SECURITY ASSESSMENT REPORT

KRION CONSULTING Pvt. Ltd – Vulnerability Assessment and Penetration Report

Krion Consulting Pvt. Ltd.

04/12/2024

Confidential and Proprietary Information

Krion6D

Index of Content

2.3 Network Security

2.3.1 Secure server conguration (e.g., disabling unused ports).

2.3.2 Firewall rules to restrict unauthorized access.

2.3.3 Intrusion Detection/Prevention Systems (IDS/IPS).

2.4 Secure Development Practices

2.4.1 Version control with access restrictions (e.g., Git).

2.4.2 Regular code reviews and automated security testing.

2.4.3 Use of a dependency scanner to check for vulnerable libraries

2.5 Database Security

2.5.1 Encrypt sensitive data at rest (e.g., passwords using bcrypt).

2.5.2 Use parameterized queries or prepared statements to avoid SQL injection.

2.5.3 Regular database backups and secure storage of backup files.

2.6 Penetration Testing

2.6.1 Test for unauthorized access scenarios.

2.6.2 Test for injection attacks and code execution.

2.6.3 Test for session management issues (e.g., session xation).

2.7 Compliance

2.7.1 Adherence to industry standards (e.g., ISO 27001).

2.7.2 GDPR, HIPAA, or other regional data protection regulations (if applicable).

2.8 Incident Response

2.8.1 Defined and tested incident response plan.

2.8.2 Logging and monitoring with alerts for suspicious activities.

2.9 Findings and Recommendations

2.9.1 Highlight vulnerabilities found during the assessment.

2.9.2 Provide actionable recommendations for mitigation.

3. Tools Used

1. Project Overview

- Project Name: Krion6D
- Domain: Construction
- Objective: Identify and mitigate security vulnerabilities in the Krion6D application/system.
- Scope: Include web application, API, database, network, or IoT devices related to the project.

2. Vulnerability Assessment and Penetration Testing Checklist

Vulnerability Assessment and Penetration Testing (VAPT) for the Krion6D construction project focuses on identifying, assessing, and mitigating potential security risks in the software and infrastructure of the project. The goal is to ensure the safety and integrity of critical data, systems, and services within the construction project environment, minimizing exposure to cyber threats.

The checklist for VAPT would typically include the following steps:

- **Information Gathering**: Identify all assets, systems, and network components related to the project, including databases, APIs, and infrastructure.
- **Vulnerability Scanning**: Use automated tools to scan for known vulnerabilities, outdated software, weak configurations, and other security issues.
- **Manual Testing**: Perform manual penetration tests to exploit vulnerabilities that automated tools may miss. This can involve simulating attacks such as SQL injection, cross-site scripting (XSS), and privilege escalation.
- **Configuration Review**: Ensure that all security configurations, such as firewalls, authentication mechanisms, and encryption protocols, are correctly implemented.
- **Risk Assessment**: Classify identified vulnerabilities based on their potential impact and likelihood, and prioritize them for remediation.
- **Reporting and Remediation**: Document findings, provide detailed recommendations for mitigating risks, and work with the development team to implement fixes.
- **Retesting**: After fixes have been applied, retest the system to ensure vulnerabilities have been properly addressed.

This process helps in securing the Krion6D construction project from cyber threats and ensures compliance with best practices for data protection and system security.

2.1 Application Security

2.1.1 Input validation and sanitization for all user inputs.

- **Validate input types**: Ensure that all inputs (e.g., form fields, query parameters) conform to the expected format (e.g., email, date, integer).
- **Sanitize inputs**: Remove or escape potentially harmful characters (e.g., <, >, ', "), especially in user-provided data.
- **Whitelist approach**: Where possible, use a whitelist of valid characters instead of a blacklist to prevent malicious input.
- **Limit input length**: Restrict the length of input to reasonable bounds to prevent buffer overflow attacks and mitigate excessive resource consumption.
- **Use frameworks and libraries**: Leverage trusted libraries and frameworks that have built-in input validation and sanitization functions.

2.1.2 Implementation of secure authentication (e.g., OAuth, JWT).

- **JWT (JSON Web Token)**: Use JWT for secure token-based authentication to avoid session hijacking and provide stateless authentication.
- **OAuth 2.0**: Implement OAuth for secure delegated access, ensuring users can log in with trusted third-party services like Google or Facebook without sharing passwords.
- **Secure password storage**: Use strong hashing algorithms like bcrypt or Argon2 for storing user passwords securely.
- **Two-factor authentication (2FA)**: Implement multi-factor authentication to add an additional layer of security.
- **Token expiration**: Ensure JWT tokens have short expiration times and are refreshed periodically to minimize the impact of stolen tokens.

2.1.3 Proper authorization and role-based access control (RBAC).

- **Define roles clearly**: Establish distinct roles (e.g., admin, user, manager) with specific access permissions to different parts of the application.
- **Enforce least privilege**: Ensure that users only have access to the resources necessary for their role and nothing more.
- **Secure sensitive routes**: Use middleware or guards to enforce role-based access control on sensitive API endpoints or pages.
- **Use groups and permissions**: Implement granular permissions within roles to allow for flexible authorization schemes.
- **Regular role reviews**: Periodically review roles and permissions to ensure they align with the current organizational structure and project requirements.

2.1.4 Protection against common vulnerabilities (OWASP Top 10):

A. SQL Injection

- **Use prepared statements**: Always use parameterized queries or prepared statements with your database queries to prevent malicious input from being executed.
- **ORM (Object-Relational Mapping)**: Use ORMs like Sequelize or TypeORM that automatically handle query sanitization to prevent SQL injection.
- **Input sanitization**: Ensure all inputs that are used in SQL queries are sanitized and validated before being processed.
- **Least privilege for DB users**: Ensure that database accounts have minimal privileges necessary for operations to prevent damage in case of an attack.

B. Cross-Site Scripting (XSS)

- **Escape output**: Ensure that any data rendered to the browser (e.g., user comments, inputs) is properly escaped to prevent malicious scripts from being executed.
- **Content Security Policy (CSP)**: Implement a strict CSP to prevent the loading of malicious scripts or resources from unauthorized domains.
- **Sanitize HTML input**: If allowing HTML input from users, sanitize it to remove any potentially dangerous scripts.
- **Avoid inline JavaScript**: Refrain from using inline JavaScript or event handlers (e.g., onclick, onmouseover) to reduce the risk of XSS attacks.

C. Cross-Site Request Forgery (CSRF)

- **Anti-CSRF tokens**: Implement anti-CSRF tokens in all forms and state-changing requests (e.g., POST, PUT, DELETE) to ensure that requests are coming from authenticated and authorized users.
- **SameSite cookies**: Use the SameSite attribute for cookies to prevent browsers from sending cookies with cross-origin requests.
- **Verify request origin**: Check the Origin and Referer headers to ensure that the request is coming from the trusted domain.
- **Use secure HTTP methods**: Avoid using GET for state-changing operations, as it can be vulnerable to CSRF attacks.

D. Insecure Direct Object References (IDOR)

- **Validate user access to objects**: Ensure that users can only access objects (e.g., records, files) that they are authorized to access, and not others.
- **Use indirect references**: Instead of exposing direct object identifiers (e.g., database IDs) in URLs, use indirect references or obfuscate the object ID.
- **Access control checks**: Implement proper authorization checks for every request that accesses or modifies an object, based on the user's permissions.
- **Log and monitor access attempts**: Keep track of access attempts to sensitive objects, especially those that are unauthorized, and alert administrators of suspicious behavior.

2.2 API Security

2.2.1 Use HTTPS for secure API communication.

- **Encrypt data in transit**: Use HTTPS to ensure that all data exchanged between the client and server is encrypted, preventing eavesdropping and man-in-the-middle (MITM) attacks.
- **Enforce HTTPS**: Redirect all HTTP traffic to HTTPS to ensure secure communication.
- **Use strong TLS configurations**: Implement secure TLS versions (e.g., TLS 1.2 or 1.3) and disable weak protocols like SSL and outdated versions of TLS.
- **Regular certificate updates**: Use trusted Certificate Authorities (CA) for SSL certificates and keep them updated to avoid downtime or warnings.

2.2.2 API keys or tokens for authentication.

- **API keys for client authentication**: Issue unique API keys to identify and authenticate applications accessing the API.
- **Token-based authentication**: Use JWT or OAuth tokens for stateless and secure user authentication.
- **Secure storage of keys**: Avoid exposing API keys in client-side code. Store them securely in environment variables or secret management services.
- **Key rotation**: Periodically rotate API keys to reduce risks if a key is compromised.
- **Revocation mechanism**: Implement functionality to revoke compromised keys or tokens.

2.2.3 Rate limiting and throttling to prevent abuse.

- **Set request limits**: Define a maximum number of requests that can be made within a specific time frame (e.g., 100 requests per minute).

- **Prevent DDoS attacks**: Use rate limiting to block abusive traffic patterns, ensuring that resources are not overwhelmed.
- **Per-user limits**: Apply rate limits based on user or API key to prevent any single user from monopolizing API resources.
- **Retry-after headers**: Inform clients when they are being throttled and when they can retry their requests.
- **Monitoring and alerts**: Set up systems to monitor unusual activity and alert administrators when limits are consistently exceeded.

2.2.4 Validation of input payloads and schemas.

- **Strict schema validation**: Define and enforce input payload schemas using tools like Joi, Yup, or JSON Schema to ensure incoming data meets expected formats.
- **Reject invalid requests**: Reject requests with missing or extra fields, incorrect data types, or unexpected payload structures.
- **Sanitize inputs**: Strip out or neutralize potentially harmful inputs to prevent injection attacks.
- **Use frameworks**: Leverage API frameworks like Express.js with middleware to validate and sanitize input payloads.
- **Error handling**: Provide detailed error messages for invalid payloads without exposing sensitive implementation details.

2.3 Network Security

2.3.1 Secure server conguration (e.g., disabling unused ports).

- **Close unused ports**: Scan the network to identify open ports and disable any that are not required for the project's functionality (e.g., using tools like Nmap).
- **Minimize running services**: Only run essential services on servers to reduce potential attack vectors.
- **Secure default settings**: Change default usernames, passwords, and configurations for server software to avoid exploitation.
- **Regular patches and updates**: Keep server operating systems and software up to date to protect against known vulnerabilities.
- **Disable unnecessary protocols**: Turn off unused network protocols (e.g., FTP, Telnet) and replace them with secure alternatives like SFTP or SSH.

2.3.2 Firewall rules to restrict unauthorized access.

- **Implement strict access control**: Define firewall rules to allow only trusted IP ranges to access sensitive systems.
- **Block all by default**: Use a default "deny all" policy and explicitly allow only necessary traffic to specific ports and services.
- **Segregate networks**: Isolate sensitive resources (e.g., databases, APIs) in separate network zones and use firewalls to regulate access between zones.
- **Monitor traffic patterns**: Continuously monitor incoming and outgoing traffic to detect anomalies and unauthorized access attempts.
- **Web Application Firewall (WAF)**: Deploy a WAF to filter and monitor HTTP traffic to protect against application-layer attacks.

2.3.3 Intrusion Detection/Prevention Systems (IDS/IPS).

- **Deploy IDS/IPS**: Use Intrusion Detection Systems (IDS) to monitor network traffic for suspicious activities and Intrusion Prevention Systems (IPS) to actively block potential threats.
- **Signature-based detection**: Use known attack signatures to identify common threats and ensure the signature database is regularly updated.
- **Anomaly-based detection**: Configure IDS/IPS to detect unusual behavior that may indicate novel or emerging attacks.
- **Integration with SIEM**: Combine IDS/IPS with a Security Information and Event Management (SIEM) system to centralize alerts and analysis.
- **Set up alerts**: Configure IDS/IPS to send real-time alerts to administrators for immediate response to potential threats.

2.4 Secure Development Practices

2.4.1 Version control with access restrictions (e.g., Git).

- **Access controls**: Restrict access to version control repositories using role-based permissions (e.g., read, write, or admin access based on the user's role).
- **Two-factor authentication (2FA)**: Enable 2FA for all developers accessing the version control system to add an additional layer of security.
- **Branch protection**: Use protected branches (e.g., main or master branch) to prevent unauthorized direct commits and enforce code reviews before merging.

- **Audit logging**: Enable logging of all access and changes to the repository to track unauthorized or suspicious activity.
- **Encryption**: Use HTTPS or SSH for secure communication with the version control server to prevent data interception.

2.4.2 Regular code reviews and automated security testing.

- **Code reviews**: Conduct regular peer reviews to identify potential security issues, enforce coding standards, and ensure the quality of the code.
- **Static Application Security Testing (SAST)**: Use tools like SonarQube, Checkmarx, or CodeQL to automatically analyze code for vulnerabilities during development.
- **Dynamic Application Security Testing (DAST)**: Simulate attacks on running applications using tools like OWASP ZAP or Burp Suite to uncover runtime vulnerabilities.
- **Secure coding standards**: Establish guidelines for secure coding practices and ensure they are followed during code reviews.
- **Continuous Integration/Continuous Deployment (CI/CD)**: Integrate security checks into the CI/CD pipeline to automatically test code for vulnerabilities before deployment.

2.4.3 Use of a dependency scanner to check for vulnerable libraries

- **Automated dependency scans**: Use tools like Snyk, Dependabot, or OWASP Dependency-Check to identify outdated or vulnerable libraries in your project.
- **Maintain updated dependencies**: Regularly update libraries and packages to their latest secure versions, ensuring compatibility with the application.
- **Monitor known vulnerabilities**: Subscribe to vulnerability databases like CVE (Common Vulnerabilities and Exposures) or NVD (National Vulnerability Database) to stay informed about newly discovered issues in third-party dependencies.
- **Minimal dependencies**: Only include libraries and packages that are absolutely necessary to reduce the attack surface.
- **Verify source integrity**: Use checksums or digital signatures to verify the authenticity of downloaded dependencies and prevent supply chain attacks.

2.5 Database Security

2.5.1 Encrypt sensitive data at rest (e.g., passwords using bcrypt).

- **Password hashing**: Use strong hashing algorithms like **bcrypt**, **Argon2**, or **PBKDF2** for storing passwords securely. Avoid using MD5 or SHA1 as they are no longer secure.
- **Encryption for sensitive fields**: Encrypt critical data such as personally identifiable information (PII) and financial records using algorithms like AES-256.
- **Database-level encryption**: Enable Transparent Data Encryption (TDE) or similar features to encrypt the entire database at rest.
- **Key management**: Store encryption keys securely using a hardware security module (HSM) or a cloud-based key management service.

2.5.2 Use parameterized queries or prepared statements to avoid SQL injection.

- **Parameterized queries**: Always use parameterized queries to prevent malicious input from being interpreted as SQL commands. For example:
- **Prepared statements**: Use prepared statements in your database queries. In frameworks like Sequelize or TypeORM, these are built-in by default.
- **Avoid dynamic SQL**: Do not concatenate user inputs into SQL queries. Use frameworks or ORMs to handle queries safely.
- **Input validation**: Validate and sanitize all user inputs before processing them in database queries, even when using prepared statements.

2.5.3 Regular database backups and secure storage of backup files.

- **Automated backups**: Schedule regular database backups to ensure data can be restored in case of accidental loss or a security breach.
- **Encrypt backups**: Always encrypt backup files to protect sensitive data in case the backup is accessed by unauthorized users.
- **Secure storage**: Store backups in a secure location, such as a dedicated backup server, cloud storage with strong access controls, or an encrypted external drive.
- **Access controls**: Restrict access to backups to only authorized personnel and monitor all access attempts.
- **Backup testing**: Periodically test backups by restoring them in a test environment to ensure data integrity and functionality.

2.6 Penetration Testing

2.6.1 Test for unauthorized access scenarios.

- **Role-based access control (RBAC) checks**: Verify that users can only access resources and perform actions permitted by their roles (e.g., admin, teacher, student).
- **Horizontal privilege escalation**: Attempt to access another user's data or functionality without authorization (e.g., using another user's ID in a request).
- **Vertical privilege escalation**: Test if a lower-privileged user can perform admin-level actions.
- **Endpoint security**: Ensure sensitive API endpoints are not accessible without proper authentication or from unauthorized IPs.
- **Directory traversal attacks**: Test file paths to ensure that attackers cannot access restricted directories or files.

2.6.2 Test for injection attacks and code execution.

- **SQL injection**: Test input fields, query parameters, and headers for SQL injection vulnerabilities using tools like **sqlmap** or manual payloads (e.g., OR 1=1).
- **Command injection**: Attempt to execute unauthorized commands on the server via vulnerable inputs.
- **Cross-Site Scripting (XSS)**: Test for XSS by injecting malicious scripts into inputs and observing their execution in the browser.
- **LDAP injection**: If the application interacts with LDAP, test for vulnerabilities by injecting malicious LDAP statements.
- **Server-Side Template Injection (SSTI)**: If using templating engines, test for code execution through input that is rendered in templates.

2.6.3 Test for session management issues (e.g., session fixation).

- **Session fixation**: Attempt to set a predefined session ID and verify if the application allows it. Ensure session IDs are regenerated upon login.
- **Session timeout**: Test for proper session expiration after a period of inactivity or when the user logs out.
- **Session ID exposure**: Check that session IDs are not transmitted in URLs or easily accessible in logs or browser storage.
- **Cross-Site Request Forgery (CSRF)**: Test if sensitive actions can be performed without user consent by crafting malicious requests.

- **Secure cookie attributes**: Verify that cookies are set with the HttpOnly, Secure, and SameSite flags to mitigate theft and misuse.

2.7 Compliance

2.7.1 Adherence to industry standards (e.g., ISO 27001).

- **Establish an Information Security Management System (ISMS)**: Implement an ISMS framework to manage sensitive information systematically and securely.
- **Risk assessment**: Conduct regular risk assessments to identify and mitigate potential threats to information security.
- **Access controls**: Implement strict access control policies to limit access to sensitive data and systems based on roles and responsibilities.
- **Incident management**: Develop and maintain an incident response plan to address security breaches promptly and effectively.
- **Certification**: Work towards ISO 27001 certification to demonstrate compliance with global information security best practices.

2.7.2 GDPR, HIPAA, or other regional data protection regulations (if applicable).

- **Data protection policies**: Develop and enforce policies to protect personal and sensitive data in compliance with relevant regulations.
- **GDPR compliance**: If operating in the EU, ensure compliance with the General Data Protection Regulation (GDPR) by:
    - Obtaining user consent before collecting or processing personal data.
    - Allowing users to access, rectify, or delete their data.
    - Appointing a Data Protection Officer (DPO) if required.
    - Implementing measures for data portability and breach notifications.
- **HIPAA compliance**: If handling healthcare data, follow the Health Insurance Portability and Accountability Act (HIPAA) by:
    - Encrypting electronic protected health information (ePHI) both in transit and at rest.
    - Implementing access controls to ensure that only authorized personnel can access ePHI.
    - Maintaining an audit trail of access and changes to ePHI.
    - Conducting regular HIPAA training for employees.

- **Local regulations**: Identify and adhere to regional laws and standards applicable to your project's operating areas (e.g., CCPA in California, PDPA in Singapore).

2.8 Incident Response

2.8.1 Defined and tested incident response plan.

- **Develop a clear plan**: Create a detailed Incident Response Plan (IRP) outlining the steps to identify, respond to, mitigate, and recover from security incidents.
- **Incident categories**: Define different types of incidents (e.g., data breach, DDoS attack, unauthorized access) and establish response protocols for each.
- **Incident response team (IRT)**: Assemble a dedicated team with clear roles and responsibilities for managing incidents.
- **Communication strategy**: Establish a communication plan to notify stakeholders, including internal teams, customers, and regulatory bodies, during and after an incident.
- **Test the plan**: Conduct regular tabletop exercises or simulations to test the effectiveness of the IRP and identify areas for improvement.
- **Post-incident review**: After an incident, perform a root cause analysis and update the IRP based on lessons learned.

2.8.2 Logging and monitoring with alerts for suspicious activities.

- **Comprehensive logging**: Enable logging of critical activities across applications, databases, servers, and network devices to maintain a detailed audit trail.
- **Centralized log management**: Use tools like ELK Stack, Splunk, or a cloud-based logging solution to centralize and manage logs for easier analysis.
- **Real-time monitoring**: Implement real-time monitoring solutions to detect suspicious activities, such as unauthorized logins, unusual data access patterns, or privilege escalations.
- **Automated alerts**: Set up automated alerts for predefined triggers, such as repeated failed login attempts, unusual traffic spikes, or access to restricted resources.
- **Threat detection tools**: Use Security Information and Event Management (SIEM) systems to analyze log data and detect potential threats.
- **Retention policy**: Define a log retention policy to store logs for a sufficient duration for forensic analysis while complying with data retention regulations.

2.9Findings and Recommendations

2.9.1 Highlight vulnerabilities found during the assessment.

- **Access control weaknesses**:
  - Unauthorized access to sensitive endpoints due to improper role-based access control (RBAC) implementation.
  - Lack of multi-factor authentication (MFA) for critical user accounts.
- **Injection vulnerabilities**:
  - Potential SQL injection points in input fields without parameterized queries.
  - Risk of Cross-Site Scripting (XSS) in user-submitted forms.
- **Session management flaws**:
  - Session IDs not being regenerated upon user login, increasing the risk of session fixation.
  - Lack of secure cookie flags (HttpOnly, Secure, SameSite) for session cookies.
- **Data encryption gaps**:
  - Sensitive data, such as passwords and personally identifiable information (PII), not encrypted at rest.
  - Inadequate encryption of data transmitted over insecure protocols.
- **Network security lapses**:
  - Open and unused ports detected during a network scan.
  - Weak firewall rules exposing unnecessary services to the internet.
- **Monitoring and logging issues**:
  - Absence of real-time alerts for critical events like unauthorized access attempts.
  - Inadequate log storage policies, risking loss of valuable forensic data.

2.9.2 Provide actionable recommendations for mitigation.

- **Access Control**:
  - Implement robust RBAC and ensure all endpoints enforce proper authorization checks.
  - Enable MFA for user accounts, especially for administrative roles.
- **Injection Prevention**:
  - Use parameterized queries and input validation to prevent SQL injection.
  - Sanitize all user inputs and encode output to mitigate XSS attacks.
- **Session Security**:
  - Regenerate session IDs on login and logout events.
  - Set secure cookie attributes (HttpOnly, Secure, SameSite) for all cookies.

- **Data Encryption**:
  - Hash passwords using strong algorithms like bcrypt before storing them in the database.
  - Use HTTPS to encrypt data in transit and enable encryption for sensitive data at rest using AES-256.
- **Network Security**:
  - Disable unused ports and services and configure a default "deny all" firewall policy.
  - Use Intrusion Detection/Prevention Systems (IDS/IPS) to monitor network traffic for malicious activities.
- **Monitoring and Logging**:
  - Implement a centralized logging system (e.g., ELK, Splunk) with real-time alerting for critical events.
  - Establish a log retention policy to ensure logs are available for auditing and forensic purposes.

3. Tools Used

- List the tools used for vulnerability scanning and penetration testing, such as:
  - OWASP ZAP
  - Burp Suite
  - Nessus
  - Metasploit