Introduction to Cybersecurity

UT Knoxville, Spring 2023, COSC 366

Alex Zhang

March 4, 2023

Contents

1 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***: proof that an entity has the necessary privilege to take the request

action; most commonly done by authenticating a principal, and lookup up its privileges
- ***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".

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

Definition 1.4.1 ▶ Adversary

An ***adversary*** is an entity that wants to violate a security policy in order to harm an asset.

- ***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

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 1.4.2 ▶ 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 identifying 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 – get security out of the box; users rarely change defaults
3. Open Design – don't rely on the secrecy of our code; 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

1.8 Review

CIA Triad, Golden principles, trusted vs. trustworthy, confidentialty vs. privacy.

TODO: add more review stuff from slides here; flesh out notes to reflect review

2 Cryptography

Cryptographic Rules to Remember

1. Do not design your own cryptographic protocols or algorithms.
2. ***Kerckhoff's Principle*** – security should only come from the secrecy of the crytographic key, NOT the algorithm or code.
3. Encryption only provides confidentiality, not integrity.

2.1 Introduction

Definition 2.1.1 ▶ Cipher, Plain Text, Cipher Text

A ***cipher*** is any encryption or decryption algorithm. A ***plaintext*** message is vulnerable, usually prior to cipher. A ***ciphertext*** message is secure, usually after cipher is applied.

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. It should be large enough such that it would be impractical to try every possible key.

For example, AES-256 uses 256 bit keys. The key space of AES-256 is 2^{256}. 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.

Some possible threats to encryption may be:

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

Some possible adversaries include:

- Passive – just listens to communications
- Active – can modify data

Definition 2.1.3 ▶ Information-Theoretic Security

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

Definition 2.1.4 ▶ Computational Security

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

Definition 2.1.5 ▶ 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 Algorithm

A ***symmetric-key algorithm*** encrypts plaintext and decrypts ciphertext using the same cryptographic key.

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. The resulting ciphertext is information theoretic so long as the following are met:

- Key must be as long as the plaintext
- Key must be random
- Key must never be reused
- Key must be kept completely secret

Many modern cryptographic algorithms work by taking a small cryptographic key and expanding 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, providing computational security.

- 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

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 ▶ Basic RSA Encryption using C#

```
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 verify 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

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 digital signatures.

2. There can be mathematical interactions between encryption and signing that can 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 hashed before signed, and 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 an arbitrary length string and outputs a unique fixed output length string.

$$H : 2^* \to 2^n$$

Usage:

- H can be applied to data of any size
- H outputs a fixed length data
- H is fast to compute

Security properties:

- ***One-way property***: Theoretically impossible to find the original message from the hash; one hashed string maps to an infinite amount of input strings
- ***Weak collision resistance***: Given an input, it's computationally impossible to find another input that produces the same hash.
- ***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, Tag

A ***message authentication code*** or ***tag*** is a short piece of information 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)
- Does NOT provide non-repudiation (at least two people will have the symmetric key)

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#

```
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

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 bits of the password, not necessarily the characters. 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 • No physical load to carry • Usually easy to recover lost access • Easily delegated (e.g. sharing Netflix passwords)	• Hard to create random passwords (theoretical key space is large, but practical key space is small) • Hard to remember good passwords • Easy to reuse 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

Passwords can be changed into cryptographic keys called ***passkeys***.

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

3.2 Password Authentication

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#

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

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

```
6
7  var salt = RandomNumberGenerator.GetBytes(32);
8
9  // hash for 100000 iterations
10 var hash = Rfc2898DeriveBytes.Pbkdf2(password, salt, 100000,
    ↪  HashAlgorithmName.SHA256, 32);
11
12 using BCrypt.Net;
13
14 // Returns a crazy string
15 var temp = BCrypt.Net.BCrypt.HashPassword(password);
16
17 // Should return true
18 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, pretty shit.

That's why account recovery should be the only way to restore an account. Some 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 authentication for multiple websites.

While SSO creates a single point of failure, most users already reuse the same password for multiple sites. If one of those sites has a breach, then the rest of their accounts are screwed. Why not just have one identity provider that handles authentication for multiple websites. It's easier for the user (don't have to create as many accounts), and in practice, most SSO 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 all of your interactions with the website as well as other metrics to determine if you're a human. If it's not sure, it makes you do the traffic light clicking thing.

3.4 Authentication Factors

Authentication factors are used to verify you are who you say you are. They rely on something 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.

4 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 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

The OS kernel mediates access to system memory, acting as a reference monitor. The kernel allocates memory using virtual addressing. The kernel also limits who can access what memory, preventing users/processes from accessing other memory. It also protects

OS memory and provides mechanisms for shared memory access.

Definition 4.2.2 ▸ Kenel Memory Segmentation

The kernel divides memory into 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 Filesystem Access Control

Definition 4.3.1 ▸ Filesystem

A ***filesystem*** is a hierarchical structuring of directories and folders. It maintains per-file meta-data, including permissions.

In the context of filesystems, subjects are users, groups or processes; actions are read/write/execute, modify metadata, etc.; objects are files, directories, etc.

A filesystem categorizes subjects as such:

- Users are identified by a user id (UID)
- Groups are identified by a group id (GID) (for role-based access control)
- Processes are identified by a process id (PID)

Definition 4.3.2 ▸ Special Actions

Actions involving the filesystem include basic ones like read, write, execute, and list folder contents. Some special actions include:

- 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)

Advanced actions include:

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

Example 4.3.3 ▶ Linux Access Control

Linux uses the ***user-group-others*** (UGO) permission model which uses 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 4.3.4 ▶ 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.

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.

Filenames are a reference to an inode. The inode is only deleted when all filename references to it are deleted.

Definition 4.4.3 ▶ Filesystem Links

Filesystem 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.

`chroot`: restricts filesystem access

- treats the specified directory as the root directory `/`
- ***chroot jail***:

wht is this

Definition 4.4.4 ▶ 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***, the policy administrator sets all the access control rules; usually tied to role-based access control (RBAC).

Definition 4.4.5 ▶ SELinux

Security-Enhanced Linux is a kernel module found in most Linux distributions. It defines access control for applications, processes, and files, providing attribute-based access control (ABAC).

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.

5 Software Security

5.1 TOCTOU Race

TOCTOU Race:

- Time-of-check (TOC): when you check whether an action is allowed
- Time-of-use (TOU): when you execute the action
- TOCTOU race: potential that action becomes disallowed between checking and executing the action; often caused by multi-threading allowing concurrent execution
- File-based TOCTOU
- TOC: Check if file exists; check if necessary permissions
- TOU: create/access file
- Although code is written in sequence, not guaranteeed to run in sequence (OS can interrupt our process)

How to fix tihs:

- Application logic race conditions: use appropriate locking (mutexes, semaphores, etc.)
- File-based race conditions: use operations that do both the check and action in one call; guarantees OS will run it right
-

What is this

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 signed, all operands are treated as signed
- 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

Firstly, we could mark the stack and heap as non-executable, preventing code injection attacks in the stack. This could cause a loss of functionality like reflection. This defense can be circumvented by using code snippets from other parts of memory, referred to as return-oriented programming. (code snippets found in `libc` are Turing complete!)

Definition 5.4.1 ▶ Stack Canary

Some architectures implement a ***stack canary***—a memory address squished between the return address and local variables. It is set to a random value for each function call, and the canary is checked before jumping to the return address.

The stack canary is an application of defense in depth. This can be worked around if the canary can be learned. This would require finding and exploiting a vulnerability to read the stack.

Definition 5.4.2 ▶ Address Space Layout Randomization (ASLR)

ASLR randomizes the location of the stack.

Now, if the attacker uploads a malicious payload to the stack, they don't know what to change the return address to. This again is an application of defense in depth. An attacker would have to find and exploit a vulnerability to read the stack. It can be worked around using a NOP sled. In addition, there are other vulnerabilities to let us know where the stack is.

We could simply **use the appropriate C library functions**. Many default C functions don't provide bounds checking for the sake of efficiency. There are equivalent functions that provide bounds checking. It does require competency on the developers' part, so it's still easy to make a mistake and allow an attack to still happen.

We could just develop in other languages that perform bounds checking for the developer. These checks may happen at runtime or compile time.This can still be circumvented in some edge cases not anticipated by the language designers.

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 conditions:

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
3. ***Trigger condition:*** 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 evolved 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 is run

2. Propagation strategy: propagates by tricking users into downloading and running the virus; often focused on social engineering
3. Trigger condition: controls when the malware's payload is deployed; 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 potentially 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 virus that hides itself as 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. Common examples include 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 decryption methods. More advanced, it could use a compiler to create new 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 the function call

- Inline Hooking: add a malicious payload to a legitimate function

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