REST Assessment Cheat Sheet

About RESTful Web Services

Web Services are an implementation of web technology used for machine to machine communication. As such they are used for Inter application communication, Web 2.0 and Mashups and by desktop and mobile applications to call a server.

RESTful web services (often called simply REST) are a light weight variant of Web Services based on the RESTful design pattern. In practice RESTful web services utilizes HTTP requests that are similar to regular HTTP calls in contrast with other Web Services technologies such as SOAP which utilizes a complex protocol.

Key relevant properties of RESTful web services

- Use of HTTP methods (`GET` , `POST` , `PUT` and `DELETE`) as the primary verb for the requested operation.
- Non-standard parameters specifications:
  - As part of the URL.
  - In headers.
- Structured parameters and responses using JSON or XML in a parameter values, request body or response body. Those are required to communicate machine useful information.
- Custom authentication and session management, often utilizing custom security tokens: this is needed as machine to machine communication does not allow for login sequences.
- Lack of formal documentation. A proposed standard for describing RESTful web services called WADL was submitted by Sun Microsystems but was never officially adapted.

The challenge of security testing RESTful web services

- Inspecting the application does not reveal the attack surface, I.e. the URLs and parameter structure used by the RESTful web service. The reasons are:
  - No application utilizes all the available functions and parameters exposed by the service
  - Those used are often activated dynamically by client side code and not as links in pages.

    - The client application is often not a web application and does not allow inspection of the activating link or even relevant code.
- The parameters are non-standard making it hard to determine what is just part of the URL or a constant header and what is a parameter worth fuzzing.
- As a machine interface the number of parameters used can be very large, for example a JSON structure may include dozens of parameters. fuzzing each one significantly lengthen the time required for testing.
- Custom authentication mechanisms require reverse engineering and make popular tools not useful as they cannot track a login session.

How to pentest a RESTful web service

Determine the attack surface through documentation - RESTful pen testing might be better off if some level of clear-box testing is allowed and you can get information about the service.

This information will ensure fuller coverage of the attack surface. Such information to look for:

- Formal service description - While for other types of web services such as SOAP a formal description, usually in WSDL is often available, this is seldom the case for REST. That said, either WSDL 2.0 or WADL can describe REST and are sometimes used.
- A developer guide for using the service may be less detailed but will commonly be found, and might even be considered *opaque-box* testing.
- Application source or configuration - in many frameworks, including dotNet ,the REST service definition might be easily obtained from configuration files rather than from code.

Collect full requests using a proxy - while always an important pen testing step, this is more important for REST based applications as the application UI may not give clues on the actual attack surface.

Note that the proxy must be able to collect full requests and not just URLs as REST services utilize more than just GET parameters.

Analyze collected requests to determine the attack surface:

- Look for non-standard parameters:
    - Look for abnormal HTTP headers - those would many times be header based parameters.
    - Determine if a URL segment has a repeating pattern across URLs. Such patterns can include a date, a number or an ID like string and indicate that the URL segment is a URL embedded parameter.

    - For example: `http://server/srv/2013-10-21/use.php`
- Look for structured parameter values - those may be JSON, XML or a non-standard structure.
- If the last element of a URL does not have an extension, it may be a parameter. This is especially true if the application technology normally uses extensions or if a previous segment does have an extension.
    - For example: `http://server/svc/Grid.asmx/GetRelatedListItems`
- Look for highly varying URL segments - a single URL segment that has many values may be parameter and not a physical directory.
    - For example if the URL `http://server/src/XXXX/page` repeats with hundreds of value for `XXXX`, chances `XXXX` is a parameter.

Verify non-standard parameters: in some cases (but not all), setting the value of a URL segment suspected of being a parameter to a value expected to be invalid can help determine if it is a path elements of a parameter. If a path element, the web server will return a *404* message, while for an invalid value to a parameter the answer would be an application level message as the value is legal at the web server level.

Analyzing collected requests to optimize fuzzing - after identifying potential parameters to fuzz, analyze the collected values for each to determine:

- Valid vs. invalid values, so that fuzzing can focus on marginal invalid values.
    - For example sending *0* for a value found to be always a positive integer.
- Sequences allowing to fuzz beyond the range presumably allocated to the current user.

Lastly, when fuzzing, don't forget to emulate the authentication mechanism used.

Related Resources

- REST Security Cheat Sheet - the other side of this cheat sheet
- YouTube: RESTful services, web security blind spot - a video presentation elaborating on most of the topics on this cheat sheet.