REST SECURE

ADDRESS REST SECURITY ISSUES.

kiuwan

MANY APPLICATIONS PROVIDE A SERVICES LAYER (TO OTHER APPLICATIONS, TO A PRESENTATION LAYER…) OR CONSUME SERVICES EXPOSED BY THIRD-PARTIES (NOT NECESSARILY TRUSTED).

REST MODEL IS A SIMPLE WAY FOR DESIGNING SUCH SERVICE LAYERS, WIDELY USED TODAY.

This document is about **REST security issues** and presents the main security problems that need attention, the attack threats and attack surface for REST, and how to handle some common security issues.

We will center our discussion around Java-based REST frameworks, but many concepts are also valid for other implementation languages, like PHP, .Net, JavaScript, Python, Ruby…

THINK REST...

REST (REpresentational State Transfer) is an architectural style to build services on top of the Web. REST simplifies interaction with a web-based system via simplified URLs, instead of complex HTTP requests.

A REST-enabled application is made of resources ("*nouns*"), each responding to different "*verbs*" (for example, HTTP methods like GET, POST, PUT, DELETE...) over well-defined URLs (typically HTTP/HTTPS URLs) which map entities in the system to endpoints. Each REST request/response to a resource uses one or more resource representation, typically negotiated between client and server (for example, using Accept and Content-Type HTTP headers). Most common representation formats include JSON, XML or raw text, with many other alternatives like YAML, Atom and HTML which are also used.

Here are the **main characteristics of a REST architecture style**:

- *Client-Server*: a pull-based interaction style: consuming components pull representations.
- *Stateless*: each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- *Cache aware*: to improve network efficiency, responses must be capable of being labeled as cacheable or non-cacheable.
- *Uniform interface*: all resources are accessed with a generic interface (e.g., HTTP GET, POST, PUT, DELETE).
- *Named resources* – the system is comprised of resources which are named using a URL.

- *Interconnected resource representations* – the representations of the resources state are interconnected using URIs, thereby enabling a client to progress from one state to another.
- *Layered components* – intermediaries, such as proxy servers, cache servers, gateways, etc., can be inserted between clients and resources to support performance, security, etc.
- In the Java/J2EE platform, a standard Java API for RESTful Web Services (JAX-RS, JSR 311) is the most common API for building and consuming REST services. There are many widely used implementations, like Jersey, Apache CXF, Restlet, or RESTEasy.
- Any services architecture is inherently “recursive”: services could aggregate information resources fetched from other services. A vulnerability in a service not only is due to implementation defects committed by service developers: the REST framework itself has much to tell us.

SECURITY CONCERNS WITH REST

REST typically uses HTTP as its underlying protocol, which brings forth the usual set of security concerns:

- A potential attacker has full control over each single bit of an HTTP request (if your app acts as REST server) or HTTP response (if your app is a REST client). We remind this just in case...
- The attacker could be at the REST client (the victim is the REST server) or at the REST server (a rogue, malicious app, the victim is the REST client, for example an application consuming resources from remote REST services).
- For an application using REST as client or server, the other side typically has full control on the resource representation, and *could inject any payload to attack resource handling* (gaining arbitrary Java code or system command execution, for example).

In a REST architecture, end-to-end processing implies a sequence of potentially vulnerable operations:

- During the mapping from/to the HTTP message and resource URL.
- When the object representing the target resource is instantiated and the requested operation is invoked.
- When producing/parsing state representation for the target resources.

- When accessing/modifying the data in the backend systems that host the resource state.

The layered sequence of transformations in REST frameworks means that a single weak link in the chain could made your application vulnerable.

UNTRUSTED INPUTS MAY CHANGE THE INTENDED REST SEMANTICS

Extracting parameters from HTTP message and getting resource URLs could be vulnerable to injection attacks that may change the semantics of the intended resource. Two classes of attacks are relevant here: *HTTP parameter/path pollution* (HPPP) and *Server-Side Request Forgery* (SSRF). Remember that **our attacker has full control over the HTTP request or the HTTP response**.

In an HPPP (*HTTP parameter/path pollution* attack), a parameter is used to compose the resource URL to be used to prepare a REST request for a resource (or generate an embedded link). The problem is that the attacker may either alter the path or add/overwrite unexpected parameters in the "query string".

Additionally, REST frameworks may use a parameter (like _method) to allow the specification of a REST verb different from the incoming HTTP method, so a GET request could be interpreted as a PUT operation. **An attacker may change the semantics of the REST resource URL**!

For example, with the following code:

```
String entity = request.getParameter("entity");
String id = request.getParameter("id");
URL urlGet = new URL("http://myserver.com/rest/" + entity + "?id=" + id);
// perform a REST query using urlGet
```

An attacker may provide the following parameters:

```
entity=../admin/users/user/badGuy
id=x&_method=PUT&isAdmin=true
```

So the operation that the application executes is like this:

```
GET url http://myserver.com/rest/../admin/users/user/badGuy?id=x&_method=PUT&isAdmin=true
```

Which is probably interpreted by myserver.com REST endpoint as this:

```
PUT /admin/users/user/badGuy?id=x&isAdmin=true
```

In a *Server-Side Request Forgery* (SSRF), the vulnerable application composes the URL using data from the HTTP request that could also affect the scheme, host and port parts of the URL, so the vulnerable application acts as an *open proxy/relay* for the attacker. Remember, **the wish for an attacker is to gain access to an interface that allows access to internal resources**.

Remember that browsers impose at least a *Same-Origin Policy* (SOP) to control cross-site reads, but HTTP clients (and REST stubs/proxies) do not enforce any kind of SOP. This is your business.

Of course, if your app is so naive as to use untrusted input directly as the prefix part of a resource URL to be retrieved, SSRF is open. But there are other enablers for SSRF. Some REST frameworks provide proxies (RESTlet Redirector) for server-side redirections that may use an input URL directly. XML External Entity vulnerabilities (more on this later) may force unintended connections to internal URLs. And when the attacker has control over the target URL, you may hear really loud laughters across your office window!

NOTE: SSRF is also the acronym for Super Simple Rest Framework

RESOURCE STATE REPRESENTATION IS UNDER ATTACK

The logical entity representing the current state is mapped to its representation (*marshalling* or *unmarshalling*), which often means converting between a Java object (e.g. a POJO) and the representation text (like XML or JSON). This marshalling/unmarshalling could be the weak link in the chain. For example, under XML representation, either an XML parser, or a higher-level XML marshalling framework (like schema-free XMLEncoder/XMLDecoder or XStream or schema-based JAXB) is used.

a) *XML representations*. There are at least three kind of potential issues with XML in Java REST frameworks here:

– *XML entity injection attacks*, either an *XML entity expansion* denial-of-service ("billion laughs" attack) or as *XML external entity injection* (XXE attacks). The problem is that common Java XML parsers use an insecure default configuration for these threats, and the REST framework may not be performing adequate configuration of the underlying parser. This opens the playground for bad consequences like denial-of-service, code injection, sensitive file disclosure, internal network port scanning...

Well known examples of such XML-based attack vectors are:

"Billion laughs" attack, 1K payload expands to 3GB, leading to directed denial-of-service:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

XXE (external entity attack), for sensitive file disclosure (WEB-INF configuration files, /etc/passwd, etc.):

```
<!DOCTYPE root [
 <!ELEMENT root ANY >
 <!ENTITY windowsfile SYSTEM "file:///c:/boot.ini">
]>
<root>
 <sometag>&windowsfile;</sometag>
</root>
```

XXE, for denial-of-service under Unix servers:

```
<!DOCTYPE foo [
  <!ELEMENT root ANY >
  <!ENTITY unixfile SYSTEM "file:///dev/random" >
]>
<root>&unixfile;</root>
```

Note: The JAXP API for XML parsers in Java recently adopted a more robust default configuration.

– If the REST framework (or REST application) uses a generic Java-XML marshalling API, like JDK XMLEncoder/XMLDecoder or XStream, that need to include method and constructor invocations during XML unmarshalling to Java objects, a potential Java *code injection* could be leveraged by the attacker.

The following xml attack payload for XMLDecoder will overwrite a sensitive file. Similar to `new PrintWriter("/myapp/WEB-INF/sensitive.dat").println("hacked!")`. The bad guys could install a JSPShell page, modify configuration files, deface web contents, etc.

```
<java>
  <object class="java.io.PrintWriter">
    <string>/myapp/WEB-INF/sensitive.dat</string>
    <void method="println">
      <string>hacked!</string>
    </void>
  </object>
</java>
```

The good news is that newer versions of some Java REST frameworks have patched this, but often a specific configuration on XML parser and XML marshalling components should be performed by the REST application. For example, Jersey fixed this bug ("*All Jersey web services that accept XML are vulnerable to XXE attacks*", bug-id JERSEY-323) only recently (September 2015)!

Be cautious. For example, RESTlet uses XMLEncoder/XMLDecoder for its ObjectRepresentation, when the media type chosen was 'application/x-java-serialized-object+xml'. This was deactivated by default, but insane developers may ignore the security warning and activate it.

b) *JSON representations*. Remember, the JSON payload to be used by our REST application is under control of the bad guys. REST frameworks provide extensions that allow more functionality at the expense of security. For example, passing partial script blocks or JavaScript functions that are executed, passing a complete script (e.g. Groovy code in the Gremlin plugin for NoSQL Neo4j database REST API), or passing source and target URLs for data replication.

DO NOT MISS CONTROLS AGAINST WELL-KNOWN ATTACKS

You need specific controls against well-known attacks, such as *Cross-Site Request Forgery* (CSRF). Your REST endpoints should enforce explicit controls for resource state-changing operations. Due to the stateless nature of REST, this could need a slight departure from conventional session-based anti-CSRF schemes, like *synchonizer token pattern*.

For protection against CSRF, a check on the mere existence of a particular HTTP header, like *X-CSRF* or *X-Requested-By*, for state-changing verbs (anything except GET, OPTIONS or HEAD) could be done at REST endpoint (for example, in an HTTP filter) to make sure that no malicious content loaded from a browser with valid session could emit unexpected requests to the target service, on behalf of a previously authenticated, trusted user. Jersey framework, for example, use a *CsrfProtectionFilter* for this. Remember that **any cross-site scripting (XSS) vulnerability in the target application could allow injection of JavaScript code** that creates XMLHttpRequest (XHR) with the anti-CSRF header, and the browser SOP will allow that (because JavaScript has same origin as the request target).

Authentication and session-management data (API keys, username/password, session tokens, etc.) should not appear in the URL, as such sensitive info could be leaked in web server logs, intermediate proxies, etc. Authentication with client-side (and server-side, of course) SSL certificates could be useful here.

As RESTful (stateless) web services often need to maintain a transactional state, an state blob is typically exchanged between client and server. **Your API should be designed to avoid *replay attacks* with the state blob**. A session token or API key could be used to keep all state information at server side, in a similar way as conventional web applications use a session cookie + server-side session data.

Another general recommendation with REST is to ***avoid encoding sensitive information in the URL*** (REST principles always tell you to avoid using GET for state-changing operations). As URLs could be logged or cached, this may result in information leakage.

DISCLOSURE OF TOO MANY DETAILS ABOUT REST API MAY HELP ATTACKERS

REST documentation could provide too much details to potential attackers. Many REST frameworks disclose full information about the REST API, either through exposed pages (without authentication), using HTML pages or WADL documents. Probably an active validation of the exposed documentation should be done, as many frameworks expose API documentation “automagically”.

Although providing detailed documentation for the REST API could be useful for API consumers, it also provides valuable information for the bad guys.

WITH GREAT POWER COMES GREAT RESPONSIBILITY

So, what could you do to avoid security issues with your REST API?

- Before choosing a REST framework, **double-check all the potential security threats** that could afflict your REST applications.
- Prepare a "**REST security guide**" with full details on:
  - How to properly configure the chosen framework.
  - How to deactivate unintended functionality (for example, blocking representation formats that should not be exposed).
  - How to handle (remediate or mitigate) each specific attack.
  - Access controls between client roles and each resource group REST verbs (access matrix may be OK).
  - Design security tests that make sure that none of the attacks seen does work.
  - How to expose REST API documentation to the world.
- If you consume REST resources from untrusted applications, **validate carefully the results** to avoid injection attacks.
- **Carefully validate the inputs**, for public REST APIs they should be considered untrusted.
- If you need to prepare URLs for REST resources, consider HPPP and SSRF attacks, and how the URL building logic handles untrusted inputs used as parts of the target URL. **Output encoding**, like URL encoding, if done properly, could close some doors.

- Do not forget the “classical” things: **XSS** (if REST contents are inserted into web pages to be consumed by user browsers), **CSRF, SQL injection**, and all of that. REST framework provides the envelope, the inside letter is also your business.

For better control during URL construction, **do not use concatenation with parts coming from untrusted inputs**. A useful facility are the “URI-builders” that work like the PreparedStatement for SQL injection, for example the *javax.ws.rs.core.UriBuilder*, or the Apache Commons HttpComponents *org.apache.http.client.utils.URIBuilder*. See URLs and Links for details on how to use UrlBuilder for building safer URLs, that automatically encode the dangerous characters to avoid HPPP attacks.

Of course, input “whitelist” validation in untrusted inputs, plus validation that the target composed URL really points to the intended resource, are also recommended neutralizations here.

If you would like to test if your applications are susceptible to these types of attacks, we encourage you see a demo or request a tailored trial of Kiuwan. We constantly update the rules for these and many more in compliance with OWASP, Nist, CWE and many others.

TRY KIUWAN FOR FREE

KIUWAN.COM/FREE

CONTACT US

CONTACT@KIUWAN.COM
+1 9045123050
LIVE CHAT: KIUWAN.COM

BECOME A PARTNER

PARTNERS@KIUWAN.COM

kiuwan