# TABLE OF CONTENTS

1. REPORT STRUCTURE
2. ABOUT THE EDITOR
3. EXECUTIVE SUMMARY
4. BACKGROUND
5. PROJECT DESCRIPTION
6. SCOPE & TARGETS
7. TEST LIMITATIONS
8. SUMMARY & ASSESSMENT
9. CONCLUSIONS
10. SETTING GOALS AND OBJECTIVES
11. IDENTIFIED VULNERABILITIES

* VULN-001: Stored XSS via HTML Attribute (HIGH)
* VULN-002: Reflected XSS (HIGH)
* VULN-003: Stored XSS (HIGH)
* VULN-004: Stored XSS via HTML Attribute (HIGH)
* VULN-005: Stored XSS via AJAX (HIGH)
* VULN-006: Reflected XSS via AJAX (HIGH)
* VULN-007: Elevation of Privilege (MEDIUM)
* VULN-008: Cookie Manipulation (MEDIUM)
* VULN-009: Cross-Site Request Forgery (XSRF) (MEDIUM)
* VULN-010: Cross Site Script Inclusion (XSSI) (MEDIUM)
* VULN-011: Path Traversal (MEDIUM)
* VULN-012: Denial of Service
* VULN-013: Code Execution (CRITICAL)
* VULN-014: Information Disclosure (MEDIUM)
* VULN-015: Phishing via Insecure JavaScript code Implementation (MEDIUM)

## **REPORT STRUCTURE**

This report comprises 2 main sections:

1. Executive Summary: A concise overview of the penetration testing work and the main findings requiring immediate attention.

2. Details of the Tests: A comprehensive breakdown of all tests conducted, including vulnerabilities identified, risk assessments, and recommendations for remediation.

## **ABOUT THE EDITOR**

Orr Amsalem a dedicated Cyber Security student currently enrolled in ECOM. My coursework covers essential areas like network security, ethical hacking, and risk management, providing me with practical skills in identifying vulnerabilities and responding to cyber threats. I use industry-standard tools and stay updated on the latest trends in cyber security. Committed to continuous learning, I am eager to contribute my knowledge and enthusiasm for safeguarding digital assets. My goal is to apply my skills in addressing vulnerabilities, such as those identified in this SQL Injection Vulnerability Assessment Report and contribute to enhancing overall security resilience.

*[LinkedIn Profile](https://www.linkedin.com/in/orramsalem/)*

## **EXECUTIVE SUMMARY**

The recent security assessment conducted on Google Gruyere uncovered various vulnerabilities, from severe to moderate levels of risk. These vulnerabilities could potentially compromise the safety of the application and the data it holds. Some of the critical issues found include the ability for attackers to run their code within the application (VULN-015) and inject harmful scripts (VULN-003), which could lead to unauthorized access and data theft.

Additionally, there were medium-level concerns like Cross-Site Request Forgery (VULN-009) and Information Disclosure (VULN-011), which may not be as severe but still pose risks to users' security and privacy.

Moreover, the assessment revealed vulnerabilities related to potential Denial of Service attacks, where attackers could disrupt the service by terminating it (VULN-013) or overloading it (VULN-014), causing availability issues.

Overall, this assessment stresses the importance of maintaining strong security measures and regularly checking for vulnerabilities. By addressing these issues and implementing the recommended measures, Google Gruyere can better protect its users and their data from potential threats and breaches.

## **BACKGROUND**

Google Gruyere is a vulnerable web application designed for educational purposes to demonstrate common security vulnerabilities and help developers understand how to secure their applications. As part of its security testing and improvement efforts, the application undergoes regular penetration testing to identify and address potential vulnerabilities.

The penetration testing on Google Gruyere was conducted by Orr Amsalem, aiming to assess the security posture of the application and identify any weaknesses that could be exploited by malicious actors. The test scenarios included various attack vectors, such as code injection, cross-site scripting (XSS), privilege escalation, and denial-of-service (DoS) attacks, to comprehensively evaluate the application's resilience against different threats.

The assessment involved examining both the web application and underlying infrastructure to identify vulnerabilities that could compromise the confidentiality, integrity, and availability of the system. By simulating real-world attack scenarios, the testing aimed to provide actionable insights and recommendations for strengthening Google Gruyere's security defenses and mitigating potential risks.

Overall, the penetration testing on Google Gruyere aimed to enhance the application's security posture, educate developers about common security pitfalls, and ultimately contribute to the development of more secure web applications. The findings and recommendations generated from the testing process are crucial for improving the application's resilience against cyber threats and ensuring the protection of user data and privacy.

## **PROJECT DESCRIPTION**

Orr Amsalem conducted a comprehensive penetration test on Google Gruyere, a vulnerable web application developed by Google for educational purposes. The primary objective of the project was to assess the security posture of Google Gruyere and identify potential vulnerabilities that could be exploited by malicious actors. The penetration test involved various methodologies, including code analysis, manual testing, and automated scanning, to thoroughly evaluate the application's security controls.

The scope of the project encompassed both the web application and its underlying infrastructure. Orr Amsalem targeted a wide range of attack vectors, such as injection flaws, cross-site scripting (XSS), privilege escalation, and denial-of-service (DoS) attacks, to simulate real-world threats and assess the effectiveness of the application's defenses. The testing process was designed to identify vulnerabilities that could compromise the confidentiality, integrity, and availability of the system.

By conducting the penetration test, Orr Amsalem aimed to provide valuable insights into the security weaknesses of Google Gruyere and recommend appropriate remediation measures to mitigate the identified risks. The project also aimed to raise awareness among developers about common security pitfalls and best practices for developing secure web applications. Ultimately, the goal of the project was to contribute to the improvement of Google Gruyere's security posture and enhance its resilience against cyber threats.

## **SCOPE & TARGETS**

The scope of the security assessment conducted by Orr Amsalem on Google Gruyere was broad and thorough, covering both the web application itself and the infrastructure it relies on. Here's a breakdown of what we focused on:

1. Web Application Security: We looked for vulnerabilities within the Google Gruyere web application, such as weaknesses that could allow hackers to inject malicious code, gain unauthorized access, or expose sensitive information. This included testing for common issues like cross-site scripting and authentication bypass.
2. Infrastructure Security: We also assessed the security of the servers, databases, and other components supporting Google Gruyere. This part of the assessment aimed to uncover any weaknesses that could be exploited to access data or disrupt services.
3. Attack Vectors: Our tests covered a wide range of potential attack methods, including file uploads, phishing attempts, and denial-of-service attacks. By simulating these real-world threats, we could see how well Google Gruyere defends against them.
4. Security Controls: We evaluated the effectiveness of the security measures in place, such as how well the application validates user input, controls access to sensitive data, and encrypts information. This helped identify areas where improvements could be made.
5. Compliance Requirements: Lastly, we made sure that Google Gruyere met relevant security standards and regulations, such as those set by OWASP and other industry guidelines.

Overall, our goal was to provide a thorough assessment of Google Gruyere's security and help strengthen its defenses against potential cyber threats.

## **TEST LIMITATIONS**

The primary limitation of the penetration testing was the controlled testing environment. While every effort was made to replicate real-world scenarios, it's important to note that the effects observed during testing may differ in a production setting. In a controlled environment, the impact of the vulnerabilities identified can be contained and managed more effectively. However, in a live production environment, the consequences of exploitation could be more severe, potentially leading to service disruptions, data breaches, or other security incidents.Top of Form

## **SUMMARY & ASSESSMENT**

The comprehensive security assessment conducted by Orr Amsalem aimed to evaluate the safety measures of Google Gruyere, a web application designed for testing purposes. The assessment looked into various aspects of security, including the application's infrastructure and layers, with the goal of uncovering any weaknesses that could be exploited by malicious actors.

During the assessment, we discovered a range of vulnerabilities across different areas of the application. These vulnerabilities, while they may sound complex, essentially represent areas where the application could be at risk. They include things like potential ways for attackers to sneak harmful code into the system, manipulate user sessions, or even disrupt the service entirely.

We assessed each vulnerability based on how severe it could be and how likely it is to be exploited. Some vulnerabilities, like File Upload XSS and Code Execution, pose the most significant risks because they could allow attackers to take control of the application entirely. Others, like Stored XSS and denial-of-service attacks, also present serious threats and need urgent attention to fix.

In addition to identifying vulnerabilities, we also looked at the existing security measures in place and provided recommendations to help strengthen them. The goal is to make sure Google Gruyere is as secure as possible and better protected against potential cyber threats.

Overall, the assessment highlights the importance of having strong security measures in place for web applications. By staying vigilant and addressing vulnerabilities promptly, we can better protect sensitive data and reduce the risk of security breaches. It's essential for organizations to prioritize security and regularly assess their systems to stay one step ahead of potential threats.  
Top of Form

## **CONCLUSIONS**

Based on the comprehensive evaluation of Google Gruyere's security, here are the key insights:

1. **Security Status**: The security assessment revealed that Google Gruyere faces a notable level of risk due to the presence of various vulnerabilities across its systems.
2. **Severity of Vulnerabilities**: The vulnerabilities identified range from minor to critical, with critical ones posing the most substantial threat to the application's security and integrity.
3. **Likelihood of Exploitation**: Several vulnerabilities, particularly those related to cross-site scripting (XSS), denial-of-service (DoS), and information disclosure, are susceptible to exploitation by malicious actors.
4. **Technical Complexity**: While some vulnerabilities require advanced technical skills to exploit, others are relatively straightforward, underlining the importance of addressing all security issues promptly.
5. **Urgency of Action**: Immediate action is imperative to address critical vulnerabilities, followed by prioritizing high and medium severity issues. Delaying fixes could leave the application vulnerable to unauthorized access, data breaches, or service disruptions.
6. **Continuous Improvement**: Security measures should be continually reviewed and updated. Regular vulnerability assessments, staying informed about emerging threats, and implementing robust security protocols are essential for ongoing protection.
7. **Team Collaboration**: A collaborative approach involving developers, administrators, and security experts is vital. Ongoing training and awareness initiatives will ensure that everyone within the organization is equipped to uphold and reinforce security best practices. In conclusion, addressing these vulnerabilities and implementing the recommended measures will significantly enhance Google Gruyere's security posture, mitigating the risk of cyberattacks and safeguarding critical data and assets.

**SETTING GOALS & OBJECTIVES**

During the penetration testing operations, the primary objectives were defined to ensure comprehensive coverage and effective evaluation of Google Gruyere's security posture. The following goals and objectives were established:

1. **Identify High-Risk Vulnerabilities**: The foremost objective was to identify and prioritize high-risk vulnerabilities within Google Gruyere's infrastructure and web application. This includes vulnerabilities that could lead to unauthorized access, data breaches, or service disruptions.
2. **Assess Security Controls**: Evaluate the effectiveness of existing security controls, including authentication mechanisms, access controls, encryption protocols, and monitoring solutions. This involved testing these controls for weaknesses and ensuring they provide adequate protection against common attack vectors.
3. **Validate Compliance**: Verify compliance with industry standards and best practices, such as OWASP guidelines and security frameworks. This ensures that Google Gruyere adheres to established security standards and mitigates known security risks effectively.
4. **Test Incident Response**: Assess the effectiveness of Google Gruyere's incident response procedures by simulating real-world attack scenarios. This involves testing the detection, containment, and recovery processes to ensure they are robust and well-coordinated.
5. **Provide Recommendations**: Offer actionable recommendations and best practices for addressing identified vulnerabilities and improving overall security posture. This includes prioritizing remediation efforts based on risk severity and providing guidance on implementing security controls effectively.
6. **Enhance Awareness**: Raise awareness among stakeholders about the importance of cybersecurity and the potential risks posed by various attack vectors. This involves providing educational materials, training sessions, and best practices to help users and administrators better understand and mitigate security threats.

By setting clear goals and objectives, the penetration testing operations aimed to provide Google Gruyere with valuable insights into its security strengths and weaknesses, enabling informed decision-making and proactive risk mitigation efforts.

**IDENTIFIED** **VULNERABILITIES**

**VULN-001: File Upload XSS (HIGH)**

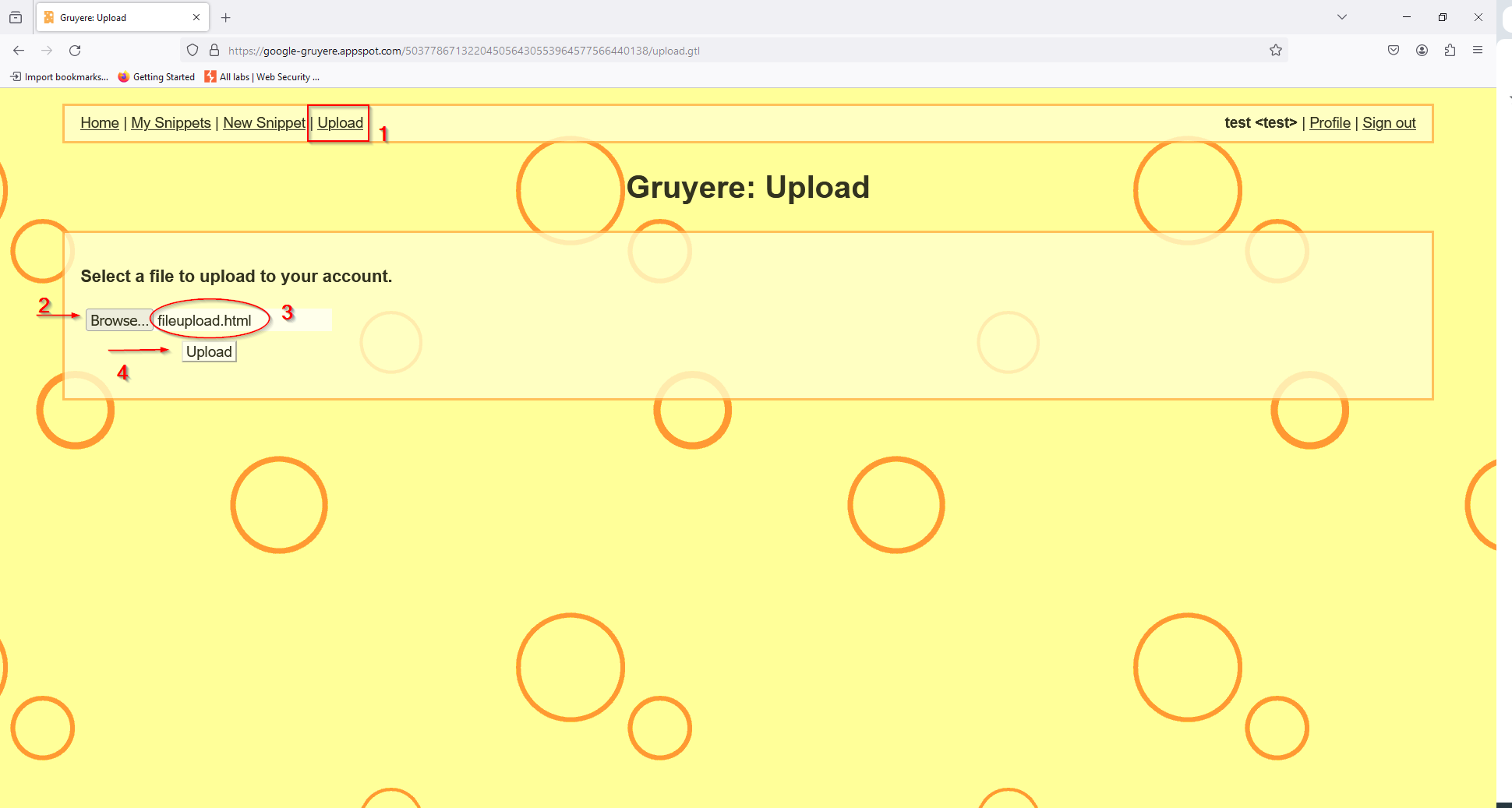
**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

**RISK:**

* General: High
* Probability: High
* Severity: High
* Fix Effort: Low

**DESCRIPTION:** File Upload XSS is a critical vulnerability that allows attackers to upload malicious files containing XSS payloads. These payloads can execute arbitrary code in the context of the web application, leading to unauthorized access, data theft, and further exploitation of the system. By exploiting this vulnerability, attackers can bypass security measures and compromise the integrity and availability of Google Gruyere and create other vulnerabilities such as DOS and SSTI.

**PROOF OF CONCEPT:** To demonstrate the File Upload XSS vulnerability, an authenticated user accesses the upload feature of the website containing a simple XSS payload showing the document cookie.



The payload used in this Proof of Concept (POC) is a basic alert script. The file contained the script below:

<!DOCTYPE html>

<html>

<h1>Orr hacked you!<h/1>

<script>

alert(document.cookie);

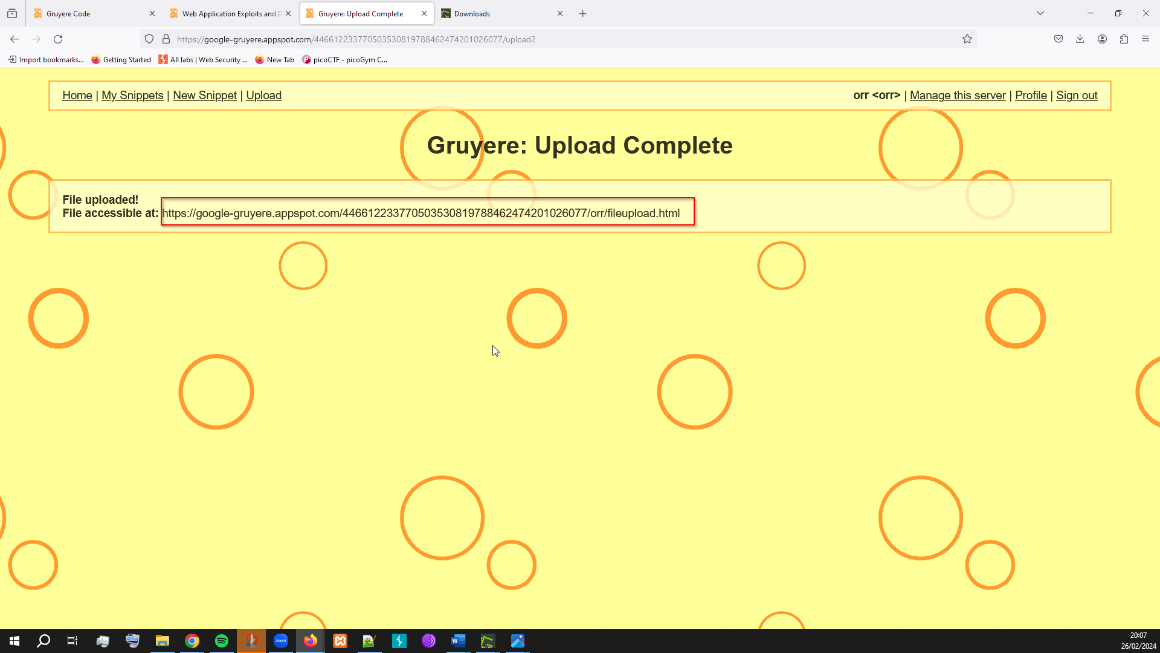
</script>

</body>

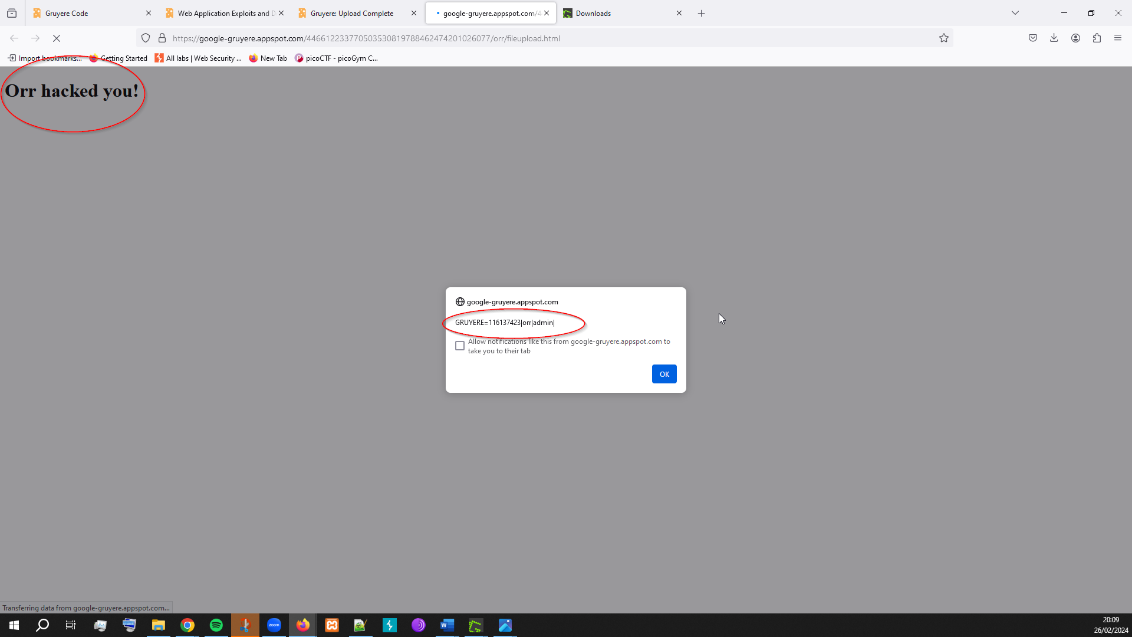
</html>   
  
Upon successful upload, the file was processed by the Google Gruyere application without proper validation, allowing the XSS payload to execute when accessed and show the message “Orr hacked you! “ followed by an alert with the content of the session cookie .

The following steps outline the POC:

1. Access the Google Gruyere application and navigate to the file upload feature.
2. Upload a file containing the XSS payload provided above.
3. Once the file is uploaded successfully, access the uploaded file's URL triggers its execution within the application.



1. The XSS payload will execute in the context of the application, displaying the message followed by a js pop up with the cookie session.



This POC demonstrates how an attacker can exploit the File Upload XSS vulnerability to execute arbitrary scripts within the application, potentially leading to further exploitation and compromise of user data.

**RECOMMENDED MITIGATIONS:**

1. Implement strict file upload validation mechanisms to ensure that only allowed file types are accepted. Additionally, consider hosting user-uploaded content on a separate domain to prevent scripts from accessing sensitive content on the main domain. This approach reduces the risk of XSS attacks by isolating user-generated content from the main application.
2. Sanitize file contents before processing by performing thorough filtering and content checking on any files uploaded to the server. Implement robust scanning and validation procedures to detect and remove any potentially malicious scripts or payloads. Files that raise doubts regarding their safety should be discarded to prevent the risk of XSS exploitation.
3. Incorporate content security policies (CSP) to further restrict the execution of scripts and enhance protection against XSS attacks. By defining strict policies regarding script execution, CSP helps mitigate the impact of XSS vulnerabilities by limiting the ability of malicious scripts to execute within the application.

It's important to note that the Google Gruyere application currently lacks these protective measures, as observed in the code review. Implementing these mitigations will help strengthen the application's security posture and mitigate the risk of file upload XSS vulnerabilities. For more detailed countermeasures, refer to the OWASP guide provided at <https://bit.ly/3xVUjqE>.

**VULN-002: Reflected XSS (HIGH)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

**RISK:**

* General: High
* Probability: High
* Severity: High
* Fix Effort: Low

**DESCRIPTION:** Reflected XSS is a critical vulnerability that allows attackers to inject malicious scripts into web applications, which are then reflected back to users in certain contexts. In the case of Google Gruyere, the vulnerability was discovered in the search functionality, where user-supplied input was not properly sanitized before being echoed back in the application's responses. This allowed an attacker to craft a malicious URL containing a script payload, which, when clicked by a victim user, would execute the script in their browser.

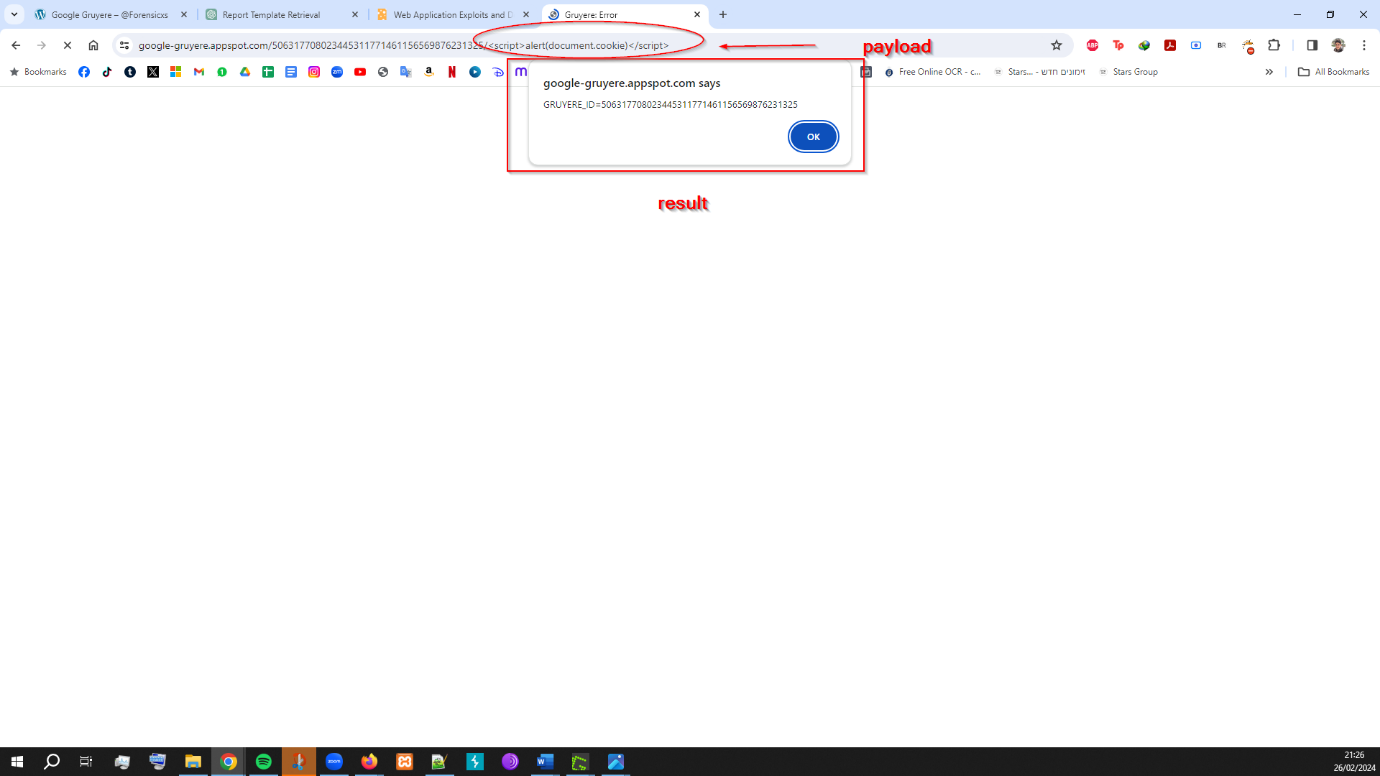
**PROOF OF CONCEPT:**

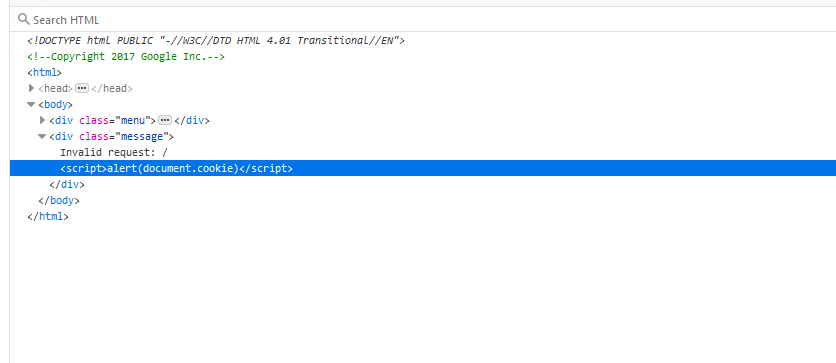
During the penetration test, it was observed that the Google Gruyere application did not properly validate and sanitize user input in the search functionality. As a result, it was possible to inject arbitrary HTML and JavaScript code into the application's responses. This allowed an attacker to craft a specially crafted URL containing the XSS payload, which, when accessed by a victim user, would trigger the execution of the malicious script in their browser.  
To demonstrate the Reflected XSS vulnerability in Google Gruyere, we crafted a URL containing a simple XSS payload “<script>alert(document.cookie)</script>”  
  
*https://google-gruyere.appspot.com/* *446612233770503530819788462474201026077/<script>alert(document.cookie)</script>*  
Upon accessing the malicious URL, the XSS payload executed within the context of the application, displaying a pop up containing the session’s cookie.

The following steps outline the POC:

1. Access the Google Gruyere application and navigate to the url bar.
2. In the URL input the following payload:

“<script>alert(document.cookie)</script>”

1. Execute the search by clicking enter.
2. The payload will be reflected back in the page, and the XSS payload will execute, displaying an alert message. 
3. We can also see the script inserted in the Elements inspector.



**RECOMMENDED MITIGATIONS:**

1. One issue is that Gruyere sends us an error message, but the script is included in the output rendered code (as seen before in the Elements inspector. One way to fix is to escape the user input that is displayed in error messages. Error messages are displayed using error.gtl, but are not escaped in the template. The part of the template that renders the message is {{message}} and it’s missing the modifier that tells it to escape user input. Add the ”: text” modifier to escape the user input. This is called manual escaping.

<div class="message">{{\_message:text}}</div>

When adding this to the error.gtl code and running it We can see that the script is escaped to text and the script is not executed.

1. In addition, implement proper input validation and output encoding to prevent XSS attacks. Ensure that user-supplied input is thoroughly validated and sanitized before being rendered in the application's output. Use input validation techniques to reject any input that contains potentially dangerous characters and apply output encoding to encode user input before displaying it in the application's response. This will help mitigate the risk of XSS attacks by preventing malicious scripts from being executed.
2. Sanitize user-supplied input to remove or encode potentially dangerous characters. Apply strict input sanitization techniques to filter out any characters or strings that could be used to inject malicious scripts into the application's output. Remove or encode special characters, such as <, >, ", ', and &, to prevent script injection attacks. By sanitizing user input effectively, you can reduce the likelihood of XSS vulnerabilities in your application.
3. Use security mechanisms such as Content Security Policy (CSP) to restrict the execution of scripts. Implement Content Security Policy (CSP) headers in your web application to define a whitelist of trusted sources for scripts, stylesheets, and other resources. By specifying a strict CSP policy, you can prevent unauthorized scripts from being executed on your web pages, thereby mitigating the risk of XSS attacks. Additionally, CSP can help detect and mitigate XSS attacks by blocking or reporting any attempts to violate the defined security policy.

By incorporating these mitigation measures into your application's development process, you can significantly enhance its resilience against Reflected XSS vulnerabilities and protect your users' sensitive information from exploitation by malicious actors.

**VULN-003: Stored XSS (HIGH)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:H

**RISK:**

* General: High
* Probability: High
* Severity: High
* Fix Effort: Medium

**DESCRIPTION:** Stored XSS is a critical vulnerability that allows attackers to inject malicious scripts into web applications, which are then stored and executed when accessed by other users. In the case of Google Gruyere, the vulnerability was discovered in the snippet creation functionality, where user-supplied input was not properly sanitized before being stored in the database. This allowed an attacker to create a malicious snippet containing a script payload, which, when viewed by a victim user, would execute the script in their browser.

**PROOF OF CONCEPT:**

During the penetration test, it was observed that the Google Gruyere application did not properly validate and sanitize user input in the snippet creation functionality. As a result, it was possible to inject arbitrary HTML and JavaScript code into the application's database. This allowed an attacker to create a snippet containing the XSS payload, which would be executed whenever another user viewed the snippet on the platform.  
  
The following steps outline the POC:

1. Access the Google Gruyere application and navigate to the snippet creation feature.
2. In the snippet content field, input the following payload:

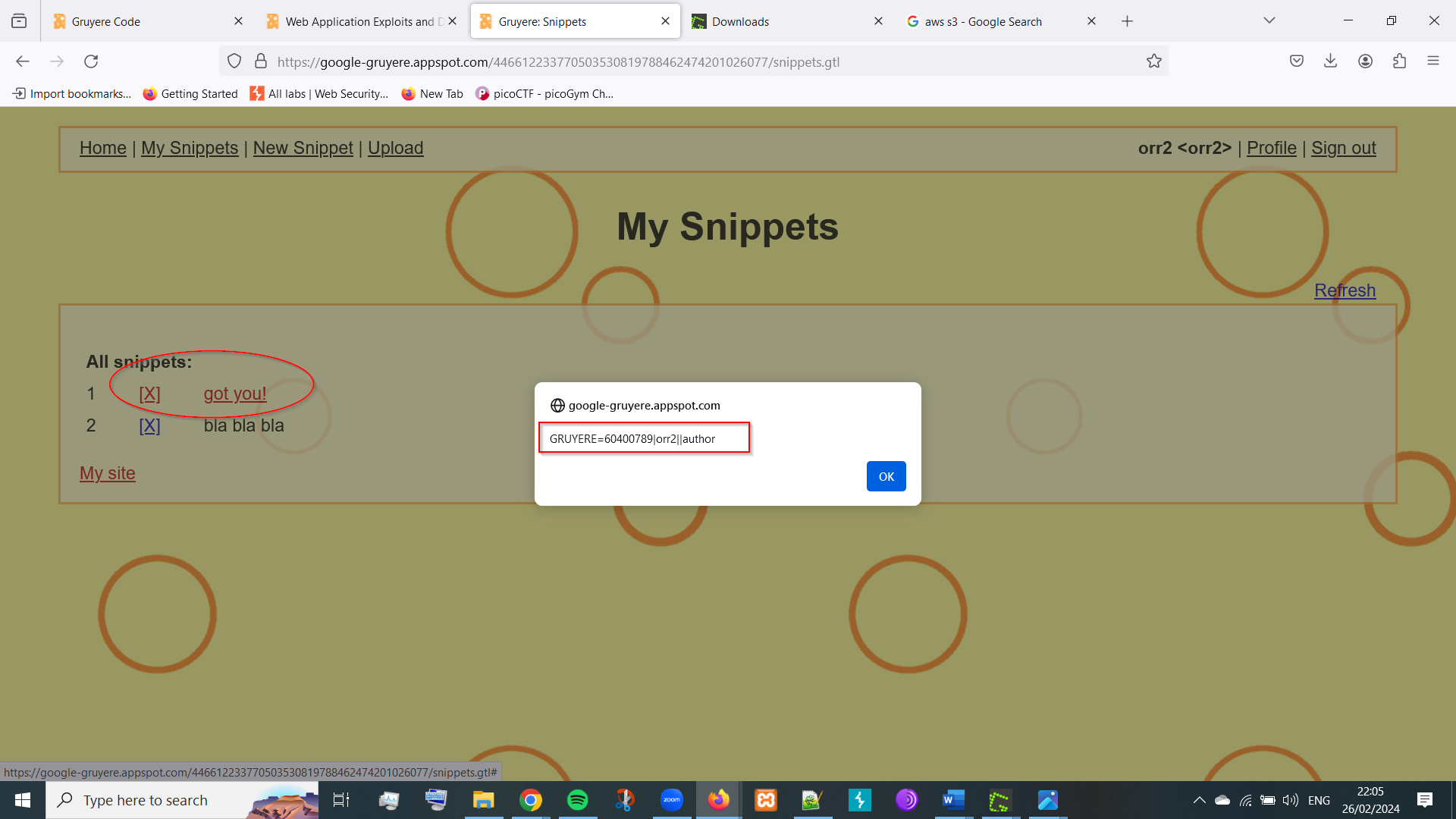
<a onmouseover="alert(document.cookie)" href="#">got you!</a>

1. Create the snippet by clicking the submit button.

A screenshot of a computer

Description automatically generated

1. The payload will be stored in the database and displayed to other users who view the snippet, triggering the execution of the malicious script in their browser.



**RECOMMENDED MITIGATIONS:**

1. Enhance the Sanitizer to effectively catch XSS threats by strengthening its capabilities. Identify weaknesses in the current Sanitizer implementation and address them to ensure it can accurately detect and block XSS payloads. Consider improving the Sanitizer's handling of attributes like "onmouseover", sensitivity to lower and upper cases and other potentially dangerous elements to prevent script execution.
2. Update the Sanitizer code to address flaws and vulnerabilities. Conduct a thorough examination of the Sanitizer implementation, including files such as gtl.py (value = sanitize.SanitizeHtml(str(value))) and sanitize.py, to identify areas where improvements can be made. Make necessary modifications to the Sanitizer code to address any identified weaknesses and enhance its ability to sanitize user input effectively.
3. Implement strict whitelists for allowed HTML tags and attributes to prevent XSS attacks. Define comprehensive whitelists of permissible HTML elements and attributes, ensuring that only safe and essential tags are allowed in user input. Enforce strict validation rules to reject any input that contains disallowed tags or attributes, thereby mitigating the risk of XSS vulnerabilities in the application.
4. Apply rigorous sanitization techniques to handle URL and CSS attributes securely. Implement robust sanitization procedures for URL and CSS attributes, ensuring that any permitted URLs or stylesheets are thoroughly validated and sanitized to prevent XSS exploits. Utilize proven HTML sanitization libraries or tools to parse input into an intermediate DOM structure and rebuild it as well-formed output, adhering to strict whitelists and sanitization rules.

By following these recommended approaches to HTML sanitization and strengthening the Sanitizer's capabilities, you can significantly reduce the risk of Stored XSS vulnerabilities in your application and protect against potential exploitation by malicious actors. Additionally, consider utilizing reputable HTML sanitization libraries, such as the one provided in the link (https://github.com/Vereyon/HtmlRuleSanitizer), to further enhance your application's security posture and safeguard user data from XSS attacks.

**VULN-004: Stored XSS via HTML Attribute (HIGH)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:H

**RISK:**

* General: High
* Probability: High
* Severity: High
* Fix Effort: Medium

**DESCRIPTION:** Stored XSS via HTML Attribute is a critical vulnerability that allows attackers to inject malicious scripts into web applications through HTML attributes, which are then stored and executed when accessed by other users. In the case of Google Gruyere, the vulnerability was discovered in the snippet creation functionality, where user-supplied input was not properly sanitized before being stored in the database. This allowed an attacker to create a malicious snippet containing a script payload within HTML attributes, which, when viewed by a victim user, would execute the script in their browser.

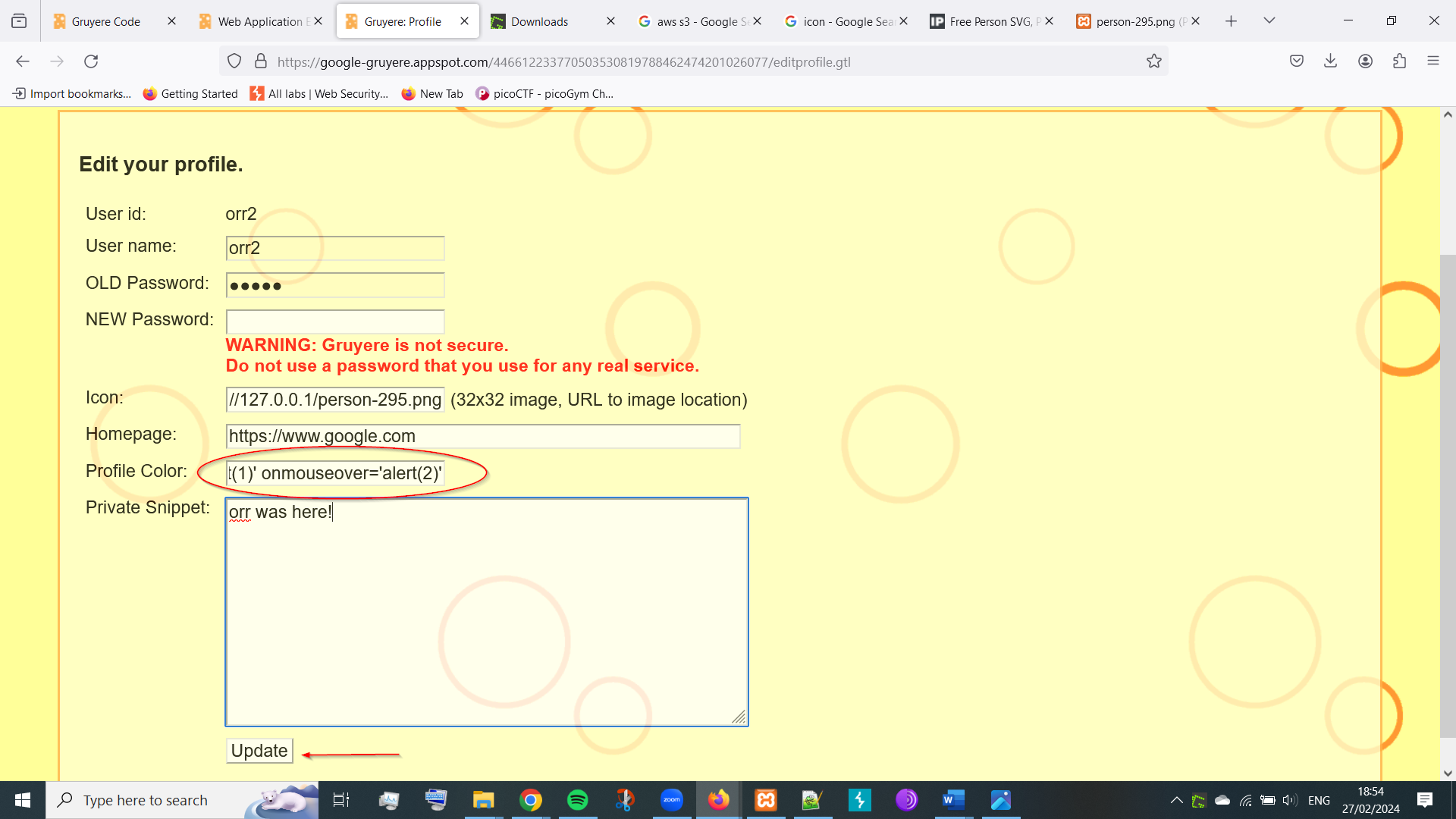
**PROOF OF CONCEPT:**

During the penetration test, it was observed that the Google Gruyere application did not properly validate and sanitize user input in the profile color functionality, specifically within HTML attributes. As a result, it was possible to inject arbitrary JavaScript code into the profile color tag, which would be executed the mouse hovers over the profile name

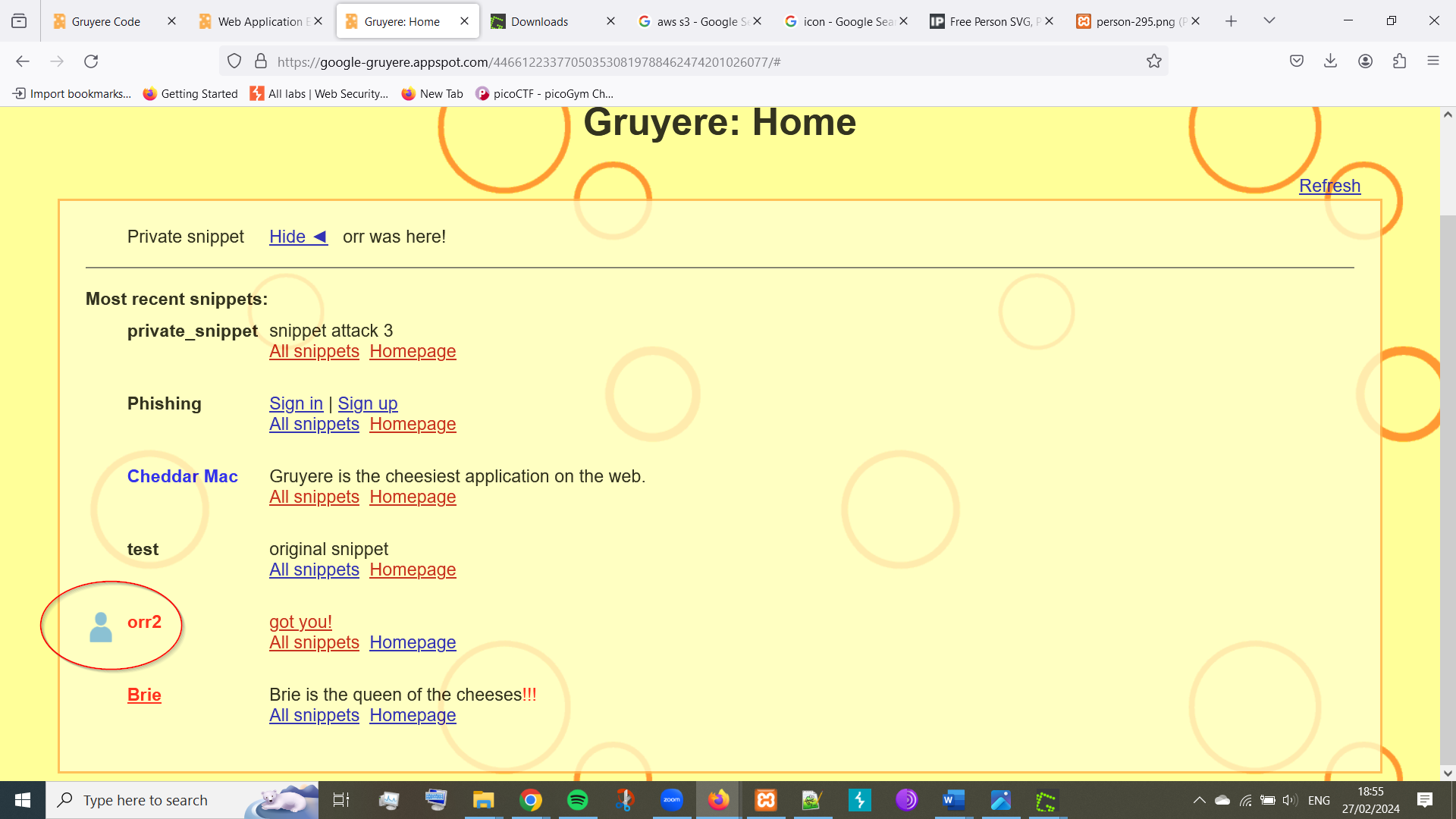
1. Access the Google Gruyere application and navigate to the profile editing feature.
2. In the profile color field, input the following payload:

red' onload='alert(1)' onmouseover='alert(2)'

1. Create the payload attack by clicking the update button.



1. The payload will be stored in the database and displayed every time the mouse hovers over the profile name triggering the execution of the malicious script and showing a pop up with the number 2.



A screenshot of a computer

Description automatically generated

**RECOMMENDED MITIGATIONS:**

1. As previously stated in earlier vulnerabilities, sanitize User-Supplied Input:
   * Prior to storing user input in the database, apply thorough sanitization routines to remove or encode potentially malicious characters, particularly those that could be used to execute XSS attacks via HTML attributes.
   * Use a combination of whitelisting and blacklisting approaches to filter out disallowed characters and prevent injection of script payloads into attribute values.
2. As mentioned earlier, implement Content Security Policy (CSP):
   * Configure a robust Content Security Policy (CSP) to define and enforce security policies for the web application, including directives to restrict the execution of scripts and mitigate the impact of XSS attacks.
   * Specify strict policies for allowing only trusted sources for scripts, stylesheets, and other external resources, thereby reducing the likelihood of successful exploitation of XSS vulnerabilities via HTML attributes.
3. Utilize Escaping Functions:
   * Augment existing escaping functions, such as cgi.escape, with additional logic to escape single and double quotes.
   * Introduce a custom escaping function, such as \_EscapeTextToHtml(), to replace cgi.escape in the gtl.py to provide more comprehensive protection against XSS attacks targeting HTML attributes.

**VULN-005: Stored XSS via AJAX (HIGH)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:H

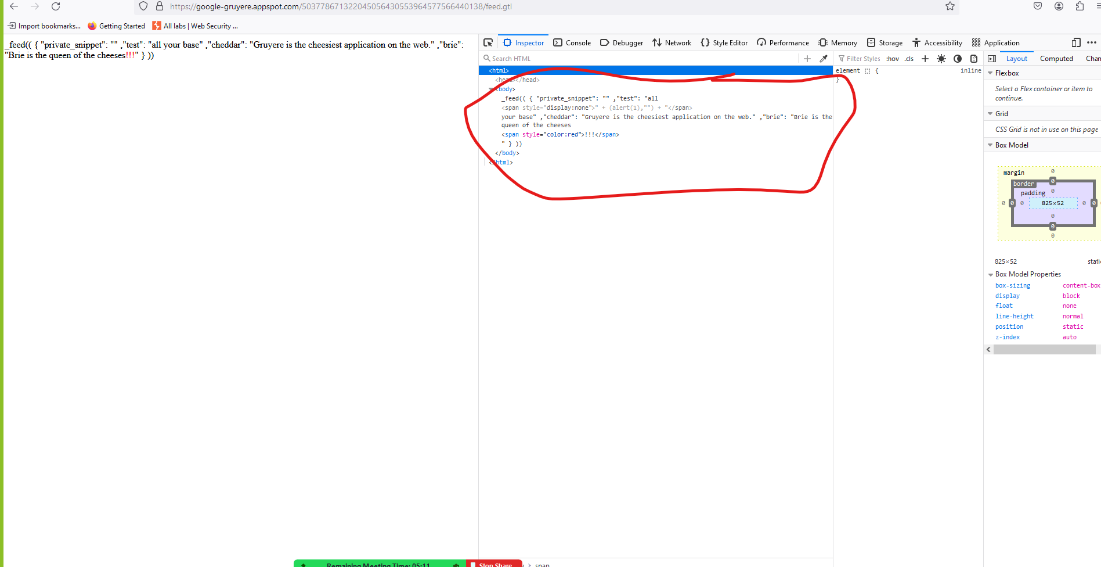
**RISK:**

* General: High
* Probability: High
* Severity: High
* Fix Effort: Medium

**DESCRIPTION:** Stored XSS via AJAX is a critical vulnerability that allows attackers to inject malicious scripts into web applications through asynchronous JavaScript requests (AJAX), which are then stored and executed when accessed by other users. In the case of Google Gruyere, the vulnerability was discovered in the AJAX functionality used for loading snippet content. Attackers could exploit this vulnerability by crafting a malicious snippet containing a script payload that would be executed when retrieved via an AJAX request.

**PROOF OF CONCEPT:**

During the penetration test, it was observed that Google Gruyere utilizes AJAX principles to implement manual refresh functionality on the home and snippets page. When a user clicks the refresh link, Gruyere fetches the **feed.gtl** file, which contains refresh data for the current page. The client-side script then uses the browser DOM API to insert the new snippets into the page, based on the fetched response. Attackers can exploit this process by injecting malicious scripts into the **feed.gtl** response, which are then evaluated and executed on the client side. The vulnerable code section in **feed.gtl** includes each user's first snippet into the response without proper sanitization or validation, making it susceptible to XSS injection attacks. By injecting a crafted payload into the response, attackers can execute arbitrary JavaScript code in the context of other users' sessions, leading to unauthorized actions, data theft, and further exploitation of the system.

1. The vulnerability can be demonstrated by injecting the following payload into the response fetched via AJAX requests on the feed.gtl source code:  
   “<span style="display:none;">" + (alert(1), "") + "</span>your base”.   
   
2. A screenshot of a computer

   Description automatically generatedWhen this payload is injected and the page is refreshed, a pop-up dialog with the message "1" will appear, indicating successful execution of the injected script.

**RECOMMENDED MITIGATIONS:**

1. Server-Side Input Sanitization:

* Implement server-side input sanitization to cleanse user-supplied data before storing or rendering it within the application.
* Apply strict validation and filtering mechanisms to remove or encode potentially malicious characters, including single and double quotes.
* Ensure that all user-generated content, such as snippets, is properly sanitized before being included in AJAX responses.

1. Escape Single and Double Quotes:

* Modify the server-side code to escape single and double quotes in addition to HTML entities when generating JSON responses for AJAX requests.
* Use appropriate escaping techniques, such as backslashes or HTML entity encoding, to prevent script injection and ensure the integrity of JSON data.

1. Client-Side JSON Parsing:
   * Refrain from using the eval() function in client-side JavaScript code, as it can introduce security vulnerabilities by executing untrusted code.Replace the usage of the eval() function with the safer alternative, JSON.parse(), to convert JSON strings into JavaScript objects. JSON.parse() provides a secure and reliable method for parsing JSON data without the risks associated with executing arbitrary JavaScript code.

More information on why not to use eval() can be found here: https://www.digitalocean.com/community/tutorials/js-eval

**VULN-006: Reflected XSS via AJAX (MEDIUM)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:L

**RISK:**

* General: Medium
* Probability: Medium
* Severity: Low
* Fix Effort: Low

**DESCRIPTION:** Reflected XSS via AJAX is a medium-severity vulnerability that allows attackers to inject and execute malicious scripts through AJAX requests, leading to the execution of arbitrary code in the context of a web application. In the case of Google Gruyere, the vulnerability was identified in the AJAX functionality used for processing search queries. Attackers could exploit this vulnerability by crafting a malicious search query containing a script payload, which would be executed when the search results are displayed to the user. Unlike stored XSS, where the malicious payload is stored on the server and served to all users, reflected XSS involves the injection of scripts directly into client-side code, typically through URL parameters.

**PROOF OF CONCEPT:**

To exploit this vulnerability, attackers craft URLs containing JavaScript payloads and trick victims into clicking on them. When the victim clicks on the malicious URL, the payload gets executed in the context of the victim's session. Unline the stored XSS via AJAX, the vulnerability resides only in the client-side code. The code processes AJAX responses and inserts them into the DOM without proper sanitization. The vulnerability is found in the code section responsible for fetching and displaying snippets based on user IDs.

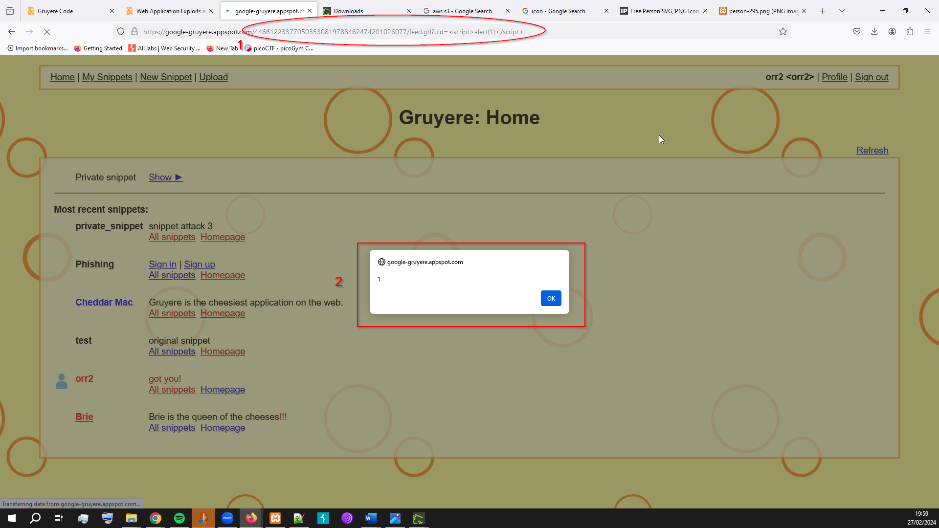
1. We can create a URL with a crafted payload in the feed.gtl user id attribute feed.gtl?uid=<script>alert(1)</script>

https://google-gruyere.appspot.com/ 446612233770503530819788462474201026077/feed.gtl?uid=<script>alert(1)</script>

or URL-encoded:

https://google-gruyere.appspot.com/ 446612233770503530819788462474201026077/feed.gtl?uid=%3Cscript%3Ealert(1)%3C/script%3E

1. When we click on the URL it triggers the AJAX request, the JavaScript payload embedded in the URL parameter gets executed in the browser and a pop up will show up with the alert “1”.



This can lead to various malicious actions, such as stealing session cookies, redirecting users to phishing sites, or performing unauthorized actions on behalf of the victim.

**RECOMMENDED MITIGATIONS:**

1. **Implement HTML Sanitization:** Integrate a robust HTML sanitizer into the application to sanitize user-generated content effectively. Considering the use of Python and the Django template language in Google Gruyere, it's recommended to utilize well-established HTML sanitization libraries that are specifically designed for Python-based web applications.

Bleach is a reputable HTML sanitizer library for Python that provides comprehensive sanitization capabilities. It is capable of performing essential sanitization tasks to prevent XSS attacks by removing or escaping potentially malicious HTML and JavaScript content from user input. <https://github.com/mozilla/bleach>

Since Google Gruyere is built on a technology similar to Django's GTL, it's advisable to complement Bleach with a Django-specific HTML sanitizer such as Leverage Django HTML Sanitizer. This additional layer enhances security by ensuring that sanitized content remains compatible with Django's template system. <https://github.com/ui/django-html_sanitizer>

1. **Utilize Template Languages:** Instead of relying on custom sanitization solutions, leverage template languages provided by frameworks like Django. Template systems often include built-in security mechanisms to automatically sanitize user input and prevent XSS vulnerabilities.  
   Django Template Language can be found here: <https://docs.djangoproject.com/en/4.0/topics/templates/>

**VULN-007: Elevation of Privilege (HIGH)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H

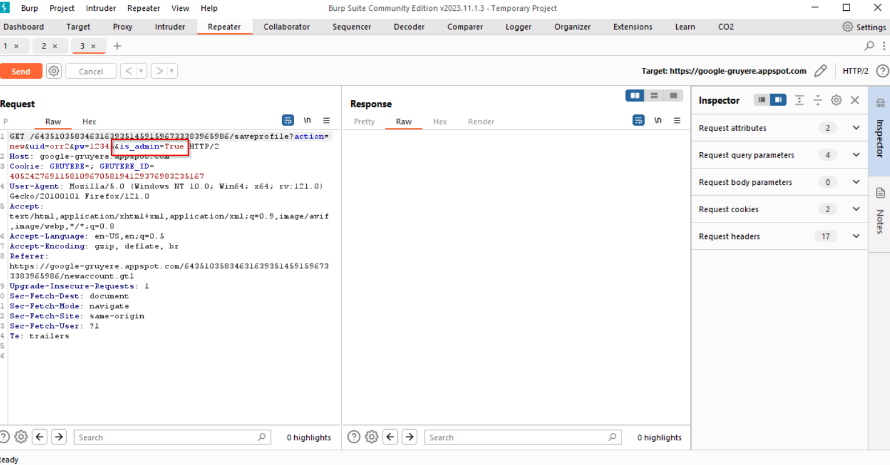
**RISK:**

* General: High
* Probability: High
* Severity: High
* Fix Effort: High

**DESCRIPTION:** Elevation of Privilege is a critical vulnerability that allows attackers to gain unauthorized access to elevated privileges within the Google Gruyere application. This vulnerability arises due to improper access controls and insufficient validation of user permissions. By exploiting this vulnerability, attackers can escalate their privileges from a regular user to an administrative user, granting them unrestricted access to sensitive features and data within the application.

**PROOF OF CONCEPT:**

During the penetration test, it was discovered that Google Gruyere did not adequately enforce access controls for certain administrative functions. While opening a new user I was able to intercept the request via burp suite and edit it to give admin access to the user.

1. Create a new user in Google Gruyere.
2. Intercept the request using burpsuite.
3. Edit the request and change is\_author to is\_admin  
   GET /446612233770503530819788462474201026077/saveprofile?action=new&uid=orr2&pw=12345&is\_author=True HTTP/2
4. Observe that the new user has access to administrative features, indicating successful exploitation of the vulnerability.

**RECOMMENDED MITIGATIONS:**

1. **Implement Granular Access Controls:**
   * Enforce granular access controls meticulously tailored to each administrative function and user role. Adopting a fine-grained approach ensures that privileges are judiciously assigned and strictly aligned with the principle of least privilege.
2. **Multi-Factor Authentication (MFA):**
   * Augment the authentication mechanism with robust multi-factor authentication (MFA) protocols to fortify user authentication and deter unauthorized access attempts. MFA adds an additional layer of security by necessitating multiple forms of verification before granting access to sensitive functionalities.
3. **Continuous Security Monitoring:**
   * Institute robust security monitoring mechanisms capable of detecting and promptly responding to anomalous access patterns and suspicious activities indicative of privilege escalation attempts. Leveraging advanced intrusion detection systems (IDS) and security information and event management (SIEM) tools can enhance the organization's threat detection capabilities.
4. **Regular Security Audits and Assessments:**
   * Conduct routine security audits and comprehensive vulnerability assessments to proactively identify and remediate access control weaknesses and other security vulnerabilities. Regular assessments serve as an invaluable proactive measure in safeguarding against potential exploitation by malicious actors.

**VULN-008: Cookie Manipulation (MEDIUM)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:L/A:N

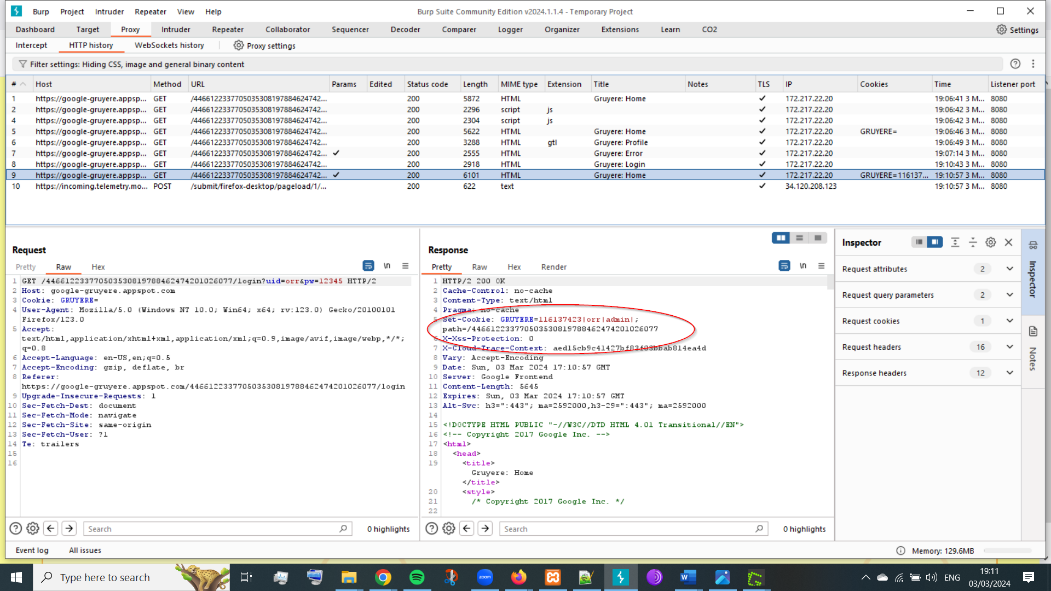
**RISK:**

* General: Medium
* Probability: Medium
* Severity: High
* Fix Effort: Medium

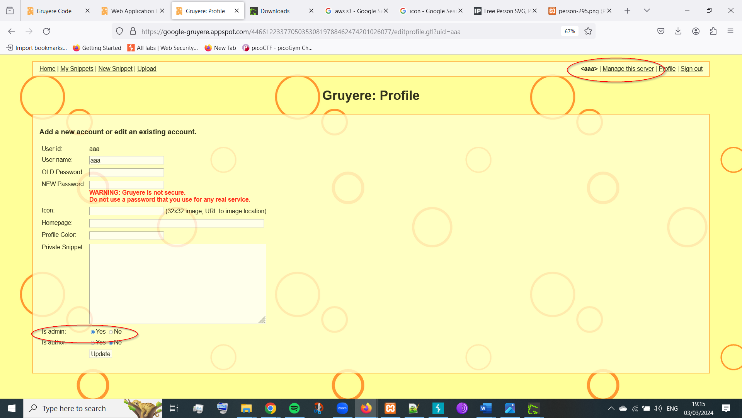
**DESCRIPTION:** Cookie Manipulation is a medium-severity vulnerability that arises due to improper handling and manipulation of cookies within the Google Gruyere application. Cookies are used to store user session information and preferences, but if not properly secured, they can be manipulated by attackers to perform unauthorized actions or access sensitive data. By exploiting this vulnerability, attackers can tamper with session cookies, hijack user sessions, and impersonate legitimate users, leading to potential data breaches and unauthorized access to the application.

**PROOF OF CONCEPT:**

During the penetration testing of Google Gruyere, a critical vulnerability related to insufficient enforcement of access controls was identified. This vulnerability enabled an attacker to manipulate session cookies, thereby granting unauthorized access to administrative functionalities within the application. The following details outline the steps involved in the exploitation of this vulnerability:

1. Log in to Google Gruyere as a regular user.
2. Use burp suite to inspect the session cookies associated with the user's session.
3. Sign up for a new account under the username aaa|admin|author.
4. A screenshot of a computer

   Description automatically generated Analyze the "Set-Cookie" header to verify cookie attributes and path restrictions, focusing on consistency across the requests.
5. Check the assigned privileges for the manipulated user account to confirm administrative access.



**RECOMMENDED MITIGATIONS:**

1. **Enhanced Cookie Security:**
   * Implement robust cookie handling mechanisms to mitigate cookie manipulation attacks. Ensure that cookies are securely generated and parsed, and enforce strict validation of cookie contents to prevent unauthorized modifications.
   * Encrypt sensitive cookie data or use cryptographic hashing with secure algorithms to enhance the integrity and confidentiality of cookie information.
   * Enforce strict cookie path and domain restrictions to prevent unauthorized access and cookie leakage across different application components or domains.
   * Implement secure randomization techniques for generating session identifiers to reduce predictability and prevent session fixation attacks.
2. **Input Sanitization and Validation:**
   * Apply rigorous input validation and sanitization techniques to all user-supplied data, including usernames and other parameters used in cookie generation and parsing.
   * Use server-side input validation routines to sanitize user inputs and reject malformed or malicious data that may attempt to bypass security controls.
   * Employ context-aware input validation to ensure that user inputs adhere to expected formats and patterns, reducing the risk of injection attacks and malicious exploitation.
3. **Secure Hashing Functions:**
   * Replace the insecure Python **hash()** function with a secure cryptographic hashing algorithm (e.g., SHA-256) for generating cookie hashes. Utilize well-established cryptographic libraries and best practices to implement secure hashing functions.
   * Ensure that the hashing algorithm used for generating cookie hashes is resistant to collision attacks and provides sufficient entropy to mitigate brute-force and rainbow table attacks.
4. **Static Cookie Secret Enhancement:**
   * Enhance the security of the static cookie secret by generating a cryptographically strong random value as the cookie secret initialization vector or salt. Avoid using static or predictable values for the cookie secret to prevent cryptographic weaknesses and exploitation.
5. **Implement Content Security Policies (CSP):**
   * Utilize Content Security Policies (CSP) to mitigate the impact of client-side attacks, such as XSS, by restricting the execution of scripts and controlling the sources from which content can be loaded.
   * Define strict CSP directives to prevent unauthorized script execution and enforce the use of secure cookie attributes (e.g., SameSite and HttpOnly) to enhance cookie security and prevent cookie-based attacks.

**VULN-009: Cross-Site Request Forgery (XSRF) (HIGH)**

CVSS: CVSS:3.1/AV:N/AC:L/PR:H/UI:R/S:U/C:H/I:H/A:H

RISK:

General: High

Probability: High

Severity: High

Fix Effort: Medium

DESCRIPTION:

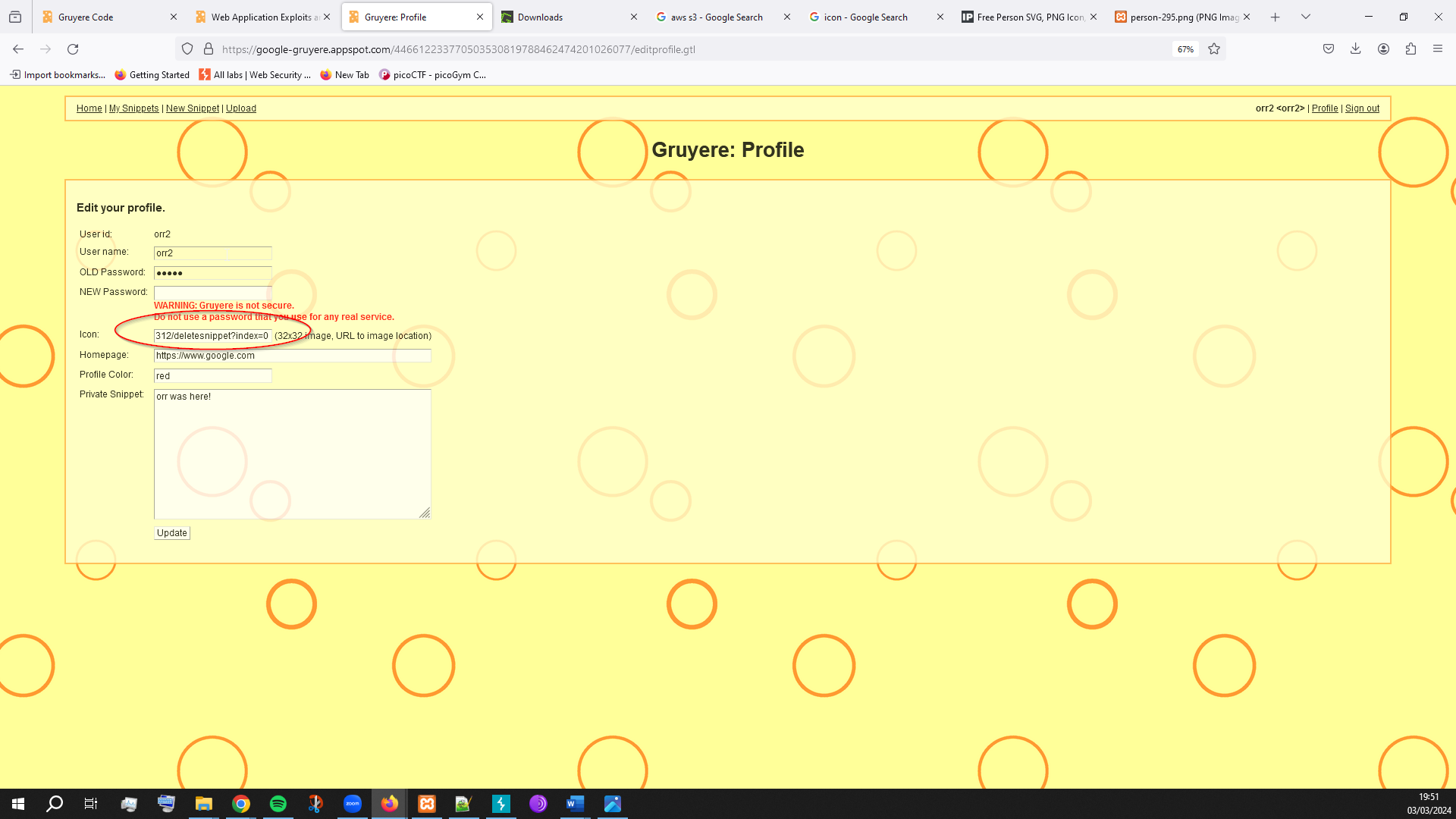
Cross-Site Request Forgery (XSRF) is a high-severity vulnerability that allows attackers to trick authenticated users into executing unintended actions on Google Gruyere. This vulnerability occurs when the web application does not sufficiently validate or authenticate incoming requests, allowing attackers to forge malicious requests on behalf of authenticated users. By exploiting this vulnerability, attackers can perform unauthorized actions, such as changing account settings, making purchases, or performing actions with elevated privileges, leading to data breaches, financial losses, and reputational damage.

**PROOF OF CONCEPT:**

We can find The URL used to delete a snippet in Google Gruyere in burpsuite simply by deleting a snippet: /deletesnippet?index=0. This endpoint allows users to delete a snippet by specifying its index in the user's snippet list. However, the lack of proper CSRF protection leaves this functionality vulnerable to exploitation by attackers. To simulate a CSRF attack, we can craft a malicious URL containing the delete snippet endpoint and embed it in a location where the victim is likely to visit, such as a phishing website, malicious email, or compromised web page. When accessing the malicious URL while authenticated to Google Gruyere, the delete snippet request is automatically executed in the context of the session, leading to the unintended deletion of their snippet.

1. Generate a malicious URL containing the delete snippet endpoint with the index parameter set to 0:

https://google-gruyere.appspot.com/ 446612233770503530819788462474201026077/deletesnippet?index=0

1. Utilize the "Edit Profile" feature in Google Gruyere to modify the user's profile and insert the malicious URL into the Gruyere icon field. This action ensures that the malicious URL is triggered whenever the victim's profile is viewed or accessed.  
   
2. Upon accessing the home page, the browser automatically sends a request to the delete snippet endpoint, resulting in the inadvertent deletion of the victim's snippet without their knowledge or consent.

**RECOMMENDED MITIGATIONS:**

1. Modify the application logic to utilize POST requests instead of GET requests for sensitive operations such as deleting snippets. POST requests should be employed when data mutation or state change is involved, ensuring that sensitive actions cannot be triggered via simple URL manipulation.
2. Introduce Cross-Site Request Forgery (CSRF) tokens as a defense mechanism against CSRF attacks. These tokens should be unique per session and per form submission, effectively preventing attackers from crafting malicious requests that can be executed by authenticated users. The tokens should be cryptographically secure and unpredictable, making it challenging for attackers to guess or replicate valid tokens.
3. Validate the CSRF tokens embedded in form submissions on the server-side to ensure their authenticity and integrity. Additionally, enforce token expiry mechanisms to mitigate the risk of token reuse or exploitation in the event of token leakage. Consider implementing token validity periods, such as expiring tokens after 24 hours, to limit their lifespan and reduce the window of vulnerability.
4. Include CSRF tokens as hidden fields within HTML forms to bind them to specific user sessions and form submissions. Ensure that these tokens are securely generated and transmitted to prevent tampering or interception by malicious actors. By embedding CSRF tokens directly in form submissions, the application can validate the authenticity of requests and reject unauthorized actions.
5. Utilize cryptographically secure methods for generating CSRF tokens, such as using random number generators or cryptographic hash functions. Avoid predictable patterns or algorithms that could enable attackers to predict or brute-force valid token values. Employing strong token generation practices enhances the security and resilience of the CSRF protection mechanism.
6. Leverage built-in CSRF protection mechanisms provided by popular web application frameworks, such as AngularJS's CSRF protection features. Framework-level CSRF protection often includes robust token generation and validation mechanisms, reducing the complexity of implementing custom CSRF defenses and ensuring adherence to established best practices.

**VULN-010: Cross Site Script Inclusion (XSSI) (MEDIUM)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:L/I:L/A:L

**RISK:**

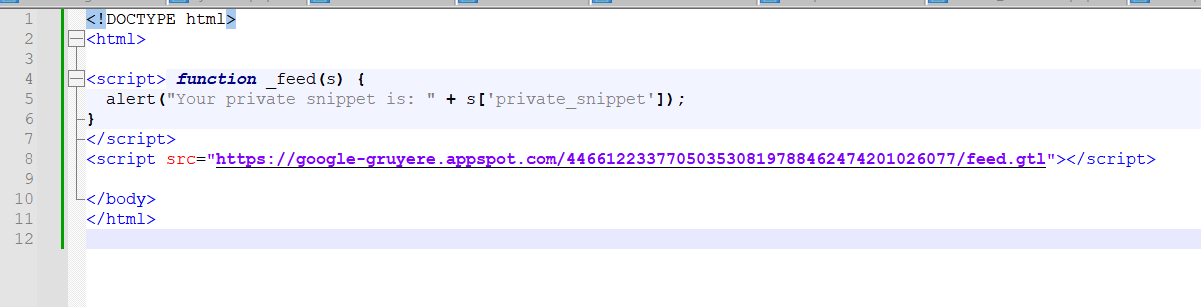
* General: Medium
* Probability: Medium
* Severity: Low
* Fix Effort: Medium

**DESCRIPTION:** Cross-Site Script Inclusion (XSSI) is a medium-severity vulnerability that occurs when a web application includes untrusted JavaScript code from external sources without proper validation or sanitization. This vulnerability can be exploited by attackers to inject malicious scripts into the application, leading to client-side attacks such as cross-site scripting (XSS), data theft, or session hijacking. While XSSI does not directly compromise the integrity or confidentiality of the application, it can facilitate other attacks and undermine the trustworthiness of the web application.

**PROOF OF CONCEPT:**

In the case of Google Gruyere, the vulnerability manifests in the **/feed.gtl** endpoint, which discloses information about private snippets without proper access control enforcement. Attackers can exploit this vulnerability to include malicious JavaScript code in legitimate web pages, thereby extracting and exfiltrating sensitive data stored within private snippets of unsuspecting users.

1. We craft a malicious HTML page containing JavaScript code which leads to feed.gtl to exploit the XSSI vulnerability:



1. When opening the html page an alert will pop up with the content of the private snippet which is our data leak.   
   A screenshot of a computer

   Description automatically generated

**RECOMMENDED MITIGATIONS:**

1. Utilize Cross-Site Request Forgery (CSRF) tokens to ensure that JSON responses containing sensitive data are only returned to authorized pages within your application. CSRF tokens help prevent unauthorized third-party sites from making malicious requests on behalf of authenticated users.
2. Limit JSON response pages to only support POST requests, which mitigates the risk of XSSI attacks by preventing the malicious script from being loaded via a script tag. By enforcing the use of POST requests, you restrict the execution of sensitive JSON responses to trusted sources within your application.
3. Implement content sanitization techniques to ensure that the JSON script content returned in responses is not executable when included in a web page. One approach is to append a non-executable prefix or wrapper to the JSON script content, which prevents it from being interpreted and executed as JavaScript code. This technique helps mitigate the risk of XSSI attacks by rendering the script inert when included in a web page.
4. Validate the Origin and Referrer headers in incoming requests to verify the origin of the request and ensure that it originates from an authorized domain. By validating these headers, you can prevent unauthorized cross-origin requests from accessing sensitive JSON responses and mitigate the risk of XSSI attacks.
5. Enforce the Same-Origin Policy (SOP) to restrict the execution of scripts to the same origin as the web page. By adhering to SOP guidelines, you prevent scripts from accessing JSON responses served from different origins, thereby reducing the likelihood of XSSI vulnerabilities.

**VULN-011: Path Traversal (HIGH)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N

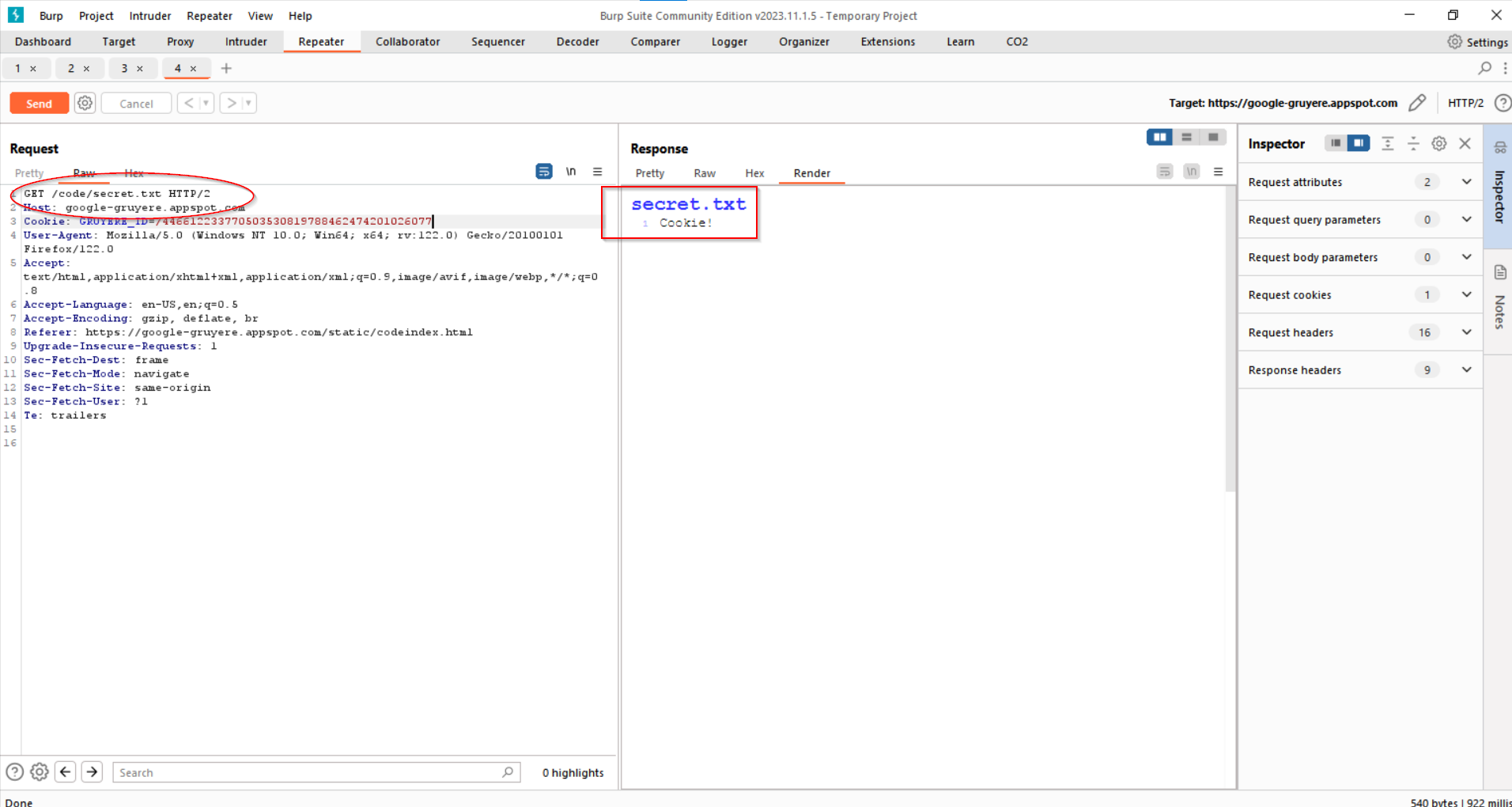
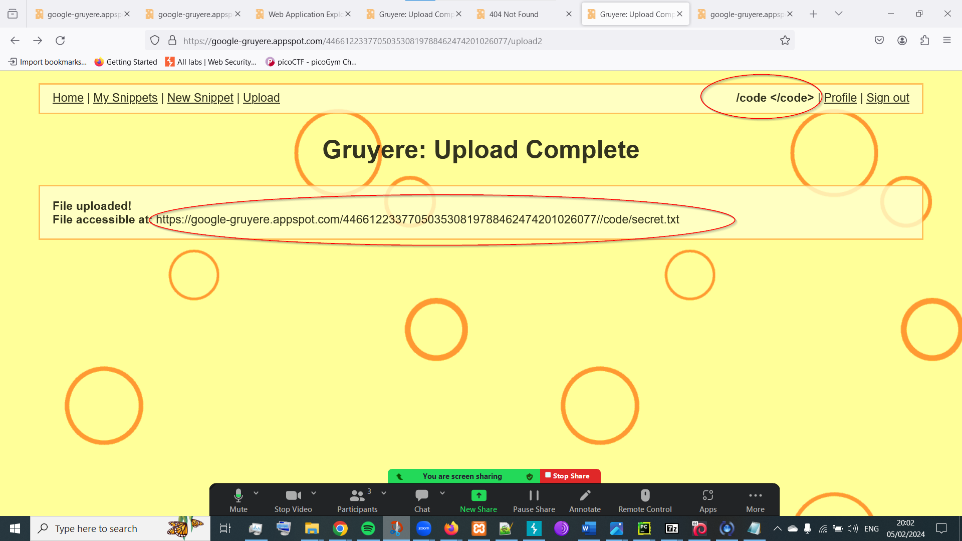
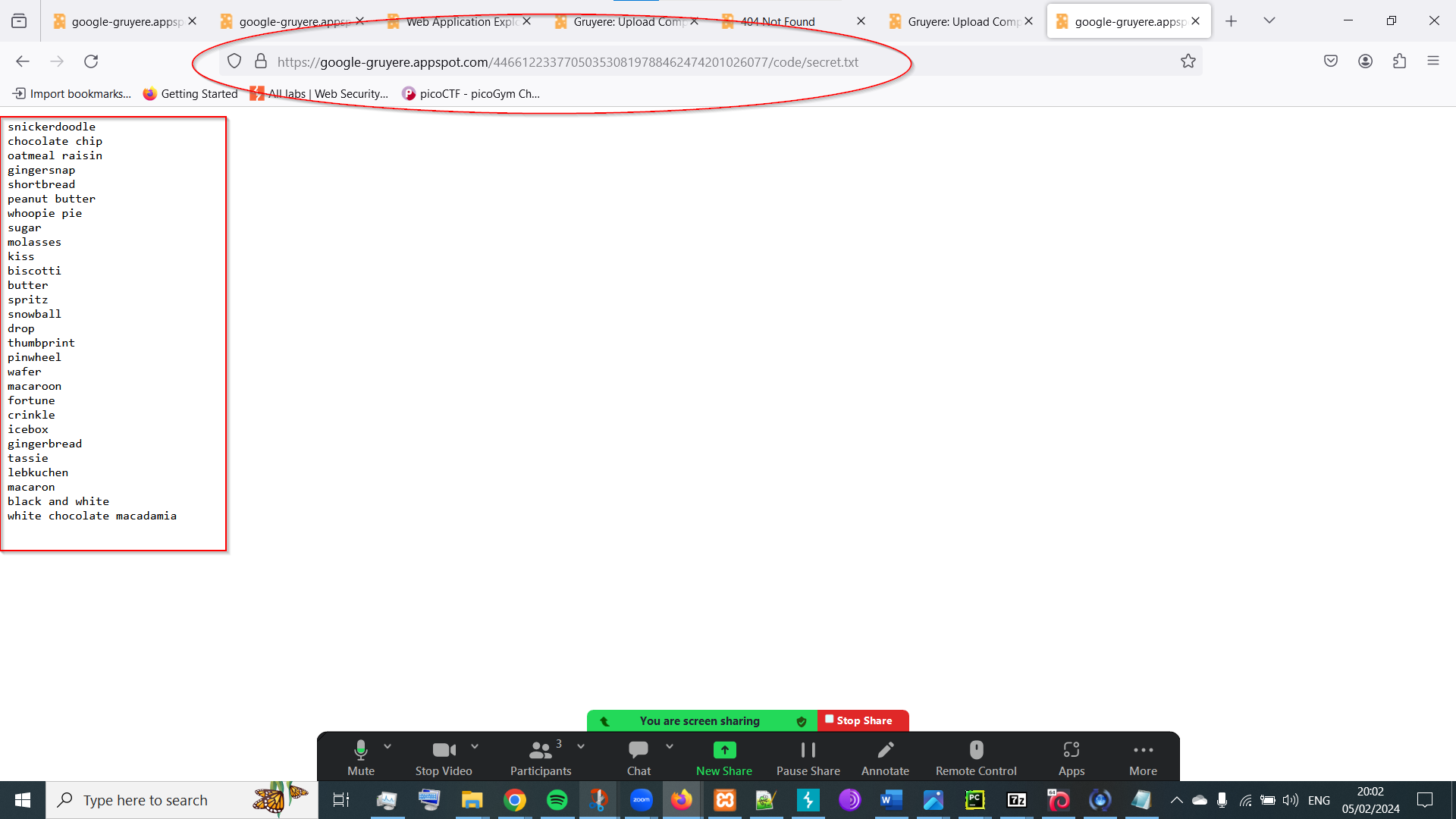
**RISK:**

* General: high
* Probability: Medium
* Severity: High
* Fix Effort: Medium

**DESCRIPTION:** Path Traversal is a medium-severity vulnerability that allows attackers to access sensitive files and directories on the server by manipulating file paths. In Google Gruyere, improper input validation and lack of access controls enable attackers to traverse directories and retrieve files containing sensitive information, such as configuration files, user data, or application logs. By exploiting this vulnerability, attackers can gain insights and access into the internal workings of the application and potentially escalate their attacks.

**PROOF OF CONCEPT:**

Using Burp Suite to intercept HTTP requests, it was observed that Gruyere is vulnerable to Path Traversal attacks. By manipulating the URL path, attackers can navigate beyond the intended directory structure and access files that should be restricted. Initially, attempting to access the "upload.gtl" module exhibits the expected behavior, where access to the code is restricted. However, by appending additional directory traversal sequences to the URL (e.g., "/upload.gtl/test" and "/upload/test/../"), it was possible to traverse the directory hierarchy and access files that should be inaccessible. Exploiting the vulnerability further, the content of the "secret.txt" file was accessed by appending “/..%2fsecret.txt " to the URL. This revealed sensitive information, confirming the presence of the Path Traversal vulnerability. With the ability to access sensitive files, we manipulate can the content of the "secret.txt" file by crafting a malicious request.

1. Access to the secret file is enabled via adding “/..%2fsecret.txt " to the url, reveals the content of the "secret.txt" file, is "Cookie!".  
   
2. Exploiting this vulnerability further, we create a new user under "/code" and creating a new file named "/secret.txt" with desired contents (in this case, a list of cookies).
3. Upon following the upload link, we observe that the content of the "secret.txt" file has been successfully changed, demonstrating the exploitation of the Path Traversal vulnerability to achieve unauthorized access and information disclosure.  
   

**RECOMMENDED MITIGATIONS:**

1. Store sensitive files outside the document root directory to prevent direct access via HTTP requests. Limit access to these files through appropriate access controls and authentication mechanisms.
2. Implement strict input validation and sanitization mechanisms to filter out any potentially malicious input from user-supplied data. Remove all but the known good data and filter out meta-characters that could be used to manipulate file paths or execute unauthorized commands.
3. Specifically remove occurrences of ".." and "../" from any user input or parameters used in file contexts to prevent directory traversal attacks. This prevents attackers from navigating up the directory hierarchy to access files outside the intended scope.
4. Configure your web server to restrict public access to only necessary directories and resources required for the proper functioning of the website. Implement access controls, permissions, and directory listings to limit exposure to sensitive files and directories.

**VULN-012: Denial of Service (DoS) (HIGH)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

**RISK:**

* General: High
* Probability: High
* Severity: High
* Fix Effort: Medium

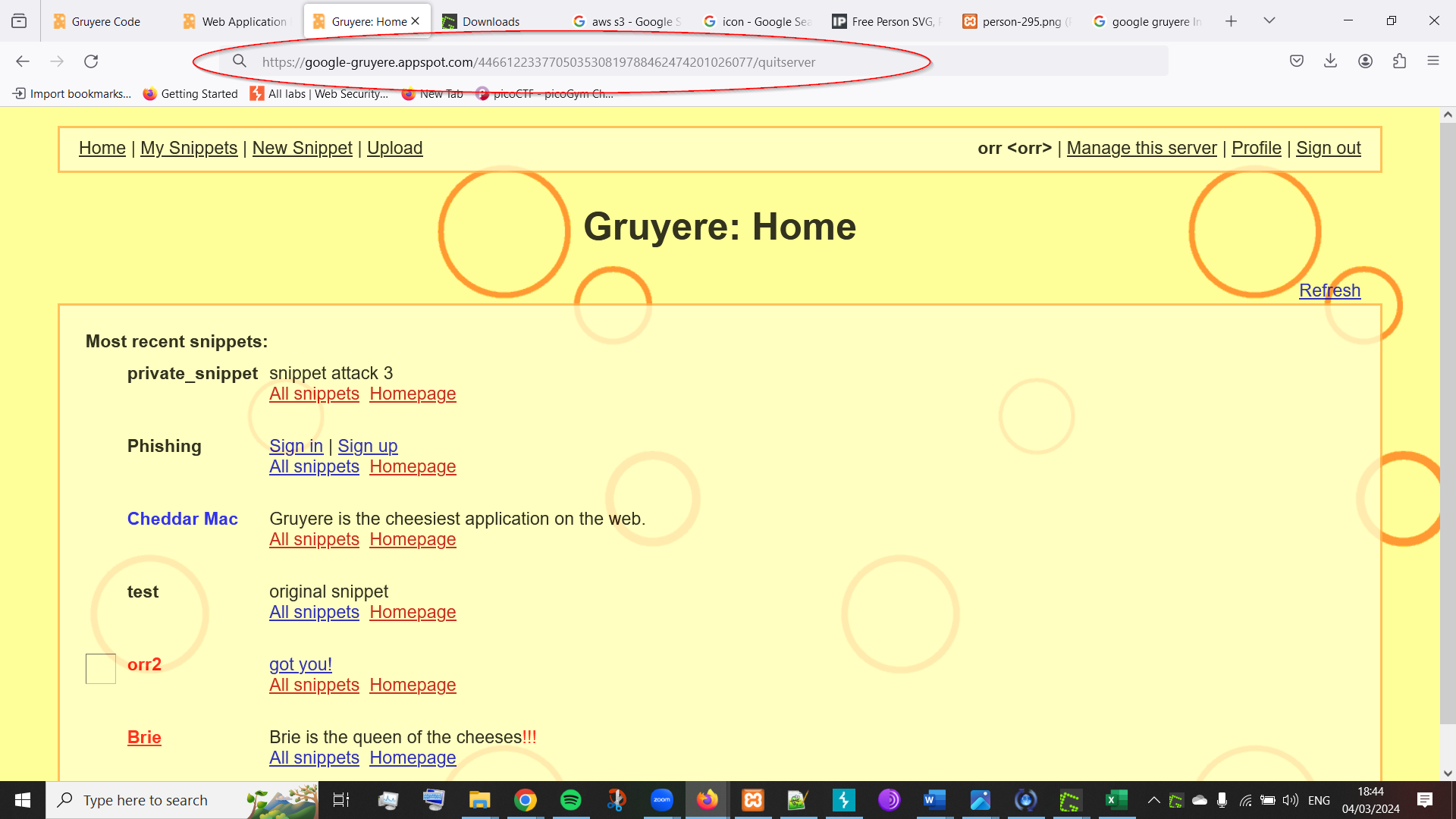
**DESCRIPTION:** The Denial of Service (DoS) attack against the Gruyere server involves two distinct methods: "Quit the Server" and "Overloading the Server." In the first method, an admin user can trigger a server shutdown command by accessing the "Quit the server" functionality in the "Manage this server" section, a regular user can simply add “/quitserver” to the end of the the URL .   
  
https://google-gruyere.appspot.com/446612233770503530819788462474201026077/ quitserver  
  
The second method involves exploiting a vulnerability in the server by overloading it with continuous requests, causing it to enter an infinite loop state. This is achieved by manipulating the "menubar.gtl" file, which is present on every page of the application, to create a loop that repeats indefinitely.

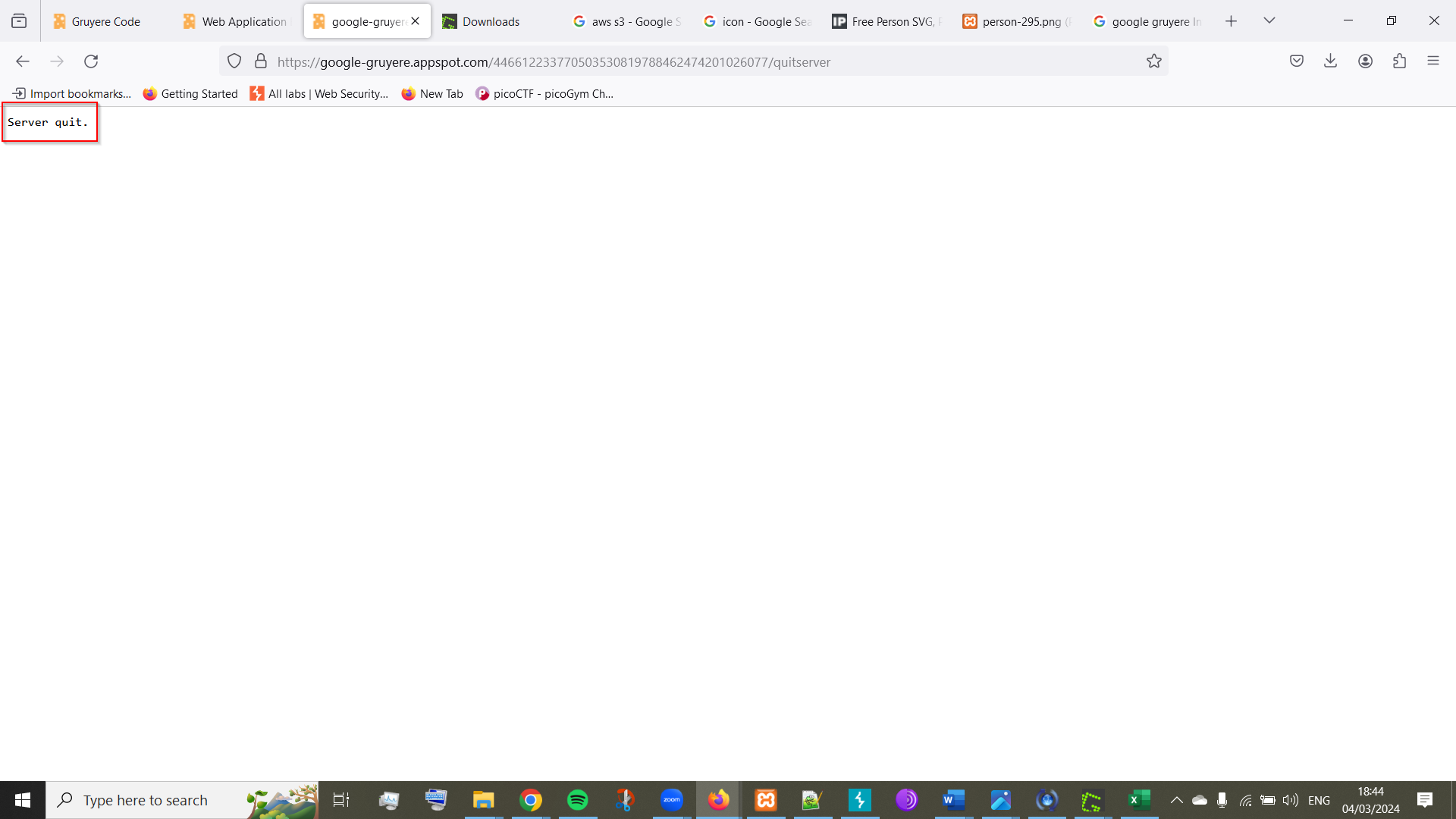
**PROOF OF CONCEPT:**

**Quit the Server (Regular User) :**

1. Log in as user.

2. Navigate to the URL.

3. Add “/quitserver” to the end of the the URL .   


4. Observe the server shutdown, resulting in denial of service. 

**Overloading the Server:**

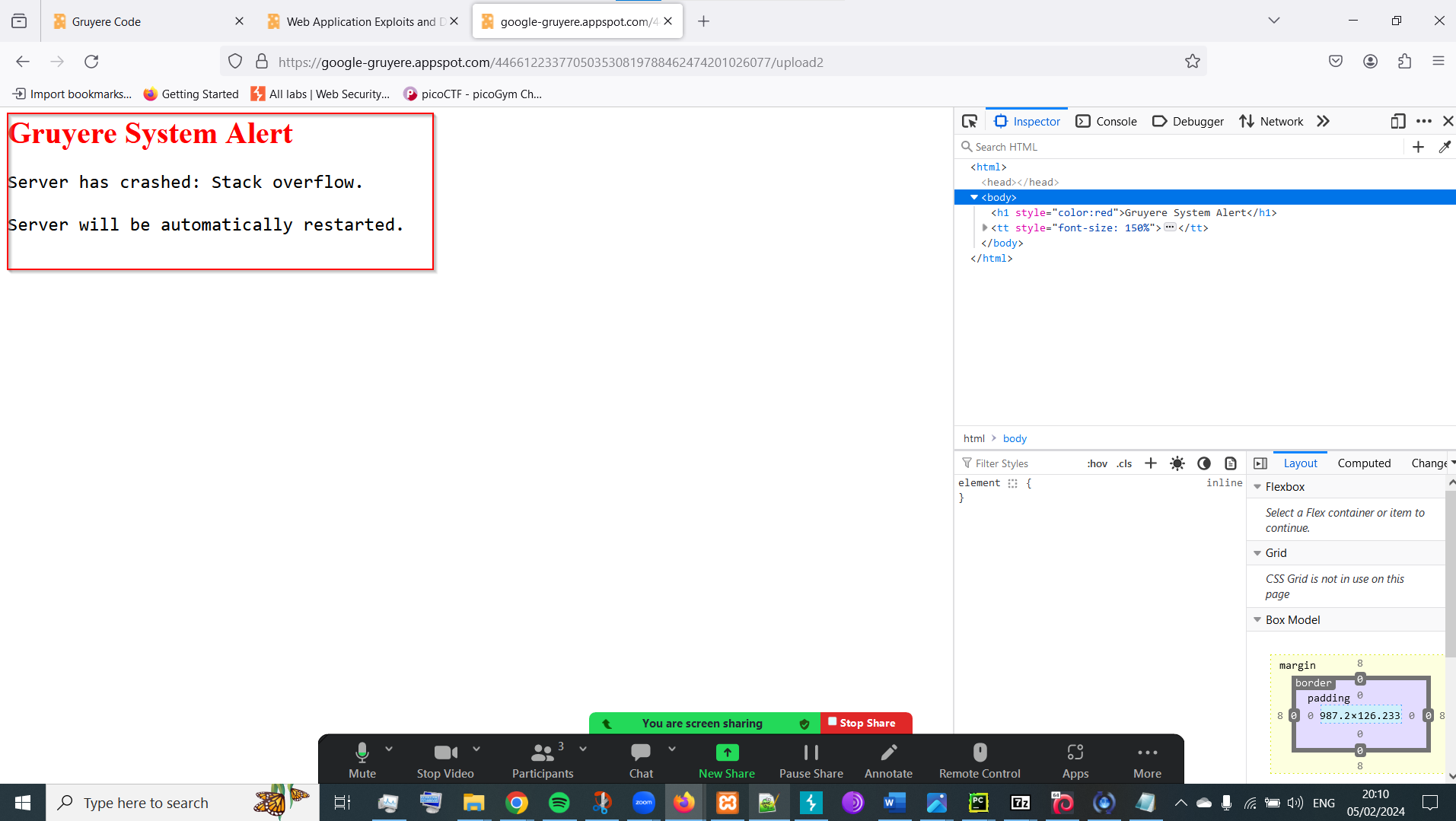
1. Create a new file under the name "menubar.gtl" that contains a loop that repeats indefinitely.

[[include:menubar.gtl]]DoS[[/include:menubar.gtl]]

2. Exploit a Path Traversal vulnerability to upload a specially crafted "menubar.gtl" file to the server (as shown in the previous vulnerability).

3. Access any page within the application to trigger the inclusion of the modified "menubar.gtl" file.

4. Observe the server becoming overloaded as the infinite loop continuously executes.



1. To stop the loop and regain access to the application, use the provided "reset" functionality.

**RECOMMENDED MITIGATIONS:**

1. The potential fix has been described earlier in the Path Traversal section.

**VULN-013: Code Execution (CRITICAL)**

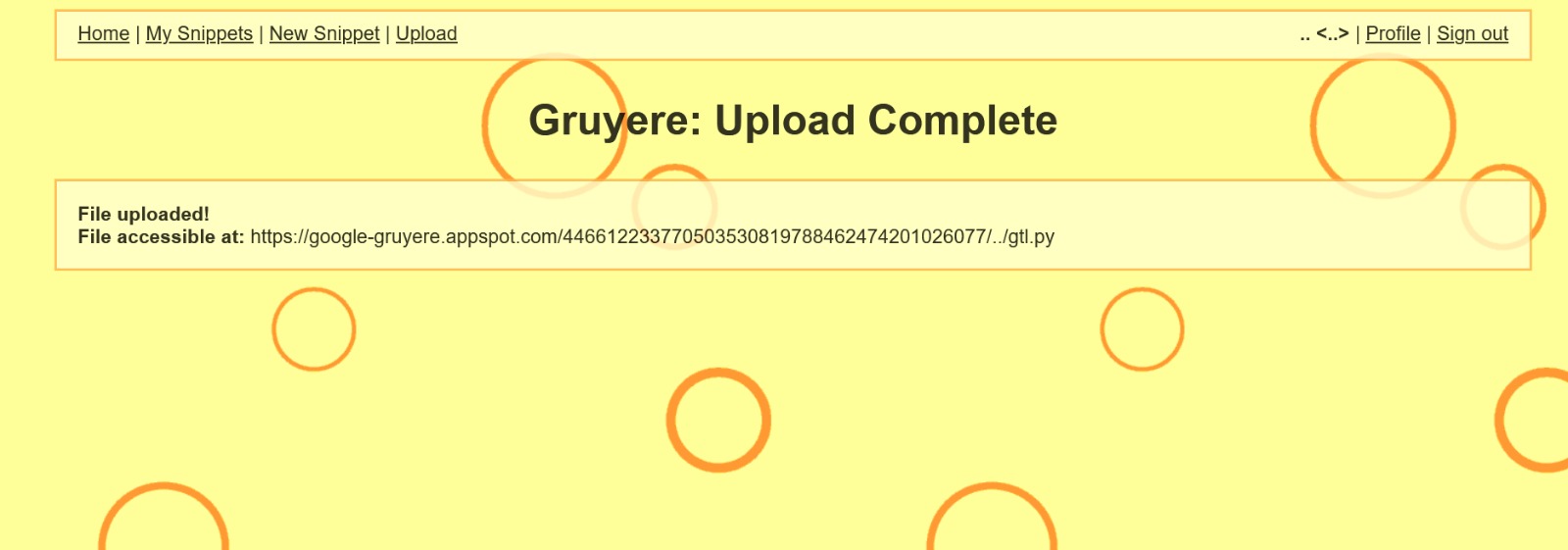
**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

**RISK:**

* General: Critical
* Probability: High
* Severity: High
* Fix Effort: Medium

**DESCRIPTION:** The code execution attack against the Gruyere application involves leveraging previously identified vulnerabilities, namely Path Traversal and Denial of Service (DoS), to gain unauthorized access to the server and execute arbitrary code. By exploiting these vulnerabilities, we can manipulate the GTL template files, which are integral to the functionality and presentation of the Gruyere web application. Since GTL is shaping the entire Gruyere web site modifying the GTL language can permanently alter the site and put it down. This allows to inject our own malicious code into critical components of the application, potentially leading to complete control over the server infrastructure. This penetration test is incomplete because google gruyere is a learning environment.

**PROOF OF CONCEPT:**

1. Use file upload to replace the gtl.py (my file contains the word “hello”).  
   
2. Use the previous DOS vulnerability to quit and restart the server
3. A screenshot of a computer

   Description automatically generatedGoogle gruyere shows us that the attack was successful.

**RECOMMENDED MITIGATIONS:**

1. **File Upload Sanitization**:
   * Implement proper input validation and sanitization mechanisms to block the upload of files with sensitive extensions, such as .py (Python files). This can be achieved by:
     + Restricting file upload functionality to accept only whitelisted file types.
     + Checking file extensions and MIME types to ensure they match the expected format.
     + Utilizing server-side validation to verify uploaded files against predefined criteria before allowing them to be stored or executed.
2. **Path Traversal Vulnerability Fix**:
   * Address the identified path traversal vulnerabilities within the Gruyere application to prevent unauthorized access to sensitive files or directories. This can be accomplished by:
     + Implementing input validation and sanitization to filter user-supplied input and prevent traversal sequences (e.g., "..") from being processed.
     + Enforcing strict file path validation to ensure that requested resources are within the intended directory structure and scope.
     + Employing proper access controls and permissions to restrict user access to only authorized directories and files.
3. **Least Privilege Principle**:
   * Modify the Gruyere application to run with minimal privileges and restrict its permissions for reading and writing files within the Gruyere directory. This can be achieved by:
     + Implementing a principle of least privilege, where the application is granted only the minimum permissions necessary to perform its intended functions.
     + Utilizing operating system-level access controls, such as file system permissions, to restrict file access rights for the Gruyere application.
     + Running the Gruyere application with a dedicated service account or user profile that has limited access rights to the file system.

**VULN-014: Information Disclosure (High)**

**CVSS:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N  
  
**RISK:**

* General Risk: Medium
* Probability: High
* Severity: Low
* Fix Effort: Low

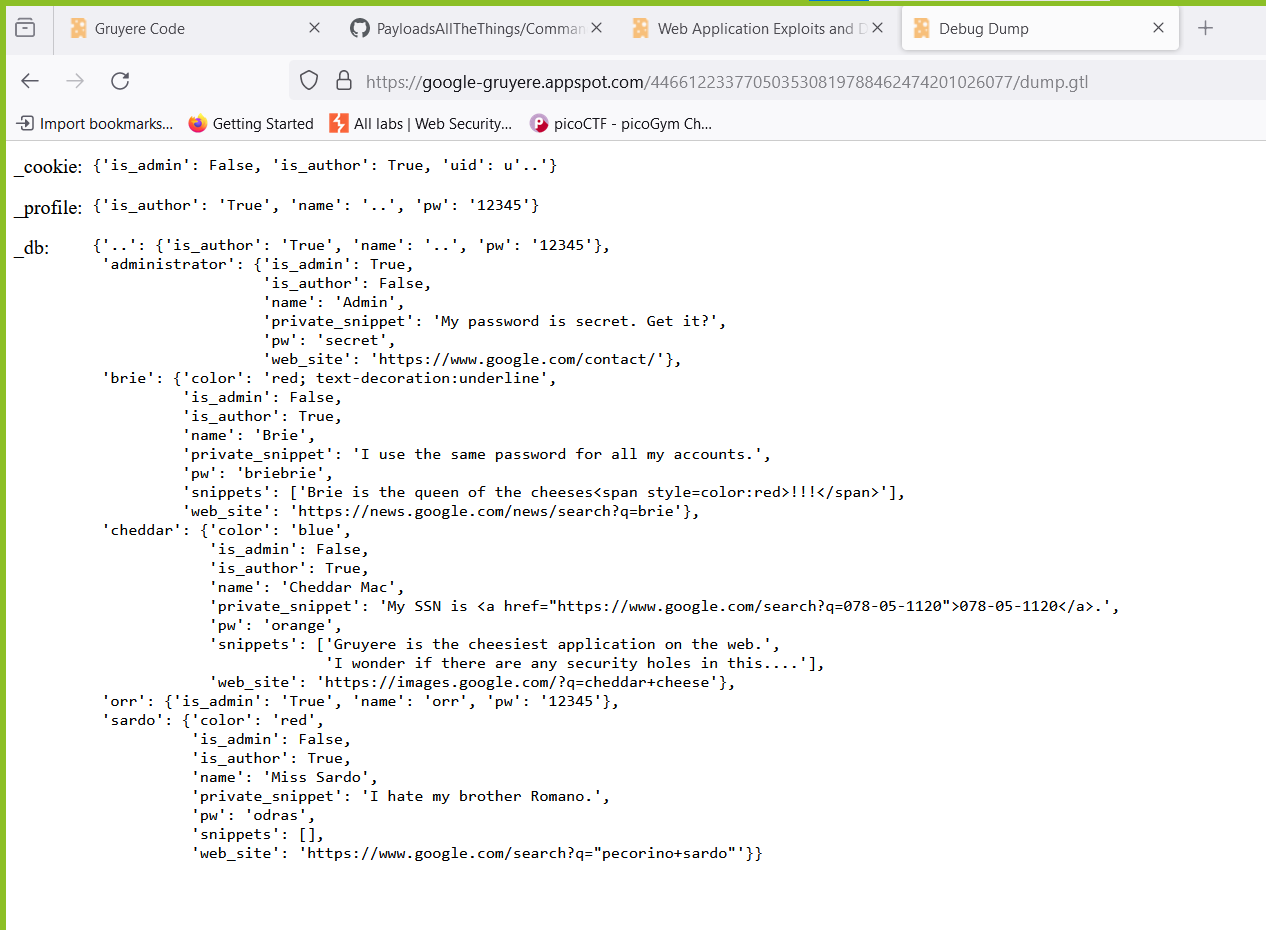
**DESCRIPTION:** The information disclosure vulnerability in Gruyere allows unauthorized access to sensitive files, such as database dumps containing clear-text usernames and passwords. By exploiting this flaw, attackers can view critical data by accessing the "dump.gtl" file or injecting code into input fields like the "new snippet" window. This vulnerability poses a medium general risk due to its high probability of exploitation, although the severity is low, and fixing it requires low effort. It highlights the need for robust access controls and input validation mechanisms to protect sensitive information from unauthorized access.

Top of Form

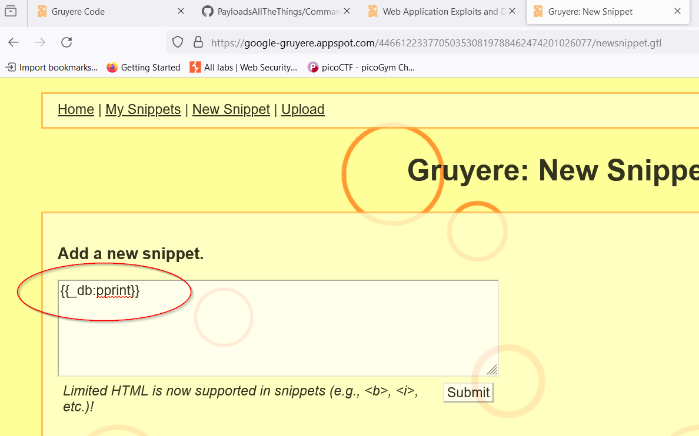
**PROOF OF CONCEPT:**

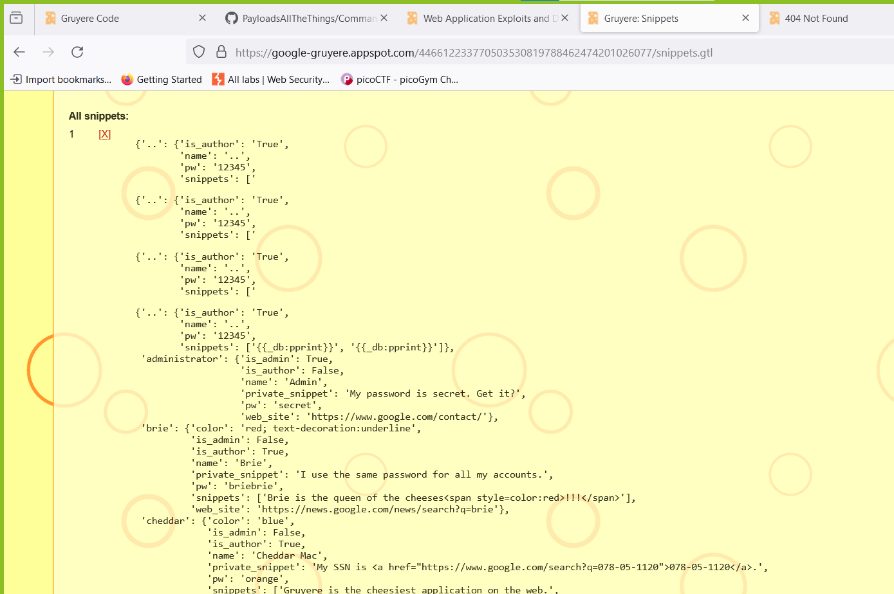
The Gruyere application contains a potentially sensitive file named "dump.gtl" that appears to function as a database dump program. Typically, a database dump contains structured data or SQL statements from a database, often used for backup or debugging purposes. Accessing the "dump.gtl" file reveals clear-text usernames and passwords, indicating a significant information disclosure vulnerability. To exploit this vulnerability, we can leverage weaknesses in the Gruyere codebase to directly access and display sensitive database content. By injecting Python functions like pprint (pretty-print) into the "new snippet" window, we can execute arbitrary code within the application's template engine, leading to the unauthorized disclosure of sensitive information.

1. Access the Gruyere application.
2. Navigate to the "/dump.gtl" URL within the application.  
     
   https://google-gruyere.appspot.com/446612233770503530819788462474201026077/dump.gtl
3. Observe the clear-text usernames and passwords exposed in the database dump.



1. Craft a new snippet using the "new snippet" window and inject the following code: **{{\_db:pprint}}**



1. Submit the snippet, and observe the database content displayed directly on the application's "my snippet" page.

**RECOMMENDED MITIGATIONS:**

1. **File Storage and Access Controls**:
   * Restrict the storage location for sensitive files such as passwords and dump files. Utilize a secure and designated directory with strict access controls to prevent unauthorized access.
   * Implement mechanisms such as IP whitelisting, port restrictions, and authentication mechanisms to control access to dump files and other sensitive resources.
2. **Password Handling**:
   * Hash and encrypt passwords before storing them in the database. Avoid storing passwords in clear text to prevent unauthorized access in the event of a data breach.
3. **Access Control for Dump Program**:
   * Restrict access to the dump program to users with administrative privileges only. Implement proper access controls and authorization mechanisms to ensure that only authorized users can execute sensitive operations.
4. **File Upload Security**:
   * Enhance file upload security by implementing strict validation and filtering mechanisms to prevent the upload of malicious files.
   * Enforce restrictions on allowed file formats and content types to mitigate the risk of uploading harmful scripts or templates.
5. **Template Parsing Vulnerability Fix**:
   * Address the design flaw in the template parsing mechanism to prevent unintended variable expansion and execution of potentially malicious code.
   * Enhance the template language to restrict database access and limit the scope of queries to mitigate the risk of SQL injection attacks and unauthorized data retrieval.

**VULN-015: Insecure JavaScript code Implementation (Medium)  
  
CVSS: 3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H**

**Risk:**

* General: Medium
* Probability: Medium
* Severity: High
* Fix Effort: Low

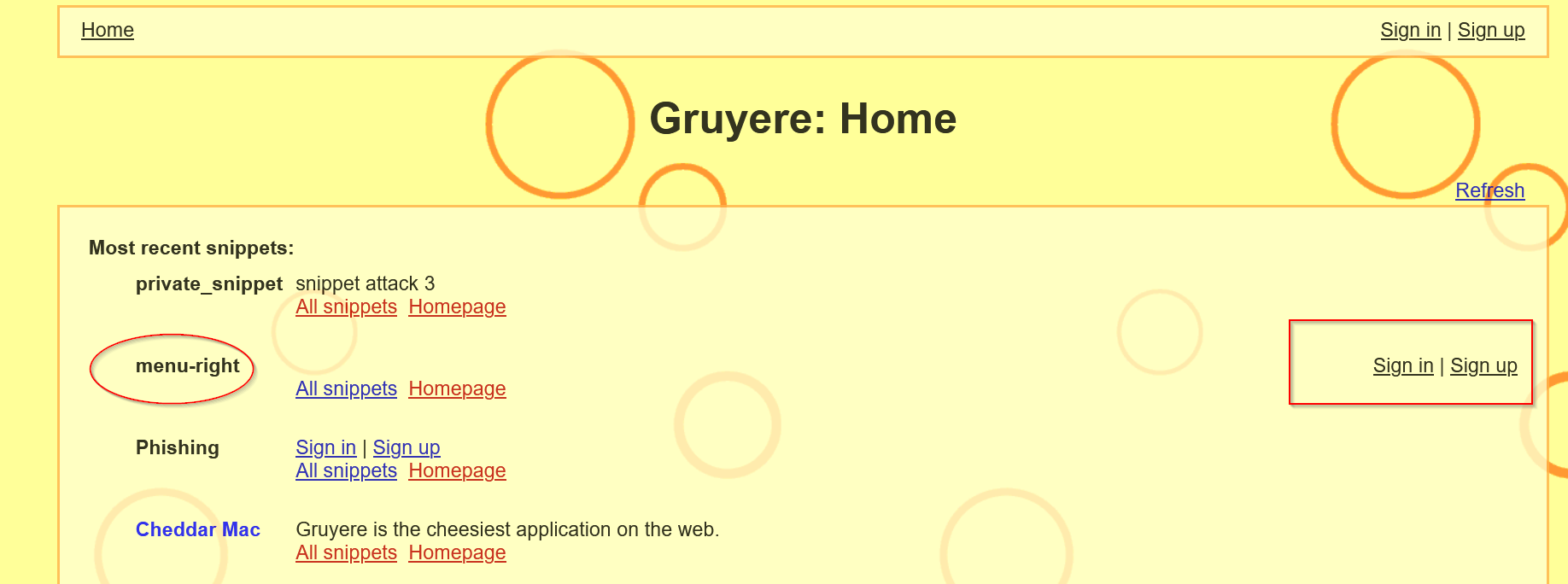
**DESCRIPTION:** The vulnerability allows an attacker to inject malicious JavaScript code into the Gruyere application, leading to the display of phishing links to users. However, successful exploitation of this vulnerability requires user interaction (clicking on the phishing links) and does not directly compromise the confidentiality, integrity, or availability of the system. Additionally, the attack complexity is low, and the attacker does not require any privileges to exploit the vulnerability. While the impact on users may vary depending on their interaction with the phishing links, the overall risk posed by this vulnerability is considered low.  
  
**PROOF OF CONCEPT:**

We exploit an insecure JavaScript implementation in Gruyere by injecting malicious code into a user profile, creating phishing links that appear legitimate to unsuspecting users. This allows to redirect users to a fraudulent site, potentially leading to data theft and compromise of sensitive information, posing a significant risk to user security and privacy.

1. Create a new user profile named "menu-right"
2. Navigate to the snippet creation section
3. Craft a snippet containing the following HTML code, which includes phishing links disguised as sign-in and sign-up links:

<a href='https://www.example.com'>Sign in</a> | <a href='https://www.example.com'>Sign up</a>



1. Submit the crafted snippet containing the malicious JavaScript code
2. Log out of the user
3. Upon successful submission, navigate to the homepage where the snippet is displayed within the Gruyere application.
4. Observe that the phishing links injected via the malicious snippet are now displayed on the page, appearing as legitimate sign-in and sign-up links.  
   

**RECOMMENDED MITIGATIONS:**

1. **Input Sanitization:**
   * Implement rigorous input validation and sanitization mechanisms to filter out and block any unauthorized JavaScript code or HTML markup submitted by users.
2. **Content Security Policy (CSP):**
   * Utilize Content Security Policy (CSP) headers to restrict the execution of inline JavaScript and mitigate the risk of XSS attacks.
3. **Code Review and Testing:**
   * Conduct thorough code reviews and security testing to identify and remediate any vulnerabilities related to JavaScript injection.
   * Implement secure coding practices and adhere to OWASP guidelines for preventing XSS vulnerabilities.