Lecture notes - Chapter 7: Secure Software Development

Topics: 4 hours

- o Secure Software Development Lifecycle (SDLC)
- o Threat Modeling
- o Code Review and Static Analysis
- o Secure Coding Practices
- o Vulnerability Management in Software Development
- o Software Security Testing
- o Security Headers

QUIZ - https://forms.gle/z9Jc87oKJbhQNJ3q7

Secure Software Development Lifecycle (SDLC)

The **Secure Software Development Lifecycle (SDLC)** is a process that integrates security into every phase of software development. It aims to produce robust, secure applications by considering security from the initial planning stages through to deployment and maintenance.

1. **Planning & Requirements:**
   - **Objective:** Define security requirements alongside functional requirements.
   - **Security Focus:** Identify potential threats and compliance requirements (e.g., GDPR, PCI-DSS). Conduct threat modeling to anticipate and mitigate security risks.
2. **Design:**
   - **Objective:** Architect the software with security in mind.
   - **Security Focus:** Incorporate security principles like least privilege, defense in depth, and secure coding practices. Design secure authentication and authorization mechanisms.
3. **Implementation (Coding):**
   - **Objective:** Write code following secure coding standards.
   - **Security Focus:** Utilize secure coding guidelines to prevent vulnerabilities such as SQL Injection, XSS, and buffer overflows. Implement code reviews and static code analysis to detect security flaws early.
4. **Testing:**
   - **Objective:** Validate that the software meets security requirements.
   - **Security Focus:** Conduct security testing including penetration testing, vulnerability scanning, and fuzz testing. Ensure that any security vulnerabilities are identified and remediated before deployment.
5. **Deployment:**
   - **Objective:** Securely release the software to the production environment.
   - **Security Focus:** Implement secure deployment practices, including environment hardening, secure configuration management, and continuous monitoring. Use automated tools to ensure consistency in deployments.

6. **Maintenance:**
    - **Objective:** Continuously monitor and update the software.
    - **Security Focus:** Apply security patches, monitor for new vulnerabilities, and maintain up-to-date documentation. Implement incident response plans to handle security breaches effectively.

Best Practices

- **Shift Left Approach:** Integrate security activities early in the development process to identify and mitigate risks before they become costly.
- **Automated Tools:** Use tools like static code analyzers, dependency checkers, and automated security testing tools to streamline the secure SDLC process.
- **Continuous Training:** Regularly train developers and project managers on secure coding practices and emerging security threats.
- **Regular Audits and Reviews:** Conduct regular security audits and code reviews to ensure ongoing compliance with security standards.

Threat Modeling

Threat Modeling is a systematic approach to identifying, assessing, and mitigating potential security threats to a system. It aims to proactively anticipate and address vulnerabilities before they can be exploited by attackers. Threat modeling is crucial for understanding how an application or system could be attacked and for designing countermeasures to prevent or mitigate these attacks.

Key Objectives of Threat Modeling

- **Understand the System:** Gain a thorough understanding of the system architecture, data flows, and potential attack vectors.
- **Identify Threats:** Recognize potential threats that could compromise the security of the system.
- **Prioritize Risks:** Assess and prioritize the risks based on the potential impact and likelihood of each threat.
- **Mitigate Vulnerabilities:** Develop strategies and implement controls to mitigate identified threats.

Threat Modeling Process

1. **Identify Assets:**
    - **Objective:** Determine what needs to be protected, such as sensitive data, intellectual property, user credentials, and system resources.
    - **Example:** In a web application, assets might include user data, payment information, and backend servers.
2. **Create an Architecture Overview:**
    - **Objective:** Document the system architecture, including components, data flows, entry points, and trust boundaries.
    - **Example:** A data flow diagram (DFD) showing how data moves between a user's

browser, a web server, and a database.

3. **Decompose the Application:**
    - **Objective:** Break down the system into smaller components to understand how data is processed and where security controls are applied.
    - **Example:** Analyzing how an API processes requests and where input validation occurs.
4. **Identify Threats:**
    - **Objective:** Use threat modeling methodologies to identify potential threats that could exploit vulnerabilities in the system.
    - **Example:** Using STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to categorize threats.
5. **Mitigate Threats:**
    - **Objective:** Develop and implement countermeasures to address identified threats.
    - **Example:** Applying encryption to protect data in transit, using multi-factor authentication to prevent spoofing.
6. **Validate and Iterate:**
    - **Objective:** Continuously review and update the threat model as the system evolves.
    - **Example:** Re-evaluating the threat model after significant changes to the system architecture.

Common Threat Modeling Methodologies

1. **STRIDE:**
    - **Overview:** A methodology developed by Microsoft to categorize threats into six categories: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege.
    - **Use Case:** Helps teams systematically consider different types of attacks that could affect their system.
2. **DREAD:**
    - **Overview:** A risk assessment model that evaluates threats based on Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability.
    - **Use Case:** Provides a scoring system to prioritize threats based on their potential impact.
3. **PASTA (Process for Attack Simulation and Threat Analysis):**
    - **Overview:** A seven-step risk-centric methodology that focuses on identifying business objectives and evaluating threats based on their potential impact on these objectives.
    - **Use Case:** Useful for organizations that want to align threat modeling with business goals and risk management.

Secure Code Review and Static Analysis

Secure Code Review and **Static Analysis** are critical components of the software development lifecycle focused on identifying and mitigating security vulnerabilities in the source code. While a secure code review involves human inspection of the code with a focus on security, static analysis leverages automated tools to analyze code without execution, detecting security issues early in the development process.

Secure Code Review

Definition:

- A secure code review is a manual or semi-automated process where developers and security experts examine the source code to identify security flaws. The goal is to ensure the code adheres to secure coding practices and is free from vulnerabilities that could be exploited by attackers.

What to Look for in a Secure Code Review:

- **Input Validation:** Ensure that all user inputs are properly validated and sanitized to prevent injection attacks.
- **Authentication and Authorization:** Verify that authentication mechanisms are robust and that access controls are properly implemented to prevent unauthorized access.
- **Data Protection:** Check for secure handling of sensitive data, including encryption in transit and at rest.
- **Error Handling:** Ensure that error messages do not reveal sensitive information that could be useful to an attacker.
- **Session Management:** Review how sessions are managed to prevent session hijacking and fixation.

Best Practices:

- **Use a Checklist:** Develop a comprehensive checklist based on common vulnerabilities and secure coding guidelines.
- **Focus on High-Risk Areas:** Prioritize the review of code that handles sensitive data, authentication, and access control.
- **Integrate with Development:** Conduct secure code reviews as part of the regular development process, rather than as an afterthought.
- **Collaborate:** Involve both developers and security experts in the review process to combine coding knowledge with security expertise.

Static Analysis for Security

Definition:

- Static analysis involves the automated inspection of source code or compiled code (such as bytecode) to identify security vulnerabilities, coding errors, and deviations from secure coding practices. The analysis is performed without executing the code.

How It Works:

- Static analysis tools scan the codebase, applying predefined rules and patterns to detect potential security issues. These tools can identify a wide range of vulnerabilities, from basic syntax errors to complex security flaws.

Key Security Issues Detected by Static Analysis:

- **Injection Flaws:** Detects potential SQL injection, command injection, and other types of injection attacks.
- **Cross-Site Scripting (XSS):** Identifies places where untrusted data is improperly handled in web applications.
- **Insecure Data Handling:** Flags improper encryption, storage, or transmission of sensitive data.
- **Buffer Overflows:** Detects situations where memory handling could lead to buffer overflow vulnerabilities.
- **Insecure API Usage:** Identifies the use of insecure functions or APIs that could expose the application to security risks.

Popular Static Analysis Tools:

- **Checkmarx:** A security-focused tool that scans source code to identify vulnerabilities.
- **SonarQube:** An open-source platform that provides continuous inspection of code quality and security.
- **Fortify Static Code Analyzer:** A tool that focuses on detecting security vulnerabilities in both source code and compiled code.
- **Bandit:** A tool specifically for finding security issues in Python code.

Best Practices:

- **Early Integration:** Incorporate static analysis into the CI/CD pipeline to catch security issues early.
- **Custom Rule Sets:** Customize the tool's rules and policies to match the security requirements of your specific project.
- **Regular Scanning:** Perform static analysis regularly, not just before release, to continuously monitor and improve code security.
- **Combine with Manual Review:** Use static analysis in conjunction with manual secure code reviews for a comprehensive security assessment.

Secure Coding Practices

Secure Coding Practices involve writing software in a manner that prevents security vulnerabilities and ensures the application can resist attacks. These practices are essential for protecting sensitive data, maintaining the integrity and availability of systems, and ensuring compliance with security standards and regulations.

Core Principles of Secure Coding

1. **Input Validation:**
   - **Objective:** Ensure that all user inputs are correctly validated, sanitized, and

filtered to prevent malicious data from being processed.

  - **Best Practices:**
    - **Whitelisting:** Allow only acceptable inputs by defining what is permitted rather than trying to block what is not.
    - **Input Length Validation:** Restrict the length of input data to prevent buffer overflows.
    - **Character Encoding:** Ensure that inputs are correctly encoded to prevent injection attacks like SQL Injection and XSS.

2. **Authentication and Access Control:**
   - **Objective:** Implement strong authentication mechanisms and enforce access control to ensure that only authorized users can access certain functions or data.
   - **Best Practices:**
     - **Multi-Factor Authentication (MFA):** Use multiple forms of verification (e.g., password and OTP) to enhance security.
     - **Role-Based Access Control (RBAC):** Assign permissions based on user roles, ensuring that users can access only what they need.
     - **Session Management:** Implement secure session management, including the use of secure cookies, session timeouts, and regeneration of session IDs upon login.
3. **Data Protection:**
   - **Objective:** Protect sensitive data both in transit and at rest to prevent unauthorized access or data breaches.
   - **Best Practices:**
     - **Encryption:** Use strong encryption algorithms (e.g., AES-256) to protect data.
     - **Hashing Passwords:** Store passwords securely using salt and hash functions (e.g., bcrypt, Argon2).
     - **Secure Storage:** Ensure sensitive information is stored securely, avoiding hard-coded credentials in the codebase.
4. **Error Handling and Logging:**
   - **Objective:** Implement robust error handling and logging practices that do not expose sensitive information or aid attackers.
   - **Best Practices:**
     - **Generic Error Messages:** Display generic error messages to users while logging detailed error information securely.
     - **Log Sensitive Data Carefully:** Avoid logging sensitive information such as passwords or personal data.
     - **Exception Handling:** Handle exceptions gracefully to prevent system crashes and security breaches.
5. **Secure Software Configuration:**
   - **Objective:** Ensure that software is configured securely to reduce the risk of misconfigurations leading to vulnerabilities.
   - **Best Practices:**
     - **Least Privilege:** Run applications with the minimum permissions required to reduce the attack surface.
     - **Disable Unnecessary Features:** Disable features and services that are not

        needed to minimize potential attack vectors.
        - **Secure Defaults:** Set secure default configurations and enforce them across environments.

6. **Code Quality and Testing:**
    - **Objective:** Maintain high code quality and rigorously test code to detect and eliminate security flaws.
    - **Best Practices:**
        - **Code Reviews:** Regularly perform secure code reviews to identify and fix security issues early.
        - **Automated Testing:** Use automated testing tools for static and dynamic analysis, as well as security-specific testing (e.g., fuzz testing).
        - **Unit and Integration Testing:** Ensure that all code paths are tested, especially those handling security-critical functions.

Software Security Testing

Types of Software Security Testing

Types of Software Security Testing

1. **Static Application Security Testing (SAST):**
    - **Definition:** SAST involves analyzing the source code, bytecode, or binary code of an application without executing it. It focuses on identifying security flaws in the code.
    - **How It Works:**
        - Analyzes the code for vulnerabilities such as SQL injection, XSS, and buffer overflows.
        - Tools scan the codebase to detect patterns and issues based on predefined rules.
    - **Tools:** SonarQube, Checkmarx, Fortify, Bandit.
    - **Best Practices:**
        - Integrate SAST into the CI/CD pipeline for continuous code analysis.
        - Regularly update rules and policies to reflect emerging threats and vulnerabilities.
2. **Dynamic Application Security Testing (DAST):**
    - **Definition:** DAST tests the application in its running state, focusing on identifying vulnerabilities that are exposed during execution.
    - **How It Works:**
        - Interacts with the application via its user interface, simulating real-world attacks to find issues such as authentication flaws, session management problems, and input validation weaknesses.
        - Tools perform automated scans and manual testing to uncover vulnerabilities.
    - **Tools:** OWASP ZAP, Burp Suite, Acunetix.
    - **Best Practices:**
        - Perform DAST in a staging or test environment to avoid impacting production systems.

        - Combine with manual testing to identify complex vulnerabilities that automated tools might miss.

3. **Penetration Testing (Pentesting):**
    - **Definition:** Penetration testing involves simulating real-world attacks on an application to identify vulnerabilities that could be exploited by attackers.
    - **How It Works:**
        - Conducts manual testing and exploitation attempts to assess the security posture of the application.
        - Testers use a variety of tools and techniques to uncover vulnerabilities and assess their potential impact.
    - **Tools:** Metasploit, Burp Suite, Nessus.
    - **Best Practices:**
        - Define clear scope and objectives for the test, including specific systems and data to be assessed.
        - Combine with other testing methods (SAST, DAST) for comprehensive coverage.
4. **Fuzz Testing:**
    - **Definition:** Fuzz testing involves sending a large volume of random or malformed data to an application to identify vulnerabilities that cause crashes, unexpected behavior, or security issues.
    - **How It Works:**
        - Uses automated tools to generate and input invalid or unexpected data into the application.
        - Monitors for abnormal responses, crashes, or other indicators of potential vulnerabilities.
    - **Tools:** AFL (American Fuzzy Lop), Peach Fuzzer.
    - **Best Practices:**
        - Incorporate fuzz testing into the development cycle to catch issues early.
        - Focus on critical components and interfaces to maximize testing effectiveness.
5. **Security Code Review:**
    - **Definition:** Security code review involves manually inspecting source code to identify security vulnerabilities and ensure adherence to secure coding practices.
    - **How It Works:**
        - Reviewers examine code for common vulnerabilities such as SQL injection, cross-site scripting (XSS), and insecure data handling.
        - Provide feedback and recommendations for improving security.
    - **Best Practices:**
        - Use a checklist based on common vulnerabilities and secure coding guidelines.
        - Involve both developers and security experts to leverage coding and security knowledge.
6. **Configuration Review:**
    - **Definition:** Configuration review assesses the security settings and configurations of software and systems to ensure they adhere to security best practices.
    - **How It Works:**

        - Evaluate system and application configurations for security settings, access controls, and compliance with policies.
        - Identify and remediate misconfigurations that could expose the application to security risks.
    - **Best Practices:**
        - Review configurations as part of the deployment process and periodically throughout the software lifecycle.
        - Use automated tools to assist with configuration analysis and validation.

Security Headers

Security Headers are HTTP response headers that help protect web applications from various types of attacks by enforcing security policies directly on the client (browser). These headers can prevent or mitigate common web vulnerabilities like Cross-Site Scripting (XSS), Clickjacking, and data injection attacks by controlling how the browser interacts with the server and the application.

Key Security Headers

1. **Content Security Policy (CSP):**
    - **Purpose:** CSP helps prevent XSS, data injection attacks, and other content-related attacks by allowing developers to define which sources of content are considered trusted.
    - **How It Works:**
        - CSP allows developers to specify the types of content (scripts, styles, images, etc.) that are allowed to be loaded by the browser from specific sources.
        - Example Directive: `Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted-scripts.example.com;`
    - **Best Practices:**
        - Start with a restrictive policy and gradually relax it as needed.
        - Use CSP in report-only mode during development to identify potential issues before enforcing the policy.
2. **X-Content-Type-Options:**
    - **Purpose:** This header prevents browsers from interpreting files as a different MIME type than what is specified in the `Content-Type` header, protecting against MIME-type sniffing attacks.
    - **How It Works:**
        - The header forces the browser to follow the declared content type strictly.
        - Example: `X-Content-Type-Options: nosniff`
    - **Best Practices:**

        - Always include this header to prevent browsers from inferring the MIME type and executing malicious scripts.
3. **Strict-Transport-Security (HSTS):**
    - **Purpose:** HSTS enforces the use of HTTPS, preventing downgrade attacks and ensuring that all communications between the client and server are encrypted.
    - **How It Works:**
        - The server instructs the browser to only interact with it using HTTPS for a specified period.
        - Example: `Strict-Transport-Security: max-age=31536000; includeSubDomains`
    - **Best Practices:**
        - Use HSTS with a long max-age (e.g., one year) and include subdomains to ensure that all communications are secure.
4. **X-Frame-Options:**
    - **Purpose:** This header prevents clickjacking attacks by controlling whether a page can be embedded in a frame, iframe, or object.
    - **How It Works:**
        - It restricts the framing of a page to protect users from UI redress attacks.
        - Example: `X-Frame-Options: DENY` or `X-Frame-Options: SAMEORIGIN`
    - **Best Practices:**
        - Set this header to `DENY` unless you have a specific need to embed the content in a frame, in which case use `SAMEORIGIN`.
5. **X-XSS-Protection:**
    - **Purpose:** This header configures the browser's built-in XSS filter to prevent Cross-Site Scripting attacks.
    - **How It Works:**
        - It instructs the browser to block or sanitize pages that appear to contain XSS attacks.
        - Example: `X-XSS-Protection: 1; mode=block`
    - **Best Practices:**
        - Although modern browsers may have better built-in protections, including this header can provide an additional layer of defense.
6. **Referrer-Policy:**
    - **Purpose:** Controls how much referrer information is included with requests made from your site, protecting user privacy and reducing information leakage.
    - **How It Works:**
        - The header restricts what information about the origin is sent in the `Referer` header.
        - Example: `Referrer-Policy: no-referrer-when-downgrade`
    - **Best Practices:**
        - Use `no-referrer`, `strict-origin`, or `strict-origin-when-cross-origin` for maximum privacy and security.
7. **Permissions-Policy (formerly Feature-Policy):**

- **Purpose:** This header allows site owners to control which features and APIs (like geolocation, camera, and microphone) can be used in the browser.
- **How It Works:**
  - It provides fine-grained control over what functionalities are allowed within the web page.
  - Example: `Permissions-Policy: geolocation=(), microphone=(), camera=()`
- **Best Practices:**
  - Disable unnecessary features to reduce the risk of exploitation through malicious scripts or iframes.

8. **Expect-CT:**
   - **Purpose:** The Expect-CT header helps detect misissued SSL/TLS certificates by instructing browsers to enforce Certificate Transparency.
   - **How It Works:**
     - It ensures that all certificates for a domain are logged in public Certificate Transparency logs.
     - Example: `Expect-CT: max-age=86400, enforce, report-uri="https://example.com/report"`
   - **Best Practices:**
     - Implement Expect-CT to monitor and detect certificate issues early, protecting against man-in-the-middle attacks.
9. **Cache-Control:**
   - **Purpose:** Controls how, and for how long, browsers and other caches store copies of resources from your site, helping to protect sensitive information.
   - **How It Works:**
     - This header can prevent sensitive data from being stored in the browser’s cache.
     - Example: `Cache-Control: no-store, no-cache, must-revalidate`
   - **Best Practices:**
     - Use restrictive caching policies on pages that contain sensitive information, such as login pages and user dashboards.

Lab: 2 hours

o Secure Code Review

o Insecure Header Configurations