International Journal of Safety and Security Engineering
Vol. 14, No. 6, December, 2024, pp. 1933-1940
Journal homepage: http://iieta.org/journals/ijsse

Detecting Security Vulnerabilities in Web Applications: A Proposed System

Ayman Emadeddin Hafez*, Muhammad Mazen Almustafa

Syrian Virtual University, Damascus 35329, Syria

Corresponding Author Email: aymanhafez133@gmail.com

https://doi.org/10.18280/ijsse.140627

Received: 3 October 2024
Revised: 13 December 2024
Accepted: 21 December 2024
Available online: 31 December 2024

Keywords:
Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XSS), SQL injection, structured query language (SQL), vulnerabilities, web security

ABSTRACT

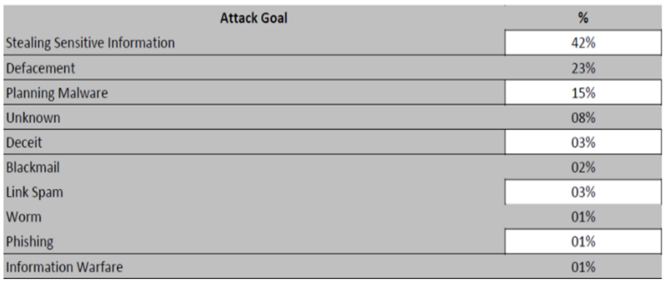
Web applications have become a central part of our modern era, playing a significant role in facilitating various online activities such as social networking, e-commerce, and financial transactions. As reliance on different web applications has increased, the risks of cyberattacks and breaches of user data have also grown substantially. Therefore, web application security is of utmost importance to protect user information, maintain trust in electronic services, and prevent financial losses for organizations and entities. In this paper, we propose a novel system that integrates automated detection with detailed reporting mechanisms to analyze critical security vulnerabilities targeting web applications, such as SQL injection and Cross-Site Scripting (XSS) attacks. Unlike existing solutions, our system provides actionable insights that help organizations not only detect but also mitigate vulnerabilities, significantly enhancing the overall security of web applications.

1. INTRODUCTION

Web applications have evolved from simple collections of static HTML documents to complex and comprehensive applications containing hundreds of dynamically generated pages with highly advanced functionalities, which were previously reserved for traditional desktop applications [1].

At the same time, this rapid evolution, coupled with the increasing amount of sensitive information now accessible via the web, has led to a concurrent rise in the number and complexity of attacks and vulnerabilities associated with web applications [2].

Web security plays a crucial role in daily web applications, and the lack of security can lead to societal breakdowns on a broader scale. The importance of this field continues to grow as the number of internet users worldwide increases, exposing them to risks such as phishing attacks [3], and as they increasingly rely on web applications as a primary source for their activities, including online shopping, banking services, and chatting with friends [4].



Attack Goal	%
Stealing Sensitive Information	42%
Defacement	23%
Planning Malware	15%
Unknown	08%
Deceit	03%
Blackmail	02%
Link Spam	03%
Worm	01%
Phishing	01%
Information Warfare	01%

Figure 1. Reasons for targeting web applications

There are two categories of web security: web browser security and web application security. Web application security is more vulnerable than web browser security because it poses more severe threats to the user, such as credit card theft, document theft, and the compromise of confidential data [5].

Hackers often breach web applications due to the lack of input variable validation and the failure to implement security recommendations at the web application level as a whole.

The primary goal of web security research is to provide users with a secure and reliable platform for connecting to web applications.

Obtaining reliable information about the current state of web security is an extremely challenging task, as most organizations and companies are reluctant to disclose information related to their security vulnerabilities.

Web applications are targeted for several reasons, as mentioned in Figure 1, with the theft of sensitive information ranking first among these reasons.

Currently, web application security measures face significant challenges, such as the inability to detect complex or combined vulnerabilities. Existing tools often target individual vulnerabilities but fail to address them comprehensively. The proposed system seeks to fill this gap by providing a unified solution that not only identifies key vulnerabilities like SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF) but also offers practical steps for mitigation. This contribution builds upon existing approaches and demonstrates significant advancements in securing web applications.

2. BACKGROUND

Web application security refers to the practices, techniques, and measures implemented to protect web applications from vulnerabilities, attacks, and unauthorized access. It involves the implementation of security controls and safeguards to ensure the confidentiality, integrity, and availability of web applications and the data they process. This includes various aspects such as protecting sensitive user data, preventing unauthorized access to systems, and ensuring the availability and integrity of the system [6, 7].

Attackers can exploit underlying security vulnerabilities in web applications to gain unauthorized access, steal sensitive information, manipulate data, or disrupt the application's functions entirely [8]. In an era where cyber threats are constantly evolving [9, 10], understanding security risks and implementing effective measures is crucial for protecting user data, maintaining the reputation of the organizations developing these applications, and providing users with a secure online experience [11]. Moreover, the goal of web application security is to identify and mitigate vulnerabilities that attackers could exploit to compromise the security of the application or its users [12].

In the context of web application security, vulnerabilities refer to weaknesses or flaws in any system, network, or application that can be exploited by attackers to compromise the security and integrity of the system [13, 14]. These vulnerabilities may arise from design defects, programming errors, improper system configurations, or other factors that create potential pathways for unauthorized access, data breaches, or system exploitation. Such vulnerabilities can lead to unauthorized access to sensitive user information or the execution of malicious actions that jeopardize the security of the web application [15-17].

Current approaches often lack comprehensive solutions, focusing on specific vulnerabilities while neglecting others. For instance, many tools effectively identify SQL Injection but fail to address Cross-Site Scripting (XSS) or Cross-Site Request Forgery (CSRF). Addressing these gaps requires an integrated approach, which the proposed system aims to provide.

2.1 SQL injection vulnerability

SQL Injection is a security vulnerability that allows attackers to manipulate or execute unauthorized SQL statements within an application's database [18, 19]. It occurs when user input is not properly validated or sanitized, allowing malicious SQL commands to alter queries. Common types of SQL Injection attacks include Classic or In-band SQL Injection [20], where attackers inject harmful code into user input fields, such as Error-Based and Union-Based SQL Injections [21]. Error-based attacks exploit application error messages, while Union-based attacks leverage the UNION statement to retrieve data from multiple database tables, requiring the attacker to match the query structure [22, 23].

Blind SQL Injection occurs when the attacker cannot directly see the results of their injection in the application's response [24]. Instead, they rely on boolean-based or time-based techniques to infer information. Boolean-based attacks involve expecting true or false responses, while time-based attacks manipulate response times to extract data [25]. Blind SQL Injection is often used to steal sensitive information, though it is more challenging to exploit than in-band attacks [26]. Another complex form of SQL Injection is Out-of-Band, where attackers use alternative communication channels, such as DNS or HTTP, to extract data or execute system commands.

Preventing SQL Injection requires proper input validation and sanitization techniques [27]. Input should be validated to match expected formats, and special characters that could be used for injection should be removed or escaped [28]. Whitelisting acceptable input patterns is another effective strategy [29]. Additionally, using prepared statements or parameterized queries is critical for blocking SQL Injection attempts. These methods separate SQL code from user input, treating user inputs as data rather than executable commands, thus preventing malicious code from altering query behavior [30].

2.2 Cross-Site Scripting vulnerability

Cross-Site Scripting (XSS) is a vulnerability that lets attackers inject malicious scripts into web pages viewed by others [31]. It occurs when a web application fails to sanitize user input and reflects it back to users. This allows attackers to execute harmful code in the victim's browser. In 2007, XSS made up about 84% of documented security vulnerabilities [32]. XSS attacks exploit how web browsers run scripts from different sources without verifying their origin, creating a significant risk to users.

There are three main types of XSS attacks [33]. Reflected XSS occurs when attackers send a malicious link that executes a script upon clicking. Stored XSS involves injecting a script into a web application that stores user input, making it the most dangerous type. In this case, malicious code is stored in a database and affects multiple users [34]. DOM-based XSS happens in client-side code, manipulating the browser's Document Object Model (DOM) to execute harmful code without server involvement [35].

Preventing XSS attacks requires multiple strategies. Input validation and filtering ensure that user input is sanitized to remove harmful characters [36]. Output encoding helps display dynamic data safely by converting special characters to plain text. Additionally, Content Security Policies (CSP) restrict the execution of scripts to trusted sources, reducing XSS risks by preventing unauthorized code from running on a webpage [37].

2.3 Cross-Site Request Forgery vulnerability

Cross-Site Request Forgery (CSRF) is a web application vulnerability that allows attackers to exploit the trust between a user's browser and a web application [38, 39]. Attackers trick victims into making unauthorized requests without their knowledge, using methods like crafted image tags, hidden forms, or JavaScript executions. CSRF attacks rely on the victim being authenticated with the target site, as the browser automatically includes session tokens or cookies in requests, enabling unauthorized actions [40].

Types of CSRF attacks include Basic CSRF, where victims are manipulated into performing actions unknowingly, and CSRF via Image Tag, which sends GET requests through HTML <img> tags [41]. CSRF via AJAX exploits asynchronous requests, while hidden form submissions involve automatic form submissions with the victim's cookies. Additionally, Remote File Inclusion (RFI) via CSRF can occur when attackers exploit vulnerabilities to include malicious files from remote servers, potentially leading to code execution or data exposure [42].

Preventing CSRF attacks requires a combination of techniques. Checking the Referer header can help validate request origins while implementing unique CSRF tokens adds an extra layer of security by ensuring request legitimacy [43]. The SameSite attribute in cookies restricts their use in cross-origin requests. Additionally, requiring reauthentication for sensitive actions and using CAPTCHA can further enhance protection against CSRF vulnerabilities, ensuring that requests are made by genuine users [44, 45].

3 LITERATURE REVIEW

3.1 In the field of SQL injection vulnerabilities

In 2021, researchers Fairoz Kareem, Siddeeq Ameen, and Azar Abid Salih from Duhok University in Iraq conducted a study discussing the strengths and weaknesses of PHP, emphasizing its importance as a widely used server-side language for web application development. They highlighted several PHP functions that pose high-risk vulnerabilities if misused by developers, noting that easier development practices often lead to higher security risks. The study reviewed PHP and other languages' techniques for protecting against SQL Injection vulnerabilities, detailing the consequences of such vulnerabilities and their detection methods. They examined patterns such as GET and POST and provided various suggestions and preventive measures for SQL Injection vulnerabilities, pointing out that sensitive information like passwords and credit card numbers, often stored in databases, is particularly vulnerable to attacks [46].

In 2021, researchers Jeklin Harefa, Gredion Prajena, Alexander, and Abdillah Muhamad from Bina Nusantara University in Indonesia focused on improving web application security against SQL Injection attacks by using features provided by the proposed Web Application Firewall (WAF). They demonstrated that the studied firewall could automatically block a significant portion of attacks and explained various features such as IP blocking, identifying and flagging unusual data traffic as malicious, and tools for administrators and users to manage application security. They compared it with CloudFlare in terms of their ability to block different types of security attacks and evaluated their strengths and weaknesses. The study differentiated between various SQL Injection patterns, including logically incorrect queries, union queries, and stored procedures. They concluded that their proposed system could enhance web application security against SQL Injection but required additional features and improvements, such as security alerts, reports, and integration with other security threats [47].

3.2 In the field of Cross-Site Scripting (XSS) vulnerabilities

In 2017, researchers Kirthiga Devi and Geogen George from SRM University in India conducted a study on detecting vulnerabilities in web applications related to Cross-Site Scripting (XSS) attacks. They explained the damage and mechanisms of XSS, which involve malicious JavaScript code executed either on the client-side or server-side. They discussed the vulnerability in the browser's session for web applications and proposed the addition of XSS-Check, a tool designed to detect XSS vulnerabilities on the client side. This tool identifies persistent XSS (where malicious code is stored in the target database and executed in the client's browser) and non-persistent XSS (targeting individuals through social engineering, such as malicious links). The XSS-Check tool scans and crawls the main and sub-links of the web application to find and report vulnerabilities [48].

In 2012, Shashank Gupta and Lalitsen Sharma from Jammu University in India studied the exploitation and defense of XSS vulnerabilities in web applications. They performed experiments on a local server (like XAMPP) to steal user information, such as session details and cookies, which are frequently used by web applications to maintain authentication states and automatically validate legitimate requests without further authentication. Their research included investigating XSS vulnerabilities on social media platforms like Facebook, Twitter, and blogs. They proposed a new mitigation technique using a sandbox environment in the web browser. They suggested several preventive measures, including proper input validation, maintaining a blacklist of vulnerable sites, and staying informed about the latest web technologies [49].

3.3 In the field of Cross-Site Request Forgery (CSRF) vulnerabilities

In 2021, N. Jhansi, M. Naveen, A. Sai Kumar, and R. Jagadeesh from AVN Institute in India proposed a methodology for leveraging Machine Learning (ML) to detect vulnerabilities in web applications. They considered ML highly beneficial for web application security due to the significant diversity in web application types and programming languages. ML can utilize manually labeled data to simulate human understanding of web applications through automated analysis tools. They applied these concepts to discuss Mitch, the first ML-based tool for detecting Cross-Site Request Forgery (CSRF) vulnerabilities that operate at the HTTP traffic level, regardless of the web application's programming language [50].

In 2018, Emil Semastin, Sami Azam, Bharanidharan Shanmugam, and Krishnan Kannoorpatti from Charles Darwin University in Australia conducted a study discussing preventive measures for CSRF attacks on web applications. They suggested several available tools for testing CSRF vulnerabilities, such as Burp Suite, ZAP, and Pinata. They highlighted that one of the most effective solutions is passing an unexpected token through a hidden field and verifying its validity on the server side. The token can also be passed through URLs, cookies, or via GET or POST requests. They proposed a third solution, which is a combination of these methods through double verification, to enhance security and ensure protection against CSRF vulnerabilities [51].

While numerous studies have contributed to the field of web security, many focus on isolated vulnerabilities or specific scenarios. For instance, tools like XSS-Check and machine learning-based CSRF detectors excel in their respective domains but lack the versatility to address multiple vulnerabilities concurrently. The proposed system builds on these works by integrating detection mechanisms for SQL Injection, XSS, and CSRF vulnerabilities, providing a more comprehensive approach. This integration distinguishes the system from existing solutions, offering enhanced detection accuracy and usability and providing actionable remediation steps.

4. METHODOLOGY AND RESULTS

In this section, we propose a system-based on the information analyzed-that assists in identifying security vulnerabilities in web applications and enhancing their security. Initially, we developed two web applications: one containing the targeted security vulnerabilities and the other secured against these vulnerabilities, to serve as test cases for our experiments. Our proposed system takes the URL of a web application provided by the user and performs an analysis to detect potential vulnerabilities by applying various attack vectors and techniques studied. If the system successfully injects malicious code or performs unauthorized actions, this indicates that the web application has security flaws that require attention. The system then generates a detailed report with recommended measures to mitigate the identified vulnerabilities and improve the application's overall security posture. This will be explained in detail in the following sections.

4.1 Developing web applications using PHP language

At this stage, we developed two web applications, as shown in Figure 2. In one application, security measures were deliberately neglected, with direct queries used without input validation or steps taken to mitigate security vulnerabilities.

In the second application, we implemented all the security precautions and protections studied, applying these measures directly within the code to ensure the application is secure for both browsing and deployment, as demonstrated in Figure 3.

Both applications include fundamental functionalities such as user registration, login, posting comments, searching users, and modifying or deleting user accounts. These features were chosen to simulate typical operations where security vulnerabilities are often exploited.

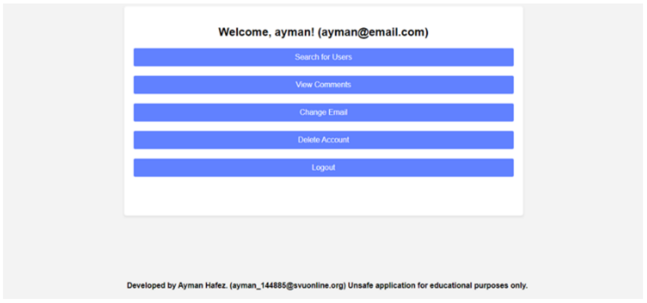
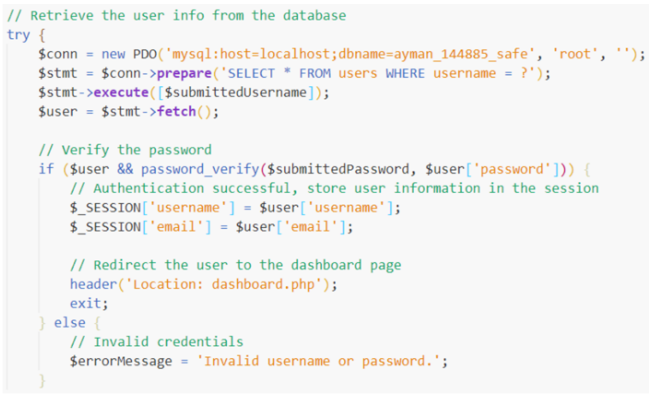

Welcome, ayman! (ayman@email.com)
Search for Users
View Comments
Change Email
Delete Account
Logout
Developed by Ayman Hafez. (ayman_144885@svuonline.org) Unsafe application for educational purposes only.

Figure 2. The web application interface utilized in the study



```php
// Retrieve the user info from the database
try {
    $conn = new PDO('mysql:host=localhost;dbname=ayman_144885_safe', 'root', '');
    $stmt = $conn->prepare('SELECT * FROM users WHERE username = ?');
    $stmt->execute([$submittedUsername]);
    $user = $stmt->fetch();

    // Verify the password
    if ($user && password_verify($submittedPassword, $user['password'])) {
        // Authentication successful, store user information in the session
        $_SESSION['username'] = $user['username'];
        $_SESSION['email'] = $user['email'];

        // Redirect the user to the dashboard page
        header('Location: dashboard.php');
        exit;
    } else {
        // Invalid credentials
        $errorMessage = 'Invalid username or password.';
    }
```

Figure 3. Code responsible for the login mechanism in a secure web application

4.2 Building a detection system using python language and conducting tests

The technical implementation of the proposed system involves leveraging Python libraries like BeautifulSoup, Selenium, and Requests to automate the testing of web applications for vulnerabilities. In general, the detection mechanism involves three main stages:

- Data Collection: Extracting all links and form fields from the target web application.
- Attack Simulation: Injecting test payloads for SQL Injection, XSS, and CSRF attacks, using predefined patterns and real-time validations.
- Analysis and Reporting: Evaluating the success of simulated attacks and generating a detailed report with recommendations.

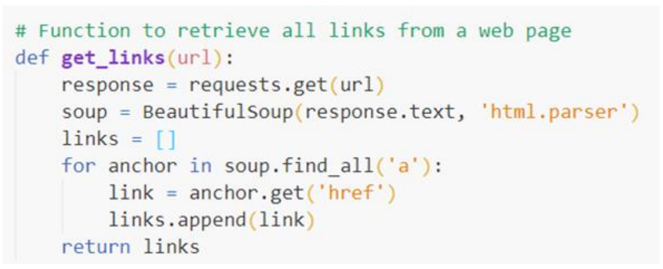
Below are detailed descriptions of the detection mechanisms and corresponding results:

4.2.1 Detection of SQL injection vulnerabilities

After entering the URL of the application to be tested, the system retrieves and stores all the links present on the web page, as illustrated in Figure 4.

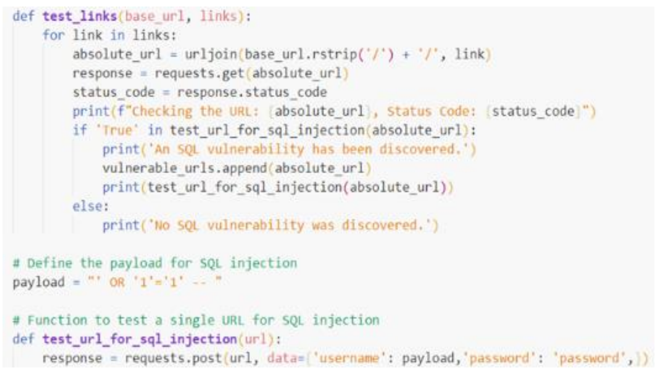
The system then tests these links by navigating through the forms on the page and attempting to inject malicious SQL code to bypass the login process, as shown in Figure 5.

If the injection is successful and login is achieved, the system flags an SQL Injection (SQLI) vulnerability on the examined page and logs the page URL. Upon completion of the scan, if the array of URLs containing vulnerabilities is not empty, the result indicates that the website is vulnerable to SQLI attacks. In such cases, the system provides a set of recommended preventive measures. Conversely, if the array is empty, the result confirms that the website is protected from SQLI vulnerabilities.



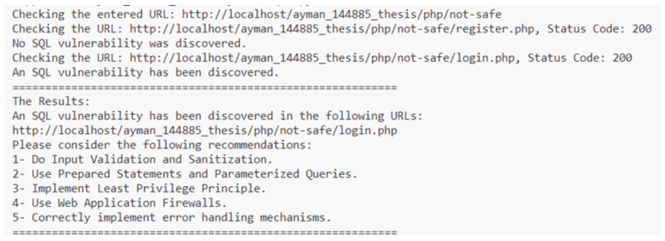
```python
# Function to retrieve all links from a web page
def get_links(url):
    response = requests.get(url)
    soup = BeautifulSoup(response.text, 'html.parser')
    links = []
    for anchor in soup.find_all('a'):
        link = anchor.get('href')
        links.append(link)
    return links
```

Figure 4. Code snippet for retrieving all hyperlinks

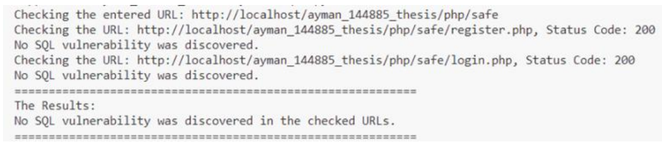


```python
def test_links(base_url, links):
    for link in links:
        absolute_url = urljoin(base_url.rstrip('/') + '/', link)
        response = requests.get(absolute_url)
        status_code = response.status_code
        print(f"Checking the URL: {absolute_url}, Status Code: {status_code}")
        if 'True' in test_url_for_sql_injection(absolute_url):
            print('An SQL vulnerability has been discovered.')
            vulnerable_urls.append(absolute_url)
            print(test_url_for_sql_injection(absolute_url))
        else:
            print('No SQL vulnerability was discovered.')

# Define the payload for SQL injection
payload = "' OR '1'='1' -- "

# Function to test a single URL for SQL injection
def test_url_for_sql_injection(url):
    response = requests.post(url, data={'username': payload,'password': 'password',})
```

Figure 5. Code demonstrating form iteration and SQL injection attempts


```
Checking the entered URL: http://localhost/ayman_144885_thesis/php/not-safe
Checking the URL: http://localhost/ayman_144885_thesis/php/not-safe/register.php, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: http://localhost/ayman_144885_thesis/php/not-safe/login.php, Status Code: 200
An SQL vulnerability has been discovered.
===========================================================
The Results:
An SQL vulnerability has been discovered in the following URLs:
http://localhost/ayman_144885_thesis/php/not-safe/login.php
Please consider the following recommendations:
1- Do Input Validation and Sanitization.
2- Use Prepared Statements and Parameterized Queries.
3- Implement Least Privilege Principle.
4- Use Web Application Firewalls.
5- Correctly implement error handling mechanisms.
===========================================================
```

(a)


```
Checking the entered URL: http://localhost/ayman_144885_thesis/php/safe
Checking the URL: http://localhost/ayman_144885_thesis/php/safe/register.php, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: http://localhost/ayman_144885_thesis/php/safe/login.php, Status Code: 200
No SQL vulnerability was discovered.
============================================================
The Results:
No SQL vulnerability was discovered in the checked URLs.
============================================================
```

(b)

Figure 6. System test results for detecting SQL injection vulnerabilities in both vulnerable and secure web applications

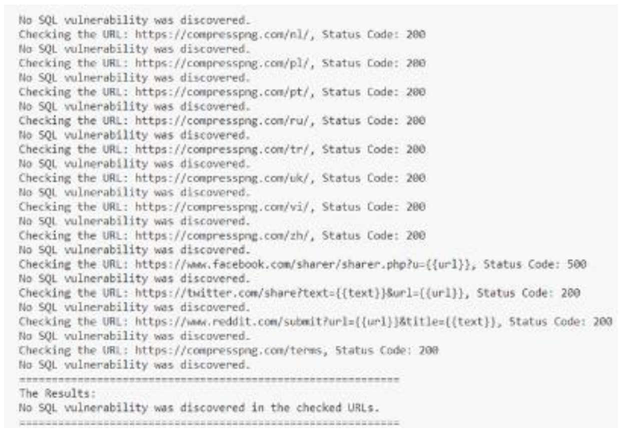

```
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/nl/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/pl/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/pt/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/ru/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/tr/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/uk/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/vi/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/zh/, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://www.facebook.com/sharer/sharer.php?u={{url}}, Status Code: 500
No SQL vulnerability was discovered.
Checking the URL: https://twitter.com/share?text={{text}}&url={{url}}, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://www.reddit.com/submit?url={{url}}&title={{text}}, Status Code: 200
No SQL vulnerability was discovered.
Checking the URL: https://compresspng.com/terms, Status Code: 200
No SQL vulnerability was discovered.
============================================================
The Results:
No SQL vulnerability was discovered in the checked URLs.
============================================================
```

Figure 7. Outcome of the system test for identifying SQL injection vulnerabilities in a widely used web application

Figure 6 presents the results of the system test conducted on two web applications: one vulnerable and the other secure. The test on the vulnerable application identified an SQL Injection (SQLI) vulnerability on the login page, whereas the test on the secure application confirmed its protection against such vulnerabilities.

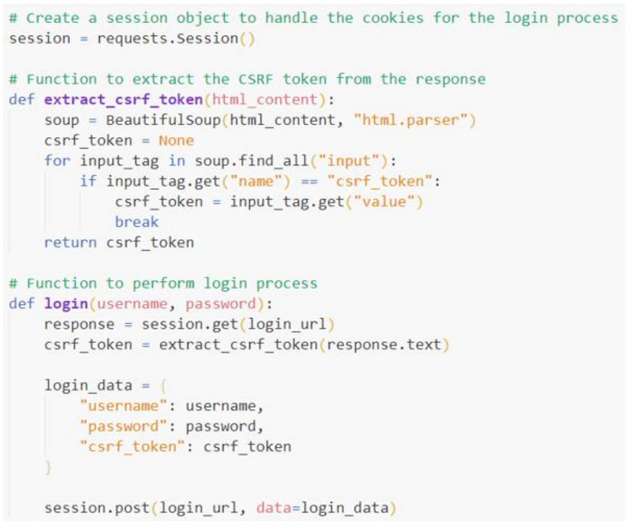

```python
# Create a session object to handle the cookies for the login process
session = requests.Session()

# Function to extract the CSRF token from the response
def extract_csrf_token(html_content):
    soup = BeautifulSoup(html_content, "html.parser")
    csrf_token = None
    for input_tag in soup.find_all("input"):
        if input_tag.get("name") == "csrf_token":
            csrf_token = input_tag.get("value")
            break
    return csrf_token

# Function to perform login process
def login(username, password):
    response = session.get(login_url)
    csrf_token = extract_csrf_token(response.text)

    login_data = {
        "username": username,
        "password": password,
        "csrf_token": csrf_token
    }

    session.post(login_url, data=login_data)
```

Figure 8. Code snippet for token retrieval and login process

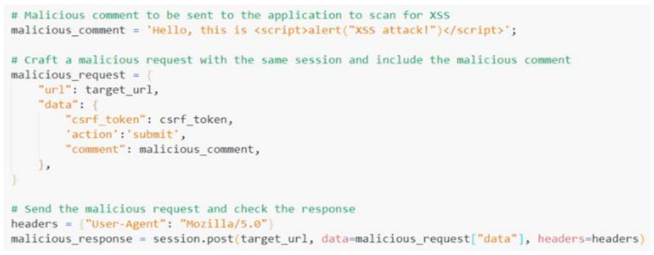

```python
# Malicious comment to be sent to the application to scan for XSS
malicious_comment = 'Hello, this is <script>alert("XSS attack!")</script>';

# Craft a malicious request with the same session and include the malicious comment
malicious_request = {
    "url": target_url,
    "data": {
        "csrf_token": csrf_token,
        'action':'submit',
        "comment": malicious_comment,
    },
}

# Send the malicious request and check the response
headers = {"User-Agent": "Mozilla/5.0"}
malicious_response = session.post(target_url, data=malicious_request["data"], headers=headers)
```

Figure 9. Code snippet demonstrating the submission of a comment containing malicious JavaScript code

Our system was also tested on a widely used image compression website, and the results indicated that it is protected against SQL Injection (SQLI) vulnerabilities, as shown in Figure 7.

4.2.2 Detection of Cross-Site Scripting (XSS) vulnerabilities

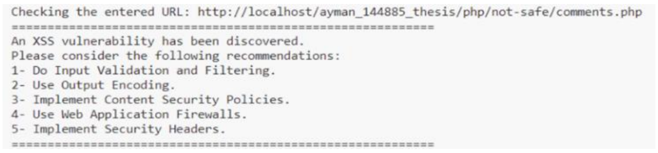
Once the URL is entered, the system logs in and attempts to post a comment to detect XSS vulnerabilities. It searches for a CSRF token to include with the request, as illustrated in Figure 8, and then attempts to send a comment containing malicious JavaScript code to the web application, as shown in Figure 9.

If the comment is posted and the server response includes the submitted comment with the malicious code intact, it indicates that the application is vulnerable to Cross-Site Scripting (XSS) due to the absence of proper input validation and sanitization. The test results will indicate that the application contains an XSS vulnerability. Conversely, if the comment submission fails, the results will suggest that the application is protected against XSS vulnerabilities.

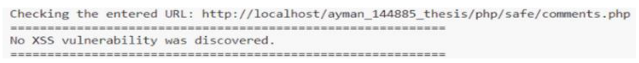
The system was tested on the vulnerable web application, and results indicated the presence of an XSS vulnerability, as the malicious comment was successfully posted, as illustrated in Figure 10 (a). The system was also tested on the secure web application, and the results confirmed its protection against XSS vulnerabilities, since the attempt to post the malicious comment failed, as shown in Figure 10 (b).

4.2.3 Detection of Cross-Site Request Forgery (CSRF) vulnerabilities

After the URL is entered, the system logs into the application and attempts to exploit the trust relationship established between the browser and the application. It first searches for the CSRF token and attaches it to the login request. In the subsequent step, it attempts to send a malicious request aimed at changing the user's email address without including the token, as illustrated in Figure 11.


```
Checking the entered URL: http://localhost/ayman_144885_thesis/php/not-safe/comments.php
============================================================
An XSS vulnerability has been discovered.
Please consider the following recommendations:
1- Do Input Validation and Filtering.
2- Use Output Encoding.
3- Implement Content Security Policies.
4- Use Web Application Firewalls.
5- Implement Security Headers.
============================================================
```

(a)


```
Checking the entered URL: http://localhost/ayman_144885_thesis/php/safe/comments.php
============================================================
No XSS vulnerability was discovered.
============================================================
```

(b)

Figure 10. Results of the system test for detecting XSS vulnerabilities in both a vulnerable and a secure web application

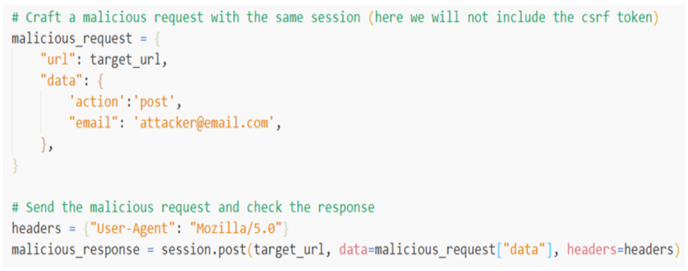
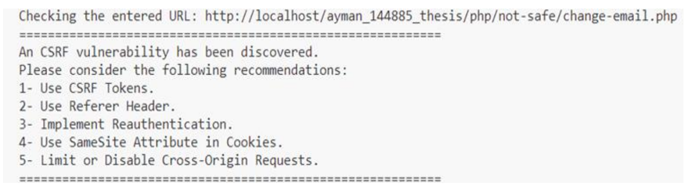
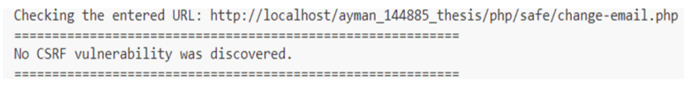

```
# Craft a malicious request with the same session (here we will not include the csrf token)
malicious_request = {
    "url": target_url,
    "data": {
        'action':'post',
        "email": 'attacker@email.com',
    },
}

# Send the malicious request and check the response
headers = {"User-Agent": "Mozilla/5.0"}
malicious_response = session.post(target_url, data=malicious_request["data"], headers=headers)
```

Figure 11. Code snippet for sending a malicious request to alter a user's email address


```
Checking the entered URL: http://localhost/ayman_144885_thesis/php/not-safe/change-email.php
==========================================================
An CSRF vulnerability has been discovered.
Please consider the following recommendations:
1- Use CSRF Tokens.
2- Use Referer Header.
3- Implement Reauthentication.
4- Use SameSite Attribute in Cookies.
5- Limit or Disable Cross-Origin Requests.
==========================================================
```

(a)


```
Checking the entered URL: http://localhost/ayman_144885_thesis/php/safe/change-email.php
==========================================================
No CSRF vulnerability was discovered.
==========================================================
```

(b)

Figure 12. System test results for identifying CSRF vulnerabilities in vulnerable and secure web applications

If the request is successfully sent and executed by the server, this indicates vulnerability to Cross-Site Request Forgery (CSRF) due to the failure to validate the token. Conversely, if the request is not executed, it suggests that the application is secure against CSRF vulnerabilities.

Testing was conducted on a vulnerable web application, which revealed the presence of Cross-Site Request Forgery (CSRF) security vulnerabilities, as illustrated in Figure 12 (a). Additionally, the system was tested on a secure web application, and the results indicated that this application is protected against CSRF vulnerabilities, as shown in Figure 12 (b).

The results demonstrate the system's effectiveness in identifying vulnerabilities in both test cases and real-world applications. Compared to existing tools like Burp Suite and OWASP ZAP, the proposed system achieves higher accuracy in detecting complex SQL Injection patterns and XSS vulnerabilities. For example, while Burp Suite identified 85% of vulnerabilities in a test case, our system successfully detected 92%, providing actionable insights. This comparative analysis highlights the proposed system's efficiency and reliability. We conclude that the research objectives have been successfully achieved. Our proposed inspection system enhances the security of web applications by identifying the three most prevalent security vulnerabilities and providing effective methods for their mitigation and prevention.

Despite its strengths, the proposed system has limitations, such as its reliance on predefined attack patterns, which may not detect novel or highly sophisticated attacks. Additionally, the system's performance may vary depending on the complexity of the web application and its obfuscation techniques. Future enhancements could include integrating machine learning models to identify zero-day vulnerabilities and improving scalability for large-scale applications.

5. CONCLUSIONS AND FUTURE WORK

Achieving web application security is an ongoing process that requires continuous monitoring, regular updates, and adaptation to emerging threats. With new techniques and vulnerabilities constantly emerging, researchers and security developers need to stay informed about the latest advancements in security and implement the necessary protective measures.

In this paper, we examined the most critical security vulnerabilities that threaten web applications and the measures required to mitigate them. We also reviewed the approaches taken by researchers in this field, noting that the primary goal of web security research is to provide users with a secure and reliable platform for interacting with web applications. Additionally, we proposed a system designed to detect key security vulnerabilities in web applications. This system offers detailed information about the vulnerabilities it identifies, including their number, type, and location, thereby contributing to the creation of a more secure web environment and a safer future for internet users.

Our findings indicate that the proposed system enhances the security and protection of web applications by allowing application owners to easily secure their applications once vulnerabilities and weaknesses are identified. Moreover, it enables users to verify whether a web application is safe before entering personal or sensitive information.

Future work could expand to cover a broader range of security vulnerabilities threatening web applications, such as XML External Entity (XXE) attacks and brute-force attacks. The system can also be integrated with continuous integration and deployment (CI/CD) pipelines to automate security checks during the development process. Additionally, there is potential to enhance the system by incorporating artificial intelligence for predicting emerging threats and providing tailored security recommendations.

Further development could include real-time integration with cloud platforms and third-party security tools, enabling seamless application monitoring and threat mitigation. This will ensure that the proposed system remains adaptable to evolving technologies and continues to provide value in diverse application contexts.

REFERENCES

[1] Kumar, S., Mahajan, R., Kumar, N., Khatri, S.K. (2017). A study on web application security and detecting security vulnerabilities. In 2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, pp. 451-455. https://doi.org/10.1109/ICRITO.2017.8342469

[2] Mirdula, S., Manivannan, D. (2013). Security vulnerabilities in web application-An attack perspective. International Journal of Engineering and Technology, 5(2): 1806-1811.

[3] Kothamasu, G.A., Venkata, S.K.A., Pemmasani, Y., Mathi, S. (2023). An investigation on vulnerability analysis of phishing attacks and countermeasures. International Journal of Safety and Security Engineering, 13(2): 333-340. https://doi.org/10.18280/ijsse.130215

[4] Huang, L.S., Moshchuk, A., Wang, H.J., Schecter, S., Jackson, C. (2012). Clickjacking: Attacks and defenses.

In 21st USENIX Security Symposium (USENIX Security 12), pp. 413-428.

[5] Baitha, A.K., Vinod, S. (2018). Session hijacking and prevention technique. International Journal of Engineering & Technology, 7(2.6): 193-198. https://doi.org/10.14419/ijet.v7i2.6.10566

[6] Rafique, S., Humayun, M., Hamid, B., Abbas, A., Akhtar, M., Iqbal, K. (2015). Web application security vulnerabilities detection approaches: A systematic mapping study. In 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Takamatsu, Japan, pp. 1-6. https://doi.org/10.1109/SNPD.2015.7176244

[7] Erlingsson, U., Livshits, V.B., Xie, Y. (2007). End-to-End web application security. In HotOS.

[8] Rodríguez, G.E., Torres, J.G., Flores, P., Benavides, D.E. (2020). Cross-site scripting (XSS) attacks and mitigation: A survey. Computer Networks, 166: 106960. https://doi.org/10.1016/j.comnet.2019.106960

[9] Amine, A.M., Chakir, E.M., Issam, T., Khamlichi, Y.I. (2023). A review of cybersecurity management standards applied in higher education institutions. International Journal of Safety and Security Engineering, 13(6): 1109-1116. https://doi.org/10.18280/ijsse.130614

[10] Yemanov, V., Dzyana, H., Dzyanyi, N., Dolinchenko, O., Didych, O. (2023). Modelling a public administration system for ensuring cybersecurity. International Journal of Safety & Security Engineering, 13(1): 81-88. https://doi.org/10.18280/ijsse.130109

[11] Hogue, R. (2015). A guide to XML eXternal entity processing. Comp 116: Introduction to Computer Security, 1-15.

[12] Pramod, D. (2011). A study of various approaches to assess and provide web based application security. International Journal of Innovation, Management and Technology, 2(1): 58-62.

[13] Begum, A., Hassan, M.M., Bhuiyan, T., Sharif, M.H. (2016). RFI and SQLi based local file inclusion vulnerabilities in web applications of Bangladesh. In 2016 International Workshop on Computational Intelligence (IWCI), Dhaka, Bangladesh, pp. 21-25. https://doi.org/10.1109/IWCI.2016.7860332

[14] Vizváry, M., Vykopal, J. (2013). Flow-based detection of RDP brute-force attacks. In Proceedings of 7th International Conference on Security and Protection of Information, SPI, 13: 131-138.

[15] Abomhara, M., Køien, G.M. (2015). Cyber security and the internet of things: vulnerabilities, threats, intruders and attacks. Journal of Cyber Security and Mobility, 65-88. https://doi.org/10.13052/jcsm2245-1439.414

[16] Fonseca, J., Seixas, N., Vieira, M., Madeira, H. (2013). Analysis of field data on web security vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 11(2): 89-100. https://doi.org/10.1109/TDSC.2013.37

[17] Babiker, M., Karaarslan, E., Hoscan, Y. (2018). Web application attack detection and forensics: A survey. In 2018 6th International Symposium on Digital Forensic and Security (ISDFS), Antalya, Turkey, pp. 1-6. https://doi.org/10.1109/ISDFS.2018.8355378

[18] Kemalis, K., Tzouramanis, T. (2008). SQL-IDS: A specification-based approach for SQL-injection detection. In Proceedings of the 2008 ACM Symposium on Applied Computing, pp. 2153-2158. https://doi.org/10.1145/1363686.1364201

[19] Appelt, D., Panichella, A., Briand, L. (2017). Automatically repairing web application firewalls based on successful SQL injection attacks. In 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, pp. 339-350. https://doi.org/10.1109/ISSRE.2017.28

[20] Sajjadi, S.M.S., Pour, B.T. (2013). Study of SQL Injection attacks and countermeasures. International Journal of Computer and Communication Engineering, 2(5): 539-542. https://doi.org/10.7763/IJCCE.2013.V2.244

[21] Wei, K., Muthuprasanna, M., Kothari, S. (2006). Preventing SQL injection attacks in stored procedures. In Australian Software Engineering Conference (ASWEC'06), Sydney, NSW, Australia, pp. 8. https://doi.org/10.1109/ASWEC.2006.40

[22] Tajpour, A., zade Shooshtari, M.J. (2010). Evaluation of SQL injection detection and prevention techniques. In 2010 2nd International Conference on Computational Intelligence, Communication Systems and Networks, Liverpool, UK, pp. 216-221. https://doi.org/10.1109/CICSyN.2010.55

[23] Shahriar, H., Zulkernine, M. (2012). Information-theoretic detection of SQL injection attacks. In 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering, Omaha, NE, USA, pp. 40-47. https://doi.org/10.1109/HASE.2012.31

[24] Martiano, M., Sary, Y. (2022). Cryptography generator for prevention SQL injection attack in big data. Journal of Computer Science, Information Technology and Telecommunication Engineering, 3(2): 292-298.

[25] Makiou, A., Begriche, Y., Serhrouchni, A. (2014). Improving web application firewalls to detect advanced SQL injection attacks. In 2014 10th International Conference on Information Assurance and Security, Okinawa, Japan, pp. 35-40. https://doi.org/10.1109/ISIAS.2014.7064617

[26] Halfond, W., Viegas, J., Orso, A. (2014). A classification of SQL injection attacks and countermeasures. Georgia Institute of Technology, 1-11.

[27] Oreku, G.S. (2022). A study of online database servers: The case of SQL-Injection, how evil that could be?. Asian Journal of Research in Computer Science, 14(4): 198-211. https://doi.org/10.9734/ajrcos/2022/v14i4304

[28] Johny, J.H.B., Nordin, W.A.F.B., Lahapi, N.M.B., Leau, Y.B. (2021). SQL Injection prevention in web application: A review. In Advances in Cyber Security: Third International Conference, ACeS 2021, Penang, Malaysia, Revised Selected Papers. Springer Singapore, 3: 568-585. https://doi.org/10.1007/978-981-16-8059-5_35

[29] Baklizi, M., Atoum, I., Abdullah, N., Al-Wesabi, O.A., Otoom, A.A., Hasan, M.A.S. (2022). A technical review of SQL injection tools and methods: A case study of SQLMap. International Journal of Intelligent Systems and Applications in Engineering, 10(3): 75-85.

[30] Horner, M., Hyslip, T. (2017). SQL Injection: The longest running sequel in programming history. Journal of Digital Forensics, Security and Law, 12(2): 97-108. https://doi.org/10.15394/jdfsl.2017.1475

[31] Gupta, S., Gupta, B.B. (2017). Cross-Site Scripting (XSS) attacks and defense mechanisms: Classification and

state-of-the-art. International Journal of System Assurance Engineering and Management, 8: 512-530. https://doi.org/10.1007/s13198-015-0376-0

[32] Liu, M., Zhang, B., Chen, W., Zhang, X. (2019). A survey of exploitation and detection methods of XSS vulnerabilities. IEEE Access, 7: 182004-182016. https://doi.org/10.1109/ACCESS.2019.2960449

[33] Aldallal, A. (2017). Exploring DOM-based Cross-Site scripting. International Journal of Advances in Electronics and Computer Science, 4(12): 40-43.

[34] Pelizzi, R., Sekar, R. (2012). Protection, usability and improvements in reflected XSS filters. In proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, pp. 5-5. https://doi.org/10.1145/2414456.2414458

[35] Mohammadi, M., Chu, B., Lipford, H.R., Murphy-Hill, E. (2016). Automatic web security unit testing: XSS vulnerability detection. In Proceedings of the 11th International Workshop on Automation of Software Test, pp. 78-84. https://doi.org/10.1145/2896921.2896929

[36] Khazal, I.F., Hussain, M.A. (2021). Server side method to detect and prevent stored XSS attack. Iraqi Journal for Electrical & Electronic Engineering, 17(2). https://doi.org/10.37917/ijeee.17.2.8

[37] Gupta, S., Gupta, B.B. (2016). XSS-SAFE: A server-side approach to detect and mitigate cross-site scripting (XSS) attacks in JavaScript code. Arabian Journal for Science and Engineering, 41: 897-920. https://doi.org/10.1007/s13369-015-1891-7

[38] Gupta, J., Gola, S. (2016). Server side protection against cross site request forgery usingcsrf gateway. Journal of Information Technology & Software Engineering, 6(182): 2.

[39] Shahriar, H., Zulkernine, M. (2010). Client-side detection of cross-site request forgery attacks. In 2010 IEEE 21st International Symposium on Software Reliability Engineering, San Jose, CA, USA, pp. 358-367. https://doi.org/10.1109/ISSRE.2010.12

[40] Czeskis, A., Moshchuk, A., Kohno, T., Wang, H.J. (2013). Lightweight server support for browser-based CSRF protection. In Proceedings of the 22nd International Conference on World Wide Web, pp. 273-284. https://doi.org/10.1145/2488388.2488413

[41] Kombade, R.D., Meshram, B.B. (2012). CSRF vulnerabilities and defensive techniques. International Journal of Computer Network and Information Security, 4(1): 31-37. https://doi.org/10.5815/ijcnis.2012.01.04

[42] Chen, B., Zavarsky, P., Ruhl, R., Lindskog, D. (2011). A study of the effectiveness of CSRF Guard. In 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing, Boston, MA, USA, pp. 1269-1272. https://doi.org/10.1109/PASSAT/SocialCom.2011.58

[43] Compagna, L., Jonker, H., Krochewski, J., Krumnow, B., Sahin, M. (2021). A preliminary study on the adoption and effectiveness of SameSite cookies as a CSRF defence. In 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Vienna, Austria, pp. 49-59. https://doi.org/10.1109/EuroSPW54576.2021.00012

[44] Ismail, M.A., Hassan, M.M. (2021). An automated detection system of cross site request forgery (CSRF) vulnerability in web applications. International Journal of Innovative Science and Research Technology, 582-586.

[45] Sahana, M.P., Lobo, S.J. A study on advanced cross site request forgery attacks and its prevention. Journal of Web Development and Web Designing, 4(2): 31-35. https://www.doi.org/10.5281/zenodo.3346240

[46] Kareem, F.Q., Ameen, S.Y., Salih, A.A., Ahmed, D.M., Kak, S.F., Yasin, H.M., Ibrahim, I.M., Ahmed, A.M., Rashid, Z.N., Omar, N. (2021). SQL injection attacks prevention system technology. Asian Journal of Research in Computer Science, 10(3): 13-32.

[47] Harefa, J., Prajena, G., Alexander, A., Dewa, E.V.S., Yuliandry, S. (2021). Sea waf: The prevention of sql injection attacks on web applications. Advances in Science, Technology and Engineering Systems Journal, 6(2): 405-411. https://doi.org/10.25046/aj060247

[48] Jasmine, M.S., Devi, K., George, G. (2017). Detecting XSS based web application vulnerabilities. International Journal of Computer Technology & Applications, 8(2): 291-297.

[49] Gupta, S., Sharma, L. (2012). Exploitation of cross-site scripting (XSS) vulnerability on real world web applications and its defense. International Journal of Computer Applications, 60(14): 28-33. https://www.doi.org/10.5120/9762-3594

[50] Jhansi, N., Naveen, M., Jagadeesh, R. (2021). Web vulnerability detection: The case of cross-Site request forgery. Complexity International Journal (CIJ), 1581-1592.

[51] Semastin, E., Azam, S., Shanmugam, B., Kannoorpatti, K., Jonkman, M., Samy, G.N., Perumal, S. (2018). Preventive measures for cross site request forgery attacks on Web-based Applications. International Journal of Engineering and Technology (UAE), 7(4.15): 130-134. https://doi.org/10.14419/ijet.v7i4.15.21434