might reveal information about our key.

Similar to encryption algorithms, there are signing algorithms that create digital signatures and verification algorithms that verify digital signatures. In addition, we still pad our messages before signing to ensure the message is formatted and to prevent some (relatively esoteric) attacks.

Data is often hashed before signed. Again, different key pairs should be used for encryption/decryption and signing/verification.

## 2.5   Cryptographic Hash Functions

> **Definition 2.5.1 ▶ Hash Function**
>
> A *hash function* takes a string of **any length** and outputs a (relatively) unique string of **fixed length**.

The output of a hash function can be called a *hash value*, *digest*, or simply *hash*. In practice, hash functions should always:

1. be applicable to data of any size,
2. output a fixed length data, and
3. be fast to compute.

Hash functions should also guarantee three essential security properties:

1. *One-way property*: Theoretically impossible to find the original message from the hash; one hashed string maps to an infinite amount of input strings (sometimes called preimage resistance)
2. *Weak collision resistance*: Given an input, it's computationally impossible to find another input that produces the same hash.
3. *Strong collision resistance*: It's computationally impossible to find any pair of distinct inputs that produce the same hash.

**Definition 2.5.2 ▶ SHA**

*SHA* is a common hash function.

**Example 2.5.3 ▶ Data Integrity**

- When sending data, first hash it and send the hash along with the data
- When receiving the data, first hash it and check that the calculated hash matches the one that was received
- TODO: finish from slides

**Example 2.5.4 ▶ Password Storage**

Instead of storing passwords in plaintext, the server stores the hashed passwords. Hashing at client and then sending to server is flawed!

- If an attacker steals the hashed passwords, an attack can simply log in using the hashed passwords.

## 2.6 Message Authentication Codes

**Definition 2.6.1 ▶ Message Authentication Code (MAC)**

A *message authentication code* is a tag (or bitstring) used for authenticating a message (i.e. came from the right person).

- Symmetric-key equivalent to digital signatures
- Provides data origin authentication and data integrity (only for two people)

In practice, message authentication codes **do not** provide non-repudiation. At least two people will have access to the key (which is used for signing and authenticating). Thus, a third party won't be able to tell who created the MAC.

**Definition 2.6.2 ▶ Hash-Based Message Authentication Code (HMAC)**

Can be used with any cryptographic hash function, only use with safe functions (e.g. HMAC-SHA256)

> **Definition 2.6.3 ▶ CMAC**
>
> Based on symmetric encryption using CBC mode

> **Definition 2.6.4 ▶ UMAC**
>
> Based on universal hashing; pick a hash function based on the key, then encrypt the digest

Again, don't reuse the same key for encryption and decryption

**Code Snippet 2.6.5 ▶ Simple HMAC in C#**

```csharp
using System.Security.Cryptography;
using System.Text;

var message = "Hello World!";
var plaintext = Encoding.ASCII.GetBytes(message);

var digest = SHA256.HashData(plaintext);
var digest2 = SHA256.HashData(plaintext);

// Verify the two digests are the same
digest.SequenceEqual(digest2)

var receivedtext = "Gello World!";
var key = RandomNumberGenerator.GetBytes(512);
var calculatedMAC = HMACSHA256.HashData(key, plaintext);
var receivedMAC = HMACSHA256.HashData(key, receivedtext);

// This will be false
calculatedMAC.SequenceEqual(receivedMAC);
```

# 3 User Authentication

*Authentication* is the process of verifying an identity is legitimate by some supporting evidence. In contrast, *identification* establishes an identity using contextual information, not necessarily having an explicit identity asserted.

## 3.1 Passwords

> **Definition 3.1.1 ▶ Username, Password**
>
> A *password* is a piece of secret information, typically a string of easily-typed characters, that is used to authenticate a user.

Passwords usually have an associated account with a username, public or secret. We are more concerned with the actual bits of the password, not necessarily the characters themselves. As such, we need to have a deterministic way of converting characters to bits, whether it be ASCII or UTF-8.

| Advantages of passwords: | Disadvantages of passwords: |
| --- | --- |
| <ul><li>Simple and easy to learn</li><li>Free for users</li><li>No physical load to carry</li><li>Usually easy to recover lost access</li><li>Easily delegated (e.g. sharing Netflix passwords)</li></ul> | <ul><li>Hard to create random passwords (theoretical space is large, but practical space is small)</li><li>Hard to remember good passwords</li><li>Users often reuse passwords</li></ul> |

To address the problem that the practical space of passwords is relatively small, some systems use *system-assigned passwords* that maximize the space of passwords.

> **Definition 3.1.2 ▶ Password Composition Policies**
>
> A *password composition policy* specifies requirements about creating a password, such as minimum and maximum password length, or character requirements and restrictions.

Although intended to improve password strength, these composition policies often backfire. It makes a lot of users recycle the same passwords. Similarly, forced password changes often backfire as most users, tech-savvy or not, will increment their password like "hello1", "hello2". This predictable pattern allows attackers to easily guess passwords whenever the next forced change happens.

> **Definition 3.1.3 ▶ Passkey**
>
> A *passkey* is a cryptographic key derived from a password.

PBKDF2 is a common function. Modern algorithms like Argon2id can also be used.

## 3.2   Password Authentication

Password authentication can be defeated by techniques such as:

1. *Online password guessing:* guesses are sent to the legitimate server
2. *Offline password guessing:* guesses are checked locally, usually accompanied by the password database itself
3. *Password capture:* attacker directly intercepts or observes the password (e.g. key logging, phishing, or simply looking over someone's shoulder as they type the password)
4. *Password interface bypass:* completely bypass the password system
5. *Defeating recovery mechanisms:* use account recovery methods to gain access to accounts

> **Example 3.2.1 ▶ Simplest (and worst) password authentication**
>
> Registration:
> - User sends a username and password to the website
> - Website stores the username and password in plaintext
>
> Authentication:
> - User sends a username and password to the website
> - Website compares the sent values with the stored values

Some of the major flaws include:

- Data can be intercepted when sending to the server; need to encrypt communication between user and server
- *Phishing:* attacker tricks user into giving the attacker their info; hard to mitigate, could

use password managers or hardware security tokens
- Online password guessing; have to rate limit authentication attempts, can also require strong passwords
- Password database theft

> **Example 3.2.2 ▶ Simple but better password authentication**
>
> Registration:
> - User sends a username and password to the website
> - Website hashes the password, then stores the info
>
> Authentication:
> - User sends a username and password to the website
> - Website hashes the received password and compares the calculated hash

This is much better, but there are still some pretty serious flaws:

- *Offline guessing:* if an attacker steals the database, then they can guess passwords until it matches one of the hashed passwords with no rate limit.
- *Rainbow Table Attack:* someone can simply create a table of password hashes for common passwords

> **Example 3.2.3 ▶ Salted Hashing**
>
> Registration:
> - User sends a username and password to website
> - Website generates some random bytes called a *salt*
> - Website hashes the received password concatenated with the salt (similar to padding)
> - Website stores username, and salt in plaintext as well as the hashed password
>
> Authentication:
> - User sends a username and password to website
> - Website retrieves the salt for the username
> - Website hashes the received password along with salt
> - Website compares the calculated hash with the stored hash

This defeats generalized rainbow table attacks. In extreme situations, a motivated adversary can steal the database and use the salt to create a specialized rainbow table targeted against one person.

We can add more features for stronger security, such as:

- *Iterated Hashing:* Repeatedly hash the salt and password a large number of times; slows down brute force guessing; has minimal impact on legitimate authentications
- *Specialized Functions:* Use memory-intense and cache-intense functions; makes it hard to accelerate, reducing password guessing rate significantly (e.g. Argon2id, bcrypt/scrypt)
- *Keyed Hash Function:* Use a hash function that requires a cryptographic key (e.g. HMAC); if key isn't stolen, it will be impossible to conduct offline guessing

**Code Snippet 3.2.4 ▶ Passwords with Salted Hashing in C#**

```csharp
using System.Security.Cryptography;
using System.Text;

var password = "hello";
var passwordBytes = Encoding.ASCII.GetBytes(password);

var salt = RandomNumberGenerator.GetBytes(32);

// hash for 100000 iterations
var hash = Rfc2898DeriveBytes.Pbkdf2(password, salt, 100000,
    ↪ HashAlgorithmName.SHA256, 32);

using BCrypt.Net;

// Returns a crazy string
var temp = BCrypt.Net.BCrypt.HashPassword(password);

// Should return true
BCrypt.Net.Bcrypt.Verify(password, temp);
```

## 3.3   Account Recovery

When a user forgets their password or their account gets compromised, it's important that the user can still recover that account. It's also important to not let the account recovery method be easier for an adversary than password authentication. Security is only as strong as its weakest link.

> **Definition 3.3.1 ▶ Password Recovery, Account Recovery**
>
> ***Password recovery*** retrieves a user's lost password. ***Account recovery*** lets a user regain access to their account, requiring a password reset.

Password recovery is a sign that the server is storing passwords in plaintext, or they are simply encrypting them without hashing. Either way, it's terrible security practice, and is a sign that the maintainers of the website have no idea how to implement good security.

In practice, account recovery should be the only recovery method available. Some popular methods include:

- ***Email-Based Recovery:*** Send a link to the email account; makes accounts only as strong as the email
- ***SMS-based recovery:*** similar to email-based recovery; prone to sim swapping
- ***Security Questions:*** in practice, it's easy for attacker to learn answers to these questions

> **Definition 3.3.2 ▶ Single Sign-On (SSO)**
>
> ***Single Sign-On*** lets a single identity provider handle user authentication for multiple services.

For example, you have probably logged into websites using your Google account. While SSO creates a single point of failure, it can be argued it's not that bad because most users already reuse the same password for multiple sites. Hence, many services let identity providers handle authentication. It's easier for the user (who doesn't have to create as many accounts), and most identity providers have top-class security.

In the Single Sign-On, there are three parties: the user, relying party, and identifying party. When trying to authenticate with the relying party, we first request an authentication token from the identifying party. We then send that token to the relying party, who verifies the token is correct by asking the identifying party.

> **Definition 3.3.3 ▶ CAPTCHA**
>
> ***CAPTCHA*** refers to any online test used to differentiate human and computer entities. It's a loose acronym of "completely automated public Turing test to tell computers and humans apart".

A popular and modern approach to CAPTCHA is ***reCAPTCHA***. It measures behavioral patterns such as interactions with the website as well as other metrics like IP address to determine

if you're a human.  If it's not sure, it makes you do the traffic light clicking thing.  If it really thinks you're a robot, it makes you do those fading traffic light clicking things.

# 3.4   Authentication Factors

Authentication factors are used to verify you are who you say you are.  They rely on one or more of the following:

- something you *know* (like a secret PIN or password)
- something you *have* (like a phone or credit card)
- something you *are* (fingerprints, behaviors)

> **Definition 3.4.1 ▶ Multi-Factor Authentication**
>
> *Single factor authentication* authenticates users using only one factor.  *Two-factor authentication (2FA)* authenticates using two distinct factors, most commonly a password and something you have.  *Three-factor authentication (3FA)* is the most secure but usually not practical.

2FA substantially increases the security of accounts, but it needs to be carefully implemented to avoid usability issues.

Something you have:

- In your possession, trivial to authenticate
- Nothing to remember
- If you don't have the item, you can't authenticate
- If someone steals the item, you have immediate access
- If you lose item, account recovery is difficult

We can use the thing you have by sending that thing a secret value.

- Service sends a secret value to your device (one-time password)
- You authenticate by entering the secret value
- Problem: easy to phish

Alternatively, we can just have some way to synchronize the secret value generation, so we only need to send a secret once.

- Service sends you a secret during account registration
- Secret is stored on your device

- The secret is used to generate the same one-time password on both the server and dev
- Authenticate by entering the value generated by your device.
- Problem: still easy to phish, but attacker would only have a one-time window

We could use some dedicated device for this.

- Dedicated device generates a public-key private-key pair
- Public key is sent to service during registration
- Device uses its private key to sign authentication requests
- Can be designed to avoid phishing and require presence
- Usually comes with some backup codes in case the device is broken; those can be prone to phishing

When it comes to something you are:

- Nothing to remember
- Nothing to carry
- Difficult to steal
- Usually requires specialized hardware; bad hardware is prone to error and/or security issues
- If you lose your biometric, how do you recover your account?
- Sharing biometrics is risky in the first place

Some examples include:

- Fingerprints – lots of poorly designed hardware prone to fingerprint lifting or forging
- Facial Recognition – similar looking people like siblings can log in too
- Iris Recognition – specialized hardware costs a lot
- Brainwave Recognition –
- Behavior Biometrics – e.g reCAPTCHA
- Walking Cadence – how you walk
- Travel Patterns – used by gov't to identify suspicious people

When using biometrics for an app, it's usually not being sent to the server. Instead, the OS will only decrypt shared secret after authenticating using biometrics.

When authentication with biometrics like fingerprints, we aren't actually sending biometric info to the server. When logging in, the website and local device create a secret passphrase, and the local device only sends the passphrase to the website when the local device verifies your biometric. This is not a form of 2FA. From the website's perspective, there's only a single factor of authentication.

# OS Security

## 4.1 Reference Monitor

> **Definition 4.1.1 ▶ Subject, Action, Object**
>
> A **subject** is any entity (such as a user or process) trying to take some **action** such as read, write, execute, start, shutdown, etc. The **object** receives the action, such as some memory, files, services, or devices.

The subject is usually authenticated, but not necessarily. We can sometimes have an unknown subject.

> **Definition 4.1.2 ▶ Policy, Reference Monitor**
>
> **Policies** define what is allowed, usually parameterized by subject, action, and object. **Reference monitors** enforce a policy by checking actions, allowing subjects to execute an action if it conforms with the policy.

Whenever a subject tries to make an action, the reference monitor gets to decide whether it should be allowed. The reference monitor has to require:

- **Complete Mediation:** all actions must go through the reference monitor (also ensures complete logs)

- **Tamper-proof:** users can't just manipulate the reference monitor; also includes tamper-proof logs

- **Verifiable:** reference monitor must be easily analyzable (means its source code has to be small!)

- **Reliable Authentication and/or Authorization:** authentication for subject identfication; authorization for bearer tokens

> **Definition 4.1.3 ▶ Permission, Capability List**
>
> A **permission** is a pair of (action, object). The list of all permissions associated with a subject can be called a **capability list**.

> **Definition 4.1.4 ▶ Capability-Based Access Control, Bearer Token**
>
> **Capability-Based Access Control** reference monitor issues **bearer tokens** which let anyone with the token take specific action(s).

An example of a bearer token would be the link-sharing system on Google Docs. The document creator can easily delegate links which allow those with the link to view or edit our document. Another example would be API tokens.

> **Definition 4.1.5 ▶ Access Control List (ACL)**
>
> The **access control list** is the set of (subject, action) pairs associated with an object.

Some other popular access control paradigms include:

- **Role-based access control (RBAC):** associates permissions with certain groups of users rather than users themselves.

- **Attribute-based access control (ABAC):** incorporates contextual information to access control decisions (e.g. subject's behavioral patterns, current location; action's time of day, current threat level; object's location on network, creation date, size)

- **Cryptographic access control:** use cryptography for access control without a reference monitor

## 4.2   Memory Protection

We care about memory protection because of concurrent execution: multiple users and multiple processes at the same time. We need to isolate resources from each process and user.

> **Definition 4.2.1 ▶ Virtual Memory**
>
> **Virtual memory** is an idealized abstraction of memory, mapping virtual addresses to physical memory addresses.

In virtual memory, the OS kernel acts as a reference monitor for memory, mediating access to

system memory. The kernel allocates memory by giving processes virtual addresses that map to physical ones. The kernel also limits who can access what memory, preventing users/processes from accessing other memory. It also protects OS memory and provides safe mechanisms for shared memory access.

> **Definition 4.2.2 ▶ Kenel Memory Segmentation, Memory Segments**
>
> The kernel divides memory into *memory segments*—contiguous regions of memory. Each segment is assigned one or more modes: read, write, execute, mode, fault.

The latter two modes are used by the OS for specific uses. The kernel sets default modes when allocating memory. Memory segments with code is marked as executable. Memory segments with the heap and stack are marked as readable and writable, not executable (prevents arbitrary code execution). In accordance with the principle of least privilege, the process can further restrict its own permissions.

## 4.3  File System Access Control

> **Definition 4.3.1 ▶ File system**
>
> A *file system* is a hierarchical structuring of directories and folders. It maintains per-file meta-data, including permissions.

In the context of file systems:

- The *subjects* are users, groups or processes (identified by some user id, group id, or process id).
- The *actions* are read/write/execute, modify metadata, etc.
- The *objects* are files, directories, links, and devices.

We can classify file system actions by three distinct categories:

1. *Basic actions* like read, write, and execution of files.

2. *Special actions* like:

    - Execute a binary as the owning user (e.g. sudo)
    - Execute as the primary group of the owning user
    - Set "sticky bit" (disables changing name or file deletion)

3. *Advanced actions* like:

- Take ownership of a file
- Change permissions
- Write attributes
- Inspect or modify memory (debugging permission)

### Example 4.3.2 ▶ Linux Access Control (user-group-other or UGO)

Linux uses the ***user-group-others*** (UGO) permission model which is a highly condensed ACL. This:

- limits subjects to either the owning user, primary group of the owning user, and everyone else, and
- limits actions to read, write, execute, and special permissions (directory travel is controlled by read; modify and write are considered the same)

When determining permissions for actions, Linux checks the following in order, letting the action happen if any one of these is true:

1. Is the subject the owning user?
2. Is the subject a member of the owning group?
3. Do the file's default permissions allow this action?

Note that permissions are not inherited (i.e. every object has its own permission list). Also, it's possible to let a subject access an object inside a directory that the subject cannot access (i.e. they can't discover the path, but can access the object if they know the path). In contrast to traditional ACLs which specifies every allowed subject action pairs associated with an object, UGO only cares about one user, one group, and lumps everyone else into "other".

### Example 4.3.3 ▶ Windows Access Control

Windows follows role-based access control (RBAC) with the roles being groups. When an object is created, it inherits the permissions of the parent folder.

## 4.4    Additional Topics

> **Definition 4.4.1 ▶ Hidden File**
>
> A **hidden file** has an attribute that indicates files which should be ignored when listing the contents of a directory.

On Linux, this is done be prepending a file name with a period. On Windows, it's part of the file metadata. This does not change any permissions regarding the file, so it should never be used as a security measure.

> **Definition 4.4.2 ▶ inode**
>
> An **inode** is a chunk of data that contains file name, metadata, and where the actual data is stored on the hard drive.

In the UNIX file system, filenames are just a reference to an inode. The inode is only deleted when all filename references to it are deleted.

> **Definition 4.4.3 ▶ File System Links**
>
> **File system links** are data that directly associate a filename with an inode. It allows for more than one file to refer to the same data on the hard drive.
> - A **hard link** is an additional filename reference to an inode, managed by the underlying file system.
> - A **symbolic link** (or symlink) is a text file that contains a single filename. The OS provides file utilities that will read and handle symlinks. It does not reference the inode, so it does not prevent the inode from being deleted.

> **Definition 4.4.4 ▶ chroot jail**
>
> In UNIX, the `chroot` command takes a specified directory and treats it as the root directory, thereby restricting file system access. This is referred to as a **chroot jail**.

The chroot jail is in-line with the principle of least privilege as it limits the impact of compromise. Nowadays, this is largely replaced by sandboxing and containerized app deployments.

`chroot`: restricts file system access

- treats the specified directory as the root directory `/`

- *chroot jail*: _____

wht is
this

> **Definition 4.4.5 ▶ Discretionary Access Control, Mandatory Access Control**
>
> In **discretionary access control**, users are responsible for assigning access control rules for their files. In **mandatory access control**, a policy administrator sets all the access control rules.

Mandatory access control usually relies on role-based access control (RBAC).

> **Definition 4.4.6 ▶ Security-Enhanced Linux (SELinux)**
>
> **Security-Enhanced Linux** is a kernel module found in most Linux distributions, supporting mandatory security policies and mandatory access control. It uses attribute-based access control (ABAC) for applications, processes, and files.

As a side effect, SELinux is often the cause of many issues when running software on Linux. **Never** disable SELinux; only ever reconfigure SELinux to make it work with your software.

# Software Security

## 5.1 TOCTOU Race

> **Definition 5.1.1 ▶ Time-of-check (TOC), Time-of-use (TOU)**
>
> *Time-of-check (TOC)* is when a reference monitor checks the permission for an action. *Time-of-use (TOU)* is when a reference monitor uses the previous check to allow an action.

A common assumption is that the permission condition being checked does not change from the time-of-check to time-of-use. However, in multi-threaded systems where processes can be interrupted or temporarily suspended, permission conditions can change between TOC and TOU (e.g. file permissions or arguments passed to routines).

> **Definition 5.1.2 ▶ Race Condition, TOCTOU Race**
>
> A *race condition* is a software vulnerability that occurs when the timing or sequence of events affects a program's behavior. A *TOCTOU race* is a specific race condition where the state of a resource or condition changes between time-of-check and time-of-use.

File operations are especially vulnerable to TOCTOU race because they rely on an external device. This means the process has to wait on I/O to and from the device. As explained by Chat-GPT:

> File operations often require interactions with external devices, such as a hard disk, to read or write data. These interactions can take a non-negligible amount of time and can lead to vulnerabilities related to TOCTOU race conditions. Specifically, if a process checks the state of a file, such as its existence or permissions, and then performs an operation on it, such as reading or writing data, it can be vulnerable to race conditions if the state of the file changes during the interval between the check and the operation. This can occur because the external device may be shared by multiple processes or may have a time delay due to its physical nature. As a result, if the process does not properly synchronize or control access to the file, it may lead to unexpected or unintended behavior. Therefore, it is important to implement appropriate locking or synchronization mechanisms when accessing files in a concurrent or multi-process environment to prevent TOCTOU race conditions.

Some common ways of fixing TOCTOU race conditions:

- Thread-locking: ensuring only one thread runs a procedure at a time

- Use calls that do both the check and action in one call; guarantees OS will run it right

- _____ slides

## 5.2 Integer-Based Vulnerabilities

C is "weakly-typed", meaning it will implicitly typecast between different integral types. It won't throw exceptions for arithmetic errors. Other languages are more likely to check for these issues and throw exceptions when they happen.

Lots of the issues come from the choice to use fixed-precision data types.

- *Overflow:* the result of an arithmetic operation is too large to store; excess bits get truncated (usually the most-significant bits get truncated)

- *Underflow:* the result of an arithmetic operation is too small to store; again, excess bits gets truncated

Casting Issues:

- When casting between signed and unsigned numbers, the underlying bits don't change

- When casting to smaller data types, bits get truncated

- Sign problems: compare signed and unsigned variables causes both to cast to unsigned numbers; sometimes treats negative numbers are being larger than positive numbers

C Type Coercion:

- If any operand is unsigned, then all operands are treated as unsigned

- The size of the resulting data type is either the size of an integer or the size of the biggest operand

- ─────────────────────────────────────────────────────────────────── slides

## 5.3   Memory

When an x86-64 system loads a binary, it stores some things into memory:

- *Text:* Machine code for the executing binary

- *Shared libraries:* machine code for common libraries; only loaded into memory once to save space (shared by all processes); read and execute permissions

- *Data:* statically allocated data like global and static variables, as well as string constants; read and write permissions

- *Stack:* stack frames and associated data; 8MB by default; read and write permissions

- *Heap:* dynamically allocated objects (e.g. malloc'ed data); read and write permissions

> **Definition 5.3.1 ▶ Stack**
>
> The *stack* is a contiguous piece of memory allocated to a program when it runs.  Since it's a fixed size, it starts from the highest address and "grows" towards the lower address.

A CPU register tracks the memory address of the top of the stack (sometimes called a stack pointer).  Each function call allocates some memory from the stack called a *stack frame*.  It stores the data needed to execute the function such as:

- the seventh or higher argument

- variadic arguments (from a function that takes a variable number of arguments)

- local variables

- data needed to return to the calling function such as the return address and stored CPU state (e.g. register before the fuction call)

## Definition 5.3.2 ▶ Buffer, Buffer Overflow

A **buffer** is an array used to read in data. It's usually a fixed size but can be dynamically allocated. A **buffer overflow** occurs when we write past the end of the buffer.

Many compilers will give a procedure's stack frame some extra unused bytes or padding to prevent buffer overflows reaching important memory like the return address.

Effects of buffer overflow can widely vary, including:

- nothing: overflow into the padding or some unused data

- minor issues: overwriting the return address to jump to another benign part of the program, or overwrites data that is still used but does not significantly modify behavior

- serious issues: changing the return address to an invalid address (SEGFAULT), or change program data to modify the program's behavior

- catastrophic issues: change return address to jump to attacker control code; can cause full machine compromise

Possible places for an attacker to inject malicious code include the stack (e.g. buffer), shared libraries (e.g. DLLs), local/global variables, environment variables, heap, and command-line arguments (`argv`). Attacking the stack is most common because the return address is right there.

## 5.4   Buffer Overflow Defenses

| Technique | Limitations |
| --- | --- |
| Mark the stack and heap as non-executable | <ul><li>Loss of functionality like reflection</li><li>Can be circumvented by using code from other parts of memory (called return-oriented programming)</li><li>Code snippets found in `libc` are Turing complete!</li></ul> |
| *Stack canary:* have some random value between the return address and local variables that changes every function call. If it unexpectedly changed, then we know a buffer overflow happened. | <ul><li>Can be worked around if the canary can be learned</li></ul> |
| *ASLR:* randomize the location of the stack | <ul><li>Can be worked around using a NOP sled</li><li>Other vulnerabilties can be used can be used to locate the stack</li></ul> |
| Simply use the safe C library functions (e.g. `strlcpy` instead of `strcpy`) | <ul><li>Requires competency from the developer to know about these functions and use them correctly</li><li>Easy to make a mistake and allow attacks to still happen</li></ul> |
| Just use another language that does bounds-checking on buffers, pointers, and type casting. These checks can happen at runtime or compile time. | <ul><li>This can still be circumvented in rare edge cases</li></ul> |

Ultimately, there is no absolute defense for buffer overflows. The best approach is to simply apply defense in depth—use as many techniques as possible to deter attacks.

## 5.5    Privilege Escalation

Why do we care about vulnerabilities? An attacker's permissions are initially quite limited. Most often, an attacker starts with no direct access to a system, and thus must access it through public interfaces. The services an attacker interacts with often have greater permissions. If an attacker can compromise the service, they can acquire that service's permissions.

Most of the time, these attacks only result in minor privilege escalation. Attackers will use privilege escalation and pivot to attacking another target with greater permissions. This involves a lot of **lateral movement**, slowly moving throughout an organization's resources until they reach their desired target. This may take months or even years.

## 5.6    Malicious Software

There are several types of software:

- **Legitimate software:** software that is designed for legitimate use
- **Harmful software:** software that is designed for legitimate use, but can cause unintentional harm due to poor design or implementation
- **Malicious software:** software designed to cause harm
- **Potentially Unwanted Software (PUS):** software bundled with other software; usually annoying or unwanted, but not necessarily harmful

How does malware get on computers?

- **Phishing:** tricking users to use harmful software
- **Repackaged software:** a legitimate program bundled with some malware; especially common in software piracy
- **Drive-by download:** non-consensual downloads from exploits in web browser, usually a result of dynamic content
- **Self-propagation:** once on a machine, malware can search for other ways to spread like through the local network or email

When describing the structure of malware, we focus on four distinct attributes:

1. **Dormancy period:** the time between the malware being on the machine and its first run
2. **Propagation strategy:** how the malware spreads to other users and machines
3. **Trigger condition:** what controls when the payload is executed (how long the malware waits to take effect)

4. ***Payload:*** what does the malware actually do?

> ### Definition 5.6.1 ▶ Computer Virus
>
> A ***computer virus*** is malware that can infect other programs or files by modifying them to include a possibly mutated copy of itself.

Viruses were the first type of malicious software.  Most of the early viruses were actually benign—early hackers were just goofing around.  Now, most viruses are designed with the specific intent to cause harm.

1. Dormancy period: dormant until its infected executable runs
2. Propagation strategy:  propagates by tricking users into downloading and running the virus; often focused on social engineering
3. Trigger condition: often triggers immediately, but can be delayed to avoid detection
4. Payload: functionality of the actual malware; often compromises existing software, services, and files; impacts can vary drastically

> ### Definition 5.6.2 ▶ Computer Worm
>
> A ***computer worm*** is malware that can propagate on its own, usually through the network.

1. Dormancy period: executes immediately

2. Propagation strategy: automatically and continuously attacks other systems; spreads between machines on the same computer network

3. Trigger condition: same as a computer virus

4. Payload: varies greatly

More specifically, worms automatically scan the network for vulnerable machines. When they find one, they will exploit a vulnerability on that machine and replicate the worm.  The machine then joins in on the process of finding new vulnerable machines on the network.

Worms have two goals which are often at odds with each other.  First, the worm wants to replicate as fast and far as possible to make it harder to remove them. However, the worm also wants to stay undetected.

Techniques for spreading include:

- Hit-list scanning: before deploying the malware, the attacker predetermines a set of po-

tentially vulnerable targets, or select servers to avoid servers where attacks could be cor-
related and detected

- Internet-scale hit lists: large hit list of every server on the internet; not particularly
stealthy, but very fast and wide-reaching

- Permutation scanning: every instance of the worm scans random points in the address
space; if it detects an infected machine, choose a new random point; avoids localized
detection

### Definition 5.6.3 ▶ Trojan Horse

A *trojan horse* is a virus that pretends to be legitimate software.

This can be fully malicious software that pretends to be legitimate software, or legitimate soft-
ware that has an additional malicious payload. Its malicious functionality may be immediately
apparent, or may take some time to notice. Trojan horses are common in piracy and pornog-
raphy software.

### Definition 5.6.4 ▶ Backdoor

A *backdoor* is software that allows an adversary to access a machine while bypassing
authentication.

Reverse shell and remote desktop tools are common examples. Often, attackers use tools that
are legitimate, but install socially engineer the user to install it.

### Definition 5.6.5 ▶ Keylogger

A *keylogger* is a program that hooks into the operating system API for getting keyboard
events.

With a keylogger, an attacker can steal sensitive information like passwords or credit card
numbers. Also applicable to other I/O devices like the mouse and monitor.

## 5.7   Detecting Viruses

> **Definition 5.7.1 ▶ Malware Signature, Behavioral Signature**
>
> A *malware signature* is a sequence of bytes known to be malicious. A *behavioral signature* refers to a sequence of actions used to identify malware.

These signatures are especially helpful in identifying known malware. However, they are limited in incurring false positives, viruses using previously unknown code or behaviors (zero-day), attackers can create code to avoid detection.

Malware signatures

- Byte sequences known to be malicious

- Search binaries for these signatures

Behavioral signatures

- Sequence of actions are known to be malicious

- Watch binaries as thy execute to detect these sequences of actions; can test binaries automatically in a VM when downloaded

- Related to intrusion detection

Preventing viruses:

- Integrity-checking: create a whitelist of allowed software binary hashes; more effective than signatures, but it's easy to block legitimate software

- Code signing: allow developers to digitally sign their exectuables; check that a binary is signed by a trusted developer; more flexible type of integrity checking

The most common type of malware detection is signature detection. A common way to circumvent this is by encrypting the payload, prepended with some code that will decrypt it whenever it runs.

- *Encrypted payload:* we encrypt the actual payload, and bundle a decryptor that that decrypts the payload when run

- *Polymorphic encrypted payloads:* we use the same encryption, but the virus changes its decryptor for each infection using a *mutation engine*. More advanced, it could use a compiler to create new decryptor implementations on-the-fly.

- *External decryption key:* Same as above, but the key is stored externally, not in the binary

- *Metamorphic Viruses:* no encryption is used, but the entire payload is recompiled with same functionality but different machine code.

> **Definition 5.7.2 ▶ Rootkit**
>
> A *rootkit* replaces part of the kernel's functionality, able to run as root and side-step the kernel's reference monitors.

This means rootkits have arbitrary access to memory, files, devices, and the network. This can be very hard to detect, as the rootkit could just overwrite the functionality used to scan for the rootkit. This often means the only way to remove it is to completely reinstall the machine.

- *System Call Hijacking:* completely replace a legitimate function call

- *Inline Hooking:* add a malicious payload to a legitimate function's precall or postcall methods

Installing rootkits:

- Superusers can simply install kernel modules

- Exploit a vulnerability in the kernel (will already be running with maximum permissions)

- Modify bootloader to load a modified kernel

- Improper memory handling for the kernel (page files written to disk, direct memory access)

There are legitimate uses for rootkits, including virus scanners, debuggers, anti-cheat, and anti-piracy software.

web stuff

Supplychain attack: Web widgets (third party code running on trusted websites) might get attacked; affecting everyone that uses that widget

Cross-site scripting: do this

> **Definition 5.7.3 ▶ Ransomware**
>
> *Ransomware* holds files or computers for ransom, whether it be deleting or leaking files.

- *Encrypting ransomware:* encrypts all the user's personal files; upon payment, the attacker (hopefully) provides software to decrypt the files

- *Non-encrypting ransomware:* Can simply modify access control rules, disable OS functionality,

Encrypting ransomware is hard to fix because, if we choose to remove the ransomware before paying, the files are still encrypted.  Non-encrypting ransomware is relatively easy to fix.  We can just access the files through a different OS, or reinstall the OS without harming the files.

Backups can be used to mitigate ransomware.

> **Definition 5.7.4 ▶ Botnet**
>
> A ***botnet*** is a set of compromised networked computer (bots), coordinated by an attacker (the botmaster).

This greatly increases the attacker's ability to conduct large-scale attacks.  It also helps avoid detection, since it's a more distributed attack.

1. Infect: attacker seeds the botnet by manually targeting some computers

2. Connect: after infection, bots connect to command and control server(s) to receive further orders (referred to as a C& C or C2 server)

3. Control: the C2 infrastructure sends commands to the bots

4. Multiply: bots automatically target other machines, attempting to infect them

Botnets are commonly used for:

- Distributed denial-of-service (DDoS) attacks

- Spamming and phishing emails

- Computational power for brute-force attacks, like cracking password databases

- Money laundering: use computers to transfer funds

- Data theft

- Cryptocurrency mining

## 5.8   Categorizing Malware

Objectives:

- Damage to the host or its data

- Data theft

- Direct financial gain

- Ongoing surveillance

- Spread of malware (lateral movement)

- Control of resources

# Certificate Management and Use Cases <span>6</span>

Although we can create theoretically secure protocols for storing and sending information, we can't be sure that the users don't mess something up. For example, managing one's public and private keys can be error-prone, especially in the hands of a non-tech-savvy user. While we try to make these secure protocols as fool-proof as possible, there will always be room for error.

## 6.1 Introduction

> **Definition 6.1.1 ▶ Key Management**
>
> ***Key management*** is the collection of mechanisms and protocols for safely and conveniently distributing such keys—includes managing not only session keys, but public keys (as discussed herein) and their corresponding long-term private keys.

Proper key management is the foundation of most secure protocols. Managing them across their lifetime is critical to a secure system. A key's "lifecycle" includes:

1. Creating a cryptographic key
2. Storing a key in a secure place
3. Replacing an old or stolen key
4. Destroying unused keys in a safe way

We must also ensure their correct usage (e.g. dissemination, verification, synchronization, recovery). We address these problems using different key infrastructures.

> **Definition 6.1.2 ▶ Public Key Infrastructure (PKI)**
>
> A ***public key infrastructure*** is a collection technologies and processes for managing public keys. It also manages their associated private keys and dictates how keys should be used by applications

There are four components in a PKI:

1. Data structures

2. Procedures and protocols for key management

3. Architectural and backend components

4. Cryptographic toolkits and key management software

PKI is an incredibly general idea that is applicable to several kinds of systems. Often, there is significant overlap in their design. In terms of the X509 specification, there are several PKI instantiations including web PKI, email S/MIME PKI, and code signing PKI. There are also per-application PKI, including email PGP PKI and email identity-based cryptography PKI.

## 6.2  X509 Certificates

The core data structure in web PKI is the certificate.

> **Definition 6.2.1 ▶ Public-Key Certificate, Certificate Authority (CA)**
>
> A **public-key certificate** associates a public key with a specific owner. In this context, the owner is referred to as the **subject**, and the association is referred to as a **binding**. Crucially, the certificate is digitally signed by a **certificate authority**—a trusted verifier of bindings between subjects and public keys.

Certificates may also be self-signed, but this signature cannot be used to verify the binding. This is sometimes used to just conform to the PKI spec.

Certificates contain several name-value pairs called **fields**. The "signed portion" of the fields includes:

- **Version:** certificate definition version (highly standardized, usually x509 version 3)
- **Serial Number:** uniquely identifies certificates
- **Issuer:** name of the CA signing the certificate
- **Validity:** date range in which the certificate is valid
- **Subject:** name of the owner of the public and private key
- **Subject Public Key Info:** the public key and its relevant information
- **Extensions:** other optional/miscellaneous info that can add specific functionality

There are two unsigned fields, which include:

- **Signature Algorithm:** algorithm and its parameters
- **Signature:** digital signature of the certificate from the CA

The subject and issuer fields are of type `name`, which is a dictionary of key value pairs, potentially including:

- Country (C)
- Organization (O)
- Organizational Unit (OU)
- Common-nome (CN)

Collectively, these components are referred to as a ***distinguished name***.

Extension fields are application-specific attributes. Common extensions include:

- `keyUsage`: specifies what the key is used for (e.g. signatures, encryption, key agreement, CRL signatures, etc.)

- `extendedKeyUsage`: even more specification for usage (e.g. code signing, TLS server authentication, etc.)

- `subjectAlternateName`: other identifiers for the subject, such as email, domain name, IP address, URI, etc.

---

**Technique 6.2.2 ▶ Acquiring a Certificate from a CA**

1. An entity generates a public-private key pair.
2. The entity sends a ***certificate request*** to a CA which includes all the fields of the certificate that will be signed.
3. The CA verifies the certificate request.
4. The CA provides a signature of the certificate.

---

Before the CA issues the signed certificate, they must verify the certificate request by the following procedure:

---

**Technique 6.2.3 ▶ Verifiying a Certificate Request**

1. Check that the request is well-formed
2. Verify that the entity actually knows the private key (usually by some zero-knowledge proof)
3. Check for evidence of ownership or control of the distinguished name identified in the `subject` field. On the web, this can include requiring the entity to host a cryptographic secret in a DNS record or on the website. Quality of verification varies wildly.

---

The above process only verifies that the binding is legitimate, not that the subject is a trusted entity!

---

**Technique 6.2.4 ▶ Validating a Certificate**

1. Verify the digital signature
2. Ensure the signature is from a trusted entity (usually a CA)
3. Check that the current date is within the validity period
4. Verify the certificate based on any recognized extensions
5. Check that the certificate hasn't been revoked

---

## 6.3   Trust Models

How do we know that a certificate is signed by a trusted entity? Different PKI systems answer the question differently. Critically, there is no one "right" approach; each has their own strengths and weaknesses.

Certificates predicate around some **trusted certificate store** that stores good, trusted certificates.

---

**Example 6.3.1 ▶ User-Controlled Trust Model**

The trusted certificate store contains self-signed certificates. Two approaches for adding a certificate to the trusted certificate store include:

1. **Out-of-band verification** of the certificate's fingerprint (i.e. digest of the certificate); the subject publishes fingerprint in a trusted out-of-band channel
2. **Trust on first use (TOFU):** trust the first certificate seen from that subject (e.g. SSH)

More advanced versions of this use the concept of **trusted introducers** that can vouch for new keys.

---

**Example 6.3.2 ▶ Single-CA**

The trusted certificate store contains one or more CAs for each application or domain. CAs only sign certificates for their associated application or domain. For example, many instant messaging services use this model.

---

> **Example 6.3.3 ▶ Strict Hierarchy**
>
> The strict hierarchy imposes a hierarchy of roles in the certificate signing model:
> - *Root CA:* signs certificates for intermediate CAs
> - *Intermediate CA:* signs end-entity or leaf certificates; can also sign certificates for other intermediate CAs
> - *End-Entity:* receives certificates from root or intermediate CAs
>
> More info about this

> **Example 6.3.4 ▶ Forest of Hierarchical Trees**
>
> Based on a collection of strict hierarchies with many root CAs. These root CAs are referred to as **trust anchors**. The list of trust anchors will be shipped with the application. This model is used for web and code signing.

## 6.4 Transport Layer Security (TLS)

A problem with network communications is that anyone "in range" of our communications can listen to anything we receive and send. As such, we implement end-to-end encryption to prevent nefarious snoopers.

> **Definition 6.4.1 ▶ Transport Layer Security (TLS)**
>
> **Transport Layer Security** is a security protocol that adds end-to-end encryption to network communication. Content is encrypted by the sender and is decrypted by the receiver.

TLS is used in HTTPS, FTPS, and general network traffic.

- *Confidentiality:* prevents attackers from viewing the contents of your message, but does not protect metadata (e.g info about endpoints, content size, message frequency, etc.)

- *Integrity:* prevents attackers from modifying or spoofing messages

- *Authentication:* provided using signed certificates; authentication of the server is almost always provided, but authentication of the client is rarely provided

Identity of the client is rarely ever established through TLS, but rather through the website's own authentication and user mechanisms.

- Sits at the session layer, carried over TCP

- Content is decrypted before being passed to the application

> **Technique 6.4.2 ▶ TLS Protocol**
>
> There are two parts that make up TLS communication:
>
> 1. *Handshake:*
>     - Negotiate an agreed upon cryptographic algorithm and key
>     - Authenticate entities using certificates and appropriate PKI
>     - Verify that both entities share the same cryptographic keys
> 2. *Data Protection:*
>     - Add MAC using an HMAC algorithm; different key for each direction
>     - Encrypt using different keys for each direction (typically AES)
>     - If an IV is needed, it is created in the handshake (unique IV for each session)

TLS has gone through several iterations, first developed as a proprietary protocol in NetScape, then improved and standardized by others like Microsoft, IETF, and WAP Forum.

## 6.5 Web PKI

As it stands, Web PKI is incredibly fragile. There are anywhere between 100–200 root CAs, any of which can sign for any website. If any root CA gets compromised, then the whole system falls. Moreover, root CAs have notoriously bad security. Many had their private keys compromised, whether it be getting hacked or simply posting their private keys online. Some even sell intermediate CA certificates that can sign for any website.

## 6.6 Secure Email

Email is an inherently insecure system. As such, people have developed measures to protect email communications.

1. Encrypting messages at rest: makes it difficult to steal messages from clients and/or mail servers

2. Using TLS: Prevents data being modified or intercepted during transmission; does not protect stored messages nor protect against forged messages

3. *Sender Policy Framework (SPF):* DNS record that whitelists IPs that can send emails; limited protection against forged emails; requires DNSSEC

4. *DomainKeys Identified Mail (DKIM):* DNS-based policy that specifies how to handle malicious messages; requires DNSSEC

5.

It's common for these approaches to be used together, but they still have significant gaps.

End-to-end encryption (alongside digital signatures) addresses the limitations of the approaches above. It is a must-have for modern email systems. It's impervious to most attacks, but the challenge here is a proper PKI system.

> ### Example 6.6.1 ▶ Email PKI: S/MIME
>
> - Based on Enterprise PKI; used mostly in enterprise
> - Cross-certification is possible but rare in practice
> - Deployments work well within companies/organizations, but poorly everywhere else

> ### Example 6.6.2 ▶ Email PKI: Pretty Good Privacy (PGP)
>
> - Based on User-Control Trust Model
> - Everyone uploads their public key to some key server (e.g. MIT key server) for a key-signing party
> - Widely considered the best if key management is done right
> - Includes key servers and trusted interface where trusted introducers can vouch for new keys.

> ### Example 6.6.3 ▶ Email PKI: Identity-Based Encryption (IBE)
>
> - Based on Single-CA domain, but adds a trusted key server
> - Uses the string of the email address as the public key; inherently binds subject to public key
> - Supports attribute-based encryption (ABE)

## Review

- **What cryptographic guarantees are provided by TLS?**
  Confidentiality and integrity is guaranteed. The server is (almost) always authenticated, but the client is only sometimes authenticated.
- **Is it possible for an attacker to get a legitimate certificate?**

Yes! An attacker can defeat the CA's identification process (which can be widely varied), or they can register a copy-cat domain like amozon.com. Since they actually own that domain, they can get a certificate for their copy-cat domain.

- **What attacks is email vulnerable to?**

Steal unencrypted emails at rest, steal emails in transmission (eavesdropping), forge messages ————————————————————————————————— more here

# Web and Browser Security

## 7.1 Introduction

> **Definition 7.1.1 ▶ Domain Name**
>
> A **domain name** is a human-readable name for an internet service.

For simplicity, we will assume that a domain name can only have one associated service/server, and vice versa (i.e. a one-to-one mapping between domain names and servers).

> **Definition 7.1.2 ▶ Domain Name System (DNS)**
>
> A **Domain Name System** maps domain names to their corresponding server's IP address. We say the DNS **resolves** a domain name to mean the DNS converts a domain name into an IP address.

It's common for computers to store the IP addresses for one or two DNS services. By default, a DNS has no integrity (that is, an attacker on the network could ). DNSSEC solves this issue but has low adoption.

what could they do tho

> **Definition 7.1.3 ▶ Top-Level Domain (TLD), Second-Level Domain, Subdomain, Fully-Qualified Domain Name (FQDN)**
>
> Consider the following domain name:
>
> $$\texttt{userlab.utk.edu}$$
>
> - edu constitutes the **top-level domain (TLD)**
> - utk constitutes the **second-level domain**
> - userlab constitutes the **subdomain** which provides further naming specification for the server. It can have several "levels" and is assumed to be controlled by its parent (sub)domain.
> - userlab.utk.edu consitutes the **fully-qualified domain name (FQDN)**, which is

> sent to the DNS to be resolved into an IP address
>
> In addition, a domain name is the combination of a second-level domain and a top-level domain.

Machines typically parse domain names from right to left, starting from the most broad identifiers and working towards the most specific ones.

> www is a common subdomain which is sort of an artifact of the internet's inception. In the early days, people wanted to separate the website from other services from the domain name. There is no guarantee that `www.website.com` hosts the same data as `website.com`. Now, most websites specify `website.com` to simply redirect to `www.website.com`.

### Technique 7.1.4 ▶ Resolving Domain Names

Suppose we have some servers with domain name `amazon.com` and Alice who wants to connect to the servers.

- Alice connects to her intermediate access point that actually connects to the internet (most commonly a router).
- Alice prompts the DNS for the IP to `amazon.com`.
- The DNS resolves the IP address and gives it to Alice.
- Alice connects to the resolved IP address.

### Definition 7.1.5 ▶ Uniform Resource Locator (URL)

A ***URL*** identifies specific resources on a server. This contrasts from a domain which only specifies which server to connect to.

### Definition 7.1.6 ▶ Scheme, Host, Port, Pathname, Query

Consider the following URL:

$$\texttt{scheme://host[:port]/pathname[?query]}$$

The square brackets denote optional fields.

- The ***scheme*** is the protocol used to communicate with the server (e.g. http, https, ftp, ftps, ws, etc.).
- The ***host*** is usually the domain name or IP address of the server.
- The ***port*** specifies the port to connect to. If omitted, it uses default values for known

schemes (e.g. 80 for http, 443 for https).
- The *pathname* identifies a specific resource on the server. It resembles a UNIX file system path, but it does not actually need to map to the underlying file system.
- The *query* is addition data that further specifies the requested resource. It always starts with ? and is a collection of key-value pairs, separated by &.

When encoding URLs, there are only a certain set of characters that is allowed. Non-allowed characters are encoded as their hexadecimal representation prepended with a %.

---

**Definition 7.1.7 ▶ Hypertext Transfer Protocol (HTTP)**

*HTTP* is a common request-response protocol for serving web content. It is built on TCP (ensures reliable data transfer) and is a stateless protocol (that is, connections are all independent of each other at the protocol level; no data is stored about the request itself).

---

HTTP was originally designed to only serve Hypertext Markup Language (HTML), but is now used to serve general content.

---

**Definition 7.1.8 ▶ HTTP Methods**

HTTP requests are sent similar to the following string:

METHOD /pathname?query HTTP VER Host:

hostname:port $< keyword >:< value >< requestbody >$

> format this properly like an actual HTTP request

- *GET:* retrieve data from the URL
- *POST:* push data to the URL
- Others include HEAD, PUT, DELETE, TRACE, CONNECT, OPTIONS, PATCH
- The *request body* is the data needed to fulfill the request.

An HTTP response follows the generic form:

HTTP/version status_code <keyword>:<value> <response body>

- *Status code:* 100s informational, 200s successful, 300s redirection, 400s client error, 500s server error

---

> - ***Response Headers:*** metadata about response in key-value pairs, separated by \r\n
> - Response body: data returned from the request

**HTTP Response**

Although the HTTP spec specifies the intended uses of each method, servers can implement and handle these methods however they choose. So there is generally no standardization of these methods between different servers.

---

**Definition 7.1.9 ▶ Hypertext Markup Language (HTML)**

***HTML*** is a markup languages based on the Standard Generalized Markup (SGML). An HTML document is composed of elements described using ***markup tags***, following a tree structure where each element can have a parent, sibling, or children elements.

---

In 1998, SGML was reworked into XML, after the creation of HTML. HTML and XML are very similar, but not the same.

---

**Definition 7.1.10 ▶ HTML Element**

```
<tag [attribute='value']>[content]</tag>
```

- Tag: the name of the element, such as p for paragraph or h1 for header.
- Attribute list: each element defines a set of attributes with well-defined behavior; HTML also supports arbitrary attributes but without well-defined behavior (up to your own implementation)
- Content: can be composed of other elements (i.e. children), textual content, or a combination of the two
- Closing tag: Only tags that can have contents need to be closed

---

**Example 7.1.11 ▶ HTML Document**

- The html tag is the root element
- head contains metadata elements
- body contains the webpage's contents
- Common content tags are p (paragraph), h1 through h6 (headers), ol (ordered list), ul (unordered list), etc.
- HTML 5 now supports ***frames*** which are used to display another entirely different

---

HTML document.

The `script` tag can be used to add client-side scripts to add interactivity to the webpage. These scripts are almost exclusively written in JavaScript.

> **Definition 7.1.12 ▶ HTTPS**
>
> *HTTPS* is simply the HTTP protocol using TLS.

What happens if multiple domains are served by the same IP address?

- This is referred to as *name-based virtual hosting*

- For HTTP, the `host` header will identify how to route the request

The problem when usnig HTTPS is to figure out which certificate to provide when establishing TLS with the client.  Also, what about routing after TLS is established?  The HTTP request is encrypted and the host header will not be reachable.

> **Definition 7.1.13 ▶ Server Name Indication (SNI)**
>
> *Server name indication (SNI)* is a TLS extension which allows clients to indicate which domain they want to communicate with.

All HTTPS requests in this TLS session will be routed to that domain.  The downside here is that SNI clearly reveals who the user is communicating with, and it often breaks many corporate security tools.  More recently, people have developed encrypted SNI where the client hello is encrypted using a public key for the domain.  This public key is hosted in DNS but requires DNSSEC to be secure.

> **Definition 7.1.14 ▶ Document Object Model (DOM)**
>
> The *document object model (DOM)* is the computer's interpretation of an HTML document which is used to actually render the HTML document. The DOM can be modified by scripts and will be immediately reflected.

The DOM is composed of several things, including:

- The HTML document itself
    - `document, document.head, document.body`
    - `document.location`
    - `document.scripts`

- `document.getElementById()`
- `document.querySelector()`, `document.querySelectorAll()`
- The window displaying the document
  - `window`, `window.document`
  - `windodw.alert`
  - `window.cookieStore`
  - `window.localStorage`, `window.sessionStorage`
  - `window.location`
- The console, typically used by developers
- `fetch()`: used to retrieve new content from the server; much of the modern web is based on `fetch()`, starting by sending a bare-bones webpage and fetching content as required

> **As a developer, never trust client-side code.** Clients can arbitrarily access, modify, and injects scripts. Big takeaway: **important data should never be stored on the client.**

**Sessions**   Originally, the web was purely static HTML webpages. As such, each request was independent of one another. At most, the `referer` header was used to link requests. In order to create more fully-featured web applications, web developers have created mechanisms for storing session data to be associated with HTTP requests, allowing HTTP requests to be linked to users.

When linking sessions to HTTP requests, we need to ask ourselves, "what session data should be stored by the server? The client?" Sessions are located by the server using a *session ID*. This identifier is used to lookup session data for a connection.

How does the session know which session ID to use for each request? HTTP is a stateless protocol. Originally, each HTML document was sent in its own TCP connection. As such, it's not possible to bind the session ID to the connection. While HTTP/1.1+ allows many HTML documents to be sent in a single TCP connection, the stateless nature of the protocol was preserved.

> Instead, each HTTP request needs to contain the session ID.

> **Definition 7.1.15 ▶ Cookies**
>
> *Cookies* refer to data stored on the web client that is used for additional functionality and/or as a means of identifying users. They usually exist between webpages and sessions on the same website.

Cookies are the standardized approach for sending session data with each HTTP request. Cookies are sent by the server in an HTTP response, and all future HTTP requests will automatically include the cookies sent by the server. Cookies can also be created, read, updated, or deleted by client-side scripts.

In an HTTP response, the set-cookie header may look something like:

```
Set-Cookie:sessionID=78ac63ea01ce23ca; Path=/; Domain=mystore.com
Set-Cookie:language=french; Path=/faculties; HttpOnly
```

As we can see, it is composed of name-value pairs separated by semicolons. Optional attributes include `domain`, `path`, `max-age`, `Secure`, `HttpOnly`, `SameSite`.

In an HTTP request, the cookie header may look something like:

```
Cookie: sessionID=78ac63ea01ce23ca; language=french
```

This includes all cookies in a single header. Once again, fields are separated by semicolons. These cookies are only sent if:

1. they are for the correct domain and path,
2. they satisfy the `Secure` and `SameSite` attributes, and
3. they have not expired.

## 7.2 Same Origin Policy

> **Definition 7.2.1 ▶ Origin, Same-Origin Policy (SOP)**
>
> An ***origin*** refers to the scheme (e.g. HTTP), host (fully-qualified domain name or IP address), and port. The ***same-origin policy*** guarantees that content from one origin will have limited access to content from another origin.

For example, `http://site.com/asdf` and `http://site.com:80/asdfzxcv` have the same origin.

> The same-origin policy is widely regarded as the most important security feature that web browsers provide.

For example, same-origin policy is applied to:

- **Frames:** scripts in one frame are only allowed access to the DOM of another frame if they share the same origin.

- **Web requests:** scripts from one origin are only allowed to make `GET`, `POST`, and `HEAD` requests to another origin, and even then have limited headers and cannot view the response bodies. Some safe headers include `Accept`, `Accept-Language`, and `Content-Language`. Some safe `Content-Type` values include plaintext and form data.

**Loosening SOP Protections**   There are some circumstance to loosen SOP protections. Regardless of the intent, extreme precaution must be taken when doing so.

- `document.domain` can be used to override the fully-qualified domain name in the origin. This allows pages from two distinct subdomains to change their domain to the same parent domain, allowing cross-page domain access. This feature is now deprecated.

- Channel messaging API allows frames to send messages to each other. The sender explicitly identifies the domain(s) they send messages to. The receiver explicitly identifies the domains they are receiving from.

- **Cross-origin resource sharing (CORS):** A header in the HTTP response indicates origins that are allowed to access the response body; can also allow additional methods and header options.

## 7.3   Cookie Attacks

If an attacker can steal a user's session identifier cookie, the attacker can use that cookie to impersonate the user! Methods include:

- Intercepting HTTP traffic; can be stopped by using TLS and/or setting the Secure attribute on cookies.

- Malicious client-side scripts: supply-chain threats make this threat increasingly common; can be prevented by setting the `HttpOnly` attribute on cookies.

- Client-side malware: malware can attack the browser and steal the cookies, either out of memory or on the disk where they are stored.

Let's suppose an attacker can't directly steal the cookies. They might still try to misuse them instead. Recall that whenever an HTTP request is made, cookies are automatically attached to that request. The browser sends these cookies. We can trigger a web request using `fetch()` to request a resource from any origin!

> **Definition 7.3.1 ▶ Cross-Site Request Forgery (CSRF)**
>
> *Cross-Site Request Forgery (CSRF)* refers to a common online attack. The user connects to a malicious website which executes a web request for the legitimate website. The browser will attach the legitimate site's cookies to this request.

Such an attack has the following steps:

1. A user connects to a legitimate website and has a cookie.

2. The user unknowingly visits a malicious website or clicks on a malicious link.

3. The malicious website contains a script that generates a request to the legitimate website.

4. The browser, unaware of the malicious intent of the request, automatically attaches the cookie from the legitimate website to this request.

5. The request is sent to the legitimate website. Because the request includes the user's legitimate session cookie, the website assumes the request is from the authenticated user and processes it.

6. The malicious site thus can perform actions on the legitimate site on behalf of the user, without their knowledge or consent. This could include actions like changing the user's email address or password, performing transactions, or any other action the user is authorized to perform on the legitimate website.

There are several methods for defense against CSRF attacks:

- Limit what can be done with the `GET`, `POST`, and `HEAD` requests; these are the only requests allowed by SOP, but can be overridden by CORS headers

- Set the `SameSite` attribute on the cookies; will only attach cookies originating from the domain that set the cookie

- Use an anti-CSRF token which itself is a cookie. Whenever the user triggers a sensitive action, include the anti-CSRF token in the request. Then the server only allows the operation if both values match.

Another approach is *cross-site scripting*. The vulnerability lies in the fact that websites often include user input in their output. This can allow attacks to inject malicious scripts.

> **Definition 7.3.2 ▶ Cross-Site Scripting (XSS)**
>
> ***Cross-Site Scripting (XSS)*** refers to a method of online attack where an attacker finds a means to get a user to run malicious code by visiting some link or website.

These attacks can result in varying outcomes including stolen cookies, stolen data stored in the browser (e.g. `LocalStorage`), redirecting the browser to an attacker-controlled domain, and/or modifying the document itself.

*Reflect XSS:*

1. An adversary includes some malicious code or script to an otherwise legitimate link

2. A user clicks on the link, where the malicious script runs while connected to the legitimate website

This requires the user to click an attacker-controlled link. This is not effective if the user directly visits the website. In addition, this only impacts a single user. A common approach use a URL shortener to hide the malicious content from the URL. Also, it's common to load some arbitrary image with `onLoad="malicous code"` as well as some code to reload the page to hide the malicious code from the URL.

*Stored XSS:*

1. The adversary directly connects to the legitimate website and uploads some malicious code (perhaps through a POST request)

2. The website stores the adversary-owned script in their database

3. A user connects to the legitimate website and sends a request to get relevant items from a database (perhaps comments on a post/thread)

4. The website sends the adversary-owned script to the user, where it is then run on the user's machine.

This does not require any error on the user's part, and works even if the user directly visits the site. Moreover, this impacts many users. Reflected XSS attacks can be used to set up Stored XSS attacks

To address XSS attacks, we never user functionality that can allow an attacker to inject malicious content. Specifically, we avoid setting the `innerHTML` attribute and avoid calling `eval()` on user input. It's best to rely on frameworks that sanitize and filter malicious content before processing user input.

## 7.4   SQL Attacks

SQL is a language used to query data in a database. SQL injection attacks happen when user input is directly included inside SQL statements executed on the database. It's similar to XSS where the attacker injects malicious code.

SQL itself stores data in ***relational databases*** which can be better thought of simply as tables. Each table has a ***schema*** that defines the columns or fields that make up the tables. In terms of OOP, tables are like classes and each row is an instantiation of that class.

Missing a lot of notes

Allows for arbitrary access to the SQL interface which means the attacker can do anything they want.

# Firewalls and Tunnels

## 8.1   Networking

> **Definition 8.1.1 ▶ Datagram, Packet**
>
> Networked data is delivered as a series of ***datagrams*** (or ***packets***) which are each composed of:
> - a ***header*** that provides information to help route the datagram, and
> - a ***payload*** that contains the actual data transmitted.

If a datagram is too large, it can be broken into fragments.

### Layer 1: The Physical Layer

This is the physical connection between computers that actually sends data between computers. This is nearly always a CAT cable that uses copper wire to transmit electrical signals representing bits. Different cat cables provide different levels of speed.

### Layer 2: Data Link

The data link defines how data is transferred over a physical medium. This converts layer 2 datagrams into electrical signals. Most devices use the ethernet protocol where devices are identified using a MAC address.

### Layer 3: Network Layer

> **Definition 8.1.2 ▶ Packet-Switched Networking, Hop**
>
> ***Packet-switched networking*** refers to a networking protocol where packets are sent via a series of connected devices. Each stop on the pathway between sender and receiver is called a ***hop***.

This means routes are not negotiated beforehand! Every packet has to "find" its route to its destination.

> The `traceroute` program can be used to see the hops between your device and the specified server.

> **Definition 8.1.3 ▶ Internet Control Message Protocol (ICMP)**
>
> *ICMP* is a protocol used to communicate among devices on the same network about problems with data transmission

## Layer 4: Transport Layer

> **Definition 8.1.4 ▶ TCP**
>
> *TCP* simulates a connection over a packet-switched network for bidirectional communication. It provides reliable and in-order packet delivery at the cost of some performance.

TCP sockets are bound ot individual connections defined by the tuple (local IP, local port, remote IP, remote port). Servers have a well-defined server-side port, but clients usually use a temporary/random port for their side of the connection. Servers accept connections using a server socket which results in the creation of a normal TCP socket for each client connection.

To establish a connection, a three-way handshake is used:

1. `SYN` packet sent by the client

2. `SYN-ACK` packet sent by the server acknowladging the first packet

3. `ACK` packet sent by client acknowledging the second packet

To confirm that data is received by the other side, all packets will set the "acknowledged" flag, denoted `ACK`. Any "holes" in the information are filled by the TCP protocol.

> **Definition 8.1.5 ▶ UDP**
>
> *UDP* is a connectionless, one-directional network protocol that does not guarantee reliability or in-order packets.

### Layer 5: Application

This is where the application handles the data.  Many have multiple sublayers (e.g.  TLS is layered below HTTP to create HTTPS).

### IPSec

Initially, IP was created without security in mind.  It did not guarantee confidentiality, integrity, nor authentication.  *IPSEC* was initially designed to secure IPv6 and was later backported to IPv4.

## 8.2   Firewall

> **Definition 8.2.1 ▶ Firewall**
>
> A *firewall* is some mechanism that inspects traffic entering of leaving a device in the network.  It can filter *inbound packets* to protect against external adversaries, and it can filter *outbound packets* to protect against malicious insiders.

### Packet Filters

The most common type of firewall is a *packet filter*, which filters traffic at a packet-level based on a list of rules.  The most common actions are ALLOW, DROP (don't allow the packet), or REJECT (drop and notify the sender).  In practice, REJECT is rarely used.

Some limitations include:

- Requires a true perimiter (can be fixed by zero-trust networking)

- Does not protect against lateral movement within the network

- Does not protect against malicious content from a benign connection (e.g.  cross-site scripting attack)

- Can be circumvented by tunneling malicious traffic through a benign connection

### Proxy Firewall

A *proxy firewall* is a proxy that is an intermediary connection between two hosts.  It can then pass all traffic to an app-level filter for inspection.  It can view all traffic, including encrypted

traffic. In addition to allowing or denying traffic, it can even modify traffic to possibly add or remove functionality. To any user, the proxy can be entirely transparent.

## 8.3   Tunnels

Tunneling is the process of transporting packets from one protocol and transmitting them using another protocol. So far, we've covered TLS and will soon explore SSH.

### Secure Shell Protocol (SSH)

> **Definition 8.3.1 ▶ SSH**
>
> **SSH** is an application-layer protocol that adds confidentiality and integrity to other application-layer protocols.

At a high level, the remote host runs an SSH server which relays all inputs to some program (most commonly the server's shell). All traffic goes through an encrypted tunnel. A single SSH connection is **multiplexed**, meaning it can be used to handle multiple protocols. It also supports port forwarding (e.g. if a server hosts a website on port 8080, we can use SSH to redirect all traffic from our port 8080 to the server's port 8080).

When establishing an encrypted tunnel, we always want to authenticate the server. Each server has a unique SSH host key which identifies the system.[1] Most of the time, the client validates the key by trust on first use (TOFU). Then, if the host's SSH key has changed, the connection is rejected.

The server also needs to authenticate the user, most commonly by password and/or public key. Passwords are not the preferred method as they are susceptible to phishing (if connected to the wrong server) and are generally less secure. Much more secure is the public-key, which conducts a challenge-response protocol to verify ownership of the private key. After the first login, the user's public key is listed as an authorized key on the server. The user's private key should be password-protected on the client.

### Virtual Private Networks

Traditionally, computers are physically networked together. The boundary to a network is well-defined, and computers within that boundary are considered trusted. Incoming and out-

---

[1]It's also possible to use PKI for SSH host keys, although rarely used in practice.

going traffic across that boundary is monitored in our firewall.

Obviously, the limitation is that the computers have to be physically connected. Instead, we can tunnel all computer networking through a remote network.

> **Definition 8.3.2 ▶ Virtual Private Network (VPN)**

VPNs do not protect against malicious insiders nor lateral movement in the network. As such, VPNs are slowly being replaced by *zero-trust networking* which works using a whitelist. All connections are distrusted by default unless explicitly authenticated.

# 9 WLAN

## 9.1 Introduction

> **Definition 9.1.1 ▶ Wireless Local Area Network (WLAN)**
>
> **WLAN** is simply a network that can be accessed wirelessly.

WLANs are accessed using the Wi-Fi[1] protocol. Wi-Fi itself is a data link layer protocol like ethernet (layer 2).

**Components**  The mobile station (STA) connects to a network using Wi-Fi. The STA connects to the network by way of an ***access point (AP)***, often part of the router. The access point is identified by a ***service set identifier*** (SSID) which we know as the network name. These are not intended to be secret.

**Connecting to an Access Point**  In this context, we can think of the mobile station (STA) as the "client" and the access point as the "server".

1. The STA probes nearby access points; this is used by the STA to determine which AP they will connect to.

2. Low-level authentication, which has since been deprecated but is still engrained in the standard. Devices now just send null responses, making this step more of a handshake than any form of authentication.

3. Association with the AP to establish the connection, so both devices know to listen for each other's communications.

4. High-level authentication: Authenticates the STA to the PA. It may also authenticate the AP to the STA. This step is omitted for open networks.

---

[1]Wi-Fi isn't an acronym, but rather a reference to the IEEE 802.11 standard

**Frames and Frame Types**    In the context of Wi-Fi, *frames* are units of data transmission or packets. They carry information necessary for devices on the network to understand data being sent and how to process it. Frames are divided into three distinct types:

1. *Data Frames* carry upper-layer protocol data, similar to an ethernet frame.

2. *Management Frames* ensure connections maintain basic service guarantees. They handle new connections between STA and AP, handle handovers between APs as STAs physically move, and handle disconnections.

3. *Control frames* communicate data and management frames and are the lowest-layer frames

## 9.2   WLAN Threats

WLAN has many of the same vulnerabilities as a physical LAN. However, it is more vulnerable as an attacker does not need physical access to the network. They would only need to be within proximity and attack devices. The issue is further exacerbated as Wi-Fi is a Broadcast Protocol—any device within proximity can receive, record, and inject Wi-Fi packets. Modern Wi-Fi supports beam forming (one-way communication), but this is only designed as an efficiency benefit and should not be relied on for security.

**Rogue AP**    A rogue AP attack establishes a copycat AP used to create a man-in-the-middle connection. This attack is only possible when there is no mutual authentication between the STA and AP.

1. The attacker sets up an AP with the same SSID as the legitimate AP. Ideally, the STA should see that the rogue AP should have a stronger signal than the legitimate AP.

2. The attacker interrupts the STA's connection. This can be done by sending a `disasso-ciate` frame to the STA. The broadcast nature of Wi-Fi makes this easy to perform.

3. The STA probes the APs and finds the rogue AP that has stronger signal than the legitimate AP.

4. The STA connects to the rogue AP, and a man-in-the-middle connection is established.

**Session Hijacking**    This attack is only possible when session encryption is not used.

1. The attacker interrupts the STA's connection, but the attacker ensures the message isn't seen by the AP.

2. The attacker spoofs the MAC address of the STA and takes over the legitimate STA's
   session.

**War Driving**    War driving involves scanning radio channels for in-range wireless networks.
The attacker often uses high-power omnidirectional antenna while literally driving around to
maximize the search radius. This can be used for:

- *Reconnaissance:* SSIDs may reveal a lot about a building's occupants, or they can be
  used to steal material to perform a brute force attack for the AP's authentication creden-
  tials.

- *Monitoring communication*, which is especially easy if the networks are unencrypted.

- *Manipulating packets:* It's possible to override a client's packet with an attacker's packet
  if the attacker can generate a stronger signal. It is also easy to perform a denial of service
  to the WLAN.

**Open Networks**    Open networks omit any form of authentication, allowing for any device
to connect. As such, packets are neither encrypted nor authenticated. Any confidential com-
munication over an open network has to rely on end-to-end encryption such as TLS. Conse-
quently, open networks are highly susceptible to rogue AP attacks.

Some open networks authenticate users by a login page (e.g. hotel Wi-Fi). However, this hap-
pens over the HTTP protocol and is not part of the Wi-Fi protocol itself. Packets are still neither
encrypted nor authenticated. This kind of authentication is susceptible to session hijacking.

Note that, even with end-to-end encryption, the DNS protocol is still plaintext and thus highly
vulnerable.

**Wired Equivalent Privacy (WEP)**    WEP encrypts all traffic using the RC4 stream cipher
with a 40-bit key and 24-bit IV. This is limited due to export restrictions, where the United
States didn't want to give away strong cryptographic schemes. However, in this configuration,
it is highly insecure.

WEP verifies integrity by using CRC-32 checksum. This protocol is highly susceptible to col-
lisions, so it isn't a cryptographically secure protocol.

WEP uses Open System Authentication (OSA) which allows anyone to connect to the system.
A random RC4 key is assigned to each connection.

A pre-shared key (PSK) between the STA and AP may be used for authentication (usually just a password). The RC4 key is based on the pre-shared key, which is the same for all clients. This renders it less secure than just using OSA!

> Ultimately, WEP is extremely insecure and should not be used.

## 9.3 EAP, Radius, WPA3

**Authentication Server (AS)**    To improve security, we can move high-level authentication to its own service. The AS can be integrated into the access point, or it can be a separate RADIUS server.

---

**Definition 9.3.1 ▶ Extensible Authentication Protocol (AEP)**

*EAP* provides a framework for implementing Wi-Fi authentication between STA and AP, and between AP and the AS. It does not specify how authentication happens, rather only facilitates how the communication happens between devices.

---

EAP can be used to implement many authentication protocols. It is now considered broken, so most networks now use Protected EAP (PEAP).

---

**Definition 9.3.2 ▶ Remote Authentication Dial-In User Service (RADIUS)**

*RADIUS* is a client/server protocol that:
- can tunnel EAP messages,
- provides authentication, authorization, and auditability,
- access request outcomes (accept, reject, challenge)

---

For example, Eduroam can use a specific university's RADIUS server to implement arbitrary authentication methods, and decouple authentication from the access points themselves.

---

**Definition 9.3.3 ▶ Wi-Fi Protected Access (WPA)**

WPA integrates AS, PEAP, and RADIUS to create a protocol more secure than WEP by:
1. significantly increasing the key length,
2. giving each packet a different key (called **key rotation**), and
3. using a MAC to protect integrity.

---

It's important to note that WPA1 and WPA2 are broken protocols, and only WPA3 should be

used. To see why, let's break down the three protocols:

1. WPA1 uses the RC4 stream cipher for confidentiality with a 128-bit key. The key is derived using the Temporal Key Integrity Protocol (TKIP) and is changed for every packet. It uses a MAC for integrity. However, flaws were discovered due to WPA's backwards compatible design.

2. WPA2 uses the AES stream cipher in CTR mode for confidentiality. It uses AES CBC-MAC for integrity. However, it is partially broken by the KRACK attack.

3. WPA3 uses AES in GCM mode for confidentiality with a 192-bit key. It uses an HMAC with SHA-384 for integrity. Simultaneous Authentication of Equals (SAI) is used for password authentication, which provides more security against brute force attacks than pre-shared key exchange (PSK). In addition, SAI provides forward security, and WPA3 protects management frames.

# Blockchain

Nowadays, we typically associate the term "blockchain"

## 10.1    Introduction

---

**Definition 10.1.1 ▶ Blockchain**

*Blockchain* is a database technology with three key properties:
1. Cryptographic append-only ledger, which stores the full history of all transactions
2. Replication
3. Distributed operation (i.e. decentralized)

---

**Cryptographic, append-only ledger**    A blockchain's ledger uses cryptography to guarantee its integrity. It does so via an *Authenticated Data Structure (ADS)*, where modifications to the data can be detected (but does not necessarily prevent modification of the ADS). An ADS produces a *verifier* that can be used to verify the data hasn't been changed since the verifier was produced. This requires access to the ADS to verify its integrity. In some ADS systems, the verifier can also be used to create proofs of inclusion and non-inclusion. In this case, access to the ADS is not required.

---

**Definition 10.1.2 ▶ Hash Chain**

A *hash chain* is an authenticated data structure based on singly linked lists where each node stores:
- its data,
- a pointer to the previous item, and
- a hash value calculated based on the value of the current item and the hash value of the previous item; the first item uses a hard-coded value, and is sometimes called the *genesis item*

---

Items in the hash chain that store transactions are referred to as *blocks*, hence the term *blockchain*.

**Replication**    Multiple entities store the blockchain ledger in its entirety as well as the verifier. If one entity modifies their ledger, the others can detect that change because their verifier values will no longer match. If the modification was malicious, the modified ledger can be restored from the replicated copies.

**Distributed Operation**    Multiple entities operate the system, often referred to as ***miners***. Each miner replicates the full blockchain, operating individually but verify and replicate each other.

To add a new block, miners undergo the following procedure:

1. An individual miner will first add a new block to their personal copy of the blockchain. The miner generates a list of transactions to add to a blockchain. They verify the legitimacy of those transactions and then create a block with those transactions. That miner adds the block to their copy of the blockchain.

2. Next, that miner announces the new block to the other miners. Each other miner verifies the legitimacy of the transactions in the new block. The other miners then add the new block to their copy of the blockchain.

In public blockchains, anybody can be a miner. Each miner votes on which blocks should be added. This requires a mechanism to fairly allocate votes.

In permissioned blockchains, the identities of the miners are defined by the system. Blocks are added directly without voting. Hence, it is more efficient than open operation.

**Miscellaneous Features**    Some blockchains also include:

- Smart contracts: algorithms stored on the blockchain that are run by miners when triggered by certain transactions. These can change the state of data on the blockchain and are subject to validation like transactions. They're commonly used to automate contractual processes.

- Non-fungible tokens (NFTs): cryptographic tokens representing physical or digital assets

**Challenges**    Blockchain systems are severely limited in terms of:

- Scalability: blockchain systems add significant overhead.

- On-chain correctness: it's hard to ensure that the digital blockchain fully represents the real world. What about counter-party risk, and how should bugs be handled when everything in a blockchain is permanent?

- Regulatory compliance: often, performance benefits of blockchain come from ignoring regulation

- Security and privacy: blockchain systems can have software vulnerabilities; public blockchains can be attacked by malicious entities

- Usability: users need to manage cryptographic keys, but key management is hard. Smart contracts need to be perfect, but development tools can't verify this.

## 10.2  Bitcoin

*Bitcoin* is a decentralized, pseudonymous cryptocurrency. A blockchain is used to store bitcoin transactions.

Each transaction stores the following information:

- List of inputs: the bitcoin that will be burned

- List of outputs: The bitcoin that results from the transaction, which is less than or equal in value to the input bitcoin

- Transaction hash: generated based on the transaction's data

**Proof of Work**    To add a block, miners race to create a block with a valid hash. What if two transactions try to spend the same bitcoin? The miners get to vote, with voting power determined by computational power. This is referred to as *proof of work*. For public blockchains, proof-of-X protocols are used to fairly allocate votes within governance and operation.

## 10.3  Blockchain Myths

**"Blockchain is always public."**    Blockchains can be public. Openness comes with additional overhead, such as the proof of work algorithms. Permissioned blockchains limit who can participate. Membership can be static or dynamic, and the ledger itself can be public or private.

**"Blockchain is fast and efficient."**    Blockchain has immense overhead. Ledgers store the entire history of every bitcoin transaction, and replication itself has a high storage cost. Decentralized operation is much slower than centralized operation. It's not the technology that's faster; it's the ignoring of regulation that's faster.

Systems using blockchain often claim to have increased efficiency. For example, many claim that it is faster to send money internationally using Bitcoin rather than through a traditional bank. This efficiency usually comes from those systems ignoring regulations and/or compliance. For example, Visa's network is much faster than Bitcoin for sending money, but that money won't be sent by Visa until proper regulatory compliance steps are taken, while Bitcoin payments ignore regulation.

**"Blockchain is immutable."** The ledger is in fact mutable. Note that miners themselves cannot change each other's copy of data, but miners can collectively agree to change the data. If only some agree, the blockchain can *fork* to create two blockchains with different ledgers. This is what happened with Ethereum and Ethereum Classic. These changes may be detectable by non-miners, so long as the ledger is publicly available.

**"Blockchain is trustless."** Although blockchain is in fact decentralized, we have only shifted our trust from a centralized entity to the bitcoin miners. This is better, but we still need to trust the software behind bitcoin, trust the connection between on-chain and off-chain assets, and trust the miners to vote fairly in transactions.

**"Blockchain removes the need for trusted third-parties."** Trusted third-parties are still needed to mediate between the real-world and on-chain ledger. For example:

- currency exchanges that convert on-chain assets to real-world money

- product vendors that convert on-chain assets to real-world products

- entities that report real-world events on the chain

- entities that handle dispute resolution (e.g. miners)

**"Smart contracts are novel and innovative."** Smart contracts are just software attached to the blockchain database. Transactions provide smart contracts with input. Their output is used to update state stored on the blockchain. The new state is validated by the miners. Any database can have smart contract equivalents.

Admittedly, companies are using smart contracts in innovative ways. For example:

- Decentralized autonamous organizations (DAOs)

- Automated supply chain tracking

**"Blockchain is wasteful."**   In reality, it's only the proof-of-work algorithms which are wasteful. Public blockchains can simply use other proof-of-X protocols. Permissioned blockchains don't need to use proof-of-X protocols.

## 10.4   Use Cases

**Cryptocurrency**   Fiat currencies are managed by central banks. These are susceptible to inflation, seizure, and general poor management. Cryptocurrencies decentralize the management of currency, removing the need for a central bank. They have public, set-in-stone rules are in place for minting additional currency, so inflation is entirely deterministic.

**Electronic Payments**   Payments are being processed by just four corporations: Visa, mastercard, discover, and American Express. Government regulations can cause significant delays (e.g. international transactions can take days to clear). Blockchain payments should be near-instantaneous.

**Supply Chain Management**   We can track products on the blockchain, and automate contracts/payments using smart contracts. Transaction history provides a permanent record, allowing for things like root-cause analysis and taint tracking. Moreover, the permanent record supports conflict resolution. This approach also removes the need for centralized infrastructure (which is nice because many producers don't trust the man).

**Multi-Organization Data Sharing**   Blockchains can act as sharing mediators to help people find data, validate access to the data, and automate things using smart contracts.

**Voting**   This is actually a terrible idea. Physical voting is always preferred to digital voting.

# Usable Security

New security features should ultimately strike a balance between security and usability. We still want the user to be able to easily and quickly accomplish the intended tasks, all while maintaining security properties including confidentiality, integrity, authentication, etc. Although mistakes in usability are fine, mistakes in security can be catastrophic.

While there is natural tension between usability and security, it is wrong to think that these two are mutually exclusive. That is, it is wrong to think that secure systems are inherently unusable, or usable systems are inherently insecure.

The people in charge of

> If users aren't doing what you want, don't blame them, blame your software or system!

## 11.1 Designing with Users in Mind

As developers, we need to align ourselves with the users' needs. We can ask ourselves:

- What roles do they have?
- What are their critical tasks?
- What expertise can you expect?
- How much of a security budget do they have? (including mental, time, and financial budgets)

From this, we can match the system design to the users. We should never try to change the user to fit our contrived design. Taking this approach will increase the likelihood of user buy-in.

**Communicating Clearly**   We also need to give users just the right amount of information to figure out the software. Concrete examples will help in conveying new ideas. This also includes making notifications actionable, giving ample information to make a decision and the ability to execute that decision.

**Educating Users**    Users rarely have time to read nor effort to read lengthy instruction manuals. As such, software should make it trivial to become educated, whether it be via integrated tutorials or inline documentation. It is critical not to treat education as a panacea. It's better to improve your design than to rely solely on education.

**Avoiding Dangerous Errors**    It should be difficult for a user to make a dangerous error. We need safe default settings that make it easy to increase security and difficult to decrease security. The path of least resistance should lead to a secure outcome.

**Avoiding Too Many Alerts**    If we overwhelm the user with too many dialog boxes, the user will stop reading them and just habitually click through them. Hence, keep dialog boxes to a minimum.

## 11.2   Testing for Usability

**Developer Evaluations**    To help determine whether a product is usable on a commercial scale, there are several methods:

- *Expert review:*

- *Cognitive walkthroughs:* Walk through the steps that a first-time user would undergo when using the software.

- *User studies:* interact with users to understand their mental models, perceptions, and requirements (including surveys, interviews, ethnographies, telemetry, lab studies, A/B testing)

# Index

## Definitions

## Examples

## Techniques

## Code Snippets