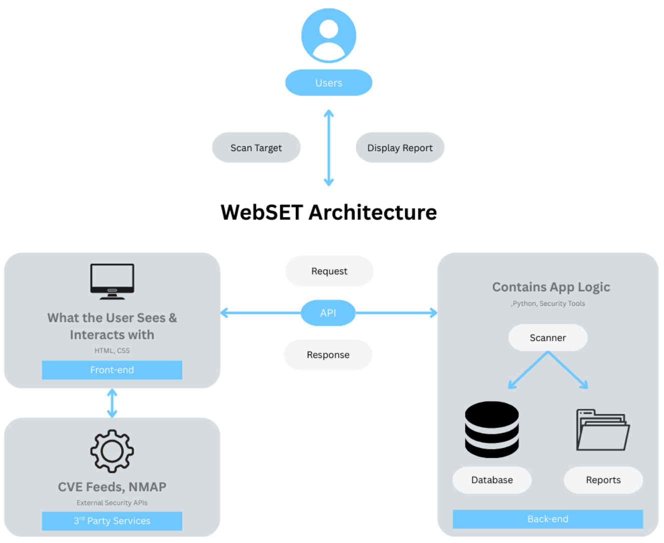
WEBSET PROJECT CONTENT

1. Test Cases

2.1 Web Application Security

Web Application Security(**WAS**) defined in relation to the WebSET product and ILC goals refers to the confidentiality, integrity and availability of the components, structure, and tech stack of Web Applications. This typically includes understanding of the Front-end (Client-side), Back-end (Server-side), Databases and software components/integrations.



2.2 OWASP Top 10 Vulnerabilities

The Open Worldwide Application Security Project (**OWASP**) is an international standard and repository for web application security vulnerabilities and remediation. The top 10 vulnerabilities provided by OWASP will provide a basis for the development of our product prototype and serve as a reputable standard and comparable benchmark for other products.

The development of test cases for WebSET's scanning engine will be structured from analysis of the OWASP top 10 (**OTT**) and subsequent Common Weakness Enumerations for software mapped to each vulnerability.

Analysis of the OTT to contextualise for WebSET's scans:

2.2.1 A01 - Broken Access Control

Broken Access Control refers to the inadequate enforcement of restrictions for authenticated users, resulting in privilege escalation or actions that are not authorised for the user role. This vulnerability can lead to unauthorised disclosure of information and stored data, or the modification, or destruction of the web application.

Static analysis can be used to identify issues pertaining to broken access controls through examining code configuration. Scanning scripts will be created to identify lack of authentication checks, hardcoded role assignments and errors or inconsistencies in validation of permissions. Database queries and function elements should be under scrutiny.

2.2.1.1 Scope

- Client-side authorisation checks
- Hardcoded credentials and tokens
- Direct object references in URLs/parameters
- Missing access control enforcement
- Privilege escalation patterns

2.2.1.2 Patterns for Detection

- Search for: `role`, `admin`, `auth`, `token`, `key`, `secret`
- Regex patterns for API keys: `[sS][kK]-[a-zA-Z0-9]{32,}`
- Look for client-side role checks without server validation
- Identify hardcoded credentials in variables

2.2.1.3 Test Cases: JavaScript

```javascript
// 1. Client-side only authorization
if (user.role === 'admin') {
    showAdminPanel();
}

// 2. Hardcoded API keys/tokens
const API_KEY = "sk-1234567890abcdef";
const authToken = "Bearer eyJ0eXAiOiJKV1QiLCJhbGc...";

// 3. Direct object references
fetch(`/api/users/${userId}/profile`);
window.location = `/admin/users/${id}/edit`;

// 4. Role-based logic in frontend
function canDeleteUser(currentUser, targetUser) {
    return currentUser.id === targetUser.id; // Missing admin check
}

// 5. Insecure parameter handling
const isAdmin = urlParams.get('admin') === 'true';
```

2.2.1.3 Test Cases: HTML

```html
<!-- 1. Hidden admin controls -->
<div id="adminPanel" style="display:none;">
    <button onclick="deleteAllUsers()">Delete All</button>
</div>

<!-- 2. Hardcoded credentials in forms -->
<input type="hidden" name="api_key" value="secret123">

<!-- 3. Direct object references in links -->
<a href="/user/profile/123">View Profile</a>
<a href="/admin/edit-user?id=456">Edit User</a>
```

2.2.2 A02 - Cryptographic Failures

Cryptographic Failures (previously “Sensitive Data Exposure”) refers to lack or failure of cryptographic measures to protect sensitive data. The vulnerability accounts for weak encryption, poor key management, or lack of encryption of data in transit and at rest.

Static analysis will examine the use of outdated or ineffective cryptographic algorithms, lack of encryption, hardcoded keys or passwords and transmission of data without secure protocols. The scans will account for effective use of salting, secure random number generation, secure storage of keys, and proper certificate validation.

2.2.2.1 Scope

- Weak or deprecated cryptographic algorithms
- Hardcoded cryptographic keys
- Improper random number generation
- Missing encryption for sensitive data
- Insecure key storage

2.2.2.2 Patterns for Detection

- Deprecated algorithms: `MD5`, `SHA1`, `DES`, `3DES`, `RC4`
- Weak patterns: `btoa()`, `atob()`, `Math.random()`
- Storage patterns: `localStorage`, `sessionStorage` with sensitive data
- Hardcoded keys: Look for `key`, `secret`, `salt` with string values

2.2.2.3 Test Cases: JavaScript

```javascript
// 1. Weak algorithms
const hash = CryptoJS.MD5(password);
const encrypted = CryptoJS.DES.encrypt(data, key);
btoa(password); // Base64 is not encryption
```

```
// 2. Hardcoded keys/salts
const encryptionKey = "mySecretKey123";
const salt = "fixedSalt";

// 3. Weak random generation
Math.random(); // For cryptographic purposes
Date.now(); // Predictable

// 4. Storing sensitive data in localStorage
localStorage.setItem('password', userPassword);
sessionStorage.setItem('creditCard', ccNumber);

// 5. Transmitting sensitive data without encryption
fetch('/api/login', {
    method: 'POST',
    body: JSON.stringify({password: pwd}) // Over HTTP
});
```

2.2.3 A03 - Injection

Injection refers to the abuse of commands, queries or input fields on a web application to execute unauthorised code or data obscured in a message. Examples include SQL injection, OS command injection, and LDAP injection, which can result in the execution of code and exploitation of the web application.

Static scanning will detect input fields and verify critical data flows to databases and sensitive information are sanitised. Input and output sanitisation and encoding should be accounted for.

2.2.3.1 Scope

- SQL injection in dynamic queries
- Cross-Site Scripting (XSS) vulnerabilities
- Command injection
- LDAP injection
- Template injection

2.2.3.2 Patterns for Detection

- Dangerous functions: `eval`, `Function`, `setTimeout` with strings
- DOM manipulation: `innerHTML`, `outerHTML`, `insertAdjacentHTML`
- SQL concatenation patterns: String concatenation with SQL keywords
- Template injection: `{{`, `${` with user input

2.2.3.3 Test Cases: JavaScript

```
// 1. SQL Injection (client-side query building)
const query = `SELECT * FROM users WHERE id = ${userId}`;
const sql = "UPDATE users SET name = '" + userName + "'";

// 2. XSS - Direct DOM manipulation
document.getElementById('content').innerHTML = userInput;
element.outerHTML = `<div>${userInput}</div>`;
$('#result').html(userData);

// 3. eval() usage
eval(userInput);
Function(userCode)();
setTimeout(userInput, 1000);

// 4. Command injection patterns
exec(`ping ${userInput}`);
system(userCommand);

// 5. Template injection
template = `Hello ${userInput}`;
```

2.2.3.4 Test Cases: HTML

```
<!-- 1. Direct user input rendering -->
<div id="userContent">{{userInput}}</div>
<script>var data = "{{untrustedData}}";</script>

<!-- 2. Unsafe attributes -->
<img src="{{userURL}}" onerror="alert('xss')">
<a href="javascript:{{userCode}}">Click</a>

<!-- 3. Form injection points -->
<input type="text" value="{{userValue}}">
```

2.2.4 A04 - Insecure Design

Insecure Design refers to weaknesses inherent in the web application's architecture. This may include incompatibilities, security gaps and lack of controls in place to ensure the application is inherently secure.

Static scans to detect Insecure Design will largely revolve around analysis of tech stack, versioning and configuration of platforms and dependencies. Controls will also be reviewed to prevent typical error handling / brute force defences such as rate limiting or throttling, as well as segmentation and isolation of key production or back-end systems.

2.2.4.1 Scope

- Lack of rate limits
- Insufficient input validation

- Lack of security controls in calculation logic
- Missing security header implementation
- Inadequate error handling

2.2.4.2 Patterns for Detection

- Missing validation patterns
- Mathematical logic without bounds checking
- Error handling that exposes sensitive information
- Absence of security controls in critical functions

2.2.4.3 Test Cases: JavaScript

```
// 1. No rate limiting on operations
function login(username, password) {
    // Direct login attempt without throttling
    return authenticateUser(username, password);
}

// 2. Insufficient validation
function processPayment(amount) {
    if (amount > 0) { // Missing upper limit, currency validation
        return chargeCard(amount);
    }
}

// 3. Mathematical logic flaws
function applyDiscount(originalPrice, discountPercent) {
    return originalPrice * (1 - discountPercent); // No validation on
discount range
}

// 4. Missing security headers
// No CSP, HSTS, or other security headers implementation

// 5. Sensitive information disclosure in errors
catch (error) {
    console.log(error); // Exposing stack traces
    alert("Database error: " + error.message);
}
```

2.2.5 A05 - Security Misconfiguration

Security Misconfiguration is a vulnerability relating to the improper management of established security controls, this includes improper definition, implementation, or maintenance. This may involve improper configuration of roles and access controls, improper alerting and triage, or exposure of sensitive information through security alerts.

Static analysis will aim to detect Security Misconfiguration by identifying configuration settings and security or error handling mechanisms are established securely.

2.2.5.1 Scope

- Default credentials usage
- Unnecessary features enabled
- Improper error handling
- Missing security headers
- Development artifacts in production

2.2.5.2 Patterns for Detection

- Debug flags: DEBUG, development, test
- Default credentials patterns
- Development comments (hardcoded API keys, backdoors)
- Localhost references in production code

2.2.5.3 Test Cases: JavaScript

```javascript
// 1. Debug mode enabled
const DEBUG = true;
console.log("Debug info:", sensitiveData);

// 2. Default credentials
const defaultUser = 'admin';
const defaultPass = 'password123';

// 3. Verbose error messages
function handleError(err) {
    return `Database connection failed: ${err.connectionString}`;
}

// 4. Development endpoints
if (window.location.hostname === 'localhost') {
    enableDebugMode();
}

// 5. Unnecessary features
// jQuery, unused libraries, development tools
```

2.2.5.4 Test Cases: HTML

```html
<!-- 1. Debug information -->
<div id="debug-info" style="display:block;">
    Database: mysql://user:pass@localhost:3306/db
</div>
```

```
<!-- 2. Development comments -->
<!-- TODO: Remove admin backdoor before production -->
<!-- API Key: abc123def456 -->

<!-- 3. Unnecessary scripts -->
<script src="dev-tools.js"></script>
<script src="debug-console.js"></script>
```

2.2.6 A06 - Vulnerable and Outdated Components

This vulnerability is centric around the improper detection, patching and version control of web application components and dependencies. This can lead to third-party compromise or supply chain compromise of the web application through service providers, libraries and frameworks such as cloud hosting or storage. Improper management of security patches may also lead to loss of support or compatibility issues between components.

Static scanning can be utilised to detect referenced versions of software and services to ensure they are updated with sufficient security.

2.2.6.1 Scope

- Outdated JavaScript libraries
- Known vulnerable dependencies
- Unmaintained packages
- Missing security patches

2.2.6.2 Patterns for Detection

- Version extraction from script tags and imports
- Known vulnerable version databases
- CDN links without SRI (Subresource Integrity)
- Development dependency patterns

2.2.6.3 Test Case: HTML

```
// 1. Outdated jQuery versions
<script src="jquery-1.4.2.min.js"></script>
// 2. Vulnerable libraries
<script src="angular-1.2.0.min.js"></script>
import 'lodash@3.10.1';
// 3. Development dependencies in production
<script src="node_modules/dev-dependency/debug.js"></script>
// 4. CDN without integrity checks
<script src="https://cdn.example.com/lib.js"></script>
```

2.2.7 A07 - Identification and Authentication Failures

Identity and Authentication Failures (previously “Broken Authentication”) refers to failures in a web application’s ability to confirm the valid identity of users, authentication, and session management. Identifying such issues early helps prevent unauthorised access, ensuring that authentication systems are properly implemented and resistant to common attack vectors like brute force or session hijacking.

In static vulnerability scanning, this vulnerability involves the analysis of source code to identify insecure practices such as hardcoded passwords, missing multi-factor authentication, poor session management, or improper credential storage.

2.2.7.1 Scope

- Weak password policies
- Session management flaws
- Missing authentication mechanisms
- Insecure password storage/handling

2.2.7.2 Patterns for Detection

- Password validation functions
- Session storage usage
- Authentication bypass patterns
- Weak token generation

2.2.7.3 Test Cases: JavaScript

```
// 1. Weak password validation
function isValidPassword(password) {
return password.length >= 4; // Too weak
}

// 2. Insecure session handling
localStorage.setItem('sessionToken'
, token); // Should use httpOnly cookies
sessionStorage.setItem('userAuth'
, authData);

// 3. Missing authentication checks
function sensitiveOperation() {

// No authentication check
return performOperation();
}

// 4. Password in client-side code
const passwordRegex = /^(?=.*[a-z]).{8,}$/; // Revealing password policy
```

```
// 5. Insecure token handling
const token = btoa(username + ":" + password); // Weak token generation
```

2.2.7.4 Test Cases: HTML

```
<!-- 1. Password autocomplete enabled -->
<input type="password" autocomplete="on">

<!-- 2. Weak password fields -->
<input type="password" maxlength="8">

<!-- 3. Session data in forms -->
<input type="hidden" name="sessionId" value="abc123">
```

2.2.8 A08 - Software and Data Integrity Failures

This vulnerability comprises the failure of an application in confirming the integrity and authenticity of files prior to their utilisation. This may allow a potential malicious actor to yield changes on critical elevated files which upon execution by Web Applications, may lead to injections and unauthorised code execution.

Static scanning may be utilised to identify any use of external scripts and media without verifying their integrity through safe hashing algorithms.

2.2.8.1 Scope

- Outdated JavaScript libraries
- Known vulnerable dependencies
- Unmaintained packages
- Missing security patches

2.2.8.2 Patterns for Detection

- Script tags without integrity attributes
- JSON.parse with untrusted input
- Dynamic script/resource loading
- eval with serialized data
- Specific regex pattern to look out for:
  - Absence of integrity checks:
    ```
    <script[^>]*src=["']https?:\/\/[^"']+["'][^>]*>(?!.*integrity=)
    <link[^>]*rel=["']stylesheet["'][^>]*href=["']https?:\/\/[^"']+
    ["'][^>]*>(?!.*integrity=)
    ```

2.2.8.3 Test Cases: JavaScript

```
// 1. Missing integrity checks
<script src="https://cdn.example.com/lib.js"></script> // No SRI
```

```
// 2. Unsafe deserialization
const userData = JSON.parse(untrustedInput);
eval(`(${serializedObject})`);
// 3. Untrusted plugin loading
const plugin = require(userProvidedPluginName);
// 4. Dynamic script loading
const script = document.createElement('script');
script.src = userProvidedURL;
document.head.appendChild(script);
```

2.2.9 A09 - Security Logging and Monitoring Failures

This vulnerability comprises the utilisation of insufficient logging, detection, and alerting of suspicious activities in web applications, allowing attacks to remain undetected for a prolonged period, hindering incident response.

In static scanning, this means code lacks proper logging mechanisms or fails to record security-relevant events. Scanners flag missing or misconfigured log statements, helping developers enhance visibility, monitor threats, and meet compliance or forensic investigation requirements effectively.

2.2.9.1 Scope

- Missing security event logging
- Insufficient monitoring of critical functions
- Information disclosure in logs
- Missing alerting mechanisms

2.2.9.2 Patterns for Detection

- Critical functions without logging
- Console logs with sensitive data
- Missing audit trails for sensitive operations
- No monitoring implementation

2.2.9.3 Test Case: JavaScript

```
// 1. Missing security logging
function login(username, password) {
if (authenticate(username, password)) {

// No logging of successful login
return true;
}

// No logging of failed attempt
return false;
}

// 2. Information disclosure in logs
console.log("Login failed for user:", username,"password:"
, password);
console.error("Database error:", fullError);

// 3. Missing monitoring for sensitive operations
function transferMoney(from, to, amount) {

// No logging of financial transaction
return process transfer(from, to, amount);
}

// 4. No rate limiting logs
function apiCall() {
// No tracking of API usage
}
```

2.2.10 A10 - Server-Side Request Forgery (SSRF)

This vulnerability occurs when a web application backend does not appropriately sanitise and validate inputs, allowing malicious actors to forge and transmit requests form such backends to destinations located within the backend's local private network. This vulnerability may present itself as a source of initial access for attackers, and may assist in various parts of the Cybersecurity Kill Chain.

In static patterns involving unsanitised inputs being passed to HTTP request functions may be detected and flagged for remediation. Thereby, it will assist developers in mitigating risks comprising internal network exposure, data leaks, and unauthorised access through proper input validation and request handling

2.2.10.1 Scope

- User-controlled URLs in requests
- Missing URL validation
- Internal resource access
- Cloud metadata access attempts

2.2.10.2 Patterns for Detection

- fetch , axios , XMLHttpRequest with user input
- URL parameters passed to HTTP functions
- Image/resource loading with user URLs
- Webhook or callback URL handling

2.2.10.3 Test Case: JavaScript

```
// 1. User-controlled URLs
function fetchUserData(url) {
return fetch(url); // No validation
}

// 2. Proxy functionality
function proxyRequest(targetUrl) {
return axios.get(targetUrl);
}

// 3. Image/resource loading
function loadImage(imageUrl) {
const img = new Image();
img.src = imageUrl; // Could be internal URL
}

// 4. Webhook functionality
function callWebhook(webhookUrl, data) {
return fetch(webhookUrl, {
method: 'POST'
,
body: JSON.stringify(data)
});
}
```

2. Hardening Recommendations

The following section outlines general hardening recommendations to provide users with post-scan advice on addressing detected vulnerabilities. Each hardening recommendation aligns with official OWASP advice to provide actionable and relevant feedback.

3.1 A01 - Broken Access Controls

To address **Broken Access Control**, developers must implement server-side authorisation as client-side checks are easily bypassed. Prioritise the following recommendations:

- **Implement Server-Side Access Control:** All access control decisions must be enforced from the server. The client should only display what is authorised by the server. Apply Zero-trust methodology and implement **deny-by-default** logic.
- **Centralise Access Control:** Consolidate authorisation processes and logic into a centralised component. This improves visibility and ease to manage and audit permissions across the web application.
- **Replace Direct Object References:** Instead, prioritise use of **indirect object references** instead of hardcoded, sequential or predictable IDs in URLs and

parameters. These IDs can be mapped to the actual database IDs on the server-side, and validate user permissions to access the requested resource.

- **Secure API and Endpoint Access:** Every API endpoint should check a user's role and permissions before processing a request. Don't trust any data sent from the client regarding user roles or privileges.
- **Remove Hardcoded Credentials:** Never hardcode credentials, API keys, or tokens in client-side code (JavaScript, HTML). Store them securely in environment variables or a secrets management system on the server.

3.2 A02 - Cryptographic Failures

To address **Cryptographic Failures**, developers should endeavor to implement strong encryption algorithms up to modern standards, ensuring data is encrypted during transit and at rest.

- **Secure in Transit - Strong Cryptographic Protocols:** Always use HTTPS with strong TLS configuration to encrypt data in transit with modern, secure ciphers.
- **Use Standardised and Secure Modern Cryptographic Algorithms:** Ensure use of algorithms such as AES-256 for symmetric encryption and SHA-256 or SHA-512 for hashing. Never use outdated or weak algorithms like MD5, SHA1, or DES.
- **Securely Store Sensitive Data**: Never store sensitive data like passwords, credit card numbers, or PII in local or session Storage. Use secure cookies for session tokens. Encrypt sensitive data at rest using stronger encryption algorithms.
- **Implement Proper Key Management**: Never hardcode encryption keys, salts, or secrets in code. Invest in secure key management or restricted access. Regularly rotate keys and use unique keys often.
- **Use Cryptographically Secure Random Generators**: Replace `Math.random()` with cryptographically secure alternatives for generating tokens, session IDs, or any security related random values.
- **Implement Proper Password Hashing**: Never use general-purpose hash functions such as MD5, SHA-256 for passwords. Instead use purpose-built password hashing algorithms such as bcrypt, scrypt, or Argon2. Always use unique salts for each password.

3.3 A03 - Injection

To address Injection vulnerabilities, developers must implement proper input validation, output encoding, and use parameterized queries and safe APIs that prevent injection attacks.

- **Use Parameterised Queries/Prepared Statements**: Always use parameterised queries or prepared statements when connecting databases. Avoid constructing SQL queries using string concatenation with any user input.
- **Implement Output Encoding**: Sanitise and encode all inputs before processing.
- **Avoid Dangerous Functions**: Never use `eval()`, `Function()` constructor, or `setTimeout()`/`setInterval()` with string arguments containing user input.

- **Use Safe DOM Manipulation Methods**: Use safe methods to replace `innerHTML`, `outerHTML`, or `insertAdjacentHTML()`. When HTML rendering is necessary, consider sanitisation libraries such as DOMPurify.
- **Implement Input Validation**: Validate all user input on both client and server-side. Configure whitelists of acceptable input patterns and reject invalid inputs.

3.4 A04 - Insecure Design

To address Insecure Design, developers must integrate security into the design phase of application development, implementing threat modeling and secure design patterns from the outset.

- **Implement Rate Limiting and Throttling**: Add rate limiting to critical or sensitive operations such as login, password reset, payment processing, and API calls. Consider exponential backoff for repeated failed attempts or implementing CAPTCHA for suspicious activity.
- **Apply Defense in Depth**: Establish multiple layers of security controls such as input validation, authentication, authorisation, and encryption.
- **Implement Comprehensive Input Validation**: Design validation rules that enforce business logic constraints such as minimum values, ranges, formats, decimals, limits.
- **Implement Secure Error Handling**: Use generic error messages when displaying to users whilst logging detailed errors securely on the server side.
- **Apply Principle of Least Privilege**: Enforce only the minimum permissions necessary to perform their functions. Regularly conduct reviews and audit permissions.

3.5 A05 - Security Misconfiguration

To address Security Misconfiguration, developers must establish secure defaults, disable unnecessary features, and maintain consistent hardening across all environments.

- **Disable Debug Mode in Production**: Remove or disable all debug flags, text output logging, and development features before public deployment.
- **Remove Development Artifacts**: Clean code of unnecessary code and comments, debugging, temp or unused libraries and dependencies, developer tools and scripts etc.
- **Implement Security Headers**: Configure and ensure use of comprehensive security headers for securing communications.
- **Use Secure Defaults**: Change all the pre–set default credentials, passwords, and API keys before deployment. Disable any default accounts.
- **Minimise Attack Surface**: Disable all unnecessary features, services, ports, and endpoints. Remove any unused dependencies or libraries.
- **Secure Configuration Management**: Store configs securely using secret management systems. Never publish sensitive configuration to version control. Use different configurations and credentials for development, staging, and production.
- **Regular Security Updates**: Establish processes for regularly updating and patching application components and dependencies.

- **Configuration Auditing**: Regularly audit configs against benchmarks and baselines. Invest in automated configuration scanning tools to detect vulnerabilities or gaps.

3.6 A06 - Vulnerable and Outdated Components

To mitigate the risk associated with A06, it is imperative for organisations to implement a strong version management and dependency hygiene policy, that proritises a robust third-party component management. These components comprise a range of libraries, frameworks, cloud SDKs, and plugins, which must be patched and updated in to ensure compliance with known industry standards and government regulations, and will inevitably curb the prospect of an attacker yielding A06 from your code. Additionally, you may also account for the following to further protect your code against exploits:

- **Inventory of Components:** Maintain an up-to-date inventory of all third-party components used in the application including their version numbers, origin/source, and license type. This includes both frontend and backend packages, where tools such as Software Composition Analysis (SCA) may assist in automating the detection process. Alternatively, Azure offers its Application Insights service, which indeed comprises a similar element.
- **Use Trusted Sources and Package Registries:** Always install packages and updates from official or trusted repositories (e.g., npm, PyPI, Maven Central). Avoid using packages from unknown developers or copied from unofficial mirrors or GitHub repos without a vetting process.
- **Update Dependencies Frequently:** Set up an automated process to check for updates to all packages (including transitive dependencies) at least weekly. Use tools like Dependabot, Renovate, or Snyk to get automatic pull requests for security fixes and version bumps.

3.7 A07 - Identity and Authentication Failures

To safeguard your web application against Identity and Authentication Failures, migrate key password validation mechanisms to server-side validation, and implement password policies in alignment with established industry standards such as the NIST Framework. Additionally consider:

- **Enforce Strong Password Policies:** Require passwords to meet complexity requirements extracted from known frameworks such as NIS, which mandates a minimum requirement of 12 characters, a mix of upper/lowercase letters, numbers, symbols, etc. Enforce account lockout after repeated failed login attempts and provide password strength feedback at creation. Integrate checks against known breached password lists using services like HaveIBeenPwned.
- **Secure Session Management:** Ensure session IDs comply with pseudorandom principles to remain unpredictable, securely generated (e.g., using cryptographically secure random number generators), and transmitted only over encrypted HTTPS traffic. In addition, Invalidate sessions on logout, idle timeout, and password change, and only store session tokens securely using technologies such as HTTP-only,

Secure, and SameSite cookies and regenerate tokens after privilege escalation or login.
- **Avoid Hardcoded Credentials and Secrets:** Do not store passwords, API keys, or secrets in source code, configuration files, or front-end assets, and any other forms of plaintext storage. Instead, use environment variables or a dedicated secrets management system (e.g., HashiCorp Vault, AWS Secrets Manager) and rotate them regularly.
- **Implement Multi-Factor Authentication (MFA):** Support MFA for all accounts, especially for administrative and high-privilege users. Prefer TOTP-based apps (e.g., Google Authenticator) or hardware security keys (e.g., YubiKey) over SMS-based methods, which are susceptible to interception.
- **Implement Single Sign On (SSO):** Use SSO through modern authentication protocols such as OAuth2, OpenID Connect, SAML, etc. to manage authentication where possible. This reduces the attack surface and simplifies identity management across distributed applications.

3.8 A08 - Software and Data Integrity Failures

To safeguard your web application against the use of unverified or malicious files, enforce strong controls around the loading, usage, and verification of external resources such as JavaScript libraries, stylesheets, and other remote assets. These files, if tampered with or sourced from untrusted locations, may lead to injection attacks or unauthorised code execution in the client's browser. Additionally consider:
- **Restrict External Resource Usage:** Restrict the use of third-party scripts, stylesheets, and other resources to pre-approved, trusted domains through well-defined allowlists. Consider the authenticity of CDNs and any external sources prior to their inclusion in your application. Wherever possible, internally host critical resources to reduce dependency on external infrastructure and mitigate the risk of content injection or supply chain compromise. Monitor all externally sourced content for changes and enforce automated controls in CI/CD pipelines to detect unauthorized updates.
- **Maintain Dependency Hygiene:** Keep all third-party libraries and packages up to date, and monitor them continuously for known vulnerabilities using software composition analysis tools. Integrate dependency checks into your build process using tools such as OWASP Dependency-Check, npm audit, or Snyk to ensure visibility into package risk.
- **Enforce Subresource Integrity (SRI):** Require all externally loaded scripts and stylesheets to include cryptographic integrity attributes that validate the authenticity of the resource before it is executed by the browser. Subresource Integrity ensures that even if a third-party server is compromised, the browser will reject any modified or malicious file whose hash does not match the expected value. Only use trusted cryptographic algorithms (e.g., SHA-384), and ensure integrity attributes are updated as part of the release process when resources are changed.

3.9 A09 - Security Logging and Monitoring Failures

Remediation for A09 predominantly places an emphasis on the logging of critical security events, which must also be monitored through tools such as Security Information and Event

Management (SIEM) to detect and respond to threats in a timely manner. However, the data to be logged must also account for the following considerations to curb any risk associated with the logging system itself:

- **Log Security-Relevant Events Consistently:** Always log important events such as failed logins, access to sensitive endpoints, permission changes, and unexpected errors, in a alignment with the who did what, when, and from where framework.
- **Avoid Logging Sensitive Information:** Never log sensitive data such as passwords, API keys, tokens, or personal information. Always sanitise logs to prevent accidental data leaks or compliance violations.
- **Implement Real-Time Alerting for Suspicious Activities:** Set up automated alerts for anomalies like repeated login failures, unusual access patterns, or privilege escalations, and ensure that they are routed to Security Analysts.
- **Centralise and Secure Log Storage:** Use a centralised, tamper-resistant logging system to collect and store logs across services in alignment with log immutability standards. Additionally adjust log retention to allow of effective post-incident forensics, and also ensure compliance with industry standards and government regulations

3.10 A10 - Server-Side Request Forgery (SSRF)

To defend your web application against SSRF, the primary control that must be implemented comprises the restriction of how and where user input can influence backend network requests. Additionally, the following strategies may further assist in this:

- **Strictly Validate and Whitelist URLs:** Never allow raw user input to be passed directly into request functions, where in lieu, you maintain an explicit allowlist of domains or IPs your app is allowed to contact backend servers. Imperatively, ensure that URLs are parsed properly and checked for disallowed patterns, including local IPs (127.0.0.1, 169.254.x.x, etc.), localhost, and encoded tricks like 0x7f000001.
- **Block Access to Internal and Metadata Addresses:** Ensure that requests cannot reach internal networks or cloud metadata services like AWS IMDS. Implement DNS resolution and IP filtering to block private or link-local ranges (10.0.0.0/8, 192.168.x.x, 169.254.x.x). Use cloud features like IMDSv2 and disable metadata access where not required.
- **Avoid Following Redirects Blindly:** Do not follow redirects from user-supplied URLs unless each redirect target is revalidated against your allowlist, to eliminate the primary threat vector of SSRP which operates through redirect chains.
- **Log and Monitor Suspicious Outbound Requests:** Log all server-side HTTP requests that are triggered by user input, and utilise SIEMs to monitor for unusual destinations, internal IPs, or metadata service access attempts. Particularly, use anomaly detection or rate-limiting to reduce exploitation attempts.