REST Security Cheat Sheet

From OWASP

Last revision (mm/dd/yy): **11/13/2017**

Introduction

REST (http://en.wikipedia.org/wiki/Representational_state_transfer) (or REpresentational State Transfer) is an architectural style first described in Roy Fielding (https://en.wikipedia.org/wiki/Roy_Fielding)'s Ph.D. dissertation on Architectural Styles and the Design of Network-based Software Architectures (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). It evolved as Fielding wrote the HTTP/1.1 and URI specs and has been proven to be well-suited for developing distributed hypermedia applications. While REST is more widely applicable, it is most commonly used within the context of communicating with services via HTTP.

The key abstraction of information in REST is a resource. A REST API resource is identified by a URI, usually a HTTP URL. REST components use connectors to perform actions on a resource by using a representation to capture the current or intended state of the resource and transferring that representation. The primary connector types are client and server, secondary connectors include cache, resolver and tunnel. In order to implement flows with REST APIs, resources are typically created, read, updated and deleted. For example, an ecommerce site may offer methods to create an empty shopping cart, to add items to the cart and to check out the cart.

Another key feature of REST applications is the use of standard HTTP verbs and error codes in the pursuit or removing unnecessary variation among different services.

Another key feature of REST applications is the use of HATEOS or Hypermedia as the Engine of Application State. This provides REST applications a self-documenting nature making it easier for developer to interact with a REST service without a-priori knowledge.

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.

See the Transport Layer Protection Cheat Sheet for additional information.

Consider the use of mutually authenticated client-side certificates to provide additional protection for highly privileged web services.

Access Control

Non-public REST services must perform access control at each API endpoint. Web services in monolithic applications implement this by means of user authentication, authorisation logic and session management. This has several drawbacks for modern architectures which compose multiple micro services following the RESTful style.

- in order to minimise latency and reduce coupling between services, the access control decision should be taken locally by REST endpoints
- user authentication should be centralised in a Identity Provider (IdP), which issues access tokens

JWT

There seems to be a convergence towards using JSON Web Tokens (https://tools.ietf.org/html/rfc7519) (JWT) as the format for security tokens. JWTs are JSON data structures containing a set of claims that can be used for access control decisions. A cryptographic signature or message authentication code (MAC) can be used to protect the integrity of the JWT.

- Ensure JWTs are integrity protected by either a signature or a MAC. Do not allow the unsecured JWTs: {"alg":"none"}. See https://tools.ietf.org/html/rfc7519#section-6.1
- In general, signatures should be preferred over MACs for integrity protection of JWTs.

If MACs are used for integrity protection, every service that is able to validate JWTs can also create new JWTs using the same key. This means that all services using the same key have to mutually trust each other. Another consequence of this is that a compromise of any service also compromises all other services sharing the same key. See https://tools.ietf.org/html/rfc7515#section-10.5 for additional information.

The relying party or token consumer validates a JWT by verifying its integrity and claims contained.

- A relying party must verify the integrity of the JWT based on its own configuration or hard-coded logic. It must not rely on the information of the JWT header to select the verification algorithm. See https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html and https://www.youtube.com/watch?v=bW5pS4e_MX8

Some claims have been standardised and should be present in JWT used for access controls. At least the following of the standard claims should be verified:

- 'iss' or issuer - is this a trusted issuer? Is it the expected owner of the signing key?
- 'aud' or audience - is the relying party in the target audience for this JWT?
- 'exp' or expiration time - is the current time before the end of the validity period of this token?
- 'nbf' or not before time - is the current time after the start of the validity period of this token?

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 organisation 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 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.

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 authorised to use the incoming HTTP method on the resource collection, action, and record

In Java EE in particular, this can be difficult to implement properly. See Bypassing Web Authentication and Authorization with HTTP Verb Tampering (http://www.aspectsecurity.com/research-presentations/bypassing-vbaac-with-http-verb-tampering) for an explanation of this common misconfiguration.

Input validation

- Do not trust input parameters/objects
- 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 regexps
- 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
- Consider logging input validation failures. Assume that someone who is performing hundreds of failed input validations per second is up to no good.
- Have a look at input validation cheat sheet for comprehensive explanation

- 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 (https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing) and similar attacks.

Validate content types

A REST request or response body should match the intended content type in the header. Otherwise this could cause misinterpretation at the consumer/producer side and lead to code injection/execution.

- Document all supported content types in your API

Validate request content types

- 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, see the cheat sheet

- Avoid accidentally exposing unintended content types by explicitly defining content types e.g. Jersey (https://jersey.github.io/) (Java) *@consumes("application/json"); @produces("application/json")*. This avoids XXE-attack vectors for example.

Send safe response content types

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.

Services including script code (e.g. JavaScript) in their responses must be especially careful to defend against header injection attack.

- ensure sending intended content type headers in your response matching your body content e.g. "application/json" and not "application/javascript"

Management endpoints

- Avoid exposing management endpoints via 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.

Error handling

- Respond with generic error messages - avoid revealing details of the failure unnecessarily
- 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
- Consider logging token validation errors in order to detect attacks
- Take care of log injection attacks by sanitising log data beforehand

Security headers

To make sure the content of a given resources is interpreted correctly by the browser, the server 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 an `X-Content-Type-Options: nosniff` to make sure the browser does not try to detect a different Content-Type than what is actually sent (can lead to XSS).

Additionally the client should send an `X-Frame-Options: deny` to protect against drag'n drop clickjacking attacks in older browsers.

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
- Be as specific as possible and as general as necessary when setting the origins of cross-domain calls

In Spring Boot (Java), for example, CORS support is disabled by default and is only enabled once the *endpoints.cors.allowed-origins* property has been set. The configuration below permits GET and POST calls from the example.com domain:

```
endpoints.cors.allowed-origins=https://example.com

endpoints.cors.allowed-methods=GET,POST
```

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

OK:

- https://example.com/resourceCollection/<id>/action

- https://twitter.com/vanderaj/lists

NOT OK:

- https://example.com/controller/<id>/action?apiKey=a53f435643de32 (API Key in URL)

HTTP Return Code

HTTP defines status code [1] (https://en.wikipedia.org/wiki/List_of_HTTP_status_codes). When designing REST API, don't just use 200 for success or 404 for error.

Here is a selection of security related REST API status codes. Use it to ensure you return the correct code.

Status code	Message	Description
200	OK	Response to a successful REST API action. The HTTP method can be GET, POST, PUT, PATCH or DELETE
201	Created	The request has been fulfilled and resource created. A URI for the created resource is returned in the Location header
202	Accepted	The request has been accepted for processing, but processing is not yet complete
400	Bad Request	The request is malformed, such as message body format error
401	Unauthorized	Wrong or no authentication ID/password provided
403	Forbidden	It's used when the authentication succeeded but authenticated user doesn't have permission to the request resource
404	Not Found	When a non-existent resource is requested
406	Unacceptable	The client presented a content type in the Accept header which is not supported by the server API
405	Method Not Allowed	The error for an unexpected HTTP method. For example, the REST API is expecting HTTP GET, but HTTP PUT is used
413	Payload too large	Use it to signal that the request size exceeded the given limit e.g. regarding file uploads
415	Unsupported Media Type	The requested content type is not supported by the REST service
429	Too Many Requests	The error is used when there may be DOS attack detected or the request is rejected due to rate limiting

Authors and primary editors

Erlend Oftedal - erlend.oftedal@owasp.org
Andrew van der Stock - vanderaj@owasp.org
Tony Hsu Hsiang Chih- Hsiang_chihi@yahoo.com
Johan Peeters - yo@johanpeeters.com
Jan Wolff - jan.wolff@owasp.org
Rocco Gränitz - rocco.graenitz@owasp.org

Other Cheatsheets

V - T - E (https://www.owasp.org/index.php?title=REST_Security_Cheat_Sheet&action=edit) **Cheat Sheets** [Collapse]

Developer / Builder	3rd Party Javascript Management · Access Control · AJAX Security Cheat Sheet · Authentication (ES) · Bean Validation Cheat Sheet · Choosing and Using Security Questions · Clickjacking Defense · Credential Stuffing Prevention Cheat Sheet · Cross-Site Request Forgery (CSRF) Prevention · Cryptographic Storage · C-Based Toolchain Hardening · Deserialization · DOM based XSS Prevention · Forgot Password · HTML5 Security · HTTP Strict Transport Security · Injection Prevention Cheat Sheet · Injection Prevention Cheat Sheet in Java · JSON Web Token (JWT) Cheat Sheet for Java · Input Validation · JAAS · LDAP Injection Prevention · Logging · Mass Assignment Cheat Sheet · .NET Security · OS Command Injection Defense Cheat Sheet · OWASP Top Ten · Password Storage · Pinning · Query Parameterization · **REST Security** · Ruby on Rails · Session Management · SAML Security · SQL Injection Prevention · Transaction Authorization · Transport Layer Protection · Unvalidated Redirects and Forwards · User Privacy Protection · Web Service Security · XSS (Cross Site Scripting) Prevention · XML External Entity (XXE) Prevention Cheat Sheet
Assessment / Breaker	Attack Surface Analysis · REST Assessment · Web Application Security Testing · XML Security Cheat Sheet · XSS Filter Evasion
Mobile	Android Testing · IOS Developer · Mobile Jailbreaking
OpSec / Defender	Virtual Patching
Draft and Beta	Application Security Architecture · Business Logic Security · Content Security Policy · Denial of Service Cheat Sheet · Grails Secure Code Review · Insecure Direct Object Reference Prevention · IOS Application Security Testing · Key Management · PHP Security · Regular Expression Security Cheatsheet · Secure Coding · Secure SDLC · Threat Modeling · Vulnerability Disclosure

All Pages In This Category