Journal of Mobile, Embedded and Distributed Systems, vol. VII, no. 2, 2015
ISSN 2067 – 4074

Securing a REST Web Service

Cristina-Elena POPA
IT&C Security Master
Department of Economic Informatics and Cybernetics
The Bucharest University of Economic Studies
cristina.popa89@gmail.com

Abstract: The aim of this paper is to present the key security requirements for Web Services. A set of security principles will be presented, as well as a study on how they can be implemented in order to ensure the service is available and non-compromised at any given time. Conclusions will be formulated at the end of this case study, based on the obtained results.

Key-Words: Web Service, REST, Security, Threat, Vulnerability, Risk, Attack

1. Introduction

A Web Service can be defined in a very generic manner as an application that is available to other applications through the internet. This means that the heterogeneity of an information system will be reduced, since applications that were not originally developed to cooperate can now exchange messages.
A more specific definition is the one provided by the World Wide Web Consortium (W3C), the international community that develops web standards: "A Web Service is a software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet-based protocols". [1]
A Web Service can be implemented using several architecture styles, the most common ones being SOAP, REST and RPC.
Representational state transfer (REST) is a software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system. [2]
For web services, a REST service usually implies answering to specific HTML requests mapped to URLs, using a specific data type. The following design principles must be taken into consideration when implementing a REST service: be stateless, use HTTP methods explicitly, transfer XML, JavaScript Object Notation (JSON) or both and expose Uniform Resource Identifiers (URIs) that have a directory-like structure.

2. Web Service Vulnerabilities

Security is crucial for any distributed computing environment and it is very important to understand the key principles for developing secure Web Services.
There are a few concepts that need to be clarified regarding the security field:

- Asset;
- Vulnerability;
- Threat;
- Attack.

An asset refers to people, property or information. The role of security is protecting the assets against threats.
A vulnerability is a weakness in the protection plan.
An attack is any attempt to destroy, expose, alter, disable, steal or gain unauthorized access to or make unauthorized use of an asset [3].
These concepts are reunited in the definition of risk: "the potential that a given threat will exploit vulnerabilities of an asset or a group of assets and thereby cause harm to the organization" [4].
When designing a network one must protect the information through risk management and the development of effective countermeasures for the existing threats. In this chapter we will present a

set of relevant security principles and threats that need to be treated.

Authentication and Authorization

Authentication is the process of determining the identity of an entity, usually using credentials such as a username and password. Authorization represents the process by which an entity is allowed to perform a requested action (for example to access a piece of information).

In a client-server architecture, the two processes are coupled, so that the server can determine the identity of a client and what it is authorized to do. Authorization occurs after successful authentication.

Session Management

A session is defined as "Stateful connection between two parties during which one or more communications take place" [5].

The session management must take into consideration that more than one session can be handled on the server side and each of them must keep the state information.

The session should also expire after a certain amount of time passes since the user authenticated himself.

Auditing and Logging

Security events need to be tracked within the application.

Logging records the important events, thus offering a trail of evidence in the case of an eventual attack.

Auditing is more focused on who did what, from a user management point of view.

Other points of view can be discussed in relation to this topic. For example, where are the log files written or who can access them. For added security, log files can be written to a write-only file (or device) and backed up.

Denial of Service

As its name suggests, the denial of service is an attack intended to make a resource (in our case a web service) unavailable to its intended users. The service needs to tell the difference between normal and malicious traffic (e.g. an attack or a large number of users connecting to the service).

Load testing tools exist that can generate traffic in order to test how will the service perform under heavy load (for example a large number of requests coming from a single IP address).

Exception Management

Exceptions represent out of the ordinary conditions that require special processing and often change the execution flow of a program.

Exception handling or management is the process of responding to the occurrence of such cases (usually with a special subroutine). If an exception is not handled in a program, it will result in an error.

Message Validation

It is important that the service validates the format of both inbound and outbound traffic. Input or output can be rejected if it does not pass the validation.

One common example of when input validation can be used for a security purpose is to prevent SQL injection.

Depending on the system that is being analyzed, message validation can be done using different techniques (escaping characters, filtering).

In the case of Web Services, the input can be verified using a validation schema.

The WSDL (Web Services Description Language) contains the set of rules used to define the interaction with the Web Service.

Message Encryption

This can refer to the protection of a message content by using an encryption algorithm to transform it from plaintext (the original message) to ciphertext (this needs to be decrypted in order to be read). Only the confidentiality of the message is protected and not its integrity or authenticity.

Another way of using encryption is to protect data that transits a network.

3. Security Solutions

In this section solutions will be proposed for each of the principles presented in the previous section.
We will base the discussion on the security solutions implemented for the Kippit Secure Digital Content Web Service [6].

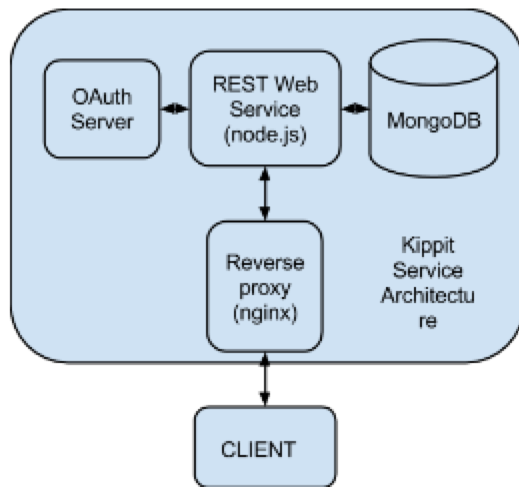

Figure 1. Kippit Service Architecture

As seen in Figure 1, the Kippit Service consists of a series of components. First of all, it includes a reverse proxy which dispatches all requests to the actual web service.
A reverse proxy, by definition, is a type of proxy server that fetches resources on behalf of a client from a server. Subsequently, the resources are returned to the client as though they originated from the proxy server itself. In Kippit's case, nginx, the reverse proxy, acts as an intermediary for the REST service to be contacted by any client.
Due to its intermediate role, it plays a crucial role in securing the service, as we will further detail.
Another component of the service, responsible for Authentication and Authorization is the OAuth Component. This ensures that user roles and session management are properly in place.
Lastly, most of our security solutions will be implemented at the service level: in this case, the web service is implemented using the Express node.js technology.
Securing the database is out of this paper's scope, as these techniques are not service specific.
Moving further, we will propose a solution for every problem highlighted in the previous chapter, implemented in one of the components listed above.

Authentication and Authorization
The service architecture currently implements an OAuth 2.0 stub for authentication and authorization. OAuth is a protocol for secure authentication, authorization and session management. In practice, it allows client tokens to be issued for third-party clients, after receiving the client approval.
For instance, in our case, the OAuth receives the username and password of a client and returns a client token in exchange. The returned client token is valid for any re-quest to the Kippit REST service made by the corresponding user in a given time frame.
The OAuth stub is currently accessible as part of the service, using the following end-points:

```
POST
https://kippit.net/rest/v0/register
POST
https://kippit.net//rest/v0/login
POST
https://kippit.net//rest/v0/logout
```

After a client token is obtained, any subsequent request must include it in a 'Authori-zation' HTTP header, allowing both authorization and session management.

Session Management
Session management is currently implemented at the OAuth level. With every token request, the OAuth component issues it so that it expires in a certain amount of time. After this period passes, the client token will be no longer valid, thus assuring the session feature.

Auditing and Logging
For auditing purposes, all requests and their corresponding results are logged. At re-quest creation time, the server randomly generates a request id, which

will uniquely identify all log messages for the corresponding REST call.
For example, if an user will call POST https://kippit.net//rest/v0/comment, the logs will look as follows:

[Mon May 04 2015 20:39:32 GMT+0300][27539586590383][Warning] Cache empty for user john123456

Notice how the request id is present in the logs, ensuring fast and reliable auditing and customer support capabilities.
Lastly, a security concern relates to logging in general. Due to the sensitive nature of the content, multiple fields, such as credit card numbers or passwords are not logged, thus minimizing a social engineering attack impact.

Denial of Service

Denial of Service attacks imply a large number of request in a short amount of time, thus making the server unavailable.
A solution for avoiding this situation is limiting the number of requests coming from an IP. This procedure is called throttling.
Kippit currently enables throttling in its nginx configuration file, disallowing more than 1 request per second, coming from the same IP, with at most 1 extra request being buffered and a maximum of 10MB of request data before being discarded.

```
http {
    limit_req_zone $binary_remote_addr
zone=kippit:10m rate=1r/s;
    ...
    server {
        ...
        location /rest/ {
            limit_req zone=one burst=1;
        }
}
```

If, for example, a script would send 50 consecutive POST requests to the service, the ones that exceed the 1 request per second limit will receive as a response HTTP error 503 (“Service unavailable”)[7].
This also helps fight brute force attacks (the simplest attack type, it tries a large number of usernames and passwords in order to gain access).

Exception Management

With Kippit, all requests are handled in a try-catch block, as follows:

```
app.use(function(req, res, next) {
trycatch(function() {
app.router(req, res, next);
}, function(er) {
console.log(er.message);
res.send(500);
});
});
```

This ensures that all requests are handled in the try catch block, whether they are new REST methods to be implemented, or already existing methods. Another advantage of this approach is the correct invisible output: no stack trace will be present on the client side, just a single 500 HTTP error.

Message Validation

All REST endpoints accept a strict set of parameters (either XML or JSON) defined in a XSD file. Message validation is currently implemented as follows: each request has a JSON format which defines how the request should look like. Then using, this format and an npm module, validation is performed.

```
// Require Sys and FileSystem
var sys = require('sys'), fs =
require('fs');

// Require package
var validate = require('commonjs-
utils/json-schema').validate;

createPostSchema =
fs.readFileSync(createPost.json');
app.post('/post', function(req,res) {
var validation = validate(req.body,
createPostSchema);


if (!validation.valid)
      throw new Exception('Invalid
input for creating a post!');
      postApi.post(req.body);
});
```

Message Encryption

HTTP as a protocol does not include encryption by default. This implied the development of a secured version of HTTP called HTTPS, which uses SSL (and later

TLS) for encrypting the client-server channel.
Kippit uses nginx for securing client connections over https. Moreover, all HTTP content is redirected to HTTPS.
The following configuration depicts the encryption solution for Kippit's nginx:

```
server {
listen 443 ssl default;
server_name kippit.net;
 ssl_certificate
/etc/nginx/ssl/kippit.crt;
 ssl_certificate_key
/etc/nginx/ssl/kippit.key;
}

# redirect all http traffic to
https
server {
 listen 80;
 server_name kippit.net;
 return 301
https://$host$request_uri;
```

As an observation, port 80 is used for HTTP traffic and 443 for HTTPS traffic.

4. Conclusions

In this article, we have briefly described the concepts of security for web services. We have shown best practices and methods of avoiding common mistakes which lead to vulnerabilities, as well as an example service that correctly handles all of these situations.
Moreover, we have managed to showcase a real world example of SOA hardening using state of the art technologies and a clear security role separation, as each component of the service handles it's appropriate possible vulnerabilities.
It is important to understand the vulnerabilities of a system before trying to secure it. In most cases these are strongly related to the components of the system and the relationship between them.

Acknowledgement

Parts of this paper were presented at The 8th International Conference on Security for Information Technology and Communications (SECITC 2015), Bucharest, Romania, 11-12 June 2015.

References

[1] Web Services Description Requirements - http://www.w3.org/TR/ws-desc-reqs/
[2] REST- http://en.wikipedia.org/wiki/Representational_state_transfer
[3] ISO/IEC 27000:2009 from ISO, via their ITTF web site
[4] "Information technology -- Security techniques-Information security risk management" ISO/IEC FIDIS 27005:2008
[5] Mario C. Jeckle, *Extending SOAP to Adhere to Session-oriented Communication Principles*, 2002
[6] https://kippit.net
[7] Hypertext Transfer Protocol -- HTTP/1.1 - http://www.w3.org/Protocols