API Security Guidelines

CWB Financial Group – Information Security Office

Document Version – V1.0

May 21, 2020

Contents

Document Summary

Brief overview

CWBFG requires all Application Programming Interfaces (APIs) to be managed and controlled.

APIs are utilized to innovate faster and create new products and services more efficiently for both internal and external consumption. Micro services architectures, API-led architectures and modern API-based applications naturally expose application logic and sensitive data, significantly expanding the attack surface. Inevitable regulatory changes will bring Open Banking to Canada and CWBFG will be required to support this ecosystem for seamless sharing of financial data. As a result, APIs will be a target of abuse and set to become the most frequent attack route for data breaches. CWBFG must prioritize API security and adhere to the following API security guidelines as an essential layer of defense.

Scope

These guidelines apply to all internal or external consumption of CWBFG APIs, and provide recommended best practices to be implemented by developers and integrators implementing digital services, if and when possible.

Out of Scope

These guidelines does not include any specifics on how to write application programming interface code. Instead, it is advised that developers leverage trusted development sources available over the Internet (e.g. OWASP, SANS, etc.)

Alignment

This API Security Guideline document aligns with ***Secure Software Design Standard***.

Guidelines

Secure Protocol (HTTPS)

- Secure REST services must only provide HTTPS endpoints. This protects authentication credentials in transit, for example passwords, API keys or JSON Web Tokens. It also allows clients to authenticate the service and guarantees integrity of the transmitted data.
- Consider the use of mutually authenticated client-side certificates to provide additional protection for high privilege web services.

Access Control

- API access starts with API keys because the required logic is implemented by most frameworks out of the box making them easy and fast to set up. However, API keys are generally used for authentication and offer little to no options for varying permissions according to use case or concept of built in “rotation”. It is recommended to use token based authentication such OAuth 2.0 which are capable of creating role based access and designed to expire/refresh.
- Non-public REST services must perform access control at each API endpoint. Web services in monolithic applications implement this by means of user authentication, authorization logic and session management.
- In order to minimize latency and reduce coupling between services, the access control decision should be taken locally by REST endpoints
- User authentication should be centralized in an Identity Provider (IDP), which issues access tokens. Scope should be used to limit the consumer access to APIs.
- Access to APIs should be restricted based on an authorization or access control rules. Role-based and/or attribute-based access control should be enforced.
- Each consuming application must be assigned a Client ID and API consumption limited to a known set of Client IDs.

OIDC Tokens

- Adopt JSON Web Tokens (JWT) as the format for Access and Identity tokens. JWTs are JSON data structures containing a set of claims that are used for access control decisions.
- A cryptographic signature must be used to protect the integrity of the JWT.
- Tokens must be unique to the user/application, have an expiration period, should be revocable and must be rotated.
- As JWTs contain details of the authenticated entity (user etc.), a disconnect can occur between the JWT and the current state of the users session, for example, if the session is terminated earlier than the expiration time due to an explicit logout or an idle timeout. When an explicit session termination event occurs, a digest or hash of any associated JWTs should be submitted to a blacklist on the API which will invalidate that JWT for any requests until the expiration of the token.
- Avoid storing access tokens, but when required, encrypt tokens at rest.
- Access tokens should be used exclusively via an HTTP Authorization header instead of encoded into a payload or URL which may be logged or cached.

Token Validation

- A relying party must verify the integrity of the JWT based on its own configuration/hard-coded logic or via a call to the Token Introspection endpoint of the authorization server. It must not rely on the information of the JWT header to select the verification algorithm due to critical vulnerabilities in many JSON Web Token libraries.
- **Access Tokens:** Claims standardized for use in JWT Access Tokens should be included and verified by each endpoint. The following must be present in all Access Tokens:
  - The iss (issuer) claim matches the identifier of the Authorization Server.
  - Verify that the aud (audience) claim is the value configured in the Authorization Server
  - The cid claim should match the Client ID that was used to request the Access Token.
  - The exp (expiry time) claim is the time at which this token will expire. Ensure this time has not already passed.
  - nbf or not before time - is the current time after the start of the validity period of this token?
  - The scope claim should match the permissions required for the endpoint being accessed.
- **ID Tokens:** Claims standardized for use in JWT ID Tokens should be included and verified by each endpoint. The following must be present in all ID Tokens:
  - The iss (issuer) claim matches the identifier of the Authorization Server.
  - The aud (audience) claim should match the Client ID that was used to request the ID Token.
  - The iat (issued at time) claim indicates when this ID token was issued.
  - The exp (expiry time) claim is the time at which this token will expire. Ensure this time has not already passed.
  - The nonce claim value should match whatever was passed when the Client requested the ID token.

SLA Based Contracts

- The rate of access to an API should be limited according to an agreed threshold on typical and/or peak use of an API by a specific Client ID.
- Consider the use case and apply either:
  - Rate Limiting: allows a certain number of requests in a given interval and rejects any requests that exceed the limit.
  - Throttling: queue the requests that exceed the threshold for retry.

API Keys

- Public REST services without access control run the risk of being farmed leading to excessive bills for bandwidth or compute cycles. API keys can be used to mitigate this risk. They are also often used by organization to monetize APIs; instead of blocking high-frequency calls, clients are given access in accordance to a purchased access plan
- API keys can reduce the impact of denial-of-service attacks. However, when they are issued to third-party clients, they are relatively easy to compromise.
  - Require API keys and secrets for every request to the protected endpoint.
  - Return 429 Too Many Requests HTTP response code if requests are coming in too quickly.
  - Revoke the API key if the client violates the usage agreement.
  - Do not rely exclusively on API keys to protect sensitive, critical or high-value resources (See Access Tokens).

Restrict HTTP methods

- Apply a whitelist of permitted HTTP Methods e.g. GET, POST, PUT.
- Reject all requests not matching the whitelist with HTTP response code 405 Method not allowed.
- Make sure the caller is authorized to use the incoming HTTP method on the resource collection, action, and record.

Mass Assignment

- Do not automatically bind incoming data and internal objects.
- Explicitly define all the parameters and payloads you are expecting.
- Use the read-only property set to true in object schemas for all properties that can be retrieved through APIs but should never be modified.
- Precisely define the schemas, types, and patterns you will accept in requests at design time and enforce them at runtime.

Input Validation

- Do not trust any API consumers, even if they are internal.
- Do not trust input parameters/objects without proper validation.
- Validate input: length / range / format and type.
- Achieve an implicit input validation by using strong types like numbers, booleans, dates, times or fixed data ranges in API parameters.
- Constrain string inputs with regex.
- Reject unexpected/illegal content.
- Make use of validation/sanitation libraries or frameworks in your specific language.
- Define an appropriate request size limit and reject requests exceeding the limit with HTTP response status 413 Request Entity Too Large.
- Log input validation failures. Assume that someone who is performing hundreds of failed input validations per second is up to no good.
- Use a secure parser for parsing the incoming messages. If you are using XML, make sure to use a parser that is not vulnerable to XXE and similar attacks.
- Add checks on compression ratios to avoid Zip bombs.
- Enforce the OWASP Core Rule Set (CRS) for checking requests and responses to detect common web application attacks.

Content Types

- A REST request or response body should match the intended content type in the header or else this could cause misinterpretation at the consumer/producer side and lead to code injection/execution.
- Reject requests containing unexpected or missing content type headers with HTTP response status 406 Unacceptable or 415 Unsupported Media Type.
- For XML content types ensure appropriate XML parser hardening.
- Avoid accidentally exposing unintended content types by explicitly defining content types. This avoids XXE-attack vectors for example.
- It is common for REST services to allow multiple response types (e.g., application/xml or application/json, and the client specifies the preferred order of response types by the Accept header in the request.

- Do NOT simply copy the Accept header to the Content-type header of the response.
- Reject the request (ideally with a 406 Not Acceptable response) if the Accept header does not specifically contain one of the allowable types

Endpoint Management

- Avoid exposing management endpoints via the Internet.
- If management endpoints must be accessible via the Internet, make sure that users must use a strong authentication mechanism, e.g. multi-factor.
- Expose management endpoints via different HTTP ports or hosts preferably on a different NIC and restricted subnet.
- Restrict access to these endpoints by firewall rules or use of access control lists.
- Keep an up-to-date inventory all API hosts.
- Limit access to anything that should not be public.
- Limit access to production data, and segregate access to production and non-production data.
- Implement additional external controls, such as API firewalls.
- Properly retire old versions of APIs or backport security fixes to them.
- Implement strict authentication, redirects, CORS, and so forth.

Error Handling

- Respond with generic error messages - avoid revealing details of the failure unnecessarily. Information required to diagnose the problem should be included in Logs and not displayed in messages to end user.
- Do not pass technical details (e.g. call stacks or other internal hints) to the client

Audit Logs

- Write audit logs before and after security related events, including when a client exceeds SLA thresholds.
- Log token validation errors in order to detect attacks.
- Take care of log injection attacks by sanitizing log data beforehand.
- Log failed attempts, denied access, input validation failures, or any failures in security policy checks.
- Ensure logs are formatted appropriately so other tools can consume them.
- Include enough detail to identify attackers.
- Avoid having sensitive data in logs — if you need the information for debugging purposes, redact it partially.
- Log activities into CWBFG log management system and integrate with other dashboards, monitoring, and alerting tools.

Security Headers

- To make sure the content of a given resources is interpreted correctly by the browser, the API response should always send the Content-Type header with the correct content type, and preferably the Content-Type header should include a charset. The server should also send the X-Content-Type-Options: no sniff security header to make sure the browser does not try to detect a different Content-Type than what is actually sent (can lead to XSS).
- The API response should send the X-Frame-Options: DENY security header to protect against drag and drop clickjacking attacks in older browsers.

Cross Origin Resoure Sharing (CORS)

- Cross-Origin Resource Sharing (CORS) is a W3C standard to flexibly specify what cross-domain requests are permitted. By delivering appropriate CORS Headers your REST API signals to the browser which domains, AKA origins, are allowed to make JavaScript calls to the REST service.
  - Disable CORS headers if cross-domain calls are not supported/expected.
  - Be as specific as possible and as general as necessary when setting the origins of cross-domain calls.

Sensitive Information In HTTP Requests

- RESTful web services should be careful to prevent leaking credentials. Passwords, security tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.
  - In POST/PUT requests sensitive data should be transferred in the request body or request headers.
  - In GET requests sensitive data should be transferred in an HTTP Header.
- API's should be carefully designed to ensure only the right amount of information is given to each consumer of an API (including errors). Filtering techniques may be required to filter out sensitive attributes. Identify all the sensitive data or Personally Identifiable Information (PII) and justify its use. If sensitive information is required by the consuming application, then tokenization must be applied wherever possible. Tokenization is the process of replacing sensitive data with unique identification symbols that retain all the essential information about the data without compromising its security.

HTTP Return Code

- When designing REST API, always use the semantically appropriate http status code for the response. Use of 200 for success or 404 for error should be limited.

Development / Testing

- MFA based authentication is required when accessing API source code.
- Secure access tokens or secrets via FIPS compliant Hardware Security Module (HSM). Avoid storing access tokens or secrets in source code, but when required, encrypt these secrets.
- Regression test against known security patterns for all APIs.
- Publish documentation targeted at API consumers, on how best to secure API keys/secrets required to access CWB APIs.
- Establish repeatable hardening and patching processes.
- Automate locating configuration flaws.
- Disable unnecessary features.
- Align to the ***Vulnerability Management Standard***.

Roles and Responsibilities

Role	Responsibility
Chief Information Security Officer	• Accountable for the creation, maintenance, and implementation of these guidelines where applicable. • Accountable to maintain written guidelines and procedures necessary to ensure implementation of and compliance to the SDLC Security standard. • Accountable to provide appropriate support and guidance to assist employees to fulfill their responsibilities of complying with these guidelines.
Sr. Manager, Information Security Program Management	• Responsible for the creation and maintenance of these guidelines and supporting policy/standard where applicable. • Responsible to have and maintain written guidelines and procedures necessary to ensure implementation of and compliance to this SDLC Security standard. • Responsible to provide support and guidance to assist employees to fulfill their responsibilities of complying with these guidelines. • Consulting with Sr. Manager, Security Governance, Risk and Compliance and Sr. Manager, Security Operations, as required.
CWB's Executive Leadership Team, Senior Leadership Team, Directors, and Managers	• Understand and comply with this SDLC security standard and supporting guidelines in its entirety. • Responsible to create and maintain processes and procedures to these guidelines and supporting policy/standards. • Responsible to ensure that all appropriate personnel are aware of and comply with this these guidelines and supporting policy/standards. • Responsible for the creation of appropriate performance standards, control practices, and procedures designed to provide reasonable assurance that all employees observe these guidelines and supporting policy/standards.
CWB employees, contractors, third-party service provider, etc.	• Understand and comply with these guidelines and supporting policy/standard in its entirety. • Implement these guidelines with supporting processes and procedures. • Report vulnerabilities and breaches.

Appendix A – Document Control

Document Status

Document Name	API Security Guidelines
Document Owner	Chief Information Security Officer
Version	Version 1.0
Publication Date	May 21, 2020
Information Classification	Internal Use
Revision Status	Final
Custodian	Sr. Manager, Information Security Program Management
Organization	CWBFG Information Security Office
Retention Period	Retain for ongoing use
Master Storage Location	

Revision History

Version	Author	Contributor	Description of Changes
Draft 0.1	Reinhardt Tonn	Vikram Singh	Document creation
Draft 0.2	Joanne Pearson	Vikram Singh	Document review and updates
Draft 1.0	Vikram Singh	Joanne Pearson	Document review by: Michael Thompson, Jose Barril, Reinhardt Tonn, Mark Doubinin, Thomas Matthews and National Leasing
Draft 2.0	Vikram Singh	Joanne Pearson	Finalize Draft for Review
Final 1.0	Joanne Pearson	Vikram Singh	Final Standard – after CISO review – Published in Keylight