Unit – II

Introduction to Cyber security: Introduction to OWASP Top 10, A1 Injection, A1 Injection Risks Root Causes and its Mitigation, A1 Injection, A2 Broken Authentication and Session Management, A7 Cross Site Scripting XSS,A3 Sensitive Data Exposure, A5 Broken Access Control, A4 XML External Entity (XEE), A6 Security Misconfiguration, A7 Missing Function Level Access Control, A8 Cross Site Request Forgery CSRF, A8 Insecure Deserialization, A9 Using Components With Known Vulnerabilities, A10 Unvalidated Redirects and Forwards, A10 Insufficient Logging and Monitoring, Secure Coding Practices, Secure Design Principles, Threat Modeling, Microsoft SDL Tool

Introduction to OWASP Top 10:

OWASP - Open Web Application Security Project.

• The OWASP Top 10 is a list of the most critical web application security risks.

• It helps developers and organizations understand and prioritize potential vulnerabilities.

List of OWASP Top 10 Application security risks:

1. Injection.

2. Broken Authentication.

3. Sensitive Data Exposure.

4. XML External Entities (XXE).

5. Broken Access Control.

6. Security Misconfiguration.

7. Cross-Site Scripting (XSS).

8. Insecure Deserialization.

9. Using Components with Known Vulnerabilities.

10. Insufficient Logging & Monitoring.

Injection:

Injection flaws allow attackers to send malicious code to an application's interpreter.
Common examples include SQL, OS, and LDAP injections.
Proper input validation and parameterized queries can mitigate this risk.

Injection Risk:

• Injection is a common vulnerability where attackers insert malicious code into an application.
• Injection attacks exploit weaknesses in interpreters, such as SQL, operating systems, or LDAP.
• Successful injection attacks can lead to data theft, unauthorized access, or even system compromise.

Root Causes of Injection:

• Insufficient input validation: Lack of proper validation allows attackers to insert malicious data.
• Improper use of interpreters: Improper handling of user-supplied input in interpreters can lead to code execution.
• Inadequate or missing security controls: Weak or missing security controls create opportunities for injection attacks.

Mitigation Strategies:

• **Input validation and sanitization:** Implement strict validation to ensure that user input meets expected criteria.
• **Parameterized queries and prepared statements:** Use parameterized queries or prepared statements to prevent code injection in database queries.
• **Use secure coding practices:** Follow secure coding guidelines and avoid concatenating user input with code or queries.
• **Least privilege principle:** Limit the privileges of the application or user accounts to minimize the potential impact of an injection attack.
• **Regular patching and updates:** Keep software, frameworks, and
libraries up to date to mitigate known vulnerabilities.
• **Security testing:** Conduct regular security assessments, including vulnerability scanning and penetration testing, to identify and fix
injection vulnerabilities.
• **Web application firewalls (WAF):** Implement a WAF to detect and block common injection attacks.

Examples of Injection Attacks:

• **SQL injection:** Attackers inject malicious SQL statements to manipulate the database or gain unauthorized access.
• **OS command injection:** Attackers execute arbitrary commands on the underlying operating system.
• **LDAP injection:** Attackers manipulate LDAP statements to bypass authentication or access unauthorized data.
• **XML or XXE injection:** Attackers exploit vulnerabilities in XML processing to read files or perform SSRF attacks.

Broken Authentication:

Weak or improper authentication and session management can lead to unauthorized access.
Common issues include weak passwords, session hijacking, and insufficient password recovery.
Secure authentication mechanisms, strong password policies, and session expiration controls are essential.

Broken Authentication and Session Management Risk:

• Broken authentication and session management vulnerabilities can lead to unauthorized access and compromise of user accounts.
• Weak or ineffective authentication mechanisms can allow attackers to bypass authentication controls.

• Insecure session management can result in session hijacking, session fixation, or session replay attacks.

Root Causes of Broken Authentication:

• **Weak password policies:** Inadequate password complexity requirements or lack of enforcement.
• **Insecure session handling:** Failure to protect session IDs, improper session timeout settings, or weak session token generation.
• **Vulnerabilities in authentication mechanisms:** Flaws in password reset, account lockout, or multi-factor authentication implementations.
• **Inadequate credential storage:** Storing passwords in plain text or using weak encryption methods.

Mitigation Strategies:

• **Strong password policies:** Enforce complex passwords, password rotation, and secure storage mechanisms.
• **Multi-factor authentication (MFA):** Implement MFA to add an extra layer of security beyond passwords.
• **Secure credential storage:** Store passwords securely using salted hashes or strong encryption algorithms.
• **Account lockout mechanisms:** Implement account lockout after a certain number of failed login attempts.
• **Properly handle password reset and account recovery:** Use secure mechanisms for resetting passwords and verifying user identities.

Best Practices for Session Management:

• **Secure session storage:** Encrypt and protect session data to prevent unauthorized access.
• **Secure session transport:** Use secure communication protocols (HTTPS) to transmit session-related data.
• **Session expiration:** Set appropriate session timeout values to automatically log users out after a period of inactivity.
• **Session monitoring and logging:** Monitor and log session activities to detect suspicious behavior or session-related attacks.

Sensitive Data Exposure:

Inadequate protection of sensitive information can lead to data breaches.
Examples include using weak encryption, storing passwords in plain text, and
exposing sensitive data through insecure channels.Employing strong encryption, using secure communication protocols, and implementing proper access controls are crucial.

Sensitive Data Exposure Risk:

• Sensitive data exposure occurs when sensitive information is not adequately protected, leading to its unauthorized disclosure or access.
• This vulnerability can result in data breaches, identity theft, financial loss, or reputational damage.
• Common examples of sensitive data include personally identifiable information (PII), financial data, healthcare records, and credentials.

Mitigation Strategies:

1. Encryption
2. Secure storage and handling
3. Secure transmission

Encryption:

• Protect sensitive data by encrypting it both at rest and in transit.
• Utilize strong encryption algorithms to safeguard data confidentiality.
• Implement secure protocols (e.g., HTTPS) for transmitting sensitive information over networks.

Secure storage and handling:

• Implement proper controls for storing and handling sensitive data.
• Use secure databases and file systems with access controls and encryption.
• Implement secure coding practices to prevent inadvertent exposure of sensitive data.

Secure transmission:

• Ensure secure transmission of sensitive data across networks.
• Use encrypted communication channels, such as VPNs or SSL/TLS, for transmitting sensitive data.
• Regularly monitor network traffic for any unauthorized access attempts or suspicious activities.

XML External Entities (XXE):

XXE vulnerabilities allow attackers to exploit the processing of XML inputs.
Attackers can read files, perform SSRF attacks, or even execute arbitrary code.
Disabling XML external entity processing or implementing strict input validation can help prevent XXE attacks.

XXE Attack Overview:

• XXE vulnerabilities occur when an application processes XML input insecurely.
• Attackers exploit XXE to read arbitrary files, perform SSRF attacks, or execute arbitrary code.
• XXE attacks can lead to data disclosure, information leakage, or system compromise.

Root Causes of XXE Vulnerabilities:

• Lack of input validation: Failing to validate or sanitize user-supplied XML inputs.
• Insecure XML parsers: Allowing external entity references without restrictions.
• Inadequate server-side configurations: Enabling entity expansion or not disabling external entity processing.

Mitigation Strategies:

• Disable XML external entity processing: Restrict or disable processing of external entities in XML parsers.
• Input validation and sanitization: Implement strict validation to reject malicious XML inputs.
• Safe XML parsing libraries: Use libraries with built-in protection against XXE attacks.

• Regular updates and patches: Keep XML parsers and dependencies up to date to address known vulnerabilities.

Broken Access Control:

Improper enforcement of access controls can allow unauthorized users to access restricted functionality or data.
Examples include insecure direct object references and insufficient authorization checks.
Implementing proper access controls, role-based access mechanisms, and authorization checks are necessary.

Broken Access Control Risk:

• Broken access control vulnerabilities occur when improper access controls are implemented, allowing unauthorized users to access restricted functionality or data.
• Attackers can exploit these vulnerabilities to perform actions beyond their authorized privileges, leading to data breaches, information
leakage, or unauthorized system manipulation.

Root Causes of Broken Access Control:

• **Insecure direct object references:** Directly referencing internal objects without proper authorization checks.
• **Insufficient authorization checks:** Failing to properly verify user permissions for specific actions or data access.
• **Lack of role-based access controls:** Not implementing granular access controls based on user roles or privileges.
• **Inadequate enforcement of access controls:** Not consistently enforcing access controls across all application components.

Mitigation Strategies:

• **Implement proper access controls:** Ensure that appropriate authorization checks are performed for every action and data access.
• **Role-based access control (RBAC):** Define and enforce roles and privileges to restrict access based on user roles.
• **Regular security testing:** Conduct comprehensive security testing, including authorization testing, to identify and address broken access control vulnerabilities.
• **Logging and monitoring:** Implement logging and monitoring mechanisms to detect and respond to unauthorized access attempts or suspicious activities.

Security Misconfiguration:

Security misconfigurations occur when systems are not securely configured.
Examples include default configurations, unnecessary services, and error messages that expose sensitive information.
Regular patching, secure configurations, and hardening systems can mitigate this risk.

Security Misconfiguration Risk:

• Security misconfigurations occur when systems, frameworks, or applications are not properly configured, leaving them vulnerable to attacks.
• Misconfigurations can lead to unauthorized access, data leaks, system compromise, or unintended exposure of sensitive information.
• Attackers often target misconfigurations to gain unauthorized access or exploit weaknesses in the configuration.

Common Types of Security Misconfigurations:

• **Default configurations:** Failure to change default settings or configurations that are known to be insecure.
• **Improper access controls:** Inadequate or misconfigured access controls, permissions, or privileges.
• **Unnecessary services and features:** Running unnecessary services or features that introduce additional attack vectors.
• Outdated or unpatched software: Failure to update or patch software and dependencies with known security vulnerabilities.

Mitigation Strategies:

• **Secure configurations:** Follow secure configuration guides and best practices for all systems, frameworks, and applications.
• **Regular security assessments:** Conduct periodic security assessments, vulnerability scans, and penetration testing to identify and address misconfigurations.
• **Patch and update management:** Keep all software, libraries, and dependencies up to date with the latest security patches and updates.
• **Least privilege principle:** Implement the principle of least privilege by granting only necessary access rights and privileges to users and components.

Cross Site Scripting (XSS):

XSS vulnerabilities allow attackers to inject malicious scripts into web pages.
These scripts can steal user data or perform actions on behalf of the user.
Proper input validation, output encoding, and secure coding practices can help prevent XSS attacks.

Cross-site Scripting (XSS) is a client-side code injection attack. The attacker aims to execute malicious scripts in a web browser of the victim by including malicious code in a legitimate web page or web application. The actual attack occurs when the victim visits the web page or web application that executes the malicious code. The web page or web application becomes a vehicle to deliver the malicious script to the user's browser. Vulnerable vehicles that are commonly used for Cross-site Scripting attacks are forums, message boards, and web pages that allow comments.

A web page or web application is vulnerable to XSS if it uses unsanitized user input in the output that it generates. This user input must then be parsed by the victim's browser. XSS attacks are possible in VBScript, ActiveX, Flash, and even CSS. However, they are most common in JavaScript, primarily because JavaScript is fundamental to most browsing experiences.

Types of XSS attacks:

According to OWASP, there are three types of XSS attacks: stored, reflected, and DOM based.

In a **stored XSS attack**, the application or API stores the unsensitized user input. Then, a victim can retrieve the stored data from the web application without that data being made safe to render in the browser.

In a **reflected XSS attack,** the application or API injects malicious code as part of the HTML input. The server returns the unescaped and unvalidated response with malicious content to the browser. The attacker can then run arbitrary HTML or JavaScript on the user's web browser.

A **DOM-based attack** is an XSS vulnerability that occurs in the Document Object Model (DOM) instead of the HTML code. In a DOM attack, the source of the data and the response of the attack are also in the DOM, and the data flow never leaves the browser.

To protect most from XSS vulnerabilities, follow three practices:

Escape user input: Escaping means to convert the key characters in the data that a web page receives to prevent the data from being interpreted in any malicious way. It doesn't allow the special characters to be rendered.

Validate user input: Treat anything that originates data from outside the system as untrusted. Validate all the input data. Use an allowlist of known, acceptable, good input.

Sanitize data: Examine and remove unwanted data, such as HTML tags that are deemed to be unsafe. Keep the safe data and remove any unsafe characters from the data.

These prevention methods cover most XSS attack vectors, but they don't cover everything. Be sure to also use static and dynamic application scanning tools in your pipeline to detect security vulnerabilities and mitigate them.

Insecure De-serialization:

Insecure de-serialization can lead to remote code execution, denial-of-service attacks, or authentication bypass.
Attackers can exploit vulnerabilities in how an application handles serialized objects.
Validating and whitelisting serialized data, as well as implementing integrity checks, can help mitigate this risk.

Insecure Deserialization Risk:

• Insecure deserialization occurs when an application fails to properly validate or sanitize data during the deserialization process.
• Attackers can exploit this vulnerability to execute arbitrary code, perform remote code execution, or conduct denial-of-service attacks.
• Insecure deserialization can lead to system compromise, data breaches, or unauthorized access to sensitive information.

Consequences of Insecure Deserialization:

• **Remote code execution:** Attackers can execute malicious code on the server or client-side, leading to complete system compromise.
• **Data tampering:** Insecure deserialization can allow attackers to modify serialized objects and manipulate data.

• **Denial-of-service (DoS):** Attackers can exploit insecure deserialization to cause resource exhaustion and disrupt application availability.

Mitigation Strategies:

• **Input validation and filtering:** Implement strict input validation and filtering during the deserialization process to prevent the execution of malicious or unexpected data.
• **Implement secure deserialization libraries:** Use deserialization libraries with built-in protection against insecure deserialization vulnerabilities.
• **Principle of least privilege:** Run deserialization code with minimal privileges and access rights.
• **Regular security updates:** Keep deserialization libraries, frameworks, and dependencies up to date with the latest security patches.

Using Components with Known Vulnerabilities:

Using outdated or vulnerable components can expose applications to security risks.
Attackers can exploit known vulnerabilities in libraries, frameworks, or other software components.
Regularly updating and patching components, using reliable sources, and monitoring for security advisories are important.

Risk of Using Components with Known Vulnerabilities:

• Using components (libraries, frameworks, plugins, etc.) with known vulnerabilities introduces security risks to the application.
• Attackers can exploit these vulnerabilities to gain unauthorized access, execute malicious code, or perform other attacks.
• Known vulnerabilities can have a significant impact on the security, stability, and reliability of the application.

Consequences of Using Components with Known Vulnerabilities:

• **Unauthorized access:** Attackers can exploit vulnerabilities to gain unauthorized access to sensitive data or systems.
• **Code execution:** Vulnerable components can allow attackers to execute arbitrary code, leading to system compromise.
• **Data breaches:** Exploiting vulnerabilities can result in data breaches, exposing sensitive information.
• **System disruption:** Attackers can exploit vulnerabilities to disrupt application functionality, leading to service outages or denial-of-service attacks.

Mitigation Strategies:

• **Regular updates and patches:** Keep all components up to date with the latest security patches and updates.
• **Vulnerability monitoring:** Stay informed about security vulnerabilities in components by monitoring security advisories and alerts.
• **Dependency management:** Conduct thorough assessments of component dependencies and ensure they are maintained and updated.
• **Security testing:** Perform security assessments, including vulnerability scanning and penetration testing, to identify and remediate vulnerabilities in components.

Insufficient Logging & Monitoring:

Insufficient logging and monitoring can prevent timely detection and response to security incidents. Lack of logs or inadequate monitoring make it difficult to identify attacks or trace malicious activities.
Implementing comprehensive logging, security event monitoring, and incident response processes is crucial.

Insufficient Logging and Monitoring Risk:

• Insufficient logging and monitoring occurs when an application fails to generate detailed logs or lacks effective monitoring mechanisms.
• This can result in the inability to detect and respond to security incidents, identify suspicious activities, or conduct effective forensic investigations.
• Insufficient logging and monitoring can lead to delayed incident response, prolonged compromise, or unauthorized access going undetected.

Consequences of Insufficient Logging and Monitoring:

• **Delayed incident response:** Inadequate logging and monitoring make it difficult to identify security incidents promptly, leading to delayed or ineffectiveincident response.
• **Unidentified attacks:** Without comprehensive logs and monitoring, attackers can perform actions without detection, resulting in extended compromise orunauthorized access.
• **Inability to conduct forensic investigations:** Insufficient logging hampers the ability to investigate security incidents, understand the scope of compromise, and gather evidence.
• **Compliance violations:** Insufficient logging and monitoring may result in non-compliance with regulatory requirements and industry standards.

Mitigation Strategies:

• **Implement comprehensive logging:** Log relevant security events, user activities, and system actions in a structured and centralized manner.
• **Real-time monitoring and alerting:** Establish monitoring mechanisms to detect and alert on suspicious activities, unauthorized access attempts, and securityincidents.
• **Security information and event management (SIEM):** Utilize SIEM solutions to aggregate and analyze logs for effective incident response and forensicinvestigations.
• **Regular log review and analysis:** Conduct regular reviews of logs to identify anomalies, security incidents, or policy violations.

Secure Coding Practices:

•Importance of Secure Coding
•Key Secure Coding Principles
•Best Practices for Secure Coding

Importance of Secure Coding:

• Secure coding practices are essential for developing software with robust security and resilience.
• Secure code helps protect against vulnerabilities and reduces the risk of security breaches.
• By following secure coding practices, organizations can safeguard sensitive data, maintain user trust, and minimize the impact of potential attacks.

Key Secure Coding Principles:

• **Input Validation:** Validate and sanitize all user input to prevent common attacks such as SQL injection and cross-site scripting (XSS).
• **Proper Error Handling:** Implement appropriate error handling mechanisms to provide meaningful error messages without revealing sensitive information.
• **Authentication and Access Controls:** Use strong authentication mechanisms and enforce access controls to ensure only authorized users can access resources.
• **Secure Communication:** Implement secure protocols (e.g., HTTPS) to protect data transmission over networks.

Best Practices for Secure Coding:

• **Regular Updates and Patching:** Keep software, libraries, and dependencies up to date with the latest security patches.
• **Code Reviews:** Conduct regular code reviews to identify and address security vulnerabilities during the development process.
• **Security Testing:** Perform comprehensive security testing, including penetration testing and vulnerability scanning, to identify and mitigate security flaws.
• **Developer Education:** Provide ongoing education and training to developers on secure coding practices, common vulnerabilities, and emerging threats.

Importance of Secure Design:

• Secure design principles help organizations build systems and applications with security in mind from the outset.
• Securely designed systems are more resilient against attacks and provide a strong foundation for protecting sensitive data.
• By incorporating secure design principles, organizations can reduce the risk of security breaches and ensure the trust of their users.

Key Secure Design Principles:

• Defense in Depth: Implement multiple layers of security controls to protect against a variety of potential threats. This includes a combination of preventive, detective, and corrective measures.
• Least Privilege: Assign the minimum level of privileges necessary for users, processes, and components to perform their intended functions. This limits potential damage in the event of a security compromise.
• Fail-Safe Defaults: Configure systems and applications with secure defaults, ensuring that the most restrictive settings are in place initially and users must explicitly grant additional access or privileges.

Threat Modeling:

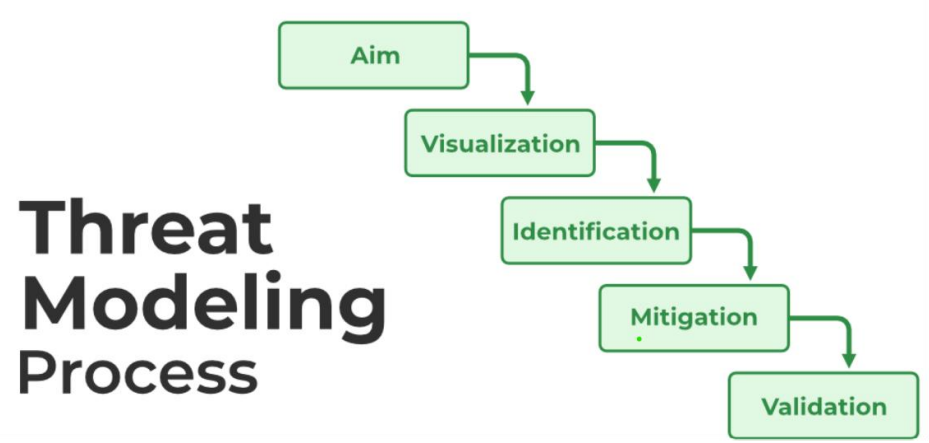
1. Importance of Threat Modeling
2. Key Steps in Threat Modeling
3. Benefits of Threat Modeling

Importance of Threat Modeling:

• Threat modeling is a systematic approach to identifying and mitigating potential threats and vulnerabilities in a system or application.
• It helps organizations proactively identify and prioritize security risks, allowing for informed decision-making and resource allocation.
• By incorporating threat modeling into the development lifecycle, organizations can build more secure and resilient systems.

Key Steps in Threat Modeling:

• Identify Assets and Scope: Identify the assets (data, systems, components) that need protection and define the scope of the threat modeling exercise.
• Identify Threats and Vulnerabilities: Analyze the system to identify potential threats and vulnerabilities that could exploit weaknesses and compromise the assets.
• Assess Risks and Impact: Evaluate the likelihood and potential impact of each identified threat, considering factors such as likelihood, severity, and business impact.
• Develop Mitigation Strategies: Develop and prioritize mitigation strategies to address the identified threats and vulnerabilities, considering cost effectiveness and feasibility.



Process of Threat Modeling:
1. **Aim:**
The target before approaching Threat Modeling must be clear within ourselves what we will achieve from Threat Modeling, that is our application must follow the CIA Triad.
Confidentiality: It helps in protecting data from unauthorized access.
Integrity: It helps in preventing restricted changes.
Availability: It helps in performing important tasks under certain attacks.

2. **Visualization:**
Here, we will deal with what we are going to build. We must have a document overview of the application which helps in making our process easier. Here we will build diagrams that will help us in making our process easier.

It can be done in two ways:

Data Flow Diagram: It helps in showing how the flow of data occurs in the system.
Process Flow Diagram: It helps in finding the process of the system that from where users interact in the system, and how the system works internally.

3. **Threat Identification:**
Here we are going to deal with how we can identify threats or what can go wrong in the process. By analyzing the images of the previous section, you have found how threats can be identified. These methods are mentioned in threat modeling methodologies..

4. **Mitigation:**
Here, we are going to deal with what we will do about the Threats. Here we will review the layers to identify the required vulnerabilities. Mitigation involves a continuous investigation of each vulnerability so that the most effective efforts can be designed.

5. **Validation:**
This is the final step in the process of Threat Modeling, here we are going to deal with whether we have done a good job or not. Have all the threats been mitigated or not? We will check the changes and as threat modeling is not a one-time activity, we have to regularly watch these things.

Benefits of Threat Modeling:

• **Proactive Risk Management:** Threat modeling enables organizations to identify and address potential risks before they can be exploited, reducing the likelihood of security
breaches.
• **Cost-Efficient Security Measures:** By focusing on high-priority threats and vulnerabilities, organizations can allocate
resources effectively and implement targeted security
measures.
• **Enhanced Collaboration:** Threat modeling encourages cross functional collaboration among stakeholders, including developers, architects, security teams, and business representatives.

Microsoft SDL (Secure Development Lifecycle) Tool:

1. Introduction to Microsoft SDL
2. Purpose of the Microsoft SDL Tool
3. Features of the Microsoft SDL Tool

Introduction to Microsoft SDL:

• The Microsoft SDL is a comprehensive framework and set of practices for integrating security into the software development process.
• It provides a structured approach to identify, prioritize, and mitigate potential security vulnerabilities during software development.
• The SDL helps organizations build secure and resilient software, reducing the risk of security breaches and enhancing user trust.

Purpose of the Microsoft SDL Tool:

• The Microsoft SDL Tool is a software security analysis tool that assists developers in identifying common security vulnerabilities in their code.
• It automates security analysis and provides recommendations for mitigating identified vulnerabilities.
• The tool helps developers identify and address security issues early in the development process, saving time and effort.

Features of the Microsoft SDL Tool:

• **Static Analysis:** The tool performs static code analysis to identify potential security vulnerabilities, such as buffer overflows, injection attacks, and cross-site scripting (XSS).
• **Vulnerability Identification:** It scans code for known vulnerabilities and provides guidance on how to fix them.
• **Integration with Development Environment:** The tool integrates with popular development environments, making it easy to incorporate security analysis into the developer's workflow.
• **Customizable Rules:** Developers can customize the tool's rules to align with their specific security requirements and coding standards.