UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Security Analysis of a web application

D&I Cofano Project
Team 14

Robert Banu
Peter Knot
Mahedi Mridul
Jesse van der Velden

29 June 2018

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Preface

This small security analysis report describes the different security aspects of the project. This document will describe the general expected behaviour of the implementation and describe which security risks may exists and how to mitigate them. The entire source code to check on your own can be found on the Gitlab page: https://git.snt.utwente.nl/module4team14/DataAndInformationProject

This is the security analysis report of the data & information project from the fourth module of the bachelor TCS and BIT at the University of Twente.

Contents

1 Login

The first thing to discuss is the login page of the web application. It is a crucial part of the designed application. For logging in Cofano wants to use their Google Business accounts. To support this feature the application makes use of the *Google OAuth Client Library for Java* which documentation can be found at: https://developers.google.com/api-client-library/java/google-oauth-java-client/oauth2. The choice was made to use this library, to not bother with the Oauth 2 spec. Therefore there is a certain trust needed into Google to implement a right Oauth 2 implementation. It is also important to implement the library itself correctly on the application.

1.1 Expected behaviour and implementation

When a user clicks on the sign in button he should be redirected to the Google Oauth website to login to their Google (Business) account. While configuring the Oauth library of Google, there was chosen to set the host domain for the email accounts to Cofano only. To test for this (while there is no access to a Cofano email), there is chosen to set this to **.utwente.nl* temporarily. This makes sure Google only shows login buttons (on their website) for accounts with that specific host domain. When Google makes a callback to the application server it also sends some more parameters. This can be used to make sure the targeted host domain is the right one, and provides the application with some handy details about the user, such as the Google ID, name and email information.

When our application gets the callback from Google and has made sure the targeted domain is right, a Java Session is created. This will send a cookie as a header to the client response, which stores the user information. In the same response the user is redirected to the protected dashboard page which can only be accessed by logged in users.
When a user directly goes to one of the dashboard URLs, while no or an invalid session is provided to the request, it is being redirected straight to the login page before loading the actual content of the dashboard page.

1.2 Possible attacks

While there was made sure there is no implementation flaw of the library at the application side carefully following the guide from Google, it still can be attacked if there is a fault in the library itself.

On the client side, as the official *farm* website doesn't implement TLS, a man in the middle can steal cookies. Furthermore does the user have the responsibility to check if the Google login page is genuine. It can be checked by checking the domain of the Google login page. However, it could theoretically also happen that even if the Google domain is right, a fake (but trusted) certificate is served, hopefully that will soon come to an end with DANE and DNSSEC.

There is actually not yet an application database involved in the login process, so no possible SQL injections can be executed.

1.3 Attack mitigation

On the server side it is needed to implement proper HTTPS TLS to prevent cookie stealing.
As the Google Oauth 2 library is provided by Google itself, making sure you always have the latest recommended version is a good practice to minimize the risks of insecure libraries. Also Google has an economic effort to make sure the library is secure.
To mitigate accidental logins by request forgery, an added secure random token with the requests is provided.
To prevent XSS attacks, the information of Google is also XML serialized by using the JSP helper method: *fn:escapeXml()*. However still the argument holds, that Google has an economic effort to make sure actually no HTML in the user data is being sent.

2 Display of tables

Because the application makes frequent use of displaying tables out of the database, it is an important thing to look into. All of the tables which display information out of the database are using the JavaScript library: *Datatables*, which documentation can be found on: https://datatables.net/manual/

2.1 Expected behaviour and implementation

The information out of the database is being retrieved by a *GET* request to the Jersey API part of the application. Because all of the information can be loaded relatively quickly and because of simplicity, it has been decided to load the entire table in one request.

2.2 Possible attacks

As the table is loaded with one simple *GET* request, it is possible to have simple a *SELECT* SQL query without input from the client. So there can be no SQL injections.

When data out of the database is being loaded by the javascript library, it could also render HTML into the table when HTML is in a SQL table column. This could lead to a (stored) XSS attack which can steal and send cookie information to an attack server.

2.3 Attack Mitigation

For displaying tables, the only thing that could go wrong is serving stored XSS attacks. As the JavaScript loads JSON through a GET request, there has been chosen to prevent this also on the client side. The JavaScript will parse all HTML code and convert it to its text equivalent which will not be interpreted as HTML code in a browser.

3 Adding, editing or deleting a table entry

Adding, editing or deleting a table entry is also a feature of the application. Because it handles user inputted data, it must handle the data securely as there are many opportunities for attacks.

3.1 Expected behaviour and implementation

It is possible to retrieve a specific table entry by going to http://server/Cofano-C/*table*/*ID*. Where *table* is the table you want to edit, and *ID* the unique identifier of the row out of the table. On this page, a JavaScript will be loaded that will call the jersey API with a GET request with the specific *ID* from the URL. When it exists it will load in the form with the data from the database. Otherwise it will show an error.
When a user wants to save data, a JSON payload body is send to the server in a PUT or POST request, depending on whether the user is editing or adding a row in a table. On success the user is being redirected to the table overview page. A DELETE request can be used to delete a specific entry.

3.2 Possible Attacks

First, there is the risk of SQL injection attacks. When a request is being made to a specific API, to request details about a table entry, in the specific URL, the *ID* is also in the GET request. When an application doesn't make use of the prepared statements/ stored procedures, there can be a risk of an SQL injection.
Next, there can be content served from the url, directly to the webpage. For example, while editing the application shows: 'Table edit: ID'. If an attacker would lead a victim to a custom prepared page, he could serve arbitrary HTML with a possibility to steal cookies. This is called a reflected XSS.
Furthermore, there can be stored XSS attacks inside the database.
Lastly there can still be CSRF attacks. It is not possible to change anything in a table with a GET request. But POST/ UPDATE/ DELETE requests could be vulnerable.

3.3 Attack Mitigation

To mitigate the possibilities of an SQL attack, there has been chosen to implement stored procedures almost everywhere to select, update, insert, or delete things from the database. For the other cases there is a usage of prepared statements.
To prevent reflected XSS, HTML tags are already filtered on the server side using *Apache's StringEscapeUtils* and also one more time in JSP. The Javascript will parse all of specific HTML code retrieved from the API, and convert it to its text equivalent which will not be interpreted as HTML code in a browser.
CSRF attacks can be mitigated by providing a secure random token on POST requests. As the application only supports a JSON payload it is not easily attackable when a user uses a modern browser, as it isn't possible with HTML code to set the content type to JSON only. Doing a CSRF attack with JavaScript is also not possible with a modern browser as the default CORS implementation prevents requests from a different website. However as browser specs can change, or because a user could use a very arbitrary browser which doesn't prevent CORS attacks by default; it is a good practice to also prevent CORS attacks on the server side. This can be done by providing a secure random token such as with the login page. Luckily Jersey already provides a basic CSRF implementation with request headers which the application uses. This is documented at: *https://jersey.github.io/apidocs/latest/jersey/org/glassfish/jersey/server/filter/CsrfProtectionFilter.html*

4 Malicious

Finally, there are some other security concerns which need to be discussed.

4.1 API Security

The API implemented with Jersey is protected by either an authorization header (for usage of other applications), or by looking into the session to check whether a user is logged in.
Currently (while writing this security analysis) it is not yet possible to generate a secure random API token. This will be implemented for the final sprint. Users could make an API token with a very simple hash/body which would undermine the entire security of the application. This is because authorized users and API-tokens could edit/ create/ delete anything as there are no different roles. Also proper backups should be made to prevent one user/ application to delete all the records. However, this isn't the scope of this project and should be done by Cofano.

4.2 Risks of the usage of external libraries

The application uses multiple different client- and server side libraries. There should be a certain trust in those libraries to make sure that they are not insecure. Most of the libraries are open source and/ or are developed by bigger companies. This doesn't mean they are secure by default, but this can provide some more trust for the usage of these libraries. Someone who would maintain the application should stay up to date with the latest news from these libraries, to make sure it doesn't use an insecure version.

Javascript and CSS files could also be loaded by CDN's to speed up the loading of these libraries as the user could already have them cached. This adds an extra trust in those CDN providers. To check libraries for their authenticity it is possible to make a checksum of the script/ css file and providing that also in the HTML tag. It needs browser support as well.
For the application there has been chosen to not do this as it needs a working internet connection and because those CDN's could track visitors of the application. Therefore all external libraries have been downloaded and imported into the project itself. At the moment of writing this security analysis, the application still makes use of one external icon library not served by our own application: *feather icons*. For the final sprint there has been chosen to download and import it to the project too.