GOOGLE'S GRUYERE

Vulnerability Analysis Report

Brief Description

Vulnerability Analysis report for the Google's Gruyere web application following OWASP's Web Security Testing Guide v4.2 reporting guidelines.

Alberto Gómez Ferrández

1. Introduction

1.1 Version Control

Version	Description	Date	Author
1.0	Initial report	04-29-2021	A. Gómez Ferrández

1.2 Table of Contents

1.3 The Team

- Alberto Gómez Ferrández
  - Computer Science Graduate
  - Cybersecurity Master's Degree

1.4 Scope

The scope of the project in limited to the public web application exposed at https://google-gruyere.appspot.com/, this is not a penetration testing for its servers or infrastructure.

1.5 Disclaimer

This document was made only for academic purposes.

2. Executive Summary

The objective of this test was to detect possible technical flaws on the company's web application. The kind of vulnerabilities that affect these applications may lead to serious business problems, system's integrity and availability and compliance with regulations like GDPR.

The tests consist in using the available data inputs on the web to create anomalous situations so we can check the correct configuration and implementation of the systems. Also, analysing the data that the server sends in the responses and its format can give hints about possible flaws.

In this test we found serious flaws regarding user information and session management that could easily lead to loss of user's data integrity and confidentiality, theft of user's identity and privilege escalation scenarios, in which regular users could access administration features, compromising the system.

This document will help to understand how authentication and authorization are easily compromised if the correct features are not implemented; And, also, show solutions with specific measures to apply so the system becomes much more robust and secure.

3. Findings

RISK LEVEL	DESCRIPTION	BASE SCORE
CRITICAL	Critical whilst posing an immediate threat. Fixing these issues should be the highest priority.	**9 - 10**
HIGH	Findings in this category pose an immediate threat and should be fixed immediately.	**7 - 8.9**
MEDIUM	May cause serious harm in combination with other security vulnerabilities. Should be fixed within short time.	**4 - 6.9**
LOW	These findings do not impose an immediate threat. Should be reviewed for their specific impact on the application and fixed accordingly.	**0.1 - 3.9**
INFO	Informational findings do not pose any threat but have solely informational purpose.	**0**

3.1 Findings Summary

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
1	Reflected XSS in *All Snippets* page	MEDIUM	5.4
2	Stored XSS in *New Snippet* page	MEDIUM	5.4
3	Stored XSS in *Private Snippet*	MEDIUM	5.4
4	Stored XSS in profile's name color	MEDIUM	5.4
5	Stored XSS in profile's icon	MEDIUM	5.4
6	File Upload Vulnerability (leading to XSS)	MEDIUM	5.4
7	CSRF to delete other users' snippets	MEDIUM	6.8

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
8	CSRF to edit other users' information	HIGH	8.7

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
9	CSRF to access administration features	CRITICAL	9.0

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
10	Insecure Admin Creation	CRITICAL	9.3

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
11	Insecure Cookie Format (privilege escalation)	CRITICAL	9.3

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
12	Default Admin Credentials	CRITICAL	9.3

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
13	Log-in Brute Force	MEDIUM	6.8

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
14	Weak Password Policy	INFO	0

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
15	Bad Log-out Functionality	MEDIUM	6.8

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
16	Missing HSTS Configuration	MEDIUM	4.8

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
17	Missing *Secure* Cookie Flag	MEDIUM	5.3

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
18	Missing *HttpOnly* Cookie Flag	MEDIUM	5.7

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
19	Missing *SameSite* Cookie Attribute	LOW	3.1

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
20	Path Traversal	MEDIUM	6.8

Ref. ID	FINDING RESULT	RISK LEVEL	SCORE
21	Missing *X-Frame-Options* HTTP Header	LOW	3.1

3.2 Findings Details

3.2.1 Reflected XSS in *All Snippets* page

Description

Using an URL to access the '*all snippets*' page of a certain user, the value of the query parameter will be added to the web page content, so JavaScript code can be injected on the URL and will be executed on every browser that loads it.

How it was exploited

With the following URL: *http://google-gruyere.appspot.com/<gruyere-id>/snippets.gtl?uid=<script>alert(1)</script>* the script is executed.

How to fix

The proper solution would be not to print query parameters on the web page. If the application looks for user's snippets on the database, the username should be retrieved from the server's response, instead of using the user ID from the URL.

3.2.2 Stored XSS in New Snippet page

Description

Snippets introduces by users will be interpreted as part of the web page HTML code, so every user that sees in the browser another user's snippet, may be a victim of an HTML/XSS injection attack.

How it was exploited

Submitting the following snippet: **.

How to fix

A possible solution would be to escape all the characters in HTML and treat them as plain text when printing them, so the users will see the snippet's plain code and not how the browser interprets it. CSPs that disallow inline JavaScript are also an option.

3.2.3 Stored XSS in Private Snippet

Description

Same as the vulnerability explained above, but on the private snippet.

How it was exploited

Submitting the following snippet: **.

How to fix

Same solutions as above XSS vulnerability.

3.2.4 Stored XSS in profile's name color

Description

From the 'edit profile' page, users can change the color in which their names are displayed. The way this function is performed is complete insecure and leads to XSS/HTML injection which every user that has the name displayed in their browsers will suffer.

How it was exploited

Inserting the payload: *blue' onMouseOver='alert(1)'*.

How to fix

Implement a whitelist of valid colors and validating user input.

3.2.5 Stored XSS in profile's icon

Description

The application allows users execute JavaScript code modifying the *img* HTML tag. This input is not correctly sanitised and allows XSS.

How it was exploited

Inserting the payload: */' onerror='alert(1)*.

Inserting the payload: *javascript:alert(1)*.

How to fix

Sanitizing user input (e.g. using a regex), to ensure it has URL format. Also, making sure that the image is not a SVG, as they allow JavaScript code execution.

3.2.6 File Upload Vulnerability (leading to XSS)

Description

JavaScript files may be uploaded on the 'file upload' page, leading to XSS when the user tries to see the content of the file from the browser. This vulnerability has greater risk and user's credentials are not needed to see his/her uploads.

How it was exploited

Uploading an HTML file with the following content: *<script>alert(1);</script>*

How to fix

User's file uploads should be quite restrictive, allowing only certain types of files and carefully checking their format when uploading them. If users' HTML files need to be displayed, all code should be escaped as on previous solutions.

3.2.7 CSRF to delete other users' snippets

Description

CSRF attacks permit attackers to perform actions on the server making calls from pages that were not designed for that purpose. This may lead to stored XSS attacks that make use of the logged-in users' credentials to perform actions in their names.

Using any of the above Stored XSS vulnerabilities, an attacker can make requests to the server to delete the snippets of the logged user.

How it was exploited

Submitting the following snippet:

<img src="http://google-gruyere.appspot.com/<grueye-id>/deletesnippet?index=0"/>

How to fix

Make sure that GET requests are not used to change the state of the server, like in this case. Implement anti-CSRF cookies to ensure requests originate from the same page it was served to. In POST requests, the forms could send back the CSRF token to the server and check that the form token match the cookie value. Set the anti CSRF cookie with the *SameSite=Strict* syntax to ensure that the browser sends the cookie only with requests initiated from within your own site.

3.2.8 CSRF to edit other users' information

Description

Like the previous CSRF attack, using any of the above Stored XSS vulnerabilities, an attacker can make requests to the server to edit the profile information of the logged user.

How it was exploited

Submitting the following snippet:

<img src="http://google-gruyere.appspot.com/<gruyere-id>/saveprofile?action=update&name=hacked&oldpw=&pw=&icon=&web_site=&color=&private_snippet=%3Cimg+src%3D%2F+onerror%3D%22alert%281%29%22%2F%3E"/>

This will create a malicious private snippet on the victim's profile that takes advantage of the XSS vulnerability presented on section 3.2.3.

How to fix

Same solutions as above CSRF vulnerability.

3.2.9 CSRF to access administration features

Description

A much more dangerous scenario is when the attacker knows how administration requests are made and uses them in their CSRF attacks, so administration tasks are launched, like restarting the server or resetting the database.

How it was exploited
Submitting the following snippet:

<img src="http://google-gruyere.appspot.com/<gruyere-id>/reset"/>

How to fix
Same solutions as above CSRF vulnerability. If only this would be a POST request, the previous payload would not exploit the vulnerability. Also, requiring reauthentication for sensitive actions helps preventing this kind of flaw.

3.2.10 Insecure Admin Creation

Description
The public website's server should not be able to create administration accounts. Based on the sign-up URI *(/saveprofile?action=new&uid=asdf&pw=asdf&is_author=True*), an attacker could figure out and try to add at the end '*&is_admin=True*' and get administration access right away.

How it was exploited
With the following URL, an administration user called 'John' is created:

http://google-gruyere.appspot.com/<gruyere-id>/saveprofile?action=new&uid=John&pw=1234&is_author=True&is_admin=True

How to fix
The server must not allow administration user creation from public requests. These users should be created internally and the '*is_admin*' parameter should not be implemented.

3.2.11 Insecure Cookie Format (privilege escalation)

Description
Having the cookies with a plain-text simple format helps the attacker understand how they are created and how a hijacking attack could be made.

Actual cookie format is 'hash|userid|admin|author', being 'hash|userid||author' if the user is not an administrator. By just signing up with a username like '<name>|admin' the application will give our user administration privileges.

How it was exploited
Creating a user with the username '*Alice|admin*'.

The resulting session cookie is '86174986|Alice|admin||author' and the server treats Alice as an administrator.

How to fix
The server should check if the user is an administrator by checking its role in the database and not basing on cookie information.

3.2.12 Default Admin Credentials

Description

Administration credentials should not be easily guessable. Using 'default' credentials like 'admin|1234' should not be used as our application would be highly vulnerable against dictionary attacks. Using this kind of attack, we discovered some administration credentials in Gruyere's web app.

How it was exploited

Default administration credentials are easily guessed. Launching a dictionary attack with basic username/password combinations led us to discover the existing admin credentials 'administrator|secret'.

How to fix

Implementing robust password policies and a non-revealing administrator username.

3.2.13 Log-in Brute Force

Description

In relation with the previous vulnerability, dictionary/brute-force attacks should be stopped with some mechanism.

How it was exploited

Launching a dictionary attack with *BurpSuite* tool.

How to fix

Common solutions are the implementation of a captcha or the increment of waiting time between requests when a single endpoint is being called massively.

3.2.14 Weak Password Policy

Description

Passwords should have several requirements, like minimum length or the use of special characters, both to obstruct brute-force and dictionary attacks.

How to fix

Informing the user about the password format on sign-up page and validating user input.

3.2.15 Bad Log-out Functionality

Description

When a user logs out, the session cookie should be revoked so it is not valid anymore. However, Gruyere's webapp session cookies can be used after the user logs out, allowing the attacker to perform actions in the user's name with those old session IDs.

How it was exploited
Saved the user's session ID, logged-out and then successfully submitting a snippet with the use of that session ID from a proxy tool like *BurpSuite*.

How to fix
Revoking the session ID on the server once the user logs out.

3.2.20 Path Traversal

Description
Path Traversal vulnerability occurs when the user manages to access server's directories that were not supposed to be accessible from the webpage. Due to poor name policies for usernames and uploaded files, the user is able to move between directories by creating profiles or uploading files with '../' in their names.

By uploading a file called like '*../../dbconfig*', we could be overwriting important configuration files. With the following URL: '*http://gruyere.com/uploads/username/../../dbconfig*', we would be accessing a config file in the webserver's main directory: '*http://gruyere.com/dbconfig*'.

How it was exploited
Uploading a file with name '../filename' or creating a user with name '../username'.

How to fix
For usernames, validating user input and not allowing special characters. For filenames, a good practice would be to rename the file on the server and not showing the uploads folder's path to the user.

3.2.16 Missing HSTS Configuration

Description
HSTS is a Security feature that allows a website to tell browsers that they should only communicate with HTTPS instead of using HTTP. This helps preventing credentials and cookie theft.

How to fix
Ensure that HSTS is configured in your web server.

3.2.17 Missing *Secure* Cookie Flag

Description
With the *Secure* flag, cookies can only be sent through HTTPS, avoiding session ID's theft due to insecure connectivity.

How to fix
Ensure that the *Secure* flag is set for all cookies.

3.2.18 Missing *HttpOnly* Cookie Flag

Description

With the *HttpOnly* flag, cookies cannot be accessed from the browser code. This prevents the attacker from stealing session IDs though JavaScript vulnerabilities.

How to fix

Ensure that the *HttpOnly* flag is set for all cookies.

3.2.19 Missing *SameSite* Cookie Attribute

Description

With the *SameSite* flag, cookies cannot be sent from a webpage that is not the one it was set on. This helps prevent Cross-Site Request Forgery (CSRF) attacks.

How to fix

Ensure that the *SameSite* flag is set for all cookies.

3.2.21 Missing *X-Frame-Options* HTTP Header / CSP frame-ancestors

Description

Proper CSP frame-ancestors directive response headers instruct the browser to not allow framing from other domains. This replaces the older X-Frame-Options HTTP headers. Disallowing framing options helps to prevent click-jacking attacks.

Appendices

I. Risk Score (CVSS)

Risk Score represents the Base Score from the CVSS system. The scores have been calculated with the CVSS v3.1 calculator available at: https://www.first.org/cvss/calculator/3.1.