Introduction to Cybersecurity

UT Knoxville, Spring 2023, COSC 366

Dr. Scott Ruoti, Alex Zhang

March 13, 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

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 legit-

imate 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);
```

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:
• Simple and easy to learn • Free for users • No physical load to carry • Usually easy to recover lost access • Easily delegated (e.g. sharing Netflix passwords)	• Hard to create random passwords (theoretical space is large, but practical space is small) • Hard to remember good passwords • Users often reuse passwords

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 pat-

terns 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	• Loss of functionality like reflection • Can be circumvented by using code from other parts of memory (called return-oriented programming) • Code snippets found in `libc` are Turing complete!
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.	• Can be worked around if the canary can be learned
ASLR: randomize the location of the stack	• Can be worked around using a NOP sled • Other vulnerabilties can be used can be used to locate the stack
Simply use the safe C library functions (e.g. `strlcpy` instead of `strcpy`)	• Requires competency from the developer to know about these functions and use them correctly • Easy to make a mistake and allow attacks to still happen
Just use another language that does bounds-checking on buffers, pointers, and type casting. These checks can happen at runtime or compile time.	• This can still be circumvented in rare edge cases

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 correlated 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 software 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 pornography 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

Index

Definitions

Examples

Code Snippets