PRACTICE SHEET II

UNIT 4

1. Discuss Key Principles and Strategies for Implementing Defensive Software Architecture in Detail

Defensive software architecture is about designing software in a way that protects it from attacks or failures. Think of it like building a house with extra security features, such as strong doors, cameras, and locks, to prevent break-ins. Here are some key principles and strategies:

- **Minimize Complexity**: Simpler systems are easier to secure. When your code or system is too complex, it becomes harder to spot security problems or fix them.
- **Use the Principle of Least Privilege**: Give each part of the system only the access it needs to do its job. For example, if a part of your system only needs to read data, don't give it permission to delete or change that data.
- **Separation of Concerns**: Split your system into different sections, each with a specific job. If one part is attacked, it won't affect other parts.
- **Validate Input**: Always check the information that users enter to ensure it's safe and correct. For example, if someone enters text where a number is expected, the system should reject it.
- **Use Security Layers**: Just like having multiple locks on a door, use different security measures at different points. If one layer is breached, the others can still protect the system.

Example: An online banking system might require a password (first layer), and then send a text message with a code to the user's phone (second layer) before allowing access.

---

2. Describe the Role of Comprehensive Code Reviews in Designing a Defensive Secure Software Architecture

Comprehensive code reviews are when developers carefully check each other's code for mistakes and security issues before it goes live. This process is very important for making software more secure because it helps to catch problems early.

Here's why they help:

- **Find Hidden Bugs**: Sometimes developers make small mistakes that can create security risks, like forgetting to validate user input. Reviewing the code helps find and fix these mistakes before they cause trouble.
- **Improve Code Quality**: By having others review the code, you make sure that it follows best practices, making the code cleaner, easier to maintain, and less vulnerable to attacks.

- **Team Collaboration**: Different people have different skills and perspectives. A code review lets the team share ideas and spot issues that one person might miss.
- **Ensure Compliance**: In many industries, software has to follow specific security rules or standards. Code reviews help ensure that the software meets these requirements.

Example: Before a new feature in an app is added, another developer reviews the code to make sure it doesn't have security holes, like allowing someone to break into the system by entering special characters.

---

3. Elaborate on the Process of Discovery Analysis and Discovery Management

Discovery analysis and **discovery management** are about finding and understanding information, like vulnerabilities, weaknesses, or potential risks in software systems. Let's break these terms down in simpler words:

Discovery Analysis:

Discovery analysis refers to the process of identifying important details or issues in your system. Think of it as an investigation or research phase where you look for potential problems that could affect your software's security or performance.

- **Identifying Issues**: In this step, you search for vulnerabilities or weaknesses in the software that hackers might try to exploit. This could be through testing tools or analyzing system behaviour.
- **Risk Assessment**: After identifying potential issues, you figure out how serious each one is. For example, a bug that allows a hacker to steal user information would be more critical than one that causes the page to load slowly.
- **Data Collection**: Gather all the information you've found from testing or reviews. This includes any bugs, vulnerabilities, or security risks.

Example: If you're building a website, discovery analysis would involve checking if there are any places where user input (like comments or form submissions) could be used to hack into the site.

Discovery Management:

Discovery management is about how you handle the problems or discoveries once you've found them. It's like making a plan for how to fix or manage the issues identified during discovery analysis.

- **Prioritize Fixes**: Decide which issues are most important and should be fixed first. For example, if one problem allows hackers to break into the system and another just causes slow performance, fixing the security issue should be a priority.
- **Plan Solutions**: Create a plan to solve or reduce the risks found. This could involve patching security holes, updating code, or adding more security features.

- **Monitor Progress**: After deciding how to fix the issues, you need to make sure the work is being done correctly and track whether it's working as planned.

Example: After finding a serious vulnerability in a banking app during discovery analysis, the development team would prioritize fixing it immediately. They would manage the process by assigning it to developers, testing the fix, and ensuring the issue is completely resolved.

---

4. Define the concepts:

I. 2FA (Two-Factor Authentication)

Two-Factor Authentication (2FA) is a security process that requires two different forms of verification to access an account or system. This significantly strengthens security by requiring two distinct factors, typically:

1. **Something you know**: This is usually a password or PIN.
2. **Something you have**: This could be a physical token, mobile phone, or an authentication app like Google Authenticator.
3. **Something you are**: In some cases, this involves biometric data, such as a fingerprint or facial recognition.

- **Example**: When you log into your bank account, after entering your password, the system sends a verification code to your phone. You enter this code to confirm your identity, adding an extra layer of protection beyond just the password.

Why it's important: 2FA mitigates the risk of compromised passwords by ensuring that even if attackers steal the password, they still need the second factor to access the account. This makes it significantly harder for cybercriminals to gain unauthorized access.

II. Cybersecurity Mitigation Strategies

Cybersecurity mitigation strategies are proactive measures taken to reduce or minimize the impact of risks, vulnerabilities, and threats to an organization's data and systems. These strategies help defend against various attack vectors, ensuring the security and integrity of the IT environment.

Common strategies include:

- **Network segmentation**: Dividing a network into smaller parts to limit the damage of a potential breach.
- **Regular software updates and patching**: Keeping systems up-to-date to fix known vulnerabilities.
- **Employee training**: Ensuring that users are aware of phishing tactics, malware, and other cyber threats.
- **Incident response plans**: Preparing a plan to detect, respond to, and recover from cyberattacks.

- **Strong access controls**: Using role-based access control (RBAC) and least-privilege models to minimize user permissions.

Example: If an organization is vulnerable to phishing attacks, a mitigation strategy would include educating employees about identifying suspicious emails, along with implementing email filters and secure email gateways.

---

5. **What mitigation strategies would you adopt to reduce or minimize the impact of potential risks, threats, or vulnerabilities on web applications?**

To minimize the impact of risks, threats, or vulnerabilities on web applications, the following strategies can be implemented:

- **Input Validation and Sanitization**:
  - **Strategy**: Ensure all user inputs are validated and sanitized to prevent SQL injection, cross-site scripting (XSS), and other injection-based attacks.
  - **Example**: Validate email formats and use parameterized queries to prevent SQL injection.
- **Use of HTTPS and SSL/TLS**:
  - **Strategy**: Encrypt all communication between the server and clients using SSL/TLS to prevent man-in-the-middle (MITM) attacks.
  - **Example**: Implement HTTPS for secure data transmission and protect sensitive information like login credentials.
- **Authentication and Access Control**:
  - **Strategy**: Implement strong authentication mechanisms, such as two-factor authentication (2FA), and enforce role-based access control (RBAC).
  - **Example**: Only allow administrators to access certain functions and enforce strong password policies.
- **Regular Security Audits and Penetration Testing**:
  - **Strategy**: Regularly audit and test the application for vulnerabilities using penetration testing and code review practices.
  - **Example**: Hire security professionals to simulate attacks and assess potential weaknesses.
- **Web Application Firewalls (WAF)**:
  - **Strategy**: Deploy a WAF to monitor and filter incoming traffic, blocking malicious requests.

- **Example**: Use WAF to prevent common threats, like SQL injection or XSS.

---

6. What is secure credential and hashing credential?

Secure Credential

A **secure credential** refers to a piece of information or a set of information used to authenticate a user, system, or application in a secure manner. This typically includes usernames, passwords, and other sensitive data like API keys, security tokens, or certificates. Secure credentials must be protected from unauthorized access, leakage, or tampering to prevent security breaches.

To ensure secure credentials, the following practices are essential:

- **Encryption**: Storing credentials in an encrypted format ensures that even if they are intercepted or stolen, they cannot be easily read.
- **Multi-factor authentication (MFA)**: Requiring additional forms of verification ensures that compromised credentials alone are not sufficient to gain access.
- **Environment variables**: Secure credentials should be stored as environment variables instead of being hardcoded in applications.

Example: Storing API keys in a secure vault like AWS Secrets Manager ensures that only authorized services can retrieve them when needed.

Hashing Credential

Hashing credentials refers to the process of converting sensitive data, like passwords, into a fixed-length string of characters, using cryptographic hash functions, before storing them. Hashing is a one-way function, meaning that it cannot be reversed to retrieve the original password. This makes it a critical component in securely storing passwords.

Common characteristics of hashing:

- **One-way encryption**: Unlike encryption, which can be decrypted, a hash cannot be reversed to its original value.
- **Salting**: A random string (called a "salt") is added to the password before hashing to make each hash unique and defend against attacks like rainbow table attacks.
- **Algorithm choice**: Strong cryptographic algorithms like **SHA-256** or **bcrypt** are used to hash credentials because they are computationally expensive to crack.

Example: When a user creates an account, their password is run through a hashing algorithm (e.g., bcrypt), and only the hashed version is stored in the database. During login, the password they enter is hashed and compared to the stored hash. If the hashes match, access is granted.

---

7. How is the vulnerability discovery process done?

The vulnerability discovery process typically involves the following steps:

- **Automated Scanning**:
  - **Step**: Use automated tools like Nessus or OpenVAS to scan systems and applications for known vulnerabilities.
  - **Example**: A vulnerability scanner identifies an outdated version of Apache web server with known security issues.
- **Manual Testing**:
  - **Step**: Security professionals manually test systems and applications to identify hidden vulnerabilities that automated tools may miss.
  - **Example**: A penetration tester identifies a logic flaw in an authentication mechanism that allows unauthorized access.
- **Code Review**:
  - **Step**: Review the application's source code to identify insecure coding practices that could introduce vulnerabilities.
  - **Example**: A code reviewer finds an insecure SQL query vulnerable to SQL injection attacks.
- **Patch Management**:
  - **Step**: Monitor security advisories and apply patches to fix known vulnerabilities in third-party software.
  - **Example**: Regularly applying patches to fix vulnerabilities in the operating system and third-party libraries.

UNIT 5

1. How to Defend Against XSS Attacks

Cross-Site Scripting (XSS) attacks occur when malicious scripts are injected into trusted websites. To defend against XSS, the following strategies can be implemented:

- **Input validation and sanitization**: Ensure that all user inputs are properly validated and sanitized. Reject or escape any suspicious characters that can be used to inject malicious scripts (e.g., <, >, &).
- **Use of HTML entity encoding**: Convert special characters in user inputs (like < to < and > to >) to prevent them from being interpreted as code.
- **Content Security Policy (CSP)**: Implement CSP to restrict the sources from which scripts can be loaded. This limits the possibility of malicious scripts being executed.
- **Use secure frameworks**: Use modern web development frameworks (like React, Angular) that automatically escape or sanitize outputs, reducing the risk of XSS.
- **Example**: If a form allows users to submit comments, sanitize the input to ensure that <script> tags are not allowed, preventing malicious code from being executed in the browser.

---

2. "Sanitizing user input is a critical step in preventing security vulnerabilities like Cross-Site Scripting (XSS)" Justify the statement.

Sanitizing user input refers to the process of cleaning and validating data before it is processed or displayed by the web application. This is crucial for preventing XSS vulnerabilities because:

- **Direct input manipulation**: Users can input malicious code into forms or URL parameters. If not sanitized, this data can be executed as part of the webpage, leading to XSS attacks.
- **Preventing injection**: Sanitizing removes or escapes characters like < and > that can be used to inject scripts into a webpage.
- **Safe output display**: When sanitized data is safely encoded and output on the page, even if a malicious script is entered, it is rendered as plain text instead of executable code.
- **Example**: A comment field without input sanitization can allow an attacker to input a <script> tag that displays a malicious pop-up or steals session cookies.

---

3. What is DOMParser Sink, BLOB Sink, and SVG Sink?

In web security, **sinks** refer to places in the code where untrusted data can be introduced, potentially leading to security vulnerabilities like XSS.

- **DOMParser Sink**: This refers to using DOMParser to parse and execute malicious XML or HTML content. If untrusted data is passed to DOMParser without sanitization, it can lead to XSS attacks.
- **BLOB Sink**: BLOBs (Binary Large Objects) can store large amounts of binary data. When BLOB URLs are created from untrusted input and rendered in the browser, they may be vulnerable to injection attacks if the input is not sanitized.
- **SVG Sink**: SVGs are often used to render vector graphics in HTML. However, SVG can contain JavaScript-like behaviour (such as embedded scripts). If an attacker injects malicious SVG data, it can lead to XSS.

Example: If an application accepts user-uploaded SVGs without sanitization, an attacker can inject malicious scripts within the SVG file.

---

4. How HTML Entity Encoding Helps to Secure Web Applications from Various Attacks

HTML entity encoding is the process of replacing special characters (like <, >, &) in user input with their HTML entity equivalents (<, >, &). This prevents browsers from interpreting the input as HTML or JavaScript.

- **Prevents XSS**: Attackers often try to inject scripts using special characters like <script>. By encoding these characters, the browser renders them as plain text instead of executable code.
- **Secures output rendering**: By encoding user input before displaying it on the page, you ensure that it is safely presented to the user.

Example: If a user tries to submit <script>alert('XSS')</script> in a form, encoding the < and > characters prevents the script from running.

---

5. What is Content Security Policy for XSS Prevention?

Content Security Policy (CSP) is a security mechanism that helps prevent XSS attacks by defining which sources of content (scripts, styles, etc.) are allowed to load on a web page. It's a header-based policy that tells the browser to only trust certain domains or sources.

- **Script-src**: Specifies which scripts can be loaded. For example, allowing scripts only from your domain.
- **Style-src**: Defines which styles can be applied.
- **Prevent inline scripts**: CSP can prevent inline scripts from executing, which is a common vector for XSS.

Example: A CSP rule like script-src 'self'; will only allow scripts from the same domain, blocking any external or injected scripts.

---

6. How to Defend Against CSRF Attacks

Cross-Site Request Forgery (CSRF) attacks trick users into unknowingly submitting malicious requests to a website where they are authenticated. Defences against CSRF include:

- **CSRF tokens**: Include a unique, secret token in every form submission or request. This token is validated on the server to ensure that the request is legitimate.
- **SameSite cookies**: Set cookies with the SameSite attribute to ensure they are only sent with requests that originate from the same domain.
- **Requiring re-authentication**: For sensitive actions (like changing a password), require the user to enter their credentials again.

Example: When a user submits a form to update their profile, a CSRF token ensures that only legitimate requests are accepted.

---

7. How to Defend Against XXE Vulnerabilities

XML External Entity (XXE) vulnerabilities occur when an application parses XML input without properly handling external entities, allowing attackers to read files or execute remote code. Defences include:

- **Disable external entity parsing**: Most modern XML parsers have settings to disable the processing of external entities.
- **Use less dangerous formats**: Where possible, avoid using XML. Use safer formats like JSON, which do not support external entities.
- **Input validation**: Validate and sanitize any XML input to ensure it does not contain malicious entities.

Example: An attacker may attempt to use an external entity reference in XML to read sensitive files. Disabling external entity parsing prevents this.

---

8. Discuss the Strategies to Protect the Organization Against Regex DoS, DOS, and DDoS Attacks

- **Regex DoS (ReDoS)**: Regular expression denial-of-service (ReDoS) attacks exploit inefficient regular expressions, causing excessive backtracking that consumes CPU resources.
    - **Defences**:
        - Optimize regular expressions to avoid excessive backtracking.
        - Use timeouts to abort regular expression evaluation after a certain period.

- **DoS (Denial of Service)**: DoS attacks overload a service by consuming all its resources.
    - **Defences**:
        - Rate limiting to prevent excessive requests from one source.
        - Use load balancers to distribute traffic and prevent single points of failure.
- **DDoS (Distributed Denial of Service)**: DDoS attacks use multiple devices (often via botnets) to overwhelm a service.
    - **Defences**:
        - Deploy DDoS protection services (e.g., Cloudflare, Akamai).
        - Use Web Application Firewalls (WAFs) to detect and block malicious traffic.

---

9. Describe the Following Terms

i. Regex DoS (ReDoS)

Regex DoS (ReDoS) is a type of denial-of-service attack that exploits vulnerabilities in poorly designed regular expressions. The attacker submits input that causes the regular expression engine to perform excessive backtracking, consuming CPU resources and causing the application to slow down or crash.

- **Example**: A regular expression designed to match email addresses could be exploited by an attacker submitting a string that forces the engine to backtrack excessively, slowing the system.

ii. Logical DoS

Logical DoS refers to a denial-of-service attack that exploits the logical flaws in an application's code or design, rather than overwhelming it with traffic. This type of attack manipulates application functionality to cause errors or crashes.

- **Example**: An attacker submits a malicious request that triggers a known bug in the application, causing it to become unresponsive without needing large amounts of traffic.

UNIT 6

1. How to Defend Against SQL Injection Attacks

SQL injection attacks occur when attackers manipulate SQL queries by injecting malicious SQL code into input fields, compromising databases. The following defences can be implemented:

- **Use of Prepared Statements (Parameterized Queries)**: This method ensures that user inputs are treated as data, not executable SQL code. Instead of directly concatenating user input into queries, use placeholders and bind the input values later.
- **Stored Procedures**: These are pre-compiled SQL code that the database executes. Using stored procedures can limit the dynamic execution of queries based on user input.
- **Input Validation and Sanitization**: Ensure that all user input is validated. For instance, restrict the input format (e.g., numerical inputs should not contain letters) and sanitize inputs to prevent malicious characters.
- **Use ORM (Object-Relational Mapping) Tools**: ORM tools such as Hibernate and Sequelize help in avoiding raw SQL queries, automatically escaping inputs and preventing SQL injection.
- **Limit Database Privileges**: Minimize the privileges assigned to database users. For example, the database account used by the application should only have permissions necessary for specific tasks (like read-only access for certain operations).

---

2. Write a Short Note on:

i. Database-Specific Defences

Database-specific defences are mechanisms that protect the database directly from SQL injection and other threats. These include:

- **Input Escaping**: Many databases provide built-in functions to escape input. For example, MySQL's mysqli_real_escape_string() escapes characters like \, preventing them from being interpreted as part of the SQL query.
- **Database Permissions**: Limit user privileges based on the principle of least privilege. For example, the application user should not have privileges to drop tables or perform administrative tasks.
- **Database Firewalls**: Some databases offer firewall functionalities that filter incoming SQL queries to block suspicious patterns and prevent SQL injection attempts.
- **Error Handling**: Ensure that detailed error messages are not exposed to users. For example, avoid displaying database error messages directly to users as they might reveal sensitive database structure.

ii. Generic Injection Defences

Generic injection defences refer to practices that can help protect against multiple types of injection attacks, including SQL injection, command injection, and more:

- **Input Validation**: Validate all incoming data to ensure it matches expected formats, types, and ranges. For instance, if a field expects a number, non-numeric characters should be rejected.
- **Output Encoding**: Convert special characters into safe equivalents before displaying user input. For example, encode special characters in HTML, JavaScript, and SQL contexts.
- **Use of Safe APIs**: When interacting with external systems, avoid dynamic queries and instead rely on safe, parameterized APIs.
- **Security Libraries and Frameworks**: Use security-focused libraries that automatically escape and validate inputs, reducing the risk of injection vulnerabilities.

---

3. Demonstrate the Concept of Principle of Least Authority

The **Principle of Least Authority (POLA)** states that each part of a system (such as a user or a software component) should have the minimum level of access or permissions necessary to perform its function. This minimizes the risk of accidental or malicious damage if one part of the system is compromised.

Example:

- **User Permissions**: If a user needs to view certain data in an application, they should only have "read" permissions. They shouldn't be able to edit, delete, or perform administrative actions unless absolutely necessary.
- **Database Access**: If an application only requires read access to a specific table, the database user should be granted read-only permissions to that table. This prevents unintended or malicious changes to other tables.

This principle is critical for minimizing the impact of potential security breaches.

---

4. How Whitelisting Commands is Useful to Secure Web Applications

Whitelisting commands involves allowing only approved, known-safe commands or inputs in an application, while blocking everything else. It is a powerful security measure to ensure that only legitimate and secure inputs are processed.

- **Prevents Malicious Commands**: By specifying a list of allowed commands or inputs, you can block unauthorized or harmful commands. This is particularly effective against injection attacks, such as SQL or command injection.

- **Restricts User Input**: For example, if a web application only allows specific commands like SELECT or UPDATE to run on a database, then harmful commands like DROP TABLE or DELETE will be blocked.
- **Reduces Attack Surface**: Whitelisting significantly narrows the range of potential attacks, as only safe operations are permitted. All other input is rejected by default, which adds a layer of security.

Example: In a web application, instead of allowing arbitrary file uploads, you whitelist specific file types (like .jpg, .png) and reject others (.exe, .php).

---

5. Discuss the Securing Third-Party Dependencies in Brief

Third-party dependencies are software libraries, packages, or modules used by applications, but they can introduce vulnerabilities if not properly managed. Securing these dependencies involves:

- **Regular Updates and Patching**: Ensure that all third-party dependencies are kept up to date, as newer versions often include security fixes for vulnerabilities. Tools like Dependabot or Snyk can automatically alert you to outdated or vulnerable dependencies.
- **Reviewing Dependencies**: Always review the security reputation of a dependency before using it. Check its maintainers, its open-source community reviews, and whether it has had any known vulnerabilities in the past.
- **Use of Dependency Management Tools**: Use tools like npm's package-lock.json or Python's requirements.txt to lock dependency versions and prevent unwanted changes.
- **Remove Unused Dependencies**: Remove any dependencies that are no longer in use to reduce the attack surface.

---

6. Discuss:

i. Evaluating Dependency Trees

A **dependency tree** is a hierarchical representation of all the packages or libraries that your project depends on, including their own dependencies (transitive dependencies). Evaluating a dependency tree involves:

- **Understanding Direct vs. Indirect Dependencies**: Identify which dependencies are directly included by your project and which are included indirectly by other dependencies.
- **Risk Assessment**: Each dependency must be assessed for potential vulnerabilities. This can be done using tools like npm audit, Snyk, or GitHub's dependency graph feature.

- **Version Compatibility**: Ensure that all dependencies are compatible with your project and with each other. Conflicting or outdated versions can introduce bugs or security issues.
- **Example**: If you depend on libraryA, which in turn depends on libraryB, you should assess both libraryA and libraryB for security vulnerabilities.

ii. Modeling a Dependency Tree

Modeling a dependency tree refers to creating a visual or structured representation of all the dependencies in a project and their relationships.

- **Hierarchy of Dependencies**: The root node is your project, and all the direct dependencies are shown as child nodes. Each child node represents a package, and its own dependencies are listed as further child nodes.
- **Tools for Modeling**: Tools like npm ls, Maven's dependency plugin, or pip's pipdeptree can generate a visual tree of dependencies, showing the hierarchy.
- **Risk Mitigation**: Once modeled, the dependency tree can be used to spot dependencies that may introduce vulnerabilities or require updates, and prioritize which to address based on their position in the hierarchy.

Example: By modeling the tree, you can see that libraryC is a transitive dependency of two other libraries, and if libraryC has a vulnerability, it affects both paths in the tree.
