B.Sc. (Hons) Computer Applications
Cyber Security Course number CABX)6003

Unit 4

by

Prof. Mohammad Ubaidullah Bokhari

Syllabus

1. Programs and Programming Unintentional (Non-malicious) Programming:

- **Oversights:** Common mistakes in programming that lead to vulnerabilities.
- **Buffer Overflow:** Exploiting memory overflows to execute malicious code.
- **Incomplete Mediation:** Insufficient validation of input data.
- **Time-of-Check to Time-of-Use (TOCTOU):** Exploiting timing discrepancies between validation and use of data.
- **Integer Overflow:** Manipulating integer values to trigger unexpected behaviors.
- **Unterminated Null-Terminated String:** Exploits due to improper string handling.
- **Race Condition:** Exploiting the timing of multiple processes to gain unauthorized access.

2. Web Application Security:

- **Web Vulnerabilities Scanning Tools:**
    - **Nikto:** Web server scanner.
    - **W3af:** Web application attack and audit framework.
- **Application Inspection Tools:**
    - **SQL Map:** Tool for automated SQL injection detection and exploitation.
    - **DVWA (Damn Vulnerable Web Application):** A platform for learning web security.

3. Core Defense Mechanisms:

- **Bypassing Client-Side Controls:** Techniques to override client-side validations.
- **Attacking Authentication:** Methods to breach authentication systems.
- **Attacking Session Management:** Exploiting sessions in web applications.
- **Attacking Access Controls:** Methods to bypass or manipulate access controls.

Understanding Non-Malicious Programming Errors

Non-malicious programming errors refer to unintentional mistakes made by developers during the software development process. These errors, while not intentionally harmful, can lead to serious security vulnerabilities if exploited by attackers. The increasing complexity of software systems and the need for rapid development contribute to such oversights, making secure coding practices essential in modern development environments.

These errors typically arise due to factors such as poor coding practices, inadequate validation mechanisms, lack of security awareness, or oversight in handling user inputs. Although they do not originate from malicious intent, their impact can be as severe as intentional attacks, leading to data breaches, unauthorized access, system crashes, or financial losses.

Why Are Non-Malicious Errors a Security Concern?

- Attackers actively seek these vulnerabilities to exploit them for unauthorized access.
- Errors such as buffer overflow, integer overflow, and race conditions can be leveraged to execute arbitrary code.
- Failure to validate user input properly can lead to serious security risks like SQL injection and Cross-Site Scripting (XSS).

Understanding and mitigating these errors is essential to ensuring robust, secure, and resilient software applications.

Common Types of Non-Malicious Programming Errors

Several categories of non-malicious programming errors contribute to vulnerabilities in applications. The most prevalent among them include buffer overflow, incomplete mediation, race conditions, integer overflow, and TOCTOU attacks. Each of these errors has unique characteristics, but they share a common theme: a lack of rigorous validation, control, or synchronization in handling data and system resources.

Below is an overview of the most frequently encountered errors in software security:

1. Oversights - Simple mistakes due to human error.
2. Buffer Overflow – Occurs when a program writes data beyond the allocated memory buffer, leading to corruption or unauthorized code execution.
3. Incomplete Mediation – Happens when input validation is insufficient, allowing attackers to manipulate data or execute unintended commands.
4. Race Conditions – Arise when two or more threads attempt to access shared resources simultaneously without proper synchronization, potentially leading to unauthorized actions.
5. Integer Overflow – Occurs when a numerical computation exceeds the storage capacity of an integer, causing unpredictable behavior or memory corruption.
6. TOCTOU (Time-of-Check to Time-of-Use) Attacks – A race condition where an attacker modifies a resource between the time it is checked and when it is used, leading to unauthorized modifications.

Each of these vulnerabilities poses a significant risk to software security and system stability if not properly mitigated.

1. Oversight

An oversight is a simple mistake or negligence in coding that can lead to security issues.

Common Causes:

- Human Error: Typographical mistakes in code.
    - Assumption Errors: Assuming data is always valid.
    - Poor Documentation: Misinterpretation of code behavior.
    - Lack of Code Reviews: Errors going unnoticed.
- Examples:
    1. Hardcoded Credentials:

```
username = "admin"
password = "password123"  # Weak and hardcoded password
```

2. Improper Input Handling:

```
const userInput = "<script>alert('XSS');</script>";
document.write(userInput);  // Leads to XSS attack
```

Security Impact:

- Exploitation of Vulnerabilities: Attacks like SQL Injection and Cross-Site Scripting (XSS).
- Data Breaches: Exposing sensitive information due to improper error handling.

2. Buffer Overflow

A buffer overflow occurs when a program attempts to store more data in a memory buffer than it can hold. This results in adjacent memory locations being overwritten, which can lead to program crashes, data corruption, or even arbitrary code execution by attackers.

How Does a Buffer Overflow Occur?

- The program does not check the size of input data before copying it into a buffer.
- User-controlled input exceeds the allocated buffer size, leading to memory corruption.
- Attackers craft malicious input to overwrite return addresses, redirecting execution flow to inject malicious code.

Example of a Buffer Overflow Vulnerability

```
#include <stdio.h>
```

```
#include <string.h>

void vulnerable_function(char *input) {

    char buffer[10];

    strcpy(buffer, input);  // No boundary check

}

int main() {

    char user_input[50];

    gets(user_input);  // Dangerous function

    vulnerable_function(user_input);

    return 0;

}
```

In this example, an attacker can input more than 10 characters, causing memory corruption.

Security Impact

- System crashes due to unexpected memory modifications.
- Arbitrary code execution, allowing attackers to take control of a system.
- Privilege escalation, granting attackers higher-level access.

Mitigation Strategies

- Use secure functions like strncpy() instead of strcpy().
- Implement boundary checks before writing to buffers.
- Enable compiler security features like Address Space Layout Randomization (ASLR) and Stack Canaries.

3. Incomplete Mediation

Incomplete mediation occurs when an application fails to validate user input adequately, allowing malicious users to manipulate data, bypass security measures, or exploit underlying system components. This type of vulnerability is one of the most common causes of injection attacks, including SQL injection and cross-site scripting (XSS).

How Incomplete Mediation Happens

- Input data is not sanitized before being processed.
- The application relies only on client-side validation, which can be easily bypassed.
- Insufficient or missing authorization checks for user privileges.

Example: SQL Injection Due to Incomplete Mediation

```
SELECT * FROM users WHERE username = '$input_username';
```

Attacker Input: ' OR '1'='1

This modifies the query to: SELECT * FROM users WHERE username = '' OR '1'='1';

- The query always returns TRUE, granting unauthorized access.

Mitigation Strategies

- Use parameterized queries instead of dynamic SQL.
- Validate and sanitize input to ensure it follows expected patterns.
- Implement strict access controls to prevent unauthorized queries.

4. Race Conditions

A race condition occurs when two or more processes attempt to access shared resources concurrently without proper synchronization, leading to unexpected behaviors or security vulnerabilities. Attackers can exploit race conditions to manipulate critical operations, escalate privileges, or access sensitive data.

How Race Conditions Occur

- Multiple threads execute the same process without proper locking mechanisms.
- An attacker injects a malicious process in a multi-threaded system to modify critical variables.
- A system checks a resource’s security state but fails to re-verify before use.

Real-World Example: The Therac-25 Radiation Machine Incident

- Due to a race condition in multi-threaded execution, the machine delivered excessive radiation doses, leading to patient deaths.

Mitigation Strategies

- Use synchronization mechanisms like mutexes or semaphores.
- Revalidate resources before executing critical operations.
- Implement transactional integrity in database operations to prevent conflicts.

5. Integer Overflow

Occurs when a mathematical operation exceeds the storage capacity of an integer variable, causing unexpected behavior.

How It Happens:

- Exceeding Data Type Limits: Manipulating integer values beyond their defined range.
- Lack of Boundary Checks: Failing to verify input values.

Example:

```
unsigned int x = 4294967295; // Max for 32-bit unsigned int
x = x + 1; // Overflow, x becomes 0
```

Security Impact:

- Application Crashes: Disrupting services (DoS attacks).
- Logic Manipulation: For example, changing account balances or quantities in e-commerce platforms.

6. Unterminated Null-Terminated String

Definition

In C/C++, a null-terminated string is expected to end with a null character (\0). Unterminated strings can cause memory leaks and undefined behavior.

Example:

```
char name[5];
strncpy(name, "Hello", 5); // No space for '\0', leading to undefined behavior
```

Impact:

- Memory Corruption: Allows attackers to read adjacent memory.
- Data Leakage: Reveals sensitive information.

Best Practices

- Non-malicious programming errors can lead to severe security breaches if left unaddressed.
- Secure coding practices, validation mechanisms, and rigorous testing are essential to minimize vulnerabilities.
- Developers must implement defensive programming strategies to protect against common software threats.

Best Practices for Secure Software Development

- Perform code reviews and security testing regularly.
- Implement input validation and proper error handling.
- Follow secure memory management techniques to prevent buffer overflows.
- Use synchronization mechanisms to prevent race conditions.
- Keep software updated and apply security patches promptly.
- By following these best practices, developers can build resilient applications that are secure against common programming errors.

Real-World Examples of Non-Malicious Programming Errors

- Heartbleed Bug (2014):
    - Type: Buffer Overflow.

  - Impact: Exposed sensitive data from OpenSSL servers, affecting websites, email servers, and VPNs.
  - Root Cause: Lack of bounds checking in heartbeat extension of OpenSSL.
- Therac-25 Radiation Therapy Machine (1985-1987)
  - Type: Race Condition.
  - Impact: Incorrect radiation doses led to patient injuries and deaths.
  - Root Cause: Unhandled multi-threaded processes, causing timing discrepancies.
- Twitter API Incomplete Mediation (2020)
  - Type: Incomplete Mediation.
  - Impact: API allowed unauthorized access to private tweets.
  - Root Cause: Insufficient validation of API requests, allowing bypass of access controls.

Case Study1: Heartbleed Bug

- Background:
  - Discovered in 2014 within the OpenSSL cryptographic library.
  - Named after the "heartbeat" extension of TLS protocol.
- Technical Details:
  - Buffer Overflow: A malicious request allowed attackers to read more memory than intended.
  - The heartbeat request allowed the server to return up to 64KB of memory content.
- Security Impact:
  - Leaked sensitive data, including usernames, passwords, private keys, and SSL certificates.
  - Affected approximately 500,000 servers worldwide.
- Mitigation:
  - Update to OpenSSL 1.0.1g or later versions.
  - Regenerate SSL certificates and change passwords.

Case Study2: Boeing 737 Max Race Condition (2018-2019)

- Background: Two fatal crashes occurred due to software errors in the MCAS (Maneuvering Characteristics Augmentation System).
- Technical Details:
  - The MCAS system relied on single sensor input, causing race conditions in flight control software.

    - Conflicting commands between manual pilot input and automated MCAS system.
- Impact:
    - 346 lives lost due to software error and lack of redundancy.
    - Financial and reputational damage to Boeing.
- Mitigation:
    - Implementing redundant sensor checks.
    - Revising software to avoid race conditions.
    - Conducting comprehensive pilot training.

Case Study3: Twitter API Bug (2020)

- Background: A bug in Twitter's API allowed third-party apps to access private information.
- Technical Details: API failed to validate certain permissions, exposing protected tweets and messages.
- Impact:
    - User privacy breach, potentially exposing sensitive information.
    - Violation of data protection regulations (GDPR, CCPA).
- Mitigation:
    - Fixed the API validation logic.
    - Conducted a security audit of all third-party integrations.

Lessons Learned from Real-World Incidents

- Common Patterns:
    - Lack of Input Validation: Allows malicious input to pass unchecked.
    - Unsafe Memory Management: Leads to buffer overflows and memory corruption.
    - Insufficient Testing: Misses edge cases and race conditions.
    - Poor Error Handling: Results in information leakage.
- Preventive Measures:
    - Adopt Secure Coding Practices: Follow OWASP guidelines.
    - Conduct Regular Code Reviews: Identify oversights and logical errors.
    - Implement Automated Testing: Use fuzzing tools and dynamic analysis.
    - Perform Security Audits: Regularly evaluate application security.

Web Application Security: Tools and Techniques for Vulnerability Assessment

Introduction to Web Application Security

Web Application Security focuses on protecting websites and web applications from cyber threats by identifying and mitigating vulnerabilities.

Importance:

- Prevent Data Breaches: Protect sensitive data from unauthorized access.
- Ensure Application Integrity: Maintain trust and reliability of web services.
- Compliance: Meet regulatory standards like OWASP, GDPR, and PCI-DSS.

Common Web Application Vulnerabilities

- SQL Injection: Manipulating database queries through user input.
- Cross-Site Scripting (XSS): Injecting malicious scripts into web pages.
- Cross-Site Request Forgery (CSRF): Trick users into performing unintended actions.
- Broken Authentication: Exploiting weak authentication mechanisms.
- Insecure Deserialization: Executing malicious code during object deserialization.

Nikto - Web Server Scanner

Nikto is an open-source web server scanner that performs comprehensive tests against web servers to identify potential security issues.

Features:

- Server Misconfigurations: Identifies default files, insecure configurations, and outdated software.
- Security Testing: Checks for known vulnerabilities and exploits.
- Web Server Fingerprinting: Detects server types and versions.
- SSL Certificate Scanning: Evaluates SSL/TLS configurations.

How Nikto Works:

1. Scanning: nikto -h http://example.com
2. Report Generation: Outputs findings in HTML, XML, and plain text formats.

Advantages:

- Open Source: Freely available.
- Extensive Test Suite: Over 6,700 vulnerability checks.
- Customization: Supports plugins and custom scripts.

Limitations:

- No Stealth Mode: Scans are easily detectable by intrusion detection systems (IDS).

- Limited Exploitation Features: Primarily focused on scanning rather than exploitation.

W3af - Web Application Attack and Audit Framework

W3af (Web Application Attack and Audit Framework) is an open-source tool designed for web application security testing.

Key Features:

- Discovery Plugins: Performs URL discovery, web spidering, and content analysis.
- Audit Plugins: Scans for SQL injection, XSS, remote file inclusion, and CSRF vulnerabilities.
- Exploitation Plugins: Allows manual or automated exploitation of discovered vulnerabilities.
- Proxy and Fuzzer: Supports traffic analysis and fuzz testing.

How W3af Works:

- Setup a Scan: w3af_console
- Configure Plugins: Choose audit, discovery, and exploitation plugins.
- Scan Target: Perform automated scans and generate reports.

Advantages:

- Modular Architecture: Easy to extend and integrate with other tools.
- Detailed Reporting: Provides actionable insights for remediation.
- Active Exploitation: Unlike Nikto, W3af can exploit vulnerabilities for pen testing.

Limitations:

- Learning Curve: Complex for beginners.
- Resource Intensive: Can consume significant system resources during scanning.

SQL Map - Automated SQL Injection Detection Tool
SQL Map is a powerful open-source tool for detecting and exploiting SQL injection vulnerabilities in web applications.

Key Features:

- Database Fingerprinting: Detects database management systems (e.g., MySQL, PostgreSQL, MSSQL).
- Automated Exploitation: Supports database takeover, data extraction, and command execution.

Supports Multiple SQL Injection Types:

- Boolean-based blind
- Time-based blind
- Union-based

- Error-based
- Stacked queries

How SQL Map Works:

- Basic SQL Injection Test: sqlmap –u "http://example.com/page?id=1" --dbs
- Extract Data: sqlmap -u "http://example.com/page?id=1" -D exampledb -T users -C username,password --dump

Advantages:

- Automated SQL Injection Testing: Reduces manual effort in security assessments.
- Extensive Payloads Library: Detects complex SQL injection scenarios.

Limitations:

- Detection Only: Does not fix vulnerabilities, requires manual remediation.
- Risk of Damage: Improper use can damage databases or systems.

DVWA - Damn Vulnerable Web Application

DVWA (Damn Vulnerable Web Application) is a purposefully vulnerable web application used for security training and practice.

Key Features:

- Simulated Vulnerabilities: Includes SQL injection, XSS, CSRF, File Inclusion, and Command Injection.
- Adjustable Security Levels: From low to high, demonstrating how different security measures affect vulnerability.
- Training Environment: Safe for pen testing, fuzzing, and manual testing.

How to Use DVWA:

- Set Up a Testing Environment: git clone https://github.com/digininja/DVWA.git
- Access DVWA Locally: Navigate to http://localhost/DVWA and start testing vulnerabilities.

Advantages:

- Hands-On Learning: Ideal for students and security professionals.
- Supports Multiple Tools: Can integrate with Burp Suite, SQL Map, and W3af.

Limitations:

- Not for Production Use: Should be restricted to safe environments to avoid accidental exposure.
- Requires Setup: Needs a local server (e.g., XAMPP, WAMP) to run.

Best Practices for Web Application Security

1. Regular Vulnerability Scanning: Use tools like Nikto, W3af, and SQL Map to identify risks early.

2. Code Reviews and Testing:

- Implement manual code reviews and automated testing.
- Use DVWA to train teams on secure coding practices.

3. Input Validation: Sanitize and validate all user inputs to prevent injection attacks.

4. Authentication and Authorization:

- Enforce strong authentication methods (MFA, OAuth).
- Apply role-based access controls (RBAC).

5. Security Patching: Regularly update web servers, frameworks, and libraries.

Web Application Security is crucial for protecting digital assets and maintaining user trust. Vulnerability scanning tools (Nikto, W3af) and application inspection tools (SQL Map, DVWA) help identify and mitigate risks. Proactive security measures and continuous learning are essential to keep web applications secure.

Real-World Examples of Web Application Security Incidents

Example 1: The Equifax Data Breach (2017)

Incident: Attackers exploited a vulnerability in Apache Struts, a web application framework.

How It Happened:

- The vulnerability allowed remote code execution (RCE).
- Lack of timely patching led to unauthorized access to sensitive data.

Impact:

- 147 million personal records, including social security numbers and financial data, were compromised.
- $700 million in fines and settlements.

Key Takeaway:

- Regularly scan web applications using tools like Nikto and W3af.
- Apply security patches immediately to prevent exploitation.

Example 2: British Airways (2018) - Magecart Attack

Incident: Attackers injected malicious JavaScript into the British Airways website through a third-party script.

How It Happened:

- Exploited incomplete mediation in web forms.
- Captured payment information as users entered it on the checkout page.

Impact:

- 380,000 payment card details were stolen.
- British Airways was fined £20 million under GDPR regulations.

Key Takeaway:

- Use application inspection tools like SQL Map to detect injection vulnerabilities.
- Regularly audit third-party integrations for security risks.

Example 3: Tesla (2020) - Bug Bounty Program Success

Incident: A security researcher discovered a vulnerability in Tesla's web application using Nikto and W3af tools.

How It Happened: Found a misconfiguration in cloud settings that could expose internal systems.

Impact:

- No breach occurred.
- The researcher was awarded $15,000 through Tesla's bug bounty program.

Key Takeaway:

- Implement vulnerability scanning and security audits.
- Encourage ethical hacking through bug bounty programs.

Exploiting SQL Injection with SQL Map

- Scenario: A web application allows users to search for products. You suspect the search functionality is vulnerable to SQL injection.
- Objective: Use SQL Map to identify and exploit the vulnerability.
- Step-by-Step Instructions:
    - Initial Test for SQL Injection: sqlmap -u "http://example.com/search?query=laptop" --dbs
    - Identify Vulnerability:
        - SQL Map detects a boolean-based blind SQL injection.
        - Lists available databases on the web server.
- Extract Sensitive Data:
    - Dump User Data: sqlmap -u "http://example.com/search?query=laptop" -D shopDB -T users -C username,password --dump
    - Results: Extracts usernames and hashed passwords.
- Identify Security Flaws:
    - Lack of Input Validation: User input directly incorporated into SQL queries.

    - No Parameterized Queries: Allows SQL injection attacks.

- Mitigation Steps:
    - Implement Parameterized Queries: SELECT * FROM products WHERE name = ?;
    - Sanitize User Inputs: Remove special characters and enforce input validation.
    - Use Web Application Firewalls (WAF): To block suspicious traffic.

Testing Vulnerabilities with DVWA

- Scenario: The Damn Vulnerable Web Application (DVWA) is set up in a virtual lab. You need to practice exploiting vulnerabilities using tools like Nikto, W3af, and SQL Map.
- Objective: Identify and exploit at least two vulnerabilities in DVWA and suggest remediation.
- Activity Steps:

1. Set Up DVWA: git clone https://github.com/digininja/DVWA.git
2. Navigate to the Application:
    - Open http://localhost/DVWA.
    - Set security level to "Low" for initial testing.
3. Explore Vulnerabilities:
    - SQL Injection: Use SQL Map to exploit database queries.
    - XSS Attack: Test input forms for script injection.
    - Command Injection: Inject system commands via the command execution feature.
4. Present Your Findings:
    - Document the Vulnerabilities: Describe how each vulnerability was identified.
    - Propose Mitigation Measures: Suggest best practices to harden the application.

Core Defense Mechanisms in Web Application Security

Core defense mechanisms are security strategies and technologies implemented to protect web applications from common attack vectors.

- Key Objectives:
    - Prevent Unauthorized Access: Ensure only authorized users can access sensitive resources.
    - Maintain Data Integrity: Prevent tampering with data during transmission and storage.
    - Enhance User Authentication: Strengthen login processes to avoid identity theft.
    - Secure Session Management: Protect session data and prevent hijacking.

What are Client-Side Controls?

Client-side controls are security mechanisms implemented in the frontend code (e.g., JavaScript, HTML). These controls perform input validation, authentication checks, and UI restrictions on the client's browser.

- Common Client-Side Controls:
    - Form Validations: Ensuring data types, length, and patterns.
    - JavaScript Validations: Preventing malicious input using scripts.
    - UI Restrictions: Disabling or hiding elements (e.g., grayed-out buttons).
- Limitation: Since client-side code is visible and modifiable, attackers can bypass these controls using developer tools or proxies.

Techniques to Override Client-Side Validations

1. Tampering with HTML Elements:

- Method: Use browser developer tools to modify HTML elements directly.
- Example: Changing an input field's 'readonly' attribute to allow modifications.

```
<input type="text" readonly value="10" id="price">
```

- Bypass:

```
document.getElementById('price').removeAttribute('readonly');
```

2. Disabling JavaScript:

- Method: Prevent JavaScript validations by disabling JavaScript in the browser settings.
- Impact: Bypasses form validations, input sanitization, and button restrictions.

3. Manipulating HTTP Requests:

- Method: Use tools like Burp Suite or Postman to modify requests before they reach the server.
- Example: Change price parameters in e-commerce sites to lower costs.

4. Intercepting Traffic Using Proxies:

- Method: Tools like OWASP ZAP and Charles Proxy allow real-time manipulation of requests and responses.
- Impact: Bypasses front-end controls by injecting malicious input or altering data.

Defense Strategies Against Client-Side Bypassing

1. Server-Side Validation: Always validate inputs on the server rather than relying solely on the client-side validation.

2. Implement Input Sanitization: Use whitelisting techniques to restrict inputs to expected formats only.

3. Secure Data Transmission: Use HTTPS to encrypt data and prevent tampering during transmission.

4. Use Content Security Policy (CSP): Helps restrict the execution of untrusted scripts and content.

What is Authentication?

Authentication is the process of verifying a user's identity before granting access to web applications or resources.

- Common Authentication Methods:
  - Username and Passwords: Traditional login method.
  - Multi-Factor Authentication (MFA): Combines passwords with additional factors (e.g., OTP, biometrics).
  - Single Sign-On (SSO): Allows one authentication session for multiple applications.

Methods to Breach Authentication Systems

1. Brute Force Attacks: Definition: Automated tools (e.g., Hydra, Medusa) try multiple combinations of usernames and passwords until successful login.

2. Credential Stuffing: Method: Attackers use previously leaked credentials to access accounts on different websites.

3. Phishing Attacks: Technique: Create fake login pages or emails to trick users into revealing credentials.

4. Bypassing Authentication Logic: Manipulate URL parameters, session tokens, or cookies to bypass login pages.

5. Exploiting Weak Password Policies: Allows simple passwords (e.g., "password123"), making brute force attacks easier.

Defense Strategies Against Authentication Attacks

1. Implement Strong Password Policies: Requirements: Minimum length, complexity, and expiration policies.

2. Multi-Factor Authentication (MFA): Adds an extra layer of security through biometrics, OTP, or security tokens.

3. Rate Limiting and Account Lockout: Prevent brute force attacks by limiting login attempts and locking accounts after multiple failures.

4. Monitor and Analyze Authentication Logs: Identify suspicious login attempts, such as unusual locations or IP addresses.

What is Session Management?

Session management involves handling user sessions securely, including session creation, maintenance, and termination.

- Common Session Management Techniques:
    - Session Cookies: Store session IDs on client browsers.
    - Tokens: Use JWT (JSON Web Tokens) or OAuth tokens for authentication.

Exploiting Sessions in Web Applications

1. Session Hijacking: Attackers steal session cookies to impersonate legitimate users.

2. Session Fixation: Force the user to use a specific session ID, allowing the attacker to take control once the user logs in.

3. Cross-Site Request Forgery (CSRF): Tricks the user's browser into executing unwanted actions while authenticated.

4. Unsecured Cookies: Storing session IDs in plaintext cookies or failing to set secure flags.

Defense Strategies Against Session Attacks

1. Secure Session IDs:

    - Use random and unique session IDs.
    - Regenerate session IDs upon authentication.

2. Implement Secure Cookie Flags:

    - HttpOnly: Prevent JavaScript access to cookies.
    - Secure: Ensures cookies are only transmitted over HTTPS.

3. Session Timeout and Logout Mechanisms: Automatically end sessions after inactivity or logouts.

4. Anti-CSRF Tokens: Generate unique tokens for each request to validate authenticity.

What are Access Controls?

Access controls define who can access what resources within an application. They enforce permissions and restrictions for users and roles.

- Types of Access Controls:

- Discretionary Access Control (DAC): Based on user identity and ownership.
- Role-Based Access Control (RBAC): Permissions based on user roles.
- Attribute-Based Access Control (ABAC): Considers user attributes, resource types, and environmental context.

Methods to Bypass or Manipulate Access Controls

1. IDOR (Insecure Direct Object Reference): Access restricted resources by manipulating URL parameters (e.g., user IDs).

2. Privilege Escalation: Technique: Exploit vulnerabilities to gain higher permissions than intended.

3. Forceful Browsing Directly access restricted pages by guessing URLs.

4. API Misconfiguration: Exposing backend data through improper API permissions.

Core defense mechanisms are crucial to prevent common attacks on web applications. Use server-side validation, strong authentication, secure session management, and robust access controls to mitigate risks. Regular security testing and best practices help maintain application integrity.

Real-World Examples of Core Defense Mechanism Failures

- Example 1: Bypassing Client-Side Controls - Uber API Flaw (2017)
- Incident: A security researcher discovered a vulnerability in Uber’s API.
- What Happened:
  - The client-side controls allowed users to change their own ride fares by modifying request parameters using browser developer tools.
  - Attackers could manipulate payment amounts, bypassing server-side validation.
- Impact: Potential financial losses and security risks for Uber.
- Key Takeaway: Always validate critical data on the server side, not just on the client side.
- Example 2: Attacking Authentication - Facebook Login Bug (2018)
- Incident: A vulnerability in Facebook's login system allowed attackers to gain access to user accounts.
- What Happened: Exploited an authentication flaw in the "View As" feature, which allowed attackers to steal access tokens and bypass authentication.
- Impact: 50 million accounts were compromised, forcing Facebook to reset access tokens.
- Key Takeaway: Regularly review authentication flows and implement multi-factor authentication (MFA) to mitigate risks.
- Example 3: Attacking Session Management - The WhatsApp Web Session Hijacking (2020)
- Incident: Attackers used session hijacking techniques to gain control of WhatsApp Web sessions.

- What Happened: Through phishing techniques, attackers acquired session cookies and used them to impersonate legitimate users.
- Impact: Unauthorized access to private messages and contact lists.
- Key Takeaway: Implement secure cookie flags (HttpOnly, Secure) and validate session integrity on the server side.
- Example 4: Attacking Access Controls - Instagram IDOR Vulnerability (2019)
- Incident: Researchers found an Insecure Direct Object Reference (IDOR) vulnerability in Instagram's API.
- What Happened: By manipulating user ID parameters in API requests, attackers could access private photos and messages of other users.
- Impact: Exposed private data and violated user privacy.
- Key Takeaway: Implement role-based access controls (RBAC) and avoid using predictable identifiers in URLs and API calls.