Team 15

Cybersecurity Low Prep

December 15, 2023

Contents

List of Figures

1 Introduction

A cybersecurity audit is a comprehensive and systematic evaluation of an organization's information systems, policies, and practices to proactively identify vulnerabilities, threats and associated mitigation options to address potential security risks before malicious actors can exploit them. Regular assessments allow organizations to adapt to changing threats, technologies, and business processes.

2 PS 1

2.1 Problem Statement

Approach to improving cyber security audit processes and outcomes of audits
Deliverables:

- Design (not develop) an application/ tool to enhance the quality of Audits
- The solution should have a multifaceted approach focusing on enriching the process of Audit for all the stakeholders i.e. Auditor, Auditee, Regulator etc.

The general steps involved in a security audit:

- Announcement letter sent by auditor to auditee for scope and timing
- Auditee sends back acknowledgement
- Pre Audit meeting set by auditor with auditee to understand process about organisation, plicies, SOPs.
- Request documents, policies from auditee on email for accountability
- Prepare an internal audit plan, not shared with external but used when we become auditee.
- Prepare pre risk control matrix [*Figure* 5],
    1. risk
    2. controls attached with it
    3. how to conduct an audit on this
- Field Work: work based on SOP, risk control matrix
    1. Sampling technique: from 1000, we pick 100, if 90 don't follow, flag for finding access management
    2. Signing of SOPs
    3. Reviewing policies
    4. chain process is followed or not
    5. Should notify immediately of any vulnerability
- Prepare draft audit report from post RCM, working paper as audit finding evidence, a meeting with people discussing the course of action, give the time to close the audit
- Give a final audit report
- Do a follow up and close the findings.

2.2 Proposed Solution

The cybersecurity audit platform is a secure system for managing cybersecurity checks. It has features like secure logins, a central dashboard for easy communication, and tools for learning. You can plan and start audits together, connect with cybersecurity tools, and automate tasks. It keeps an eye on things in real-time, makes reports automatically, and helps communicate securely with regulators. The people being audited can use a portal for self-checks. The platform also considers performance, feedback, and risk. For safety, it uses encryption, regularly checks for problems, and follows rules. The design is easy to use, and the main system is built for safety and flexibility. If there's a problem, it has backups in place. In simple terms, it's a tool that helps keep track of cybersecurity, making sure everything is safe and follows the rules. It has the following features,

- **User Authentication and Authorization**:
  - Implement multi-factor authentication (MFA) for added security.
  - Utilize strong password policies and secure password storage practices.
  - Integrate with identity providers for single sign-on (SSO) capabilities.
  - Use technologies like OAuth 2.0, OpenID Connect, and MFA tools (e.g., Google Authenticator).
  - Define roles (Auditor, Auditee, Regulator) with specific permissions.
  - Implement granular access controls based on roles.
  - Utilize Role-Based Access Control (RBAC) and technologies like JSON Web Tokens (JWT).

- **Dashboard and Communication Hub**:
  1. **Centralized Dashboard**
     - Display ongoing audits, deadlines, and key metrics.
     - Include widgets for quick access to relevant information.
     - Develop using React.js, Angular, or Vue.js.
  2. **Messaging System**
     - Implement a secure messaging system for direct communication.
     - Allow document and information exchange within the platform.
     - Implement a secure messaging system using technologies like WebSocket with end-to-end encryption.

- **Personnel Training**:
  - Dedicated section for users to enroll in workshops.
  - Repository of training materials, webinars, and resources.
  - Integrate e-learning platforms like Moodle or custom-built modules using SCORM.

- **Collaborative Planning and Scope definition**:
  1. **Initiating Audits**
     - Allow auditees to initiate the audit process through a structured form.
     - Provide templates and guidelines for consistent information submission.
     - Use customizable forms with technologies like HTML forms or dynamic form builders.
  2. **Joint Planning**
     - Enable collaboration for defining and refining audit scope.
     - Include discussion forums and comment sections.
     - Implement real-time collaboration using tools like Google Docs or collaborative editing libraries.

- **Tech Integration**:
  1. **Tools**
     - Integrate cybersecurity tools for vulnerability scanning, risk assessment, etc.
     - Provide APIs for seamless integration with other tools.

  - Integrate with tools like Nessus for vulnerability scanning, and use RESTful APIs for connectivity.
2. **Automation**
  - Automate routine tasks like data collection and report generation.
  - Ensure a user-friendly interface for interacting with automated processes.
  - Utilize Ansible, Puppet, or Chef for task automation.

- **Continuous Montioring and Reporting**:
  - Implement real-time monitoring for security incidents.
  - Integrate with SIEM systems.
  - Integrate with SIEM tools such as Splunk or ELK Stack.
  - Generate automated reports for auditors, auditees, and regulators.
  - Reports should include KPIs, identified risks, and areas for improvement.
  - Use reporting tools like JasperReports or Tableau for automated report generation.

- **Regulator work**:
  - Establish a secure channel for direct communication with regulators.
  - Enable document sharing on compliance concerns.
  - Use encrypted email services or secure messaging platforms.
  - Allow auditors to submit final reports directly to regulators.
  - Implement secure file transfer protocols.
  - Implement secure file transfer protocols such as SFTP or HTTPS.

- **Auditee empowerment**:
  - Provide auditees with a portal for self-assessments.
  - Implement digital signature capabilities for accountability. Implement digital signature features using cryptographic libraries like OpenSSL.
  - Develop using web technologies like React.js for a responsive and user-friendly interface.

- **Performance Metrics and Continuous Imporvement**:
  - Define and display KPIs on the dashboard.
  - Metrics may include audit completion times, identified vulnerabilities, and compliance status.
  - Use data visualization tools like D3.js or Chart.js for displaying key performance indicators.
  - Implement a feedback mechanism for continuous improvement.
  - Integrate feedback forms using technologies like Formik or SurveyJS.

- **Risk-Based Approach**:
  - Integrate risk assessment tools for prioritizing audit activities.
  - Highlight critical assets and high-risk areas.
  - Integrate with tools like OpenVAS or Qualys for risk assessment.
  - Use risk scoring algorithms and frameworks like FAIR (Factor Analysis of Information Risk).

- **Security Measures**:
  - Implement end-to-end encryption for data in transit and at rest.
  - Conduct regular security audits and vulnerability assessments.
  - Implement end-to-end encryption using libraries like OpenSSL or CryptoJS.
  - Use tools like OWASP ZAP or Nessus for regular security audits.

- **Interface**:
  - Design a clean and intuitive user interface.
  - Include tooltips, guides, and documentation for user assistance.
  - Ant Design for a clean and intuitive UI.
  - Provide tooltips using libraries like Tippy.js or native HTML tooltips.

3 PS 2

3.1 Problem Statement

Approach for reducing vulnerabilities in products, software and Applications
Deliverables:

- Developing a security baseline to safeguard different dimensions of application.
- Focus should be on a proactive approach to building applications with a focus on preventing security vulnerabilities and minimizing the risk of exploitation.

3.2 Proposed Solution

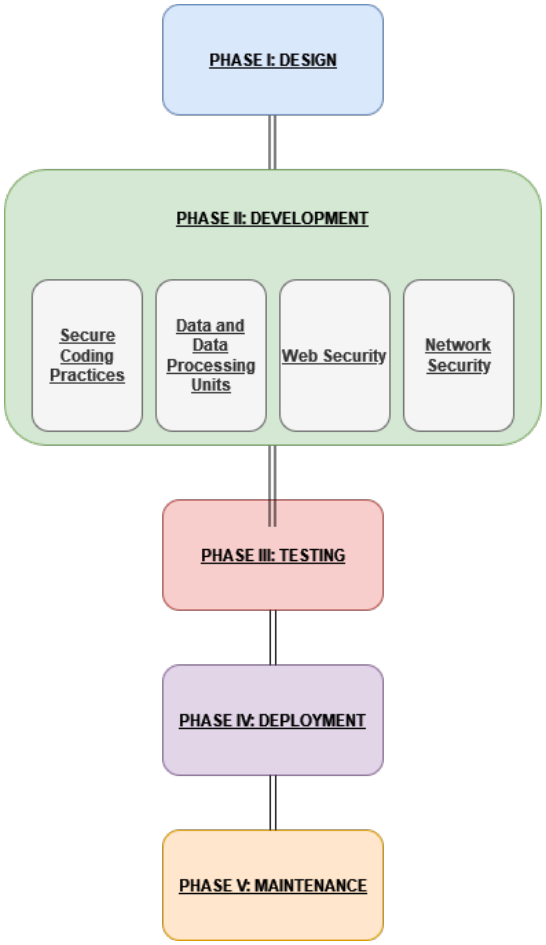


Figure 1: Security Baseline Architecture

- Security Baseline: Guidelines to be followed during development, alongside the software development lifecycle to ensure minimum attack surface.
- Automation of detection and mitigation of vulnerabilities in products, software and applications.

Security Baseline

1. **Phase-I: Design and Architecture**:
   - **Threat Modeling**
     (a) In the application design phase, model all possible threats and implement mitigations for vulnerability surface area minimization.
     (b) Consistently apply threat modelling across all software development stages.
     (c) Continuously update threat models as system details emerge, addressing new attack vectors. Identify, determine, and rank risks using a threat categorization framework like STRIDE.
   - **STRIDE**
     (a) **Spoofing Identity**
       i. Implement OAuth 2.0, OpenID Connect, and SAML protocols for token-based user authentication. Additionally, integrate with SSO providers like Okta for streamlined business user access.
       ii. Leverage PKI for secure digital certificate management, ensuring robust authentication of users and devices within the network.
       iii. Incorporate biometric authentication methods like fingerprint and facial recognition to enhance overall security.
       iv. Deploy the FIDO2 framework, including WebAuthn, for passwordless and phishing-resistant browser-based authentication.
       v. Employ Risk-Based Authentication (RBA) systems that dynamically adjust authentication strength based on various risk factors.
       vi. Strengthen your authentication system by validating email addresses, banning disposable accounts, securing password resets, implementing MFA, securing logout functions, and preventing enumeration vulnerabilities through generic error messages and CAPTCHA.
     (b) **Tampering with Data**
       i. Cryptographic Hash Functions: Use strong hash functions like SHA-256 for data integrity checks.
       ii. Role-Based Access Control (RBAC): Implement RBAC to define roles and permissions for data access and modification.
       iii. End-to-End Encryption: Employ E2E encryption protocols, such as PGP for emails or TLS for web traffic, to protect data in transit.
       iv. End-to-end Encrypted Data Streams: Use protocols like QUIC or DTLS for securing data streams, ensuring end-to-end encryption that prevents intermediate node tampering.
       v. Advanced File Integrity Checkers: Utilize tools like AIDE or Tripwire, configured for real-time monitoring, with cryptographic hash checking for critical system and application files.
     (c) **Repudiation**
       i. Repudiation threats are mitigated by implementing strong auditing and tracking of user actions.
       ii. Immutable Logging: Use secure and tamper-evident logging mechanisms, such as blockchain-based logging or write-once-read-many (WORM) storage.
       iii. Timestamping Services: Incorporate trusted timestamping and digital signature services to validate the time and origin of transactions.
       iv. Comprehensive Event Correlation: Use SIEM systems with advanced event correlation features to track and link activities across different logs and sources, creating an indisputable audit trail.
       v. Distributed Ledger Technology (DLT): Use DLT for creating decentralized and immutable logs. DLTs, like Directed Acyclic Graphs (DAG), can offer scalability and faster processing, suitable for high-volume logging.
       vi. Zero Knowledge Proof (ZKP) Systems: Implement ZKP mechanisms, allowing verification of actions without revealing the actual data.
     (d) **Information Disclosure**

i. Implement data classification and segmentation, restricting access based on sensitivity.
ii. Enhance HTTP header security by disabling revealing information, utilizing clean URLs, and generic cookie parameters.
iii. Disable client-side error reporting and sanitize client-side files from sensitive data.
iv. Adopt a zero-trust architecture for comprehensive authentication, authorization, and encryption.
v. Implement Attribute-Based Access Control (ABAC) for fine-grained access decisions.
vi. Use advanced data obfuscation techniques like format-preserving encryption (FPE) or tokenization for sensitive data protection.
vii. Explore Fully Homomorphic Encryption (FHE) for secure computation on encrypted data in untrusted computational environments.

(e) **Denial of Service**

i. Load Balancers and CDN: Use load balancers and Content Delivery Networks (CDNs) to distribute traffic and mitigate DoS attacks.
ii. Intrusion Prevention Systems (IPS): Deploy IPS to prevent DoS attacks.
iii. A thorough code review process can prevent malicious regex from entering the codebase. Look for patterns indicating extensive backtracking, such as (a[ab]*)+. Use open-source tools or regex performance testers to identify and prevent these regex.
iv. Anomaly Detection Systems: Use machine learning-based anomaly detection to identify and respond to unusual traffic patterns.
v. State-of-the-Art DDoS Mitigation: Implement advanced DoS mitigation techniques, like Anycast routing and sinkholing, with standard services.
vi. BGP Route Hijacking Mitigation: Use RPKI (Resource Public Key Infrastructure) to secure BGP routing against route hijacking.

(f) **Elevation of Privilege**

i. This happens when an attacker gains higher-level permissions than what they are entitled to.
ii. Mitigate privilege escalation with Privileged Access Management (PAM), monitoring and controlling privileged accounts.
iii. Conduct regular static and dynamic code analysis to identify and address potential vulnerabilities leading to privilege escalation.
iv. Employ exploit mitigation frameworks such as Microsoft's EMET and Google's Safe Browsing to enhance security.
v. Implement Runtime Application Self-Protection (RASP) solutions for real-time attack detection and prevention in the application's runtime environment.

Insecure Design is ranked 4th in the OWASP TOP TEN Framework* [1], these are the strategies to mitigate the risk of the same:

(a) Automated Security Scanning: Integrate automated security tools like SAST (Static Application Security Testing), DAST (Dynamic Application Security Testing), and IAST (Interactive Application Security Testing) in the CI/CD pipeline.
(b) Security Gateways in Deployment: Establish security checkpoints in the deployment pipeline to ensure no insecure code is deployed
(c) Integrate real-time software composition analysis SCA tools like Black Duck or Snyk into the development pipeline to continuously monitor and manage open-source components

2. **Phase-II Development**:

(a) **Secure Coding Practices**

- Validate and sanitize user inputs to prevent injection attacks (e.g., SQL injection, cross-site scripting) (OWASP TOP 10 Vulnerability)*.
- Implement server-side input validation to ensure data integrity.
- Use parameterized queries and prepared statements to prevent SQL injection vulnerabilities in database interactions.
- Avoid dynamic SQL queries with user-supplied input.
- Encode output data to protect against cross-site scripting (XSS) attacks.

- Implement context-aware encoding based on the HTML, JavaScript, or other output contexts.
- Use strong and well-established authentication mechanisms (e.g., OAuth, OpenID).
- Implement authorization checks at both the server and client sides to enforce proper access controls.
- Use secure, random session identifiers and tokens.
- Implement session timeout and rotation strategies.
- Store session data securely and avoid client-side storage of sensitive information.
- Implement detailed error messages for developers and user-friendly messages for end-users.
  i. Log errors securely to prevent exposure of sensitive information.
  ii. Avoid displaying stack traces or detailed error messages in production.
- Use strong encryption algorithms for sensitive data at rest and in transit.
- Implement Transport Layer Security (TLS) for secure communication.
- Encrypt sensitive information in databases and configuration files.
- Implement anti-CSRF tokens in forms and requests.
- Ensure that state-changing requests are protected against CSRF attacks.
- Utilize HTTP security headers such as Content Security Policy (CSP), Strict-Transport-Security (HSTS), and X-Content-Type-Options.
- Integrate log monitoring and alerting for real-time detection of security incidents.
- Secure APIs with proper authentication (e.g., API keys, OAuth tokens) and Validate and sanitize inputs for API requests.
- Implement rate limiting and throttling to protect against abuse.
- Validate and sanitize inputs for API requests.
- Validate file types and extensions during upload, implement proper file size limits and conduct security scans on uploaded files.
- Store uploaded files in a secure location with restricted access.
- Use environment variables or secure vaults for storing sensitive information.
- Regularly audit and update configuration settings based on security best practices
- Implement dynamic and context-aware access control using ABAC (Attribute-Based Access Control) models to prevent broken access control (OWASP TOP 10 Vulnerability)*.
- Implement Perfect Forward Secrecy (PFS) in TLS sessions and homomorphic encryption for data processing without decryption to mitigate Cryptographic Failures (OWASP TOP 10 Vulnerability)*.
- Use tools like Ansible, Chef, or Puppet for automated and consistent configuration management to prevent security misconfigurations (OWASP TOP 10 Vulnerability)*.
- Maintain a denylist of internal or sensitive URLs and IP addresses that the application should not be able to access. This includes private, loopback, and link-local addresses to prevent server-side forgery (OWASP TOP 10 Vulnerability)*.
- Integrate application logs with Security Information and Event Management (SIEM) systems like Splunk or IBM QRadar for advanced analysis to prevent secure logging and monitoring failures (OWASP TOP 10 Vulnerability)*
- Utilize memory-safe languages like Rust for developing critical components, reducing the risk of deserialization vulnerabilities and software and data Integrity failures (OWASP TOP 10 Vulnerability)*
- Utilize advanced SSO protocols like SAML 2.0 or OpenID Connect for centralized authentication management to reduce risk of Identification and Authentication Failures (OWASP TOP 10 Vulnerability)*.
- Integrate advanced SCA tools into the development pipeline for real-time alerts on vulnerable dependencies to mitigate the risk of vulnerable and outdated components.

(b) **Data and Data Processing Units**

i. Authentication, Access Controls, and IDOR
- Implement strong authentication (multi-factor, biometric, token-based) and access controls with the principle of least privilege.
- Use strong hashing algorithms for password protection.
- Employ encryption protocols (e.g., TLS) for securing data in transit and at rest.

- Implement secure key management practices for enhanced data protection.
- Utilize robust input validation, secure indirect references (token-based access control), and DLP solutions.
- Monitor network patterns for suspicious activities and restrict access to sensitive data based on user roles.

ii. Remote Code Execution (SQL Injections and XSS)
- Sanitize user input and apply content security policies.
- Regularly scan and validate web applications for vulnerabilities.
- Use parameterized queries, prepared statements, input validation, and proper escaping to prevent SQL injections.

iii. Insecure File Uploads, Compliance, and Data Exfiltration Controls
- Restrict file types, enforce size limits, and validate contents during uploads.
- Store uploads in an isolated location from the web root for enhanced security.
- Ensure compliance with data protection laws in operating countries.
- Prevent PII exposure in exploitable formats.
- Consider outsourcing data storage to compliant specialists for enhanced security.

iv. Deep Learning Models (Security and Maintenance)
- Evaluate models against adversarial attacks and prioritize interpretability.
- Adopt Differential Privacy for individual user data protection.
- Use data preprocessing and outlier detection for training data quality.
- Train models in secure environments, regularly audit, and secure supporting infrastructure.
- Implement RBAC for model access, secure model APIs, and use HTTPS and API keys for communication.
- Monitor and log API activities for anomaly detection, continuously monitor data drift and schedule regular model retraining.
- Keep third-party libraries updated, conduct thorough integration tests, use version control for tracking changes, and maintain records for auditing.

(c) **Network Security**
- Implement network segmentation to isolate critical components, using firewalls and VLANs for traffic control.
- Enforce strong encryption protocols like TLS/SSL, regularly updating standards for resilience.
- Configure firewalls to restrict unnecessary traffic and employ intrusion detection and prevention systems.
- Implement WPA3 for Wi-Fi security and use VPNs with strong encryption for remote access.
- Set up centralized logging, keep network devices up to date with security patches and schedule regular vulnerability assessments.

(d) **Third-party codes**
- Dependency Management Tools enable specifying third-party dependencies in a configuration file, downloading them on demand during the build process. They facilitate the easy addition of new dependencies and rebuilding the software stack in different environments.
- Operating System Patches, issued by vendors like Red Hat and Microsoft, provide security updates. Track the version of each library, ensuring timely server upgrades.
- Integrity Checks, implemented by dependency managers and patching tools, use checksums as digital fingerprints to ensure the delivery of uncorrupted software components.
- Software Tools, such as npm audit for Node.js, bundler-audit gem for Ruby, and OWASP's dependency-check for Java and .NET, help cross-check against vulnerability databases.
- Stay Alert to Security Issues by monitoring security alerts from hosting providers and software vendors.

(e) **Authentication (cookies should be treated with a high level of security)**
- Apply the Secure attribute to prevent the cookie from being transmitted over non-HTTPS connections.
- Define an explicit expiration for persistent cookies used for authentication or session management Expire the cookie in the browser and destroy the server-side session object.

- Generate a strong pseudo-random number if the cookie's value is an identifier used to retrieve data
- Encrypt the cookie if it is descriptive Include a Keyed-Hash Message Authentication Code (HMAC)1 to protect against manipulation
- Rate-limiting places a ceiling on the number of requests a user may make within a period
- Explore blockchain-based decentralized identity solutions that allow users to control their identity and credentials, reducing reliance on centralized identity providers and minimizing single points of failure.
- Use Deep Packet Inspection(DPI) to examine and analyze network traffic at the application layer to detect unusual patterns or anomalies during the authentication process.
- Integrate advanced anomaly detection systems that use sophisticated algorithms to identify deviations from normal behaviour.
- If you're uploading files with a known file type consider adding some file type checking in your code. Make sure the Content-Type header in the HTTP request of the upload matches the expected file type, but be aware that an attacker can easily spoof the header.

(f) **Authorisation rules**

- Access control lists- is a list of permissions, specifying the actions that each user or account can perform on that object
- You should attempt to centralize access control decisions in your codebase, There are many ways of implementing authorization rules such as using function or method decorators, URL checking, or inserting inline assertions in the code.
- Get in the habit of writing new unit tests describing access control rules as you add features to your site Providing Role-Based Access Control and Ownership-Based Access Control

(g) **Web Security (This section deals with mitigation of significant web attacks and vulnerabilities)**

i. **Race conditions**

- Implement synchronization mechanisms like barriers, mutexes, and semaphores for effective multi-threading.
- Minimize shared data, ensure internal consistency in session handling, and guarantee atomicity for state changes.
- Consider client-side state management with encryption, like JWTs, to simplify server-side complexities.

ii. **Template Injections**

- Utilize secure template engines with built-in injection prevention and a proven security track record. Implement a robust Content Security Policy (CSP) to restrict content execution in the browser and mitigate injection attack impacts.
- Opt for "logic-less" template engines like Mustache to separate logic and presentation, reducing exposure to template-based attacks. users' code only in a sandboxed environment, removing potentially dangerous modules and functions for enhanced security.

iii. **HTTP Request Smuggling**

- Implement end-to-end HTTP/2, disabling HTTP downgrading; if not possible, validate rewritten requests against HTTP/1.1 specifications.
- Normalize ambiguous requests at the front-end server and reject any remaining ambiguity at the back-end server.
- Disable unnecessary HTTP methods like TRACE and OPTIONS, and conduct rigorous checks for conflicts in Transfer-Encoding and Content-Length headers.
- Terminate requests that cannot be confidently parsed within a specified timeframe to prevent potential smuggling attacks.

iv. **Active Directory**

- Keep systems, especially Active Directory (AD), updated and patched. Use network segmentation and firewalls to control AD server traffic.
- Encrypt data in transit with Secure LDAP, enable SMB signing and implement Privileged Access Management (PAM) for monitoring elevated accounts.
- Limit administrative account use, disable legacy protocols and enforce security settings with Group Policy Objects (GPOs) in AD.

v. **Enumeration**
- Implement rate limiting and CAPTCHAs for login attempts to thwart automated attacks.
- Employ firewalls and IPS to detect and control enumeration attempts.
- Avoid revealing sensitive information in public-facing responses, disable unnecessary directory listings, and use randomized strings for usernames or IDs to enhance security.

vi. **Deserialisation**
- Minimize deserialization of user input unless essential.
- Implement a digital signature to ensure data integrity before deserialization.
- Use up-to-date and secure serialization libraries, following best practices.
- Opt for custom serialization methods to include only necessary fields and avoid unnecessary or sensitive data exposure.

vii. **GraphQL API Vulnerabilities**
- For a secure GraphQL API, disable introspection for non-public use and review the schema to prevent unintended field exposure.
- Limit query depth, set operation and byte limits, and ensure JSON-encoded POST queries with a secure CSRF token mechanism.
- Employ WAFs configured for GraphQL-specific protection.

viii. **Web Cache Poisoning**
- For Web Cache Poisoning prevention, consider disabling caching or restricting it to static responses.
- Implement strong cache control headers, normalize cache keys, and understand third-party technology implications.
- Separate user input from application control logic, avoid fat GET requests and use security headers like X-Content-Type-Options and X-Frame Options.
- Regularly patch client-side vulnerabilities and evaluate the necessity of absolute URLs.

ix. **HTTP Host Header Attack**
- Prevent HTTP host header attacks by avoiding the use of the Host header in server-side code.
- If necessary, check it against a whitelist, avoid supporting additional headers like X-Forwarded-Host, and configure load balancers to forward requests only to permitted domains.
- When using absolute URLs, manually specify the current domain in a configuration file to eliminate the threat of password reset poisoning.

x. **Prototype Pollution**
- Mitigate prototype pollution by sanitizing property keys before merging into objects, using a whitelist or blacklist.
- Prevent changes to prototype objects and object inheritance. Avoid unsafe recursive merge functions, opt for deep copies, and employ JavaScript linters to identify and prevent unsafe code patterns.

xi. **Web Socket-based attacks**
- Secure WebSockets by using the wss:// protocol with TLS.
- Authenticate users before establishing connections, employing HTTP-based authentication.
- Apply authorization checks, and hard code the WebSocket endpoint URL, avoiding user-controllable data in the URL.
- Treat received WebSocket data as untrusted, ensuring safe handling on both server and client ends to prevent vulnerabilities like SQL injection and cross-site scripting.

xii. **CORS-Based Attacks**
- Secure against CORS attacks by properly specifying origins in the Access-Control-Allow-Origin header.
- Implement server-side checks for the Origin header, avoiding wildcards (*) for sensitive resources.
- Exercise caution with Access-Control-Allow-Origin: null.
- Enable Access-Control-Allow-Credentials only if essential with a strong authentication system. Use CSP headers to mitigate data theft or XSS attacks by specifying allowed resource origins.

xiii. **JWT Attacks**

- Perform robust signature verification, safeguard keys, and consider unexpected algorithms.
- Enforce a strict whitelist for the jku header to control permitted hosts.
- Avoid storing sensitive information in JWTs due to their easy decoding.
- Set HttpOnly and Secure attributes for HTTPS transmission when using cookies.
- Always set an expiration date for issued tokens.
- Avoid sending tokens in URL parameters.
- Include an audience claim to specify the intended recipient, preventing misuse.
- Enable token revocation by the issuing server, especially on logout.

xiv. **DOM Attacks**

- To prevent DOM attacks, ensure that objects or functions are as expected and not DOM nodes.
- Avoid code referencing global variables with the logical OR operator ||.
- Utilize a trusted library like DOMPurify for DOM-clobbering vulnerabilities.
- Refrain from dynamically writing untrusted data to DOM-data fields or parsing untrusted strings as JSON.
- Avoid incorporating untrusted data into XPath queries.

3. **Phase-III: Testing**:

(a) **Static Analysis**

- Regular Source Code Scans: Employ static analysis tools to regularly scan the source code, ensuring security checks before making changes.
- Binary Analysis for Security Weaknesses: Analyze compiled binaries without execution to detect vulnerabilities like memory leaks and buffer overflows, preventing code injection and remote execution attacks.
- User Input Handling: Review code for unvalidated inputs and improper user data handling, mitigating injection attacks and preventing XSS attacks by treating configurations as code.
- Configuration Security Scans: Regularly scan and correct insecure configurations treating configurations as code, mitigating Insecure Direct Object Reference through strict user permissions and input validation practices.

(b) **Dynamic Analysis**

- Runtime Session Handling Analysis: Conduct runtime analysis for flaws in session handling mechanisms, ensuring secure session management through identification and rectification of weaknesses.
- Authentication and Authorization Testing: Test authentication and authorization controls during runtime, identifying and rectifying weaknesses in these security measures.
- Continuous Monitoring for Unvalidated Inputs: Continuously monitor unvalidated inputs during application runtime, identifying and preventing injection vulnerabilities in real-time.
- Crafted Fuzz Testing Inputs: Use well-crafted inputs to thoroughly test the application and record metadata in the crashed state, providing engineers with details to understand the crash's nature, prepare fixes, or prioritize bugs. Open-source tools like OSS-Fuzz can be used to implement the same.
- Fuzz Testing Integration: Integrate automated fuzz testing into the CI pipeline, parameterizing fuzzing tests for web applications, APIs, file input formats, and network protocols to uncover vulnerabilities.

4. **Phase-IV: Deployment**:

(a) **Deployment**

- Employ firewalls for network segmentation, VLANs, and security policy enforcement at the application level.
- Implement Software-Defined Networking (SDN) for dynamic segmentation adjustments, including micro-segmentation.
- Define limited administrative roles, designate multiple global admins, and enforce multi-factor authentication.
- Restrict user consent for unmanaged applications and ensure end-to-end encryption for secure data transmission.

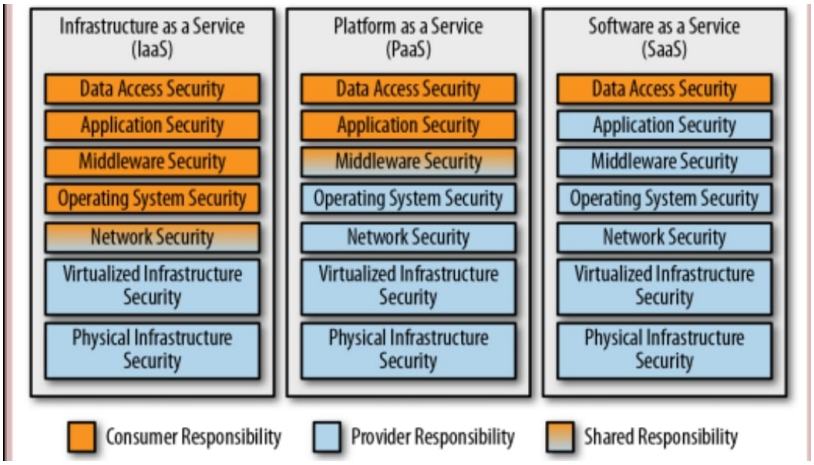


Figure 2: Cloud shared responsibility model

- Apply strong encryption algorithms for data at rest and configure firewalls to allow only necessary traffic.
- Regularly review and update firewall rules based on security policies.
- Implement runtime protections using tools like Falco for continuous anomaly detection.
- Keep container runtimes updated to leverage the latest security features and enhancements.

(b) **Cloud**

- Employ AWS Config, Azure Policy, or Google Cloud Security Command Center for continuous compliance monitoring and automated remediation of non-compliant resources. Utilize endpoint detection and response (EDR) solutions with machine learning and behavioral analysis for threat detection and response. Implement egress filtering to mitigate data exfiltration risks and prevent watering hole attacks. Enhance security with a web application firewall (WAF) to protect against common programming errors and vulnerabilities in dependencies. Minimize attack surface using bastion hosts, which serve as dedicated, hardened hosts for other machines to hide behind.

5. **Phase-V: Maintenance**:

(a) **Host and Network Security**

- Disable unnecessary services and applications.
- Apply the principle of least privilege to user accounts.
- Implement firewalls, network segmentation, and intrusion detection/prevention systems.

(b) **Secure Container Practices**

- Scan and update container images regularly.
- Sign and verify container images for integrity.
- Pod Security and Runtime Measures:

(c) **Define/enforce Pod Security Policies**

- Use runtime security tools like Falco for anomaly detection.

(d) **Vendor Security (For Third-Party Services)**

- Conduct security assessments.'
- Establish contractual agreements on security practices.
- Regularly review and analyze audit logs for security incidents.

(e) **Log Analysis**

- Establish a Baseline: Analyze log data for behaviour patterns, employing time-series analysis to discern daily, weekly, and seasonal trends. Enhance anomaly detection by correlating data from security devices, network equipment, servers, and applications.

- Automated Log Centralization: Use ELK Stack or Splunk for centralized log data, ensuring a comprehensive view of network activities.
- Structured Logging (JSON): Promote JSON for structured logs, enhancing automated analysis through simplified parsing.
- Real-Time Monitoring: Utilize tools for continuous log analysis, promptly identifying anomalies as they occur for swift response.
- Statistical and ML Algorithms: Implement k-means clustering, isolation forests, and CEP tools like Apache Flink for real-time log analysis.
- Alert Configuration in SIEM: Configure alerts in SIEM systems, prioritizing based on potential impact and source reliability for timely responses.

Automation

- **Static Testing:** Static testing is integral during application development, parsing the codebase before each change to detect vulnerabilities. This proactive measure aims to mitigate risks like injections, remote code executions, enumeration attacks, information leaks, and memory leaks.
- **Dynamic Testing:** Post-development, dynamic testing takes over, employing attack generation and fuzzing for comprehensive application testing. The crash metadata is stored to pinpoint vulnerabilities. Automated attack generation, using deep learning models and the Mitre attack framework, ensures realistic scenarios. Fuzzing engines like OSS-Fuzz are employed, utilizing well-crafted payloads developed by deep learning models during application development.
- **Vulnerability Ranking:** Vulnerabilities are ranked based on severity, difficulty, and impact, prioritizing the resolution of more severe issues.
- **Automated Patch Management:** Integrate an automated patch management system to promptly apply security patches and updates. Regularly scan for vulnerabilities in third-party libraries and dependencies, ensuring a proactive stance against emerging threats.
- **Collaboration with Threat Intelligence:** Integrate with threat intelligence feeds to stay updated on the latest threats and vulnerabilities. Automation aids in correlating threat intelligence data with the organization's environment, enhancing threat detection capabilities.
- **Intrusion Detection System (IDS):** Deploy an Intrusion Detection System to safeguard the application from potential malicious actors. Automated responses based on IDS alerts enhance the ability to prevent unauthorized access and detect suspicious activities.
- **Continuous Improvement through Feedback:** Implement an automated feedback loop that gathers insights from incidents, security assessments, and user feedback. Use this information to continually refine and improve the automation-driven security baseline.

4 PS 3

4.1 Problem Statement

Approach for application development that leads to applications having features to detect, report and respond to attempts of attacks
Deliverables:

- Suggest the elements/techniques that can be incorporated into the development process that can detect, report, and respond effectively to various forms of cyber threats
- The suggested elements/techniques should be provided along with the action plan to implement the same

Secure Software Development Lifecycle (SSDLC)

In traditional Software Development Life Cycles (SDLC), security measures were often an afterthought, typically introduced late in the development process. This approach led to the discovery of bugs, flaws, and vulnerabilities at advanced stages, making their resolution significantly more costly and time-consuming. Commonly, security wasn't a focal point during the testing phase, resulting in end-users encountering bugs post-deployment. In contrast, Secure SDLC models integrate security considerations at every stage of development.

Why Secure SDLC Matters

According to a study by IBM's Systems and Sciences Institute, the cost implications of fixing bugs vary dramatically across the SDLC phases. Fixing a bug during implementation can be up to six times more expensive than addressing it during the design phase. This cost escalates to 15 times more during the testing phase and can be up to 100 times higher during maintenance and operations. The diagram below illustrates these cost disparities:

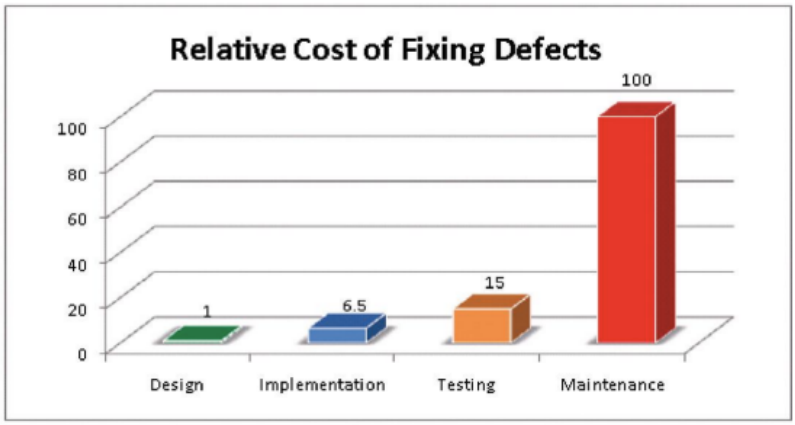


Figure 3: Relative cost of fixing defects [2]

Beyond accelerating development and reducing costs, integrating security throughout the SDLC can significantly lower business risk by early detection and mitigation of vulnerabilities. Security integration examples across various SDLC stages include architectural analysis during the design phase, code reviews and scanning during development, and conducting security assessments like penetration tests before deployment.

In traditional, waterfall-based models, security testing often occurred at the end of the lifecycle, just before production. However, in more agile methodologies, a "security by design" approach is adopted. For instance, if penetration testing in a waterfall model reveals an SQL injection flaw, rectifying this bug could necessitate a complete cycle of redesign, application of changes, and retesting. In contrast, agile methodologies facilitate proactive discussions during the planning phase to prevent such flaws. Decisions about using parameterization to mitigate SQL injection risks can prevent the need for extensive rollbacks, thus incurring only the cost of a planning discussion rather than a full cycle of remediation.

4.2 Proposed Solution

Implementing Secure Software Development Lifecycle (SSDLC)

Secure SDLC integrates security processes throughout all lifecycle phases, as highlighted in the Intro To DevSecOps. It emphasizes the inclusion of security testing tools and the concurrent development of security and functional requirements.

Understanding Security Posture

- **Gap Analysis**: Conduct a comprehensive gap analysis to evaluate the existing security activities and policies within the organization. This analysis helps in identifying the effectiveness of current practices and where improvements are needed

- **Software Security Initiatives (SSI)**: Establish realistic and measurable goals for security initiatives. Examples include setting up Secure Coding Standards and creating playbooks for data handling. Utilize project management tools to track the progress of these initiatives
- **Formalizing Security Processes**: Once a security program or standard is initiated, dedicate a period for engineers to familiarize themselves with it. Collect feedback before full enforcement. Ensure that every policy is backed by defined procedures for effective implementation
- **Security Training and Tools**: Invest in comprehensive security training for engineers and provide them with the necessary tools. Prioritize training early in the process, ideally before the implementation or onboarding of new tools

SSDLC Processes

After establishing a solid security posture, focus on embedding security within the SDLC.

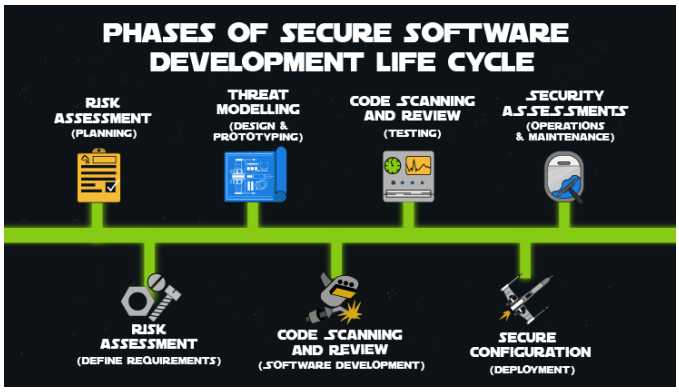


Figure 4: Phases of Secure-SDLC [3]

- **Risk Matrix and Assessment**: During the early stages of SDLC, prioritize identifying security considerations alongside functional requirements. This 'security by design' approach should be integrated into the planning and requirements stages. For instance, in a user interface like a blog, ensure that users can only perform actions that are intended, like viewing a blog entry without being able to edit or delete it
- **Threat Modelling**: Conduct threat modeling to identify potential threats, especially where there are no adequate safeguards. Effective during the design stage, it helps in contrasting what should not happen against the intended behavior and interaction of the software. For example, verifying user requests for account information to prevent unauthorized access
- **Code Scanning / Review**: Implement both manual and automated code reviews. Leverage technologies for Static and Dynamic Security testing during the development stages. This ensures that the code being written is continuously evaluated for security vulnerabilities
- **Security Assessments**: Incorporate practices like Penetration Testing and Vulnerability Assessments. These assessments, performed during the Operations and Maintenance phase, help in identifying potential exploitation paths in the application. While Vulnerability Assessments provide a hypothetical analysis, Penetration Testing actively attempts to exploit the identified flaws to validate their presence and impact

To develop a comprehensive and secure application development approach, various techniques and best practices should be integrated into the development lifecycle.

4.2.1 Risk Assessment in Secure SDLC

Risk assessment in the SDLC is crucial for identifying and mitigating potential threats. It evaluates the likelihood and impact of threats exploiting vulnerabilities in software, considering factors like design flaws or poorly reviewed code.

- **Assumption of Attack**: Begin by assuming the software will be attacked. Consider motivators for threat actors, such as the value of data, security levels of dependencies, client profiles, and distribution scale of the software
- **Defining Acceptable Risk**: Determine the acceptable level of risk based on factors like potential data loss versus the cost of fixing security bugs. Involve various stakeholders in this decision-making process to understand the trade-offs and implications of risk
- **Risk Evaluation**:
  - Worst-Case Scenario Analysis: Consider the most severe outcomes of successful attacks, like ransomware incidents
  - Valuation of Data: Determine the value of potentially stolen data, considering user identity, credentials, network control, and other assets
  - Attack Complexity: Assess the difficulty of successfully executing an attack
  - Impact Assessment: Evaluate the impact of different types of attacks on different user groups and systems
  - Accessibility of Target: Analyze the target's network accessibility, authentication requirements, and vulnerability to external requests
- **Types of Risk Assessments**:
  - Qualitative Risk Assessment: Classify risks into thresholds like "Low", "Medium", and "High", using a formula like Risk = Severity x Likelihood
  - Quantitative Risk Assessment: Assign numerical values to risk levels, using tools or calculations specific to the company's processes
- **Timing in SDLC**: Perform risk assessments early in the SDLC, during planning and requirement phases, to anticipate potential issues like data exfiltration. Use quantitative risk analysis post-development to evaluate specific risks and determine appropriate spending on security controls
- **Risk Matrix Development**: Create a Risk Matrix to visualize and prioritize risks based on their severity and likelihood like shown in *Figure* 5[1]

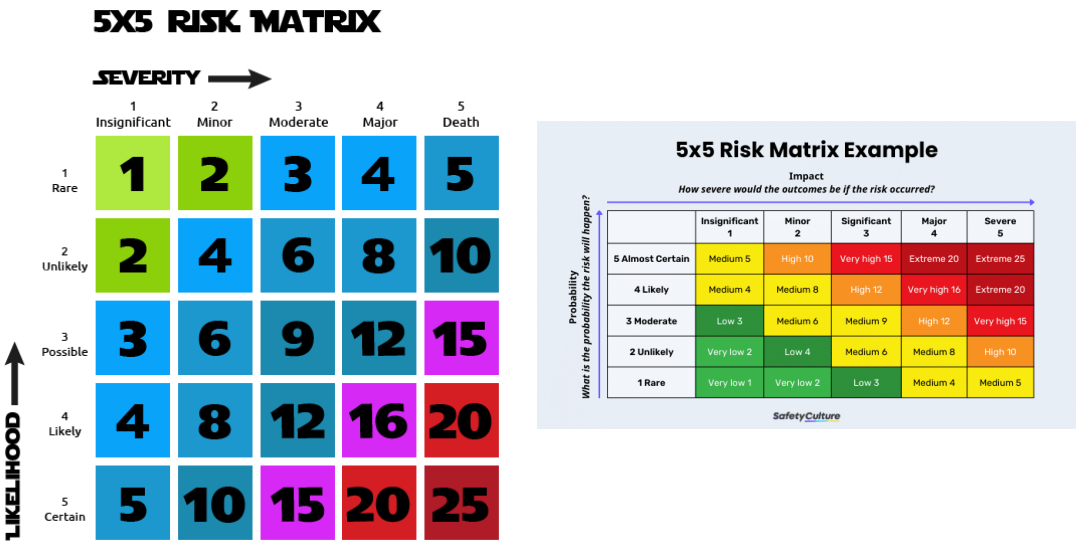


Figure 5: Risk Matrix [3, 4]

[1]More details can be found here: [4] or Link

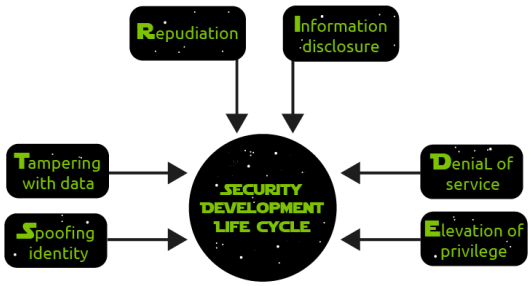
Continuous Improvement

- **Review and Adaptation:** Regularly update risk assessments to reflect changes in the threat landscape, business processes, or technological advancements
- **Stakeholder Engagement:** Include diverse groups in the risk assessment process, such as IT, security, business units, and possibly legal teams, for comprehensive risk identification and management
- **Documentation and Communication:** Maintain detailed records of risk assessments, including methodologies, findings, and decisions made. Communicate these details to all relevant parties for transparency and informed decision-making
- By systematically assessing and managing risks throughout the SDLC, organizations can proactively address potential security threats, aligning their security posture with their broader business objectives and regulatory requirements

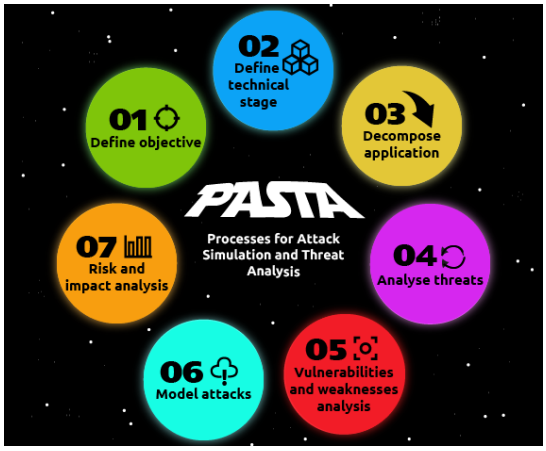
4.2.2 Threat Modeling

Threat modelling, a critical component of Secure SDLC, is most effective when integrated into the design phase, prior to coding. It involves identifying potential security threats and prioritizing countermeasures to protect high-risk or valuable data assets identified during risk assessment.

- **Start Early**: Incorporate threat modeling at the beginning of the software development process to identify and address potential threats early, minimizing the cost and effort required for later remediation
- **Diverse Stakeholders**: Engage various stakeholders including developers, architects, security experts, business representatives, and end-users to identify a wider range of threats and develop more effective mitigation strategies
- **Structured Methodology**: Use structured approaches like STRIDE, DREAD, OCTAVE, or PASTA for identifying and prioritizing threats



(a) STRIDE [3]



(b) PASTA [3]

Figure 6: Threat Modelling

- **Continuous Review and Update**: Regularly revisit the threat model to ensure its continued effectiveness as the threat landscape evolves

4.2.3 Code Reviews

- **Security Focus**: Regular code reviews should prioritize identifying security vulnerabilities. Utilize tools for Static Application Security Testing (SAST) to automatically scan the code. These tools can detect vulnerabilities such as buffer overflows, SQL injection, and cross-site scripting

- **Developer Training**: Implement a comprehensive training program for developers in secure coding practices. Focus on common security risks relevant to their development area. For instance, web developers should be trained on vulnerabilities like cross-site scripting and SQL injection
- **Issue Remediation and Verification**: Establish a process for the remediation of identified security issues. This includes re-testing the code to confirm the effectiveness of the fixes. Document the entire process, capturing the nature of the vulnerabilities, the remediation steps taken, and the results of post-remediation testing

4.2.4 Dependency Scanning

- **Regular Scanning**: Implement regular scans using tools like OWASP Dependency-Check. These tools help identify known vulnerabilities in third-party libraries and dependencies used in your software
- **Investigation and Action**: Upon detecting vulnerabilities, conduct a thorough investigation to understand the impact and develop a mitigation plan. Update or replace vulnerable components as necessary

4.2.5 Security Testing

- **CI/CD Integration**: Integrate automated security testing tools into the Continuous Integration/Continuous Deployment (CI/CD) pipeline. Use tools like OWASP ZAP, Burp Suite, and Nessus for dynamic (DAST) and static (SAST) application security testing
- **Comprehensive Testing Coverage**: Ensure that the security testing covers critical areas such as authentication mechanisms, authorization protocols, data validation processes, encryption standards, and error handling procedures

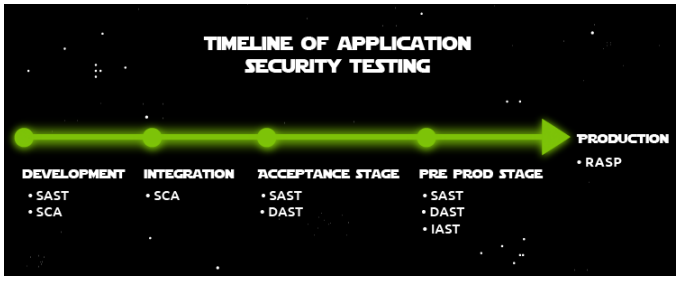


Figure 7: Timeline of Security Testing [3]

4.2.6 Incident Response Plan

- **Well-Defined Plan**: Develop a comprehensive incident response plan detailing the roles, responsibilities, and procedures to follow in case of a security incident
- **System Monitoring and Immediate Action**: Use intrusion detection systems to monitor for suspicious activities. In the event of an incident, immediately isolate affected systems and apply necessary patches or fixes

4.2.7 Patch Management

- **Vulnerability Assessment**:
  - Regular Scanning: Implement a routine for scanning systems and software using tools like OpenVAS. This includes not just the primary software but also any third-party components or libraries it depends on.
  - Identification: Identify vulnerabilities that could be exploited by attackers. This includes assessing the severity and potential impact of each vulnerability.
  - Documentation: Maintain detailed records of the identified vulnerabilities, including their nature, severity, and potential impact on the system.

- **Timely Remediation**: Upon discovering vulnerabilities, prioritize their remediation. Update software to patch vulnerabilities and, if necessary, isolate and remediate affected systems immediately to prevent exploitation
- **Update and Patch Deployment**:
  - Prioritization: Based on the assessment, prioritize the vulnerabilities that need immediate attention. This prioritization should consider the severity of the vulnerability and its potential impact on the system.
  - Patch Selection and Testing: Choose appropriate patches or updates to address the vulnerabilities. Before full deployment, test these patches in a controlled environment to ensure they do not adversely affect the system's functionality.
- **Remediation Process**:
  - Immediate Action: In cases where vulnerabilities are critical, take immediate action to remediate them. This might involve updating the software, modifying configurations, or applying patches.
  - Isolation: If necessary, isolate affected systems to prevent the vulnerability from being exploited until the patch is applied. This might involve temporarily restricting access to certain features or components of the software.
- **Post-Remediation Testing and Monitoring**:
  - Testing: After applying patches, conduct thorough testing to ensure that the vulnerabilities have been effectively addressed and that there are no new issues introduced by the patches.
  - Continuous Monitoring: Continuously monitor the software and systems for any signs of security breaches or new vulnerabilities, especially in the immediate aftermath of applying a patch

4.2.8 Feedback Loop

- **Collaborative Approach**: Establish a feedback loop between development and security teams to continuously improve security practices and respond to new threats
- **Establishing Communication Channels**: Set up a structured feedback loop between the development and security teams. This can be facilitated through regular meetings, shared reporting systems, and open channels of communication
- **Continuous Improvement**: Use the feedback to continuously improve security practices, update policies, and adapt to emerging threats and technologies

Application Security in SDLC

Middleware as a Security Solution

Integration

- **Package-Based Middleware**: Develop a middleware package that can be integrated with different applications, providing a centralized security solution
- **API Gateway**: Implement an API gateway that acts as a single point of entry for all application traffic, equipped with security measures to monitor and control API requests and responses

Features

- **Security Management**: The middleware will handle security analysis, reporting, and response, relieving individual applications of these tasks
- **Efficiency**: This centralized approach reduces redundancy and conserves resources compared to having each application manage its security

Application Security in SDLC

Artificial Intelligence and Machine Learning

- **Baseline Parameter Setting**: Utilize AI to establish baseline parameters for normal application behavior, creating a reference point for identifying anomalies
- **Anomaly Detection**: Implement AI-driven systems to monitor and detect deviations from baseline behaviors, indicating potential security threats
- **Monitoring and Analysis**: Continuously track all application requests and responses. Use machine learning algorithms to analyze this data for signs of security breaches or malicious activities

User Security Level Management

- **Implementation**:
  - Develop a system where each user is assigned a security level based on their interaction with the application
  - Increase the security level of users engaging in suspicious activities or attempting unauthorized actions
- **Response Mechanism**:
  - Warnings: Issue warnings to users who reach a predefined security level (e.g., level 3).
  - Temporary Ban: Implement automatic temporary bans for users who exceed a critical security level (e.g., level 5), preventing access to the application for a designated period

Implementation Plan

- **Middleware Development and Deployment**:
  - Design and develop the middleware security package and API gateway
  - Integrate the middleware with existing applications, ensuring compatibility and minimal disruption
- **AI and ML Integration**:
  - Train AI models using historical application data to establish baseline behaviors
  - Integrate the AI systems into the middleware for real-time anomaly detection and threat analysis
- **User Security Level System**:
  - Develop algorithms for assessing user behavior and dynamically adjusting security levels
  - Set up automated systems for user warnings and temporary bans based on security level thresholds
- **Testing and Evaluation**:
  - Conduct comprehensive testing of the middleware, AI systems, and user security level mechanisms
  - Monitor and evaluate the performance, ensuring effective detection and response to security issues
- **Continuous Improvement**:
  - Regularly update AI models and middleware features based on evolving threats and user feedback
  - Maintain an agile approach to adapt quickly to new security challenges and technological advancements

5 References

[1] *Owasp top 10*, `https://owasp.org/www-project-top-ten/`.

[2] M. Dawson, D. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (sdlc)," *Journal of Information Systems Technology and Planning*, vol. 3, pp. 49–53, Jan. 2010.

[3] *Tryhackme*, `https://tryhackme.com/`.

[4] *Safety culture*, `https://safetyculture.com/topics/risk-assessment/5x5-risk-matrix/`.