Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

Web Development with Secure HTML5 Coding Practices

Madhuri N. Gedam
Research Scholar: Department of Computer Engineering
Veermata Jijabai Technological Institute (VJTI)
Mumbai, India

Bandu B. Meshram
Professor, Department of Computer Engineering
Veermata Jijabai Technological Institute (VJTI)
Mumbai, India

***Abstract*—Due to a rapid increase in the web application's demands, its security is of prime importance throughout the development lifecycle emphasizing the protection of user data and fostering trustworthiness. Web development using the HTML5 framework and its powerful APIs such as Web Storage API, WebSockets, geolocation, local storage, and File API has emerged as a substitute for vulnerable additional plug-ins such as ActiveX, Flash, and Silverlight. This paper is focused on finding vulnerabilities and attacks in HTML5 and its APIs and provided the defense mechanisms to develop secure, robust, and reliable web development.**

Keywords— HTML5; Web Application Development; SQLI; XSS; CSRF; Web Socket

I. INTRODUCTION

In today's cyber age, web applications serve as the main entry points for most businesses and are insecure by default. Many individuals and organizations rely on websites for many real-life applications [3]. Due to non-adherence to secure coding standards and practices by developers, the systems remain vulnerable to attacks by hackers [3][11]. Web Applications are the gateways for most businesses in today's cyber era. Different technologies and programming languages are used for developing these web applications. State-of-the-art software technologies incorporate a security framework that allows software developers to include security features in web applications [5][6].

HTML is a markup language that combines hypertext with markup and is used to create web pages. HTML5, the latest version of HTML, enhances markup, and introduces APIs and DOM. The dynamic operation on the websites can be performed using plug-ins like ActiveX and Flash. Attackers prefer to use plug-ins, which slow down websites, to attempt an attack. HTML5 supports some new APIs and replaces plug-ins in website creation but has side effects and potential attack methods [11].

Ensuring security is a fundamental aspect when designing any system. The introduction of server-side languages brought about new security concerns as web servers became susceptible to vulnerabilities. As blogging and web services gained popularity, web applications became prime targets for attackers. Consequently, various novel attack vectors emerged, such as cross-site scripting (XSS), SQL injection, insecure direct object reference, remote malicious file inclusion, cross-site request forgery, access control weaknesses, data confidentiality breaches, and inadequate error handling. It is imperative to address these issues to safeguard the integrity and confidentiality of the system [13][14][19][20].

The paper organization is given as follows- Section 2 describes the related work to vulnerabilities in web applications. In Section 3, various kinds of web attacks XSS, CSRF, Clickjacking and UI exploits, SQL injection, HTML Injection, Web Messaging and Web Workers injections, Web Sockets, and Protocol/Schema/APIs attacks with HTML5 are given. The proposed work is given in the form of defense mechanisms against attacks given in section 4. The conclusion of this research work is in section 5.

II. LITERATURE SURVEY

HTML is a popular writing and display format for webpages that is expanding along with the web [11].

A. Vulnerabilities in HTML5

Web applications designed with HTML5 are susceptible to a number of vulnerabilities as shown in Table 1.

TABLE I. HTML5 VULNERABILITIES

Vulnerability	Description
V1:Cross-Site-Scripting (XSS)	Attackers are able to insert harmful scripts into websites [11][12].
V2:Cross-Site-Request-Forgery (CSRF)	Unlawful activities are carried out on a website when users are logged in [7].
V3:Insecure Direct Object References (IDOR)	Inadequate access restrictions or direct object references being exposed [19].
V4:Security of HTML5 APIs	Unknown vulnerabilities exist in HTML APIs [20][21].
V5:Clickjacking	Deceiving people into taking undesirable actions without their permission [10][17].
V6:SQL Injection	Improper handling of user-supplied input within SQL queries [13][16].
V7:HTML Injection	Injecting malicious HTML code into vulnerable areas of a website [19][23].
V8:Web messaging and web workers' injections	Run code from different origins and potentially expose sensitive data or resources [22][24].

V1: Cross-Site Scripting (XSS)

This kind of input validation vulnerability forces the victim's device to run malicious code by injecting it into the

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

browser of the victim [1][5]. Every time a user requests a specific function on a web server, a stored malicious script gets executed [4]. This attack's objective is to steal a user's identity data and carry out harmful actions such user impersonation, keylogging, phishing, and webcam activation [14]. XSS attacks can be performed using the XSSer, FoxyProxy, and Burp Suite tools [1]. Static and dynamic analysis are the main focus of traditional XSS detection techniques, which are inefficient against payload floods [12][24].

V2: Cross-Site Request Forgery (CSRF)

CSRF is an attack that forces logged-in users to do unwanted activities on a web application [4][5][7].

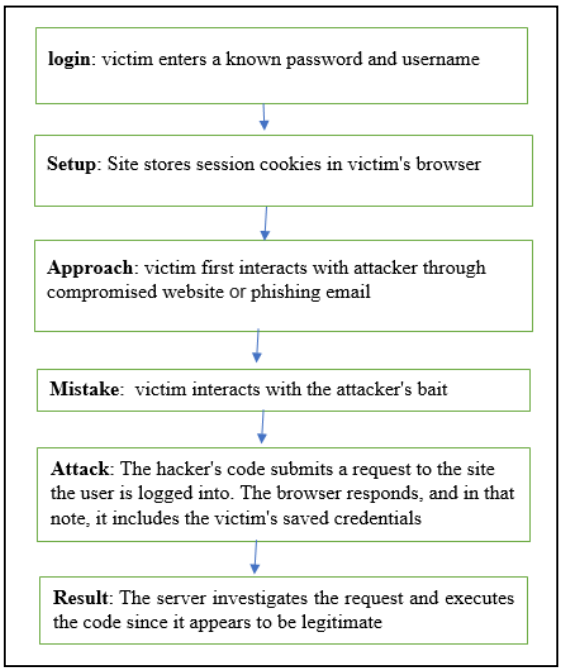

Fig. 1. Cross-Site Request Forgery attack flow.

CSRF is also known as Session Riding, Cross-site Reference Forgery (XSRF) attack, and Sea Surf attack [4]. The systematic attack flow of cross-site request forgery attacks is shown in Figure 1.

V3: Insecure Direct Object References (IDOR)

Attackers perform this attack to access sensitive data or carry out operations bypassing authorization [9][15].

V4: Security of HTML5 APIs

Secure implementation of HTML5 APIs such as WebSockets, Geolocation, local storage, and File API which can be a source of vulnerabilities is essential [20][21][22].

V5: Clickjacking

Clickjacking tricks users into clicking on elements of a web page unknowingly, potentially leading to unintended actions or exposing sensitive information. It is also known as a UI redress attack or a "UI-layer" attack [1][10][17].

V6: SQL Injection

Web applications that interface with databases frequently come under SQL injection attacks. An attacker can take advantage of the vulnerability by inserting SQL code into user-input fields or parameters that are used in SQL queries. Injected SQL code has the ability to alter the logic of the query or carry out unauthorized operations on the database [13][16].

V7: HTML Injection

This attack takes place by injecting malicious HTML code into vulnerable areas of a website. The vulnerability is caused due to improper validation, encoding, and sanitization of the user input. An attacker can inject malicious HTML code that gets executed in the context of other users' browsers [19][23].

V8: Web Messaging and Web Workers Injections

HTML5 is having a new interframe communication system called Web Messaging. By postMessage() call parent frame/domain can call with the iframe. The iframe can be loaded on cross-domain. Hence, creating issues with data/information validation & data leakage by cross-posting is possible.

B. *Attacks Performed on HTML5*

Web applications developed using HTML5 can be targeted by various attacks. There are various attacks performed on web applications developed using HTML5.

A1: Cross-Site Scripting (XSS)

Web applications created with HTML5, as well as other web technologies, are susceptible to cross-site scripting (XSS), an attack vector. When a hacker is able to insert dangerous scripts into web pages that other users view, XSS attacks take place [5][14][24].

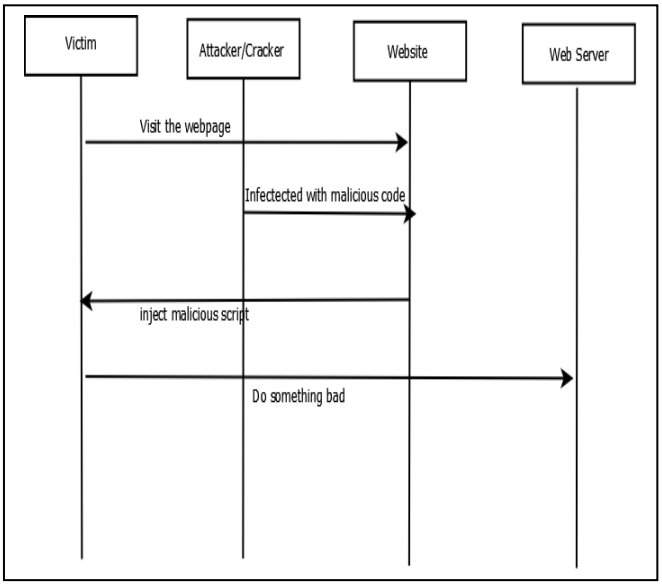

Fig. 2. Schematic diagram of an XSS attack.

XSS attack scenario on an HTML5 web application is shown in Figure 2.

Step 1: A form field where users enter a username.
Step 2: The application takes the user's input and displays it on a page, without properly sanitizing or validating the input.
Step 3: An attacker enters a script as their name, such as <script>malicious code</script>.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

Step 4: When the page is viewed by other users, the script is executed within their browser, allowing the attacker to perform various malicious actions, such as stealing sensitive information, manipulating the content of the page, or redirecting users to another malicious website.

Following are the XSS vulnerable tags and vulnerable objects/methods.

vulnerable HTML tags	Vulnerable objects/methods
<form>, <input> <video>,<img> <embed>, <button> <manifest>	innerHTML, Onload , mouse events (onmouseover ,onerror) ,alert()

A2: Cross-Site Request Forgery (CSRF)

CSRF attack happens when an attacker deceives a victim into unintentionally executing an action on a web application on which the victim is authorized [7]. Figure 3 shows the systematic CSRF attack flow.

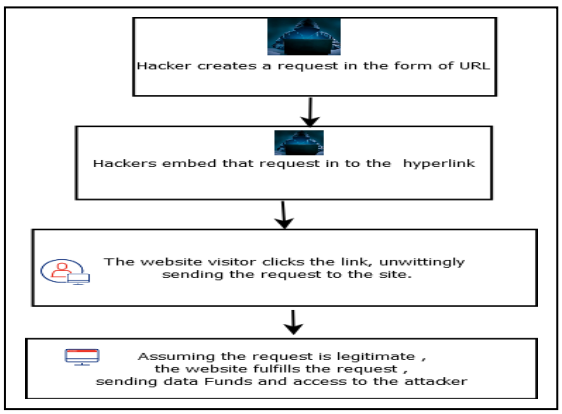

Fig. 3. CSRF attack Flow.

CSRF attack scenario on the web application is as follows.

Step 1: HTML5 web application includes a feature that allows users to change their email addresses.
Step 2: When a user wants to change their email address, they visit a specific URL or click a button that triggers the action.
Step 3: The action to change the email address is a simple HTTP POST request to a specific endpoint on the server.
Step 4: An attacker creates a malicious website or sends a crafted email to the victim, containing a form or JavaScript code that automatically submits a request to the email address change endpoint of the web application.
Step 5: If the victim is authenticated in the web application and visits the attacker's website or clicks on the link in the email, the malicious request is sent to the web application, changing the victim's email address without their knowledge or consent.

Following are the CSRF vulnerable tags and vulnerable objects/methods.

vulnerable HTML tags	Vulnerable objects/methods
Get, Post request <form>, <input> <script> ,<svg> <img>	Onload , Open() , send(), XMLHttpRequest

A3: Insecure Direct Object References (IDOR)

IDOR attacks exploit the improper implementation of access controls, allowing an attacker to directly access and manipulate sensitive data or resources.

IDOR attack scenario on the web application is as follows.

1. Direct Object References: In web applications, various resources such as user profiles, documents, or database records are typically assigned unique identifiers. These identifiers are used to access or retrieve the corresponding resources. For example, a user profile might be accessed using a URL like http://test.com/user/profile?id=12.

2. Insufficient Access Controls: Insecure Direct Object References arise when the application fails to properly enforce access controls or authorization checks on the server side.

3. Manipulating Object References: An attacker can exploit this vulnerability by manipulating the object references in the application's requests. The attacker changed the identifier value in the URL or manipulated parameters in form submissions to access restricted resources.

4. Unauthorized Access: The attacker bypasses access controls and directly accesses another user's private information or confidential documents [9][15].

A4: Insecure Direct Object References (IDOR)

WebSockets, geolocation, local storage, and the File API are powerful APIs provided by HTML5 that enhance web applications with additional functionalities. It can be vulnerable to certain attacks if not implemented and used securely.

1. WebSockets

Cross-Site WebSocket Hijacking (CSWSH): An attacker tricks a user's browser into making unintended WebSocket connections, allowing them to read or modify data transmitted over the WebSocket connection.

WebSocket Injection: Malicious actors inject unauthorized messages or data into a WebSocket connection, potentially disrupting the application or manipulating its behavior [20][21][22].

2. Geolocation API

Geolocation Spoofing: Attackers can forge or manipulate GPS coordinates, leading to incorrect location information and allowing attackers to perform location-based attacks, such as fake check-ins or targeted scams.

3. Local Storage

Cross-Site Scripting (XSS): If user input is not properly validated or sanitized before being stored in local storage, it can lead to XSS vulnerabilities. Attackers can inject malicious scripts that execute when the data is retrieved from local storage, compromising the user's session or stealing sensitive information.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

4. File API

Path Traversal: If file paths are not properly validated, attackers can manipulate file path parameters to access files outside of the intended directory, potentially exposing sensitive information or executing malicious code.

5. Malicious File Upload

Attackers may upload files containing malware or malicious scripts to the server, which can later be executed or distributed to other users.

Following are the API Vulnerable Tags and Vulnerable objects/methods.

vulnerable HTML tags	Vulnerable objects/methods
<script> <button> HTTP request	XMLHttpRequest , Open() , onload , mouse events

A5: Clickjacking Attack

An attacker uses an iframe to lure visitors into clicking links that will perform inappropriate actions in this type of web attack [1]. Unexpected outcomes result from an object on the iframe registering the click event [10].

Following are the Clickjacking attack scenarios.

1. Overlaying Content: Attackers create a transparent or disguised layer over a web page, positioning elements such as buttons, links, or form fields in a way that makes them invisible or barely noticeable to the user. The attacker then entices the user to interact with the disguised elements, while the actual clicks are intercepted and applied to hidden elements of the attacker's choosing.

2. Framing: Attackers load a target web application within an invisible or disguised iframe on their malicious website. The attacker overlays their own content, such as buttons or forms, on top of the iframe. When the user interacts with the seemingly harmless elements, they are actually interacting with the underlying web application, potentially performing unintended actions.

3. UI Redress: Attackers use CSS or other styling techniques to manipulate the appearance of web elements. They can make elements invisible, overlap them with other elements, or move them off-screen, making it difficult for users to detect their presence. By deceiving users into clicking on the disguised elements, attackers can perform actions on their behalf.

4. Invisible Frames: Attackers create invisible iframes and position them over clickable elements on a web page. When users click on what appears to be a legitimate element, they are interacting with the hidden iframe, executing malicious actions without their knowledge [10][18].

Following are the Click-Jacking vulnerable tags.

vulnerable HTML tags
<style> <iframe>

A6: SQL injection

An attacker exploits the vulnerability by injecting SQL code into user input fields or parameters that are used in SQL queries. The injected SQL code can manipulate the query's logic or execute unauthorized actions on the database [13].

There are different SQL injection methods -

1. Blind Injection: From a true/false question's response, logical inferences can be made.

2. Logically Incorrect Queries: retrieving data from various error messages in order to exploit and inject

3. Piggy-Backed Queries: A dangerous query is added to an already-injected query.

4. Stored Procedure: executing database's built-in functions using dangerous SQL Injection scripts/codes

5. Tautology: SQL injection queries are injected so they always result in a true statement

6. Timing Attack: observing the database response time in order to get information

7. Union Query: Using UNION, a malicious query combined with a secure query to obtain more table-related data [2][16].

Syntax

Login page where the user enters their username and password. The server-side code might construct an SQL query like:

```
SELECT * FROM users WHERE username = '<user_input>' AND password = '<user_input>'
```

If the user input is not properly validated or sanitized, an attacker can enter malicious input like

' OR '1'='1' --

The resulting query

```
SELECT * FROM users WHERE username = '' OR '1'='1' --' AND password = ''
```

The injected SQL code ' OR '1'='1' -- always evaluates to true, effectively bypassing the authentication and allowing the attacker to log in without a valid username or password.

Following are the SQL Injection vulnerable tags and vulnerable objects/methods.

vulnerable HTML tags	Vulnerable objects/methods
<input> <textarea> HTTP header, Connection request	Connection.query()

A7: HTML Injection

An attacker exploits the vulnerability by injecting HTML code into user input fields or parameters that are later displayed on web pages. The injected code can include JavaScript, which allows the attacker to execute arbitrary actions on the victim's browser [19][23].

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

For example, imagine a comment section on a web application where users can enter comments that are later displayed on the page without proper validation.

```
<div class="comment">User Comment: <user_input></div>
```

If the user input is not properly sanitized, an attacker can

```
script>
 // Malicious code to perform unauthorized actions
</script>
```

inject malicious HTML code like:

The injected JavaScript code will be executed when other users view the comment, leading to potential security breaches.

HTML injection attacks can have severe consequences, including stealing sensitive information, session hijacking, defacement of the website, phishing attacks, or spreading malware.

vulnerable HTML tags
<input>,<script> <svg>,<img>

A8: Web Messaging and Web Workers Injections

Potential attack scenarios involving Web Messaging and Web Workers in a web application.

1. Cross-Origin Resource Sharing (CORS) Attacks: Web Messaging allows communication between different origins (domains) using the postMessage method. If the web application doesn't properly implement Cross-Origin Resource Sharing (CORS) policies, an attacker could exploit this to perform Cross-Site Scripting (XSS) attacks. By injecting malicious scripts into the web application, the attacker can steal user data, manipulate the application's behavior, or perform other malicious activities.

2. Denial-of-Service (DoS) Attacks: Web Workers enable concurrent execution of scripts in the background, which improves performance. However, an attacker can abuse this feature to launch Denial-of-Service attacks. By creating numerous Web Workers or by sending a high volume of messages to existing workers, the attacker can exhaust system resources, causing the application to become unresponsive or crash.

3. Data Leakage Attacks: Web Messaging and Web Workers rely on the exchange of messages between different components of a web application. If sensitive information is included in these messages, an attacker may intercept and exploit it. This could lead to the exposure of user data, session tokens, or other confidential information.

4. Man-in-the-Middle (MITM) Attacks: If the communication between Web Workers or between the main application and workers is not properly secured, an attacker can perform Man-in-the-Middle attacks. By intercepting and modifying the messages being sent, the attacker can manipulate the application's behavior, inject malicious code, or capture sensitive data.

Following are the web messaging and web workers' injection vulnerable tags and vulnerable objects/methods.

vulnerable HTML tags	Vulnerable objects/methods
<button> <img> <video> <iframe> <script>	Worker.addEventListner() Worker.postMessage() innerHTML , eval()

III. Proposed Defensive Techniques for Secure Web Development

It is important to apply security standard practices for developing web applications. Following are the defensive mechanisms for secure web development.

D1: Cross-Site Scripting (XSS) Protection

Content Security Policy (CSP) is used to control scripts and content to execute on your web application. Use a robust and secure framework that automatically applies output encoding to user input to minimize the risk of XSS vulnerabilities.

Various measures to mitigate XSS attacks are as follows.

1. Use HTTPOnly and Secure Flags for Cookies

Set the HttpOnly and Secure flags on cookies to prevent client-side scripts from accessing them, thus reducing the risk of session hijacking through XSS attacks. This can be done on the server side when creating or setting cookies.

2. Input Validation and Sanitization

```
// Example code in a server-side language
res.cookie('sessionID', sessionId, { httpOnly: true, secure: true });
```

Validate and sanitize all user inputs, especially those used in dynamic content generation, like username and password fields. Use a whitelist approach for input validation, allowing only specific characters that are required for the input.

3. Output Encoding

```
// Example code in a server-side language
const sanitizeInput = (input) => {
 // Implement your input sanitization logic here
 return sanitizedInput;
};
```

Encode all dynamic content that is inserted into the HTML, to prevent script execution. Utilize functions like encodeURIComponent or server-side templating engines that automatically encode data.

```
// Example code in a server-side language
const user = getUserFromDatabase(username);
res.render('user-profile, { username: encodeURIComponent(user.username) });
```

4. Content Security Policy (CSP)

Implement a Content Security Policy to restrict the sources from which the browser can load resources (scripts, styles,

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

images) on your web page. This can help prevent the execution of malicious scripts injected through XSS attacks.

```
<!-- Example CSP header -->
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' https://trusted-cdn.com;">
```

5. HTTP Header Security

Set the following HTTP security headers in the server response to enhance protection.

```
// Example code in a server-side language
app.use ((req, res, next) => {
res.setHeader('X-XSS-Protection', '1; mode=block');
res.setHeader('X-Content-Type-Options', 'nosniff');
res.setHeader('X-Frame-Options', 'DENY');
next();
});
```

6. Use CSRF Tokens

Employ Cross-Site Request Forgery (CSRF) tokens to ensure that only authorized requests from your website can modify user data. This helps protect against CSRF attacks that could be used in combination with XSS.

```
<! -- Include the CSRF token in  login form -->
<form action="/login" method="post">
 <input type="hidden" name="_csrf" value="{{csrfToken }}">
 <!-- other form fields -->
 <button type="submit">Login</button>
</form>
```

D2: Cross-Site Request Forgery (CSRF) Protection

Implement measures like CSRF tokens to validate and authenticate user requests, ensuring that requests originate from trusted sources and preventing unauthorized actions. Verify the origin and integrity of requests to mitigate the risk of CSRF attacks. To prevent CSRF attacks in HTML5 web applications, the following measures can be implemented.

1. Generate and Store CSRF Token on the Server

On the server side, generate a CSRF token and associate it with the user's session. Store the token securely in the session or as a separate cookie.

```
// Example code in a server-side language
const csrf = require('csrf');
const cookieParser = require('cookie-parser');
const express = require('express');

const app = express ();
app.use(cookieParser ());
app.use (csrf ({ cookie: true }));

app.get('/login', (req, res) => {
 const csrfToken = req.csrfToken();
 res.render('login', { csrfToken });
});
```

2. Include CSRF Token in the Login Form

In the HTML5 login form, include the CSRF token as a hidden input field.

```
<!-- Example of including CSRF token in the login form -->
<form action="/login" method="post">
 <input type="hidden" name="_csrf" value="{{csrfToken }}">
 <!-- other form fields: username and password -->
 <button type="submit">Login</button>
</form>
```

3. Verify CSRF Token on the Server

When the user submits a form, the CSRF token is sent to the server as part of the request payload. Validate the token's authenticity on the server before processing the request.

```
// Example code in a server-side language
app.post('/login', (req, res) => {
 const { username, password, _csrf } = req.body;
 if (!req.csrfToken() || req.csrfToken() !== _csrf) {
  return res.status(403).send('CSRF token validation failed.');
 }
 // Perform login authentication and other logic
 // ...
});
```

D3: Insecure Direct Object References (IDOR) prevention

To prevent IDOR attacks, it is essential to implement proper access controls and authorization mechanisms in web applications.

1. Context-Based Access Controls: Ensure that every request to access a sensitive resource is validated against the user's permissions and privileges. This should be enforced on the server side, regardless of any client-side restrictions implemented using HTML5.

2. Indirect Object References: Avoid directly exposing internal object references or identifiers in URLs or form parameters. Instead, use an indirect reference, such as a unique session identifier, which is then mapped to the actual resource on the server side. This helps prevent attackers from easily manipulating object references.

3. Role-Based Access Control (RBAC): Implement a role-based access control model, where users are assigned specific roles or privileges, and access to resources is granted based on those roles. This ensures that users can only access resources that are authorized for their role.

D4: Security of HTML5 APIs

To mitigate these vulnerabilities and ensure the secure usage of these APIs, here are some recommended practices:

1. *WebSockets*:

- Validate and sanitize user input to prevent injection attacks.
- Implement server-side checks to ensure the WebSocket connection is legitimate.
- Encrypt sensitive data transmitted over WebSockets to protect it from interception.

2. *Geolocation API*:

• Obtain user consent before accessing their location data.

• Verify the accuracy and integrity of geolocation data received from the API.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

• Regularly review and update the accuracy of geolocation data stored by the application.

3. Local Storage:

• Sanitize and validate user input before storing it in local storage.

• Avoid storing sensitive information in local storage whenever possible.

• Implement content security policies and input validation to prevent XSS attacks.

4. File API:

• Validate file types and extensions before accepting file uploads.

• Set appropriate file upload size limits to prevent resource exhaustion attacks.

• Store uploaded files in a secure location with restricted access permissions.

D5: Clickjacking Attack Prevention

To protect against clickjacking attacks, consider implementing the following preventive measures:

1. X-Frame-Options: Set the X-Frame-Options response header to prevent web applications from being framed within iframes on other domains. The header can be set to "DENY" to disallow framing altogether or "SAMEORIGIN" to allow framing only from the same origin.

2. Content Security Policy (CSP): Implement a robust CSP to control which domains are allowed to embed your web application within iframes. Use the "frame-ancestors" directive to specify trusted origins that can frame your application.

3. Frame Busting Techniques: Include frame-busting JavaScript code within your web application to prevent it from being loaded within iframes on other domains. This code can break out of frames and redirect the user to the intended page if the application is being framed.

4. JavaScript Event Validation: Implement client-side JavaScript checks to validate user actions and prevent interactions with hidden or overlaid elements. Verify that the element being clicked is visible, not overlapped, and positioned correctly.

5. Visual Indicators: Use visual cues or indicators to clearly show when the page is being framed or overlaid with external content. This can help users detect and avoid interacting with malicious elements.

D6: SQL injection

SQL injection attacks can have severe consequences, including unauthorized access to sensitive data, data manipulation, database compromise, or even complete system compromise. To protect against SQL injection attacks in a web application developed in HTML5, the following security measures can be implemented.

Parameterized Queries: Use prepared statements or parameterized queries with placeholders instead of concatenating user input directly into SQL queries. This ensures that user input is treated as data rather than executable code.

Input Validation and Sanitization: Validate and sanitize user input on the server side to reject or neutralize potentially malicious characters or patterns.

Least Privilege Principle: Assign minimal database privileges to the application's database user account. Restrict the account's permissions to only the necessary database operations.

Principle of Least Exposure: Limit the amount of sensitive information exposed to the web application, minimizing the potential impact of a successful SQL injection attack.

D7: HTML injection

To protect against HTML injection attacks in a web application, the following security measures should be implemented.

Input Validation and Sanitization: Validate and sanitize user input on the server side to reject or neutralize potentially malicious HTML code or tags. Use proper encoding techniques

```
const userInput = document.getElementById('userInput').value;
 const sanitizedInput = escapeHTML(userInput);

 const userInputDisplay = document.getElementById('userInputDisplay');
 userInputDisplay.innerHTML = `<p>User Input: ${sanitizedInput}</p>`;
```

to render user input safely.

Output Encoding: Properly encode user-generated content when displaying it on web pages. This ensures that any HTML tags or special characters are treated as literal text rather than executable code.

Content Security Policy (CSP): Implement a Content Security Policy that defines which sources of content (scripts, stylesheets, etc.) are allowed to be loaded and executed on your

```
<!-- Example CSP header -->
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'
https://trusted-cdn.com;">
```

web pages. This can help mitigate the impact of XSS attacks.

D8: Defense Mechanism for Web Messaging and Web Workers Injections

1. Web Messaging Injection

Web Messaging allows communication between different browsing contexts. For e.g., between an iframe and its parent

```
const sanitizedMessage = escapeHTML(event.data);
 // Process the sanitizedMessage as needed.
```

window can be misused if not handled securely.

2. Web Workers Injection:

Web Workers run scripts in the background without affecting the main thread. To prevent injection attacks, avoid using user input or untrusted sources directly within a Web Worker script.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 12 Issue 07, July-2023

```
const userInput = document. getElementById('userInput').value;
    const sanitizedInput = escapeHTML(userInput);
    const worker = new Worker("defensiveWorker.js");
    // Send the sanitized user input to the Web Worker.
    worker.postMessage(sanitizedInput);
```

To protect against these attacks, consider the following security practices:

• Implement proper CORS policies to restrict communication between different origins.

• Sanitize and validate input data to prevent XSS attacks.

• Implement rate limiting and monitoring mechanisms to prevent DoS attacks.

• Avoid sending sensitive information through messages or encrypting the data before sending.

• Use secure communication channels (e.g., HTTPS) to prevent MITM attacks.

IV. CONCLUSION

Web application security is of prime concern in today's era and demands a proactive approach throughout the entire software development life cycle. This paper is focusing on the crucial task of developing secure, robust, and reliable web applications through a detailed exploration of HTML5 and its powerful APIs such as Web Storage API, WebSockets, geolocation, local storage, and File API. HTML5 provides dynamic functions to web pages without additional plug-ins such as ActiveX, Flash, and Silverlight. Most attacks on web applications use such plug-ins. HTML5 provides abilities that can be substituted for plug-ins, so hackers focus their attacks on HTML5. Hence the vulnerabilities and attacks in HTML5 and its APIs are demonstrated and a suitable defense mechanism is provided to secure coding with HTML5. This paper serves as a stepping stone towards a more secure web ecosystem and aims to inspire further advancements in the field of web development.

REFERENCES

[1] Voo Teck En Vinesha Selvarajah, “Cross-Site Scripting (XSS)”,2022, IEEE.

[2] Jean-Paul A. Yaacoub, Hassan N. Noura, Ola Salman, Ali Chehab, “A Survey on Ethical Hacking: Issues And Challenges”, A PREPRINT - MARCH 30, 2021.

[3] Rodríguez, G. E., Torres, J. G., Flores, P., & Benavides, D. E, “Cross-Site Scripting (XSS) Attacks And Mitigation: A Survey” Computer Networks, Elsevier, pp. 1-43, 2019.

[4] Ajjarapu Kusuma Priyanka, Siddemsetty Sai Smruthi, “WebApplication Vulnerabilities: Exploitation and Prevention”, (ICIRCA-2020) 2020 IEEE.

[5] Egor A.Efremov, Maria V. Pogrebnyak, Maria Skvortsova, “ HTML5 Security issue”, IEEE,2021

[6] Nirmal K, B. Janet, R. Kumar, “Web Application Vulnerabilities – The Hacker’s Treasure”, International Conference on Inventive Research in Computing Applications, (ICIRCA), IEEE, pp-58-62,2018.

[7] M.] Ksenia Pegueroa,, Xiuzhen Cheng, “CSRF protection in JavaScript frameworks and the security of JavaScript applications”, High-Confidence Computing, Elsevier,2021.

[8] Pei Wang, Bjarki Ágúst Guðmundsson, Krzysztof Kotowicz,“Adopting Trusted Types in ProductionWeb Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study”, European Symposium on Security and Privacy Workshops (EuroS&PW), 2021, pp-60-73 IEEE.

[9] Nisal Madhushan Vithanage Neera Jeyamohan, “WebGuardia – An Integrated Penetration Testing System to Detect Web Application Vulnerabilities”, pp-221-227, 2016 IEEE.

[10] Kanpata Sudhakara Rao, Naman Jain, Nikhil Limaje, “Two for the price of one: A combined browser defense against XSS and clickjacking”, 2016 IEEE.

[11] M.K. Gupta, M.C. Govil, G. Singh, Predicting cross-site scripting (XSS) security vulnerabilities in web applications, in 2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE), 2015, pp. 162–167.

[12] Zhou, Y., & Wang, P. (2019). An ensemble learning approach for XSS attack detection with domain knowledge and threat intelligence. Computers & Security, 82, 261–269.

[13] Tao Zhang; Xi Guo,” Research on SQL Injection Vulnerabilities and Its Detection Methods”, International Conference on Data Science and Business Analytics (ICDSBA), IEEE 2020.

[14] Vikas K. Malviya, Saket Saurav, Atul Gupta, “On Security Issues in Web Applications through Cross Site Scripting (XSS)”, Asia-Pacific Software Engineering Conference, 2013 IEEE.

[15] Owasp.org, "Testing for Insecure Direct Object References (OTGAUTHZ-004) - OWASP", 2015. [Online] https://www.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_%28OTG-AUTHZ-004%29.

[16] Lwin Khin Shar and Hee Beng Kuan Tan, “Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities”, ICSE 2012,2012 IEEE.

[17] Asra Kalim, P.K.Jha, Deepak Singh Thomar, “Novel Detection Technique in Frame jacking We Application “, ICCAKM 2021, IEEE.

[18] Rakhi Sinha, Dolly Uppal, Dharmendra Singh, Rakesh Rathi, “Clickjacking: Existing Defenses and Some Novel Approaches”, 2014, IEEE.

[19] https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/11-Client-side_Testing/03-Testing_for_HTML_Injection

[20] Jussi-Pekka Erkkilä, “WebSocket Security Analysis “, Aalto University T-110.5291 Seminar on Network Security, 2012.

[21] Vanessa Wang, Frank Salim, Peter Moskovits, “The Definitive Guide to HTML5 WebSocket”, Springer, 2013.

[22] Qigang Liu, Xiangyang Sun, “Research of Web Real-Time Communication Based on Web Socket”, 2012 SciRes, IJCNS.

[23] G. Deepaa, P. Santhi Thilagama, “Securing Web Applications from Injection and Logic Vulnerabilities: Approaches and Challenges” Information and Software Technology, Elsevier 2016.

[24] Wenbo Mei, Zhaohua Long, “Research and Defense of Cross-Site WebSocket Hijacking Vulnerability”, 2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA), pp.591-594, 2020 IEEE.

.