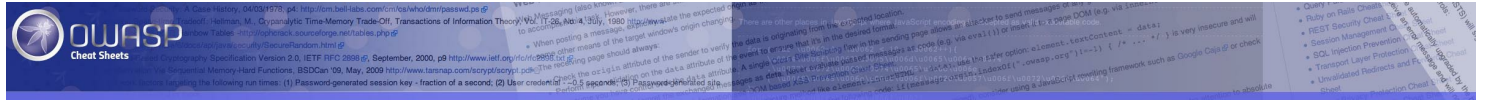


REST Security Cheat Sheet

 www.owasp.org/index.php/REST_Security_Cheat_Sheet



Last revision (mm/dd/yy): **04/10/2016**

Introduction

REST (or REpresentational State Transfer) is a means of expressing specific entities in a system by URL path elements. REST is not an architecture but it is an architectural style to build services on top of the Web. REST allows interaction with a web-based system via simplified URLs rather than complex request body or `POST` parameters to request specific items from the system. This document serves as a guide (although not exhaustive) of best practices to help REST-based services.

Authentication and session management

RESTful web services should use session-based authentication, either by establishing a session token via a `POST` or by using an API key as a `POST` body argument or as a cookie. Usernames, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.

OK:

- <https://example.com/resourceCollection//action>
- <https://twitter.com/vanderaj/lists>

NOT OK:

- <https://example.com/controller//action?apiKey=a53f435643de32> (API Key in URL)
- <http://example.com/controller//action?apiKey=a53f435643de32> (transaction not protected by TLS; API Key in URL)

OAuth for Authentication and Authorization

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

<https://tools.ietf.org/html/rfc6749>

Protect Session State

Many web services are written to be as stateless as possible. This usually ends up with a state blob being sent as part of the transaction.

- Consider using only the session token or API key to maintain client state in a server-side cache. This is directly equivalent to how normal web apps do it, and there's a reason why this is moderately safe.
- Anti-replay. Attackers will cut and paste a blob and become someone else. Consider using a time limited encryption key, keyed against the session token or API key, date and time, and incoming IP address. In

general, implement some protection of local client storage of the authentication token to mitigate replay attacks.

- Don't make it easy to decrypt; change the internal state to be much better than it should be.

In short, even if you have a brochureware web site, don't put in <https://example.com/users/2313/edit?isAdmin=false&debug=false&allowCSRPanel=false> as you will quickly end up with a lot of admins, and help desk helpers, and "developers".

Authorization

Anti-farming

Many RESTful web services are put up, and then farmed, such as a price matching website or aggregation service. There's no technical method of preventing this use, so strongly consider means to encourage it as a business model by making high velocity farming is possible for a fee, or contractually limiting service using terms and conditions. CAPTCHAs and similar methods can help reduce simpler adversaries, but not well funded or technically competent adversaries. Using mutually assured client side TLS certificates may be a method of limiting access to trusted organizations, but this is by no means certain, particularly if certificates are posted deliberately or by accident to the Internet.

Protect HTTP methods

RESTful API often use GET (read), POST (create), PUT (replace/update) and DELETE (to delete a record). Not all of these are valid choices for every single resource collection, user, or action. Make sure the incoming HTTP method is valid for the session token/API key and associated resource collection, action, and record. For example, if you have an RESTful API for a library, it's not okay to allow anonymous users to DELETE book catalog entries, but it's fine for them to GET a book catalog entry. On the other hand, for the librarian, both of these are valid uses.

Whitelist allowable methods

It is common with RESTful services to allow multiple methods for a given URL for different operations on that entity. For example, a GET request might read the entity while PUT would update an existing entity, POST would create a new entity, and DELETE would delete an existing entity. It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (for example, a 403 Forbidden).

In Java EE in particular, this can be difficult to implement properly. See [Bypassing Web Authentication and Authorization with HTTP Verb Tampering](#) for an explanation of this common misconfiguration.

Protect privileged actions and sensitive resource collections

Not every user has a right to every web service. This is vital, as you don't want administrative web services to be misused:

- <https://example.com/admin/exportAllData>

The session token or API key should be sent along as a cookie or body parameter to ensure that privileged collections or actions are properly protected from unauthorized use.

Protect against cross-site request forgery

For resources exposed by RESTful web services, it's important to make sure any PUT, POST, and DELETE request is protected from Cross Site Request Forgery. Typically one would use a token-based approach. See [Cross-Site Request Forgery \(CSRF\) Prevention Cheat Sheet](#) for more information on how to implement CSRF-protection.

CSRF is easily achieved even using random tokens if any XSS exists within your application, so please make sure you understand [how to prevent XSS](#).

Insecure direct object references

It may seem obvious, but if you had a bank account REST web service, you'd have to make sure there is adequate checking of primary and foreign keys:

- [https://example.com/account/325365436/transfer?amount=\\$100.00&toAccount=473846376](https://example.com/account/325365436/transfer?amount=$100.00&toAccount=473846376)

In this case, it would be possible to transfer money from any account to any other account, which is clearly absurd. Not even a random token makes this safe.

- <https://example.com/invoice/2362365>

In this case, it would be possible to get a copy of all invoices.

This is essentially a data-contextual access control enforcement need. A URL or even a POSTed form should NEVER contain an access control "key" or similar that provides automatic verification. **A data contextual check needs to be done, server side, with each request.**

Input validation

Input validation 101

Everything you know about input validation applies to RESTful web services, but add 10% because automated tools can easily fuzz your interfaces for hours on end at high velocity. So:

- Assist the user > Reject input > Sanitize (filtering) > No input validation

Assisting the user makes the most sense, as the most common scenario is "problem exists between keyboard and computer" (PEBKAC). Help the user input high quality data into your web services, such as ensuring a Zip code makes sense for the supplied address, or the date makes sense. If not, reject that input. If they continue on, or it's a text field or some other difficult to validate field, input sanitization is a losing proposition but still better than XSS or SQL injection. If you're already reduced to sanitization or no input validation, make sure output encoding is very strong for your application.

Log input validation failures, particularly if you assume that client-side code you wrote is going to call your web services. The reality is that anyone can call your web services, so assume that someone who is performing hundreds of failed input validations per second is up to no good. Also consider rate limiting the API to a certain number of requests per hour or day to prevent abuse.

Secure parsing

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.

Strong typing

It's difficult to perform most attacks if the only allowed values are true or false, or a number, or one of a small number of acceptable values. Strongly type incoming data as quickly as possible.

Validate incoming content-types

When POSTing or PUTting new data, the client will specify the Content-Type (e.g. `application/xml` or `application/json`) of the incoming data. The server should never assume the Content-Type; it should always check that the Content-Type header and the content are the same type. A lack of Content-Type header or an unexpected Content-Type header should result in the server rejecting the content with a `406 Not Acceptable`

response.

Validate response 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.

Because there are many MIME types for the typical response types, it's important to document for clients specifically which MIME types should be used.

XML input validation

XML-based services must ensure that they are protected against common XML based attacks by using secure XML-parsing. This typically means protecting against XML External Entity attacks, XML-signature wrapping etc. See <http://ws-attacks.org> for examples of such attacks.

Framework-Provided Validation

Many frameworks, such as [Jersey](#), allow for validation constraints to be enforced automatically by the framework at request or response time. (See [Bean Validation Support](#) for more information). While this does not validate the structure of JSON or XML data before being unmarshaled, it does provide automatic validation after unmarshaling, but before the data is presented to the application.

Output encoding

Send 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.

JSON encoding

A key concern with JSON encoders is preventing arbitrary JavaScript remote code execution within the browser... or, if you're using `node.js`, on the server. It's vital that you use a proper JSON serializer to encode user-supplied data properly to prevent the execution of user-supplied input on the browser.

When inserting values into the browser DOM, strongly consider using `.value/.innerText/.textContent` rather than `.innerHTML` updates, as this protects against simple DOM XSS attacks.

XML encoding

XML should never be built by string concatenation. It should always be constructed using an XML serializer. This ensures that the XML content sent to the browser is parseable and does not contain XML injection. For more information, please see the [Web Service Security Cheat Sheet](#).

Cryptography

Data in transit

Unless the public information is completely read-only, the use of TLS should be mandated, particularly where credentials, updates, deletions, and any value transactions are performed. The overhead of TLS is negligible on modern hardware, with a minor latency increase that is more than compensated by safety for the end user.

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

Data in storage

Leading practices are recommended as per any web application when it comes to correctly handling stored sensitive or regulated data. For more information, please see [OWASP Top 10 2010 - A7 Insecure Cryptographic Storage](#).

Message Integrity

In addition to HTTPS/TLS, JSON Web Token (JWT) is an open standard ([RFC 7519](#)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. JWT can not only be used to ensure the message integrity but also authentication of both message sender/receiver. The JWT includes the digital signature hash value of the message body to ensure the message integrity during the transmission.

<https://jwt.io/introduction/>

HTTP Return Code

HTTP defines status code [\[1\]](#). When design REST API, don't just use 200 for success or 404 for error.

Here are some guideline to consider for each REST API status return code. Proper error handle may help to validate the incoming requests and better identify the potential security risks.

- 200 OK - Response to a successful REST API action. The HTTP method can be GET, POST, PUT, PATCH or DELETE.
- 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
- 405 Method Not Allowed - The error checking for unexpected HTTP method. For example, the RestAPI is expecting HTTP GET, but HTTP PUT is used.
- 429 Too Many Requests - The error is used when there may be DOS attack detected or the request is rejected due to rate limiting

Related articles

OWASP Cheat Sheets Project Homepage

- [OWASP Cheat Sheet Series](#)

Developer / Builder

Assessment / Breaker

Mobile

- [IOS Developer](#)
- [Mobile Jailbreaking](#)

OpSec / Defender

- [Virtual Patching](#)

Draft and Beta

[All Pages In This Category](#)

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

Other cheatsheets

V - T - [E Cheat Sheets](#)

Developer / Builder

Assessment / Breaker

Mobile

- [IOS Developer](#)
- [Mobile Jailbreaking](#)

OpSec / Defender

- [Virtual Patching](#)

Draft and Beta

[All Pages In This Category](#)
