IMIT 6.14 Application and Web Security Specifications

Document Version: 1.0
Last Reviewed: August 2024

1. Purpose

To define the deliverables, and their associated roles and responsibilities, needed to establish the minimum technical security requirements for secure government applications, including web and mobile applications.

This document provides detailed security specifications to support the IMIT 6.14 Application and Web Security Standard. Both the standard requirements and these specifications MUST be followed.

2. Resources

BC Government API Guidelines	Province of British Columbia API guidelines.
Common Vulnerability Scoring System	Standard used to score the severity of a software vulnerability.
Defensible Security Framework	Critical security controls (assessment and tools).
IMIT 6.11 Security Threat and Risk Assessment Standard	Requirements to assess (identify, analyze, and evaluate), define planned treatments, and report security threats and risks in information systems.
IMIT 6.14 Application and Web Security Standard	Corresponding standard for these specifications.
IMIT 6.23 Asset Management Security Standard	Baseline security controls for managing physical IT assets and information assets to protect B.C. government information and information systems.
IMIT 6.27 Operations Security Standard	Security framework for secure IT operations management.
IMIT 6.29 System Acquisition, Development, and Maintenance Security Standard	Security guidance to preserve the integrity and accuracy of information systems and information systems' lifecycle management.
Information Security Glossary	List of information security terms and definitions.
Mobile App Development	Province of British Columbia guidelines for in-house app development for mobile devices.

NIST 800-218	Secure Software Development Framework (SSDF) Version 1.1. Recommendations for mitigating risk of software vulnerabilities. Describes fundamental, sound practices for the SSDF.
NIST 800-64	NIST Special Publication 800-64 Revision 2, Security Considerations in the System Development Life Cycle (SDLC). Focuses on the information security components of the SDLC.
OCIO Patch Guidelines	Current patching expectations for government assets and ISB expected patch mitigation plan for vulnerable systems based on risk rating.
OWASP API Security Project Top 10 vulnerabilities	Guidance on top 10 API vulnerabilities and mitigation measures for those vulnerabilities.
OWASP Cheat Sheet for Database Security	Guidance on securely configuring and using the SQL and NoSQL databases intended for application developers responsible for managing the databases.
OWASP Code Review Guide	Guidance on best practices in secure code review, and how it can be used within a secure software development lifecycle (S-SDLC).
OWASP Logging Cheat Sheet	Guidance on building application logging mechanisms, especially related to security logging.
Payment Card Industry Data Security Standard (PCI DSS) Version 4.0	Global standard that provides a baseline of technical and operational requirements designated to protect payment data.

3. Specifications

3.1 Secure software development

The OCIO (for enterprise systems) and ministries (for ministry systems) MUST ensure:

1. The custom-built application is protected against known attacks based on the attack risk ratings identified during a security threat and risk assessment. See IMIT 6.11 Security Threat and Risk Assessment Standard for details on how to conduct a security threat and risk assessment and complete a Statement of Acceptable Risks (SoAR).[1]
2. Per the IMIT 6.29 System Acquisition, Development, and Maintenance Security Standard, security controls for the custom-built application are documented in the system security plan for the custom-built application.
3. A software development life cycle (SDLC) process is followed regardless of methodology adopted (for example, DevOps, Agile, Waterfall) for software development. See Appendix A: Secure software development life cycle (SDLC) for details.
4. Digital signatures are obtained from CITZ Digital Office DevOps and Cloud Services – Platform Services for applications developed for mobile devices. See Mobile App Development for details.

[1] A SoAR is part of the system security plan for the application. See the IMIT 6.29 System Acquisition, Development, and Maintenance Security Standard for details.

3.1.1 Secure coding

1. An application is developed using one of the following secure coding practices:
   - B.C. Government security best practices for apps https://docs.developer.gov.bc.ca/security-best-practices-for-apps/#secure-coding-guideline
   - CERT – The Software Engineering Institute at Carnegie Mellon University http://www.cert.org/
   - NIST – The National Institute of Standards and Technology http://csrc.nist.gov/
   - OWASP – The Open Web Application Security Project http://www.owasp.org/
   - SANS Institute – Escal Institute of Advanced Technologies http://www.sans.org/

   Also, the guidance appropriate to the type of code being developed is followed:

   - For Kubernetes-based custom code (for example, OpenShift), follow the security context constraints (SCC) https://docs.openshift.com/container-platform/4.13/security/seccomp-profiles.html
   - For WebAssembly (Wasm)-based custom code https://webassembly.org/docs/security/
   - For web-based custom code, follow the OWASP guidance https://owasp.org/www-project-web-security-testing-guide/stable/
   - For WebSocket-based custom code, follow the OWASP guidance https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/10-Testing_WebSockets
   - For progressive web application (PWA)-based custom code, follow the security guidance in Appendix B: PWA security guidance.
   - For application programming interface (API)-based custom code, follow the security guidance from BC Government API Guidelines.

2. Insecure application programming interfaces (APIs) are not used in applications, particularly for public-facing web applications. See Appendix C: 10 API security guidelines and best practices.
3. Custom code accessing databases is configured to:
    a. Connect to the database using TLSv1.2 or higher with modern ciphers like AES-GCM or ChaCha20.
    b. Verify that the digital certificate is correct.
4. Server-side application files that are accessible for downloading or inspection by clients are authenticated.
5. Accounts, user IDs, and passwords are NOT embedded in the source code.

3.1.2 Secure code review requirements

1. Before release, the results of the security threat and risk assessment are used to identify the appropriate mix of assessments and frequency of assessments on a case-by-case basis for the custom code:
    a. At minimum, the custom code is reviewed, tested, and remediated for common coding vulnerabilities listed in Appendix D: Common coding vulnerabilities.
    b. If web applications and application interfaces are used, the custom-code is tested and remediated for interface vulnerabilities listed in Appendix E: Web application and application interface vulnerabilities.
2. Custom code is reviewed. Automated source code analysis is recommended, although manual analysis is acceptable. The combination of both, however, can provide the best results. When the review is:
    a. Automated, the author of the custom code is NOT the author of the code review scripts used to conduct the analysis.

b. Manual, the review is conducted by individuals[2] who know the application architecture—but are not the code author to provide maximum unbiased insight.

3. The code review is conducted at the following frequency:

Risk to application	**Information Sensitivity**	**Code Review Frequency**
Low	Public	• After a significant change in application or annually at minimum.
Low	Confidential, that is Protected A, B, or C	• After a significant change in application. • Every 6–9 months at minimum if the assessment is conducted manually. • More frequently, if the assessment is automated.
Medium,[3] to High or Critical	Public, or Confidential (Protected A, B, or C)	• After a significant change in application. • Every 6–9 months at minimum if the assessment is conducted manually. • More frequently, if the assessment is automated.

4. A public-facing web application is reviewed:
   a. At the frequency dictated by the sensitivity of the information that it will collect, process, or transmit, or its criticality to business operations, or both.
   b. After any significant change.
5. If automated scanning is performed on a public-facing application, the frequency for the automated scans is equivalent to frequency for manual scans, or better.

[2] These individuals need to have the necessary skills and secure coding knowledge to effectively evaluate the code.
[3] Applications and public-facing web applications implemented on shared infrastructure are considered to be at medium or higher risk.

6. Various assessments are considered when deploying a new or significantly changed application. See Appendix F: Assessment guidelines for information on the types of assessments to be conducted.

3.2 Secure software maintenance

The OCIO (for enterprise systems) and ministries (for ministry systems) MUST ensure:

3.2.1 Security patches

1. To document a security threat and risk assessment in a Statement of Acceptable Risks (SoAR) when the application of a security patch for a critical or high-risk vulnerability is delayed for more than 5 weeks from the release of the patch. The SoAR must include:
   a. Reason for the delay
   b. The planned timeline to apply the security patch
   c. The mitigative risk controls that will be implemented
2. The criteria for risk ranking of security vulnerabilities is based on:
   a. Common Vulnerability Scoring System (CVSS)
   b. Vendor-supplied patch classification designation
   c. Assessment of business risk

3.2.2 Security vulnerabilities management

1. A security vulnerability in the application that is known to result in critical or high risks is identified, prioritized, and recorded in an appropriate tool[4] to enable the risks to be monitored and tracked.
   a. Security vulnerabilities of third-party software applications used to develop the custom-built application MUST also be identified and documented.
2. Patch management activities include (but are not limited to):
   a. Ensuring patches are from authorized sources

[4] The appropriate tool can be a risk register, or a security software tool adopted by the SDLC team to scan code for vulnerabilities that is appropriate to the SDLC methodology adopted (for example, DevOps, Agile, Waterfall).

b. Assessing the business impact of implementing (or not implementing) patches
c. Adequate testing of patches
d. Identifying appropriate timing and method of applying patches
e. Reporting on patch management activities
f. Making contingency plans for failures during patch management activities

3.3 Protection of the production environment

The OCIO (for enterprise systems) and ministries (for ministry systems) MUST ensure:

1. Per IMIT 6.27 Operations Security Standard, a security threat and risk assessment is conducted to identify the security controls required to segregate the production environment from non-production environments. The controls MUST include, but are not limited to:
   a. Separate access controls for the production and non-production environments
   b. Removal or disabling of accounts for an information system that are dormant or inactive for more than 45 days so they cannot be used to login to the system
   c. Storing database connection configurations (also known as connection strings), and database credentials separately and in encrypted format
2. Separation of duties per the IMIT 6.27 Operations Security Standard is achieved by ensuring no individual is responsible for all the tasks in the different phases of a SDLC. This is done by:
   a. Fully automating[5] the SDLC processes.
      OR
   b. Ensuring the development team has sufficient human resources to ensure the review, testing, and approval of code is independent of the code author.

[5] The scripts, rules, code, and configurations used to automate the SDLC processes MUST also be reviewed independently of the author of the script, rule, code, or configuration to ensure their integrity.

3. New software, software modifications, and security patches are approved for deployment and documented in the system security plan.
4. Production data that is classified as Protected B or Protected C is:
   a. Approved via an appropriate approval process if use of production data is required in a non-production environment.
   b. Removed from the non-production environment once testing is complete (per the IMIT 6.29 System Acquisition, Development, and Maintenance Security Standard).

3.3.1 Attack prevention

1. Verification tests are conducted for automated scanning annually to ensure the automated scans work as intended.
2. A web-application firewall is configured and implemented to protect and isolate the back-end networks and production systems from known attacks on public-facing web applications.
3. All services not essential for the function of the application are disabled to help reduce the need to apply vendor-supplied security patches.

3.3.2 Attack detection

1. Per the IMIT 6.27 Operations Security Standard, applications log the following events[6] where possible:
   a. Input validation failures, like protocol violations, unacceptable encodings, invalid parameter names and values
   b. Output validation failures, like database record set mismatch, invalid data encoding
   c. Authentication successes and failures
   d. Authorization (access control) failures
   e. Session management failures, like cookie session identification value modification

[6] Based on the OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

f. Application errors
g. System events like syntax and runtime errors
h. Connectivity problems
i. Performance issues
j. Third party service error messages
k. File system errors
l. File upload virus detection
m. Configuration changes
n. Application and related systems startups and shutdowns
o. Logging initialization (starting, stopping, or pausing)
p. Use of higher-risk functionality like:
    i. Network connections
    ii. Addition or deletion of users
    iii. Changes to privileges
    iv. Assigning users to tokens
    v. Adding or deleting tokens
    vi. Use of systems administrative privileges
    vii. Access by application administrators
    viii. All actions by users with administrative privileges
    ix. Access to payment cardholder data
    x. Use of data encryption keys or key changes
    xi. Creation and deletion of system-level objects
    xii. Data import and export including screen-based reports
    xiii. Submission of user-generated content—especially file uploads
q. Legal and other opt-ins like:
    i. Permissions for mobile phone capabilities
    ii. Terms of use
    iii. Terms and conditions
    iv. Personal data usage content
    v. Permission to receive marketing communications

2. The logs record, at minimum, the following event attributes:
    a. Log date and time (international format)
    b. Event date and time
    c. Interaction identifier[7]
    d. Application identifier like name and version
    e. Application address like:
        i. Cluster/hostname or server IP address and port number
        ii. Workstation identity
        iii. Local device identifier
    f. Service name like name and protocol
    g. Geolocation
    h. Window/form/page like entry point URL and HTTP method for a web application, dialogue box name
    i. Code location like script name, module name
    j. Source address like:
        i. User’s device/machine identifier
        ii. User’s IP address
        iii. Cell/RF tower ID
        iv. Mobile telephone number
    k. User identity (if authenticated or otherwise known) like user database table primary key value, username, licence number
    l. Event type and severity[8]
    m. Security relevant event flag (if the logs contain non-security event data too)
    n. Description

[7] The "Interaction identifier" is a method of linking all (relevant) events for a single user interaction (for example, desktop application form submission, web page request, mobile app button click, web service call).
[8] The OCIO and ministries should have a consistent, and documented approach to classification of events (type, confidence, severity), the syntax of descriptions, and field lengths and data types, including the format used for dates/times.

4. Revision history

These specifications are reviewed annually and updated as needed.

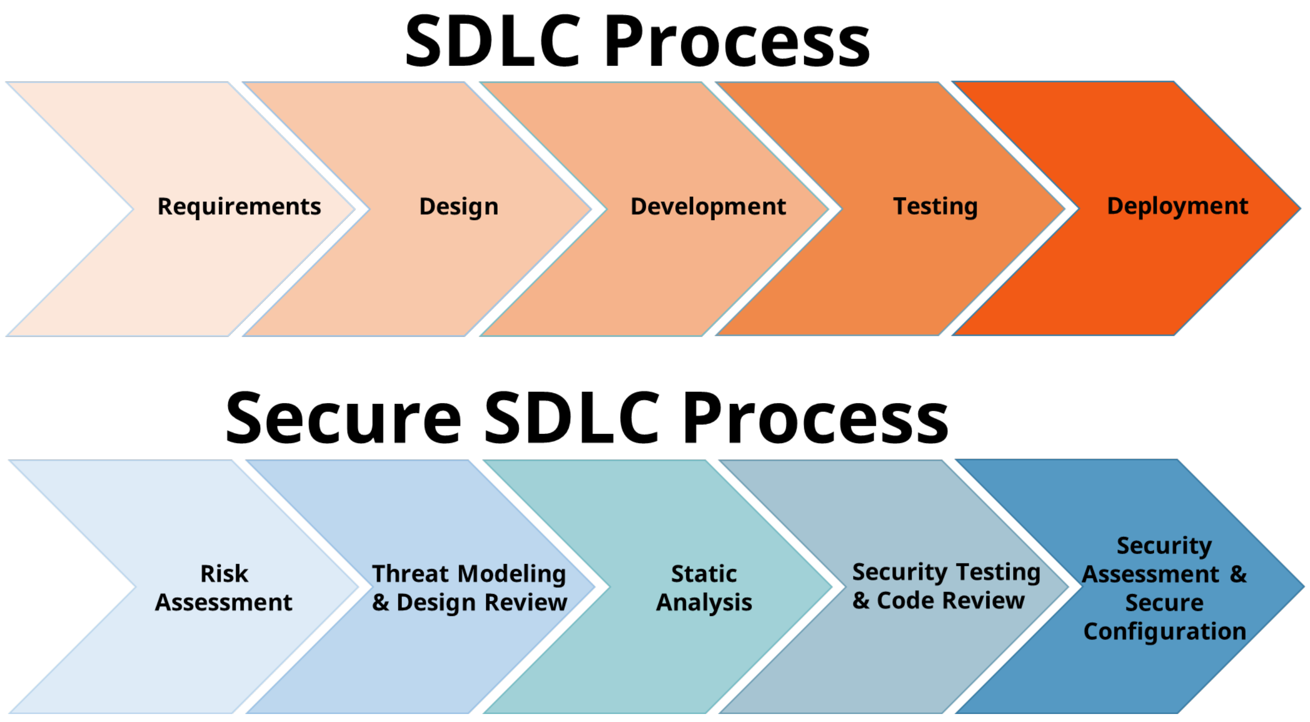
Version	Revision Date	Author	Description of Revisions
1.0	August 2024	S. Gopaldas Johnston	New.

5. Contact

For questions regarding these specifications, contact:

Cybersecurity and Digital Trust Branch, Office of the Chief Information Officer
Ministry of Citizens' Services
Email: InfoSecAdvisoryServices@gov.bc.ca

Appendix A: Secure software development life cycle (SDLC)



In SDLC, the phases that are common regardless of the SDLC methodology[9] adopted are requirements, design, development, testing, and deployment. For each phase, security processes MUST be integrated to ensure the software being developed meets the security requirements of the B.C. government. The security processes for each SDLC phase are described below.

1. Requirements

Identify and evaluate the security requirements based on the business and functional requirements for the software. For example, if the business requirement is to protect the confidentiality of the data that is handled by the software, the security requirements may include incorporating digital signatures/encryption, access control, authentication, and

[9] Examples of SDLC methodologies are DevOps, Agile, Waterfall, Spiral, Rapid Prototyping, Incremental, and etc.

audit logging components into the software. Determine the security risks the software is likely to face during operation by:

- Identifying the security classification label for the data the software will interact with following the IMIT 6.18 Information Security Classification Standard.
- Performing a rigorous security threat and risk assessment for high-risk areas, such as protecting sensitive data and safeguarding identification, authentication, and access control, including credential management to assess preliminary vulnerabilities and threats. See IMIT 6.11 Security Threat and Risk Assessment Standard for details on how to conduct a security threat and risk assessment and to complete an SoAR.
- Reviewing vulnerability reports on software components that will be integrated into the software to inform the security threat and risk assessment.
- Recording the response to each risk, including how mitigations are to be achieved and what the rationales are for approved exceptions to security requirements.

2. Design

In this phase, a threat modelling process is invoked to ensure threats are detected and a mitigation plan is created to protect the software against those threats. The software design is also reviewed to verify:

- Compliance with the security requirements to mitigate the risks identified in the requirements phase.
- Effectiveness of the security controls by conducting iterative assessments.

Also verify that third party software that will be integrated into the software complies with the security requirements by:

- Reviewing and evaluating third party software components in the context of their expected use.
- Obtaining provenance information for each software component (for example, software bill of materials (SBOM), source composition analysis, binary software composition analysis) to analyze and assess the risk that the component may introduce.

3. Development

In this phase of the SDLC, when the actual development starts, and the product is built, check open-source libraries for vulnerabilities before using them. Identify programming vulnerabilities to ensure common vulnerabilities identified in Appendix D: Common coding vulnerabilities and Appendix E: Web application and application interface vulnerabilities are not present in the software by conducting:

- Static analysis.
- Software composition analysis.

4. Testing

In modern SDLC methodologies, testing is part of the activities in all phases of the SDLC. Security testing and code review is also conducted to identify security vulnerabilities. The tests would include:

- Dynamic code scans.
- Interactive application security testing.
- Fuzz testing.

5. Deployment

Once the product is tested and ready to be deployed, conduct a security assessment again before releasing it. Manage the configuration of the system. Institute processes and procedures for assured operations and continuous monitoring of the application's security controls. Conduct periodic penetrating testing of the application's security controls based on the sensitivity of the information that is stored, processed, or transmitted by the application.

Appendix B: PWA security guidance

Progressive web applications (PWAs) are emerging technology and there is no universally accepted security framework to help secure them yet. To secure a PWA:

1. Follow the guidance provided by web browser providers (for example, Google, Mozilla, Microsoft, Apple) on building secure PWAs.
2. Use the built-in browser security features (HTTPS) to protect the PWA.
3. Define a manifest[10] for the PWA to make it less vulnerable to cross-site scripting attacks.
4. Configure the PWA to prevent the service workers[11]:
    a. From having access to:
        i. Document object model (DOM).
        ii. Local or session storage.
    b. From reading and setting a set of forbidden headers.
5. Configure the PWA to automatically clear any cached data when it is closed or upon logout.
6. Test the PWA for common web vulnerabilities.

[10] A manifest is a JSON file within the PWA. It contains all the necessary information for the PWA to be downloaded and presented.
[11] Service workers are go-betweens the front end and back end of the PWA. They provide developers the ability to add native-like features to the PWA.

Appendix C: 10 API security guidelines and best practices

Application programming interfaces (APIs) are a pivotal element in today's digital world, thanks to the rise of cloud computing and a shift from monolithic applications to microservices. As more businesses open up access to data and services through APIs, these vectors present an attractive target for data theft and attacks on software. Insecure APIs are a serious threat—they are commonly the most exposed component of a network, predisposed to denial-of-service attacks, and easy to reverse-engineer.

The following best practices can help expand and elevate the security of APIs:

1. **Understand the full scope of secure API consumption.**
   a. Understand how APIs work and the correct way to integrate them.
   b. Read API documentation thoroughly and pay particular attention to the process and security aspects of the API's function and routines, such as required authentication, call processes, data formats and any potential error messages to expect.
   c. Build a threat model for the API to understand the attack surface, identify potential security issues and incorporate appropriate security mitigations from the beginning.
2. **Validate the data.**
   a. Never assume API data has been cleansed or correctly validated.
   b. Implement data cleaning and validation routines to prevent standard injection flaws and cross-site request forgery attacks.
   c. Use debug tools to examine the API's data flow and keep track of errors and anomalies.
3. **Choose the web service API: Simple Object Access Protocol (SOAP), Representational State Transfer (REST), Remote Procedural Call (RPC), gRPC, or GraphQL.**
   a. SOAP is a communications protocol and security is applied at the message level via digital signatures and encrypted parts within the message itself.

b. REST is a set of architectural principles for data transmission and relies heavily on access control rules associated with the API's universal resource identifier (URI)[12], such as HTTP tags and the URL path.
c. RPC is a software communication protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.
d. gRPC is a modern open-source high performance RPC framework that works across different platforms.
e. GraphQL is an open-source language for querying data and provides a flexible and intuitive syntax to describe data requirements and interactions.

4. **Record APIs in an API registry.**
   a. An API is an information asset and MUST be documented in a registry to comply with the IMIT 6.23 Asset Management Security Standard.
   b. Register custom-built API by publishing it to the BC. Government API Registry.
5. **Assess API risks.**
   a. Perform a risk assessment for all APIs in the registry to identify all systems and data impacted if an API is compromised.
   b. Define a treatment plan and the controls required to reduce the risks to an acceptable level.
   c. Document review dates and repeat assessments whenever new threats arise or when an API is modified.
   d. Establish measures to ensure they adequately meet security policies and are not vulnerable to known risks as per the OWASP API Security Project Top 10 vulnerabilities.

[12] URI is a character sequence that identifies a logical (abstract) or physical resource. It is usually, but not always connected to the internet. It distinguishes one resource from another and enable internet protocols to facilitate interactions between and among these resources.

6. **Be diligent about API documentation.**
    a. Each API MUST have a document or manual that contains all technical API requirements, including its functions, classes, return types, arguments, and integration processes.
    b. Follow the guidelines for API documentation in the BC Government API Guidelines.
7. **Lock down access to APIs.**
    a. Introduce and test controls to manage who can access internal data and systems through the API.
    b. Keep APIs behind a firewall, a web application firewall, or an API gateway.
8. **Specify authentication and access.**
    a. Require client-side applications to include a token in the API call, so the service can validate the client.
    b. Use standards such as OAuth 2.0 and JSON web tokens to authenticate API traffic.
    c. Define access control rules, or grant types, which determine which users, groups, and roles can access specific API resources—always follow the principle of least privilege.
9. **Stash API keys.**
    a. Avoid embedding API keys directly in their code or in files within the application's source tree.
    b. Store API keys in environment variables or in files outside of the application's source tree.
    c. Use a secrets management service to protect and manage an application's API keys.
    d. Delete unneeded keys to minimize exposure to attack.
    e. Periodically regenerate keys, particularly if a breach is suspected to have occurred.

10. **Add Artificial Intelligence (AI) to API monitoring and threat detection.**
    a. AI-enabled behaviour analysis benchmarks normal API traffic and provides visibility into how users access and consume APIs to help fine-tune threshold settings for context security checks.
    b. Create a plan to handle the alerts produced by threat detection and other security controls that indicate an API attack.

Appendix D: Common coding vulnerabilities

The following coding vulnerabilities are based on the Payment Card Industry Data Security Standard (PCI DSS) Version 4.0, Requirement 6. The vulnerabilities and testing procedures are subject to change as secure coding techniques change.

Vulnerability	**Testing procedure**
Injection flaw	Validate input to verify user data. User data should not be able to modify meaning of commands and queries, or utilize parameterized queries to prevent attacks like SQL injection attacks, operating system (OS) command injection, lightweight directory access protocol (LDAP), and XPath injection flaws.
Buffer overflow	Validate buffer boundaries and truncate input strings.
Insecure cryptographic storage	Validate that cryptographic functions are used properly when used to protect stored data.
Unsecured communication channels	Validate that all authenticated and sensitive communications are properly encrypted.
Incorrect error handling	Validate that sensitive information is not leaked via error messages.

Appendix E: Web application and application interface vulnerabilities

The following coding vulnerabilities are based on the Payment Card Industry Data Security Standard (PCI DSS) Version 4.0, Requirement 6. The vulnerabilities and testing procedures are subject to change as secure coding techniques change.

Vulnerability	**Testing procedure**
Injection flaw	Validate input to verify user data. User data should not be able to modify meaning of commands and queries, or utilize parameterized to prevent attacks like SQL injection attacks, operating system (OS) command injection, lightweight directory access protocol (LDAP), and XPath injection flaws.
Buffer overflow	Validate buffer boundaries and truncate input strings.
Insecure cryptographic storage	Validate that cryptographic functions are used properly when used to protect stored data.
Unsecured communication channels	Validate that all authenticated and sensitive communications are properly encrypted.
Incorrect error handling	Validate that sensitive information is not leaked via error messages.

Appendix F: Assessment guidelines

This appendix provides a list of the assessments, including the tests, scans, and reports, that may be necessary and should be considered when developing and deploying applications. Automated scanning tools may assist in the assessment of applications.

Network vulnerability scans to identify/report on:

- Assets inappropriately accessible from externally and internally connected network devices.

Server vulnerability scans to identify/report on:

- Unauthorized software.
- Inappropriately opened server ports, enabled protocols, and enabled services.
- Misconfigured or inappropriately enabled high risk services (for example, ftp and telnet).
- Inappropriate stored credentials within batch jobs, scripts, or plain text files.
- Inappropriate local accounts that exist with non-expiring passwords.
- Inadequate encryption methods and level used.
- Ability to gain unauthorized access to encryption keys.
- Weak server passwords that might be determined via password cracking.
- Missing patches.

Application tests to identify/report on:

- Insecure API calls or responses.
- Insecure cross application interfaces.
- Insecure coding of customized code within COTS products or code that use COTS APIs.
- Insecure coding and functionality of sensitive application functions or of any privileged access interfaces such as application administrator screens.
- Ability to execute commands or inject code (for example, OS commands, SQL injection, Cross-site Scripting, LDAP injection).
- Inadequate session management controls.

- Ability to perform URL path traversal attacks.
- Ability to cause overflow conditions (for example, parameter overflow and buffer overflow).
- Ability to perform character encoding attacks.
- Ability to compromise an application by supplying inappropriate input values (that is, fuzz testing).
- Security flaws within Web Services (REST-based and SOAP) used by the application.

Static code analysis to identify/report on:

- The existence of Logic Bombs or Backdoors.
- Enabled debugging features.
- Credentials inappropriately stored within code.
- Use of weak encryption algorithms.
- Insecure use of client-provided data (lacking input validation).
- Use of language-specific coding standards.
- Potential for injection attacks.
- Insecure use of user sessions.
- Insecure handling of file uploads.
- Insecure configuration of SSL/TLS calls.

Middleware scans to identify/report on:

- Inappropriate configuration settings.
- Unauthorized directories that can be traversed or displayed (that is, directory enumeration).
- Unauthorized server-side application files that are accessible for downloading or inspection by clients (for example, viewing php, jsp, or asp file contents).
- Unnecessary product information displayed (for example, installed modules).
- Unnecessary accounts or features enabled.
- Missing patches.
- Default passwords.

Database scans to identify/report on:

- Inappropriate configuration settings.
- Unnecessary accounts or features enabled.
- Excessive privileges granted to database objects or to database OS files.
- Inappropriate local accounts with non-expiring passwords.
- Credentials inappropriately stored within batch jobs or scripts.
- Inadequate segregation of duties.
- Existence of privileged utilities or enabled debugging features in the production environment.
- Inadequacy of encryption method and level used.
- Ability to gain unauthorized access to encryption keys.
- Weak strength database passwords that might be determined via password cracking.
- Database replication over insecure channels.
- Ability to read, modify, copy, or remove configuration data, logs, and access control information.
- Adequacy of controls for all entrance and exit points of an application.
- Missing patches.
- Default passwords.

Application penetration tests to identify/report on:

- Inappropriate configuration settings.
- Unnecessary accounts or features enabled.
- Excessive privileges.
- Ability to bypass normal application access paths.
- Inadequate session management controls.
- Inadequate segregation of infrastructure.
- Weak application passwords that might be determined via password cracking.
- Inadequate asset segregation by purpose and environment.
- Privileged access not via the administrative gateway.

- Ability to escalate privileges.
- Ability to infiltrate data.
- Ability to store malicious content.
- Ability to gain unauthorized access to data and to installed product files.
- Ability to gain unauthorized access to encryption keys.
- Ability to gain unauthorized access to administrative interfaces and tools.
- Ability to read, modify, copy, or remove configuration data, logs, and access control information.
- Privileged utilities or enabled debugging features in the production environment.
- Inadequate encryption method and level used.
- Ability to gain unauthorized access to encryption keys.
- Vulnerability to common attacks, such as DDoS, and session replay.
- Inappropriate access or application functionality which is not restricted based on accesses granted to user roles.
- Adequacy of controls for all entrance and exit points of an application.
- Default passwords.

Dependency Analysis to identify/report on:

- Third party libraries and dependencies used, and any vulnerabilities associated with them.
- Licenses associated with third party libraries, and any legal exposure created by them.

Application recovery exercises to identify/report on:

- Ability to perform full recoveries and point-in-time recoveries.
- Acceptable levels of business data loss for a point-in-time recovery.
- Length and severity of outage.
- Alignment of support contracts with recovery objectives.

Incident response exercises to identify/report on:

- Logging details and retention requirements as specified in the Province's ARCS/ORCS.
- Ability to generate and receive expected notifications and alerts.
- Ability to change application and infrastructure privileged access credentials in the event of a breach.
- Internal and cross-government response procedures.

Audit exercise to identify/report on:

- Adequacy of security assessments.
- Missing evidence that is required to pass an audit.
- Adequacy of the application documentation.