Goyal

by Bhargavi Goyal

Submission date: 21-Jun-2026 11:21AM (UTC+0530)
Submission ID: 2986819534
File name: Bhargavi_Goyal_Thesis.pdf (1.49M)
Word count: 10218
Character count: 57432

DECONSTRUCTING CROSS-SITE SCRIPTING (XSS) EXPLOITATION: A STAGE-WISE EVALUATION OF COOKIE AND BROWSER-BASED DEFENCE MECHANISMS IN MODERN WEB APPLICATIONS

Thesis Submitted
in Partial Fulfillment of the Requirements for the
Degree of

MASTER OF TECHNOLOGY

in

Computer Science Engineering

by

Bhargavi Goyal
(Roll No. 2k24/CSE/01)

Under the Supervision of

Prof. Manoj Kumar
Assistant Professor
Department of Computer Science Engineering
Delhi Technological University

Department of Computer Science Engineering

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-110042, India

June, 2026

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-110042

ACKNOWLEDGEMENTS

I wish to express my sincerest gratitude to **Prof. Manoj Kumar** for his continuous guidance and mentorship that he provided me during the project. He showed me the path to achieve my targets by explaining all the tasks to be done and explained to me the importance of this project as well as its academic and practical relevance. He was always ready to help me and clear my doubts regarding any hurdles in this project. Without his constant support and motivation, this project would not have been successful.

Place: Delhi

Bhargavi Goyal

Date: _____

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

I, **Bhargavi Goyal**, Roll No. **2k24/CSE/01**, student of M.Tech (Computer Science Engineering), hereby declare that the project dissertation titled "**Deconstructing Cross-Site Scripting (XSS) Exploitation: A Stage-Wise Evaluation of Cookie and Browser-Based Defence Mechanisms in Modern Web Applications**" which is submitted by me to the Computer Science Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of degree of Master of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma, Associateship, Fellowship or other similar title or recognition.

Candidate's Signature

This is to certify that the student has incorporated all the corrections suggested by the examiners in the thesis and the statement made by the candidate is correct to the best of our knowledge.

Signature of Supervisor(s)

Signature of External Examiner

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-110042

CERTIFICATE

I hereby certify that the Project Dissertation titled "**Deconstructing Cross-Site Scripting (XSS) Exploitation: A Stage-Wise Evaluation of Cookie and Browser-Based Defence Mechanisms in Modern Web Applications**" which is submitted by **Bhargavi Goyal**, Roll No. **2k24/CSE/01**, Computer Science Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is a record of the project work carried out by the student under my supervision.

To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: Delhi

Date: _____

Prof. Manoj Kumar

SUPERVISOR

Deconstructing Cross-Site Scripting (XSS) Exploitation: A Stage-Wise Evaluation of Cookie and Browser-Based Defence Mechanisms in Modern Web Applications

Bhargavi Goyal

ABSTRACT

Cross-Site Scripting (XSS) is a dangerous problem in web applications in which attackers inject malicious scripts and bypass security controls. In the worst scenario, the attacker steals the session and login token and illegally takes access to the victim's account. In this study, XSS attacks are described in detail, and the attack is divided into four major stages: persistence, execution, egress, and replay. Previous studies mainly focus on detecting XSS attacks and classifying them, whereas this study explains the whole process of how an attack progresses from vulnerability to session hijack.

By implementing the demo prototype for a web application, this study demonstrates that a stored XSS attack leads to session-cookie exposure and session hijacking in a controlled research setting. The study also examines the effectiveness of preventing HttpOnly, SameSite and CSP-like security prevention attacks at each stage. The findings demonstrate that cookie-based protection protects tokens or cookies from theft, and browser-based protection provides higher security by preventing scripts from executing. However, the effectiveness of these protections depends on real-world factors such as proper configuration, deployment, and the complexity of the application.

According to the study for strong security, a "defence in depth" strategy should be used, which means that at different stages, multiple security layers should be used to prevent an attack. Using a combination of demonstration and evaluation, the research presents a systematic overview of the XSS attack and the ways to strengthen the security of Web applications in the highly complex situation.

LIST OF PUBLICATIONS

1. **Stored Cross-Site Scripting to Session Hijacking: Demonstration and Defensive Analysis.**
2. **Defence Mechanisms Against Cross-Site Scripting (XSS): A Review of Cookie and Browser-Based Protections.**

TABLE OF CONTENTS

List of Tables

List of Figures

LIST OF SYMBOLS, ABBREVIATIONS AND NOMENCLATURE

XSS	Cross-Site Scripting	CSP	Content Security Policy
DOM	Document Object Model	HTTP	Hypertext Transfer Protocol

CHAPTER 1

INTRODUCTION

With its prevalence and impact, Cross-Site Scripting (XSS) stands out as one of the most significant security risks in today's web applications. While coding securely, developing secure Web frameworks, and adding JavaScript protections to browsers have advanced, XSS continues to be listed in the OWASP Top 10 persistently because it's easy to create, easy to exploit, and can be very harmful. The vulnerability arises because web applications do not adequately sanitise, validate or encode input from a user that is controlled by an attacker, instead allowing them to enter malicious scripts into the protected browser, which will be run under the domain of the trusted web application.

Modern web application architectures have introduced many new opportunities for XSS attacks to increase the attack surface. The shift from traditional server-rendered applications to highly dynamic, API-driven, and JavaScript-intensive applications has resulted in a huge increase in the attack surface for XSS. Today's applications are heavily dependent on the use of client-side frameworks, third-party scripts and browser APIs that create a number of execution environments and intricate data flows. This means that attackers are free from the constraints of traditional input injection and can now rely on the ability to make advanced attacks by exploiting DOM manipulation, client-side sinks, and browser storage.

One of the most severe side effects of XSS is session hijacking, in which the attacker generates a script on the victim's browser to be able to access confidential session data like cookies, tokens, etc. These session IDs are soft and might be utilised once more without the characterisation of a card or ID. In addition to its role as an attack over script injection, XSS is also a big threat to the integrity, security and effectiveness of an authentication and authorisation system. Modern Web security has two types of protection mechanisms: one uses cookies, and the other is browser-enforced; both are employed to combat these threats. Advances like HttpOnly, Secure and SameSite are there for a reason – to keep the integrity of cookies during a session, and make sure that no authentication data is leaked upon being passed or accessed. Like Content Security Policy (CSP), Trusted Types, and Subresource Integrity (SRI), these are browser-level mechanisms whose only purpose is to prevent script execution and reduce unsafe runtime behaviour.

None of these mechanisms exists in real applications; they are not fully adopted and are misconfigured. This gap between theory and practical security is due to a need for an in-depth study that would be able to figure out the effectiveness of each discussed defence strategy and what they really do in real-time and space scenarios to protect against XSS attacks.

The primary focus of this thesis is a deconstructive and comparative analysis of types of Cross-Site Scripting (XSS) exploits, examination of the transition of a stored XSS attack to become a session hijacking attack, and examination of protecting against XSS attacks using cookie-based and browser-based defences at various stages of the attack lifecycle.

1.1 Background and Objectives

Web applications have become essential in processing critical user data, such as credentials, personal details and transactions. But the growing development of client-side activities, dynamic content creation, and integration with third-party systems has created a larger attack surface for web applications. One such common and significant injection attack is Cross-Site Scripting (XSS). XSS vulnerabilities enable the insertion of malicious scripts into legitimate web applications that run in the victim's browser with the permissions of the website. Because the browser will execute code injected by an attacker as part of the trusted web application, the attacker can breach application trust boundaries and perform malicious operations, such as stealing sensitive information and reusing sessions. A serious impact of XSS is session hijacking, in which the adversary uses XSS vulnerabilities to access or reuse session identifiers (cookies) to impersonate the victim. Real-world tests reveal that without adequate input sanitation and cookie security measures, attackers can steal user session IDs and gain access to the system without the user's credentials.

Although there are several ways of protecting systems from XSS, such as secure programming practices and browser-side policies, the prevalence of XSS is still high, owing to a lack of compliance, misconfigurations and ever-growing complexity of applications. Most of the research studied so far has centered on the detection and classification of XSS attacks, with little work done on the whole process of exploiting XSS from injection to session hijacking. In this research, we propose a staged-based model of XSS exploitation, dividing the exploitation process into four stages: persistence, execution, egress and replay. The stage-wise approach allows a more precise view of vulnerability propagation amongst stages, and offers insights into the interaction between different stages and various defence techniques. The main focus of this study is to assess the efficacy of cookie-based and browser-based defence measures in this stage-wise model. The research approach, which mixes demonstration and analysis, seeks to better understand XSS attack behaviour and develop a proven defence strategy for contemporary web applications.

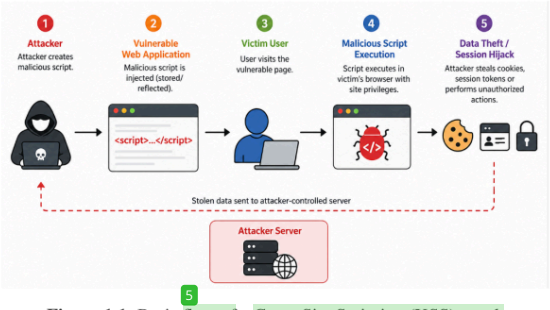

Figure 1.1. Basic flow of a Cross-Site Scripting (XSS) attack

1.2 Scope

This research examines the behaviour of persistent XSS attacks in an experimental setup and assesses the escalation of the attack to session hijacking. The research particularly evaluates the role of cookie-based and browser-based control mechanisms, such as HttpOnly, SameSite and Content Security Policy (CSP), respectively, in thwarting various phases of the attack. It only considers client-side and session-based attacks, as opposed to other types of attacks such as SQL injections or server-based attacks. Moreover, the experimental environment is created as a simple environment to showcase the attack chain, and does not account for the complexities found in enterprise production systems.

1.3 Research Questions and Problem Statement

Even with the advancements of Web security technologies, Cross-Site Scripting (XSS) is still one of the most common vulnerabilities in today's Web applications. Some current mitigation measures, such as safe coding, policies that are implemented by browsers, and security attributes of cookies, have successfully addressed some, but not all, risks. Whilst most work is currently concentrated on vulnerability detection, payload generation or payload classification, few studies have been able to get to grips with the entire exploitation lifecycle of XSS or the effectiveness of XSS security mechanisms at various points in the attack.

The situation gets worse when XSS prompts to session hijacking, in which the attackers use the vulnerabilities in session management and the trust of the browsers to steal user identities. Securing is possible with varying security mechanisms like "HttpOnly", "SameSite" and "Content Security Policy" (CSP), and their security levels vary in relation to deployment and configuration in actual use. This leaves a security protection to the service divide.

This prompted us to conduct the present research to examine the severity of XSS as presented in a staged attack approach and to measure XSS defence mechanisms throughout the different stages of the attack. Some of the questions that we can answer

after the thorough research and through this thesis are as follows:

Q1: What are the stages of a persistent XSS vulnerability in a modern web application towards session hijacking?

Q2: Are cookies effective in preventing session hijacking attacks or replays?

Q3: How effective are browser-based defences for defending against the XSS execution and data exfiltration attacks?

Q4: What are the current protection mechanisms available to defend against XSS, and what mechanisms will need to be implemented in the future to complete the cycle of attacks against XSS?

1.4 Significance of the Study

The relevance of this research is based upon a stage-based analysis of XSS exploits and defenses. This work is the first to treat a stored XSS attack as a sequential attack by looking at the whole attack life cycle from an XSS attack to a session hijacking scenario, although earlier works looked at payload detection, or browser security policies. The study examines the processes of an attack, tracking it during the 'persistence', 'execution', 'egress' and 'replay' stages, to give a better sense of how vulnerabilities ripple and weaknesses are overcome.

The research adds on work of Web Security by providing a detailed and comparative analysis of the Cookie security and the Browser security in a consistent experimental setup. The study does not consider defence mechanisms as separate entities, but examines the interactions of mechanisms and how they reinforce each other at various stages of the exploitation. This method allows for practical insights into how to develop and implement a defence-in-depth approach for developers, security researchers, and organizations interested in defense-in-depth.

Moreover, findings showcase some shortcomings in the configuration of policies, incomplete adoption, and dependence on legacy systems, pointing to the potential for future enhancements of browser security and secure Web development practices. As such, the aim of the study is not just to illustrate exploitation of XSS, but also to aid in creating more powerful and realistic defensive solutions for contemporary Web applications.

CHAPTER 2

TECHNICAL BACKGROUND

14
Cross-Site Scripting (XSS) is a client-side web security vulnerability [1, 2] in which untrusted input is interpreted by the browser as executable script [1]. Unlike many server-side attacks that directly target a database or operating system, XSS abuses the relationship between a trusted web application and the user's browser. When a vulnerable application stores, reflects, or dynamically inserts malicious input into a page without proper validation and encoding, the browser may execute that input as JavaScript. Since the script runs inside the origin of the trusted website [2], it can interact with page content, read accessible browser-side data, modify the Document Object Model (DOM), and send outbound network requests.

The technical importance of XSS comes from the fact that modern web applications rely heavily on browser-side execution [3]. Dynamic interfaces, asynchronous requests, client-side routing, third-party scripts and browser extensions [4], and token-based session handling have increased the amount of logic executed in the browser. This renders the browser as not just a display tool, but also an execution environment where application state, authentication and user actions are executed. Thus, an XSS attack has the potential to impact the user-session level of confidentiality, integrity and availability. XSS is approached as a multistage euphoric technique in this thesis, starting with an insecure web-app input and potentially leading to a session hijacking [2, 5].

2.1 Web Application Architecture and Browser Execution

This style, where the web application is divided into two components—one in the client and the other in the server—is a typical model for a web application. The application code is stored on the server, and the server is responsible for authentication, business logic implementation, and responding to the client. The client receives HTML, CSS, symbolic instructions, JavaScript, and other resources and displays these elements in an interactive interface. In "traditional" applications, the majority of content was server-generated. However, in more modern usage, much of the user interface can be constructed or developed 'within the browser' with JS. This change will enhance user-friendliness yet also create more vulnerability.

The browser processes a page by parsing HTML, applying CSS rules, executing JavaScript, and building the DOM. The DOM is a structured representation of the webpage [6] that scripts can read and modify. When a script executes, it can add new elements, change form values, read visible page content, attach event handlers, and initiate network requests. These abilities are necessary for legitimate application

features, but they become dangerous when an attacker can inject script code into the same execution environment [3].

Browsers apply the Same-Origin Policy (SOP) [7] to restrict how scripts from one origin interact with resources from another origin. An origin is normally defined by the scheme, host, and port of a URL. SOP prevents a malicious page from freely reading sensitive responses from another website. However, XSS is powerful because the injected script does not run as an external untrusted origin; it runs within the vulnerable application's own origin. As a result, the browser treats the malicious code as part of the trusted application. This is why even strong browser isolation rules may not fully protect a vulnerable site from XSS.

Modern applications also use APIs, browser storage, cookies, and client-side frameworks. JavaScript frameworks often update the DOM dynamically based on user input or server responses. If data is inserted into the DOM using unsafe methods, such as direct HTML assignment [6] without sanitisation, it may create DOM-based XSS. Similarly, if server-rendered pages include untrusted data without context-aware output encoding, stored or reflected XSS may occur. Therefore, secure design requires both server-side and client-side awareness.

2.2 Types of Cross-Site Scripting

XSS vulnerabilities are commonly classified into stored XSS, reflected XSS, and DOM-based XSS [1, 8]. Stored XSS occurs when malicious input is saved [1] by the application and later displayed to users. For example, a comment, profile field, message, or feedback form may store attacker-controlled content in a database. When another user views the affected page, the script executes automatically. Stored XSS is often considered highly severe because the payload persists and may affect many users over time.

Reflected XSS occurs when malicious input is included in an immediate response from the server. A common example is a crafted URL containing a script payload [9, 10] in a query parameter. If the server reflects this value into the page without proper encoding, the victim's browser may execute it after opening the link. Reflected XSS usually depends on social engineering, phishing, or user interaction, because the victim must access the crafted request. Although it may not persist in the application database, it can still compromise sessions if executed in an authenticated context.

DOM-based XSS occurs when the vulnerability exists primarily in client-side JavaScript [11]. In this case, the server may not directly insert the malicious payload into the response. Instead, JavaScript reads untrusted data from a source such as the URL fragment, query string, local storage, or postMessage event, and writes it into a dangerous sink such as innerHTML, document.write, or unsafe script construction. DOM-based XSS is especially relevant in modern single-page applications, where much of the rendering logic happens in the browser.

These categories are useful for understanding where the vulnerability is introduced, but in practice they can overlap. A stored payload may later be executed through a DOM sink, or a reflected payload may be processed by client-side routing code. For this reason, this thesis focuses not only on the category of XSS but also on the stages

of exploitation. The stage-wise view makes it possible to understand how an injected payload moves from storage or reflection to execution, egress, and replay.

2.3 Session Management and Cookie Security

Web applications use session management such that identity stays intact between multiple requests. Because HTTP is a stateless protocol, the server has to connect each request to the authenticated user using a session ID, a token or through some similar credentials. A cookie is a common mechanism that is used to store session identifiers in the browser [5]. When a user logs in, the server sends a cookie to the browser, and the browser automatically sends that cookie to the same website with future requests. Because of this process, the server, without asking for the credentials from the user again and again, recognises it.

During an XSS attack, this session cookie becomes a valuable target for the cookie attackers. If JavaScript can read a session cookie, an attacker may exfiltrate it to an external endpoint. The attacker may then replay the cookie in another browser or request tool to impersonate the victim. This is the technical basis of session hijacking in many XSS scenarios [12]. Even if the attacker cannot read the user's password, stealing a valid session identifier may be enough to access the account until the session expires or is invalidated.

Cookie attributes help reduce this risk. The HttpOnly attribute prevents JavaScript from reading the cookie [13, 14] through document.cookie. This is useful against direct cookie theft during the egress stage. The Secure attribute ensures that the cookie is sent only over HTTPS, reducing exposure over insecure channels. The SameSite attribute restricts when cookies are sent with cross-site requests and can reduce some cross-site request and session abuse scenarios. However, these attributes do not remove the XSS vulnerability itself. If any script executes inside a trusted website, then it can modify page content, perform an action on behalf of the user or misuse the functionality of the application.

Besides cookies, some applications can store tokens in localStorage or sessionStorage. This browser storage mechanism is accessible via JavaScript; if an XSS vulnerability is present, this becomes risky. In API-driven applications, token-based systems are very common, but storing sensitive tokens in this storage where scripts can access them can increase the impact of client-side injection [15, 16]. Where tokens are stored is important, but it is also important that they are not accessible by any scripts.

2.4 Browser-Based Defence Mechanisms

The main goal of browser-based defences is to prevent the execution of malicious scripts. Content Security Policy (CSP) is the most important security mechanism of the browser, which is used to mitigate XSS [17]. CSP permits the website to allow scripts, styles, images and sources from other sources. A strict CSP can block the inline scripts, restrict the origin of external scripts, and decrease the execution of injected scripts. But it is important to configure CSP properly. If unsafe-inline or broad wildcard sources are allowed in a policy, then protection becomes limited.

Trusted Types is a browser security feature that reduces DOM-based attacks [6]. It

prevents the direct use of browsers' dangerous DOM sinks, such as innerHTML and eval. The developer has to define a trusted policy that specifies approved rules. Based on these policies, content is approved to insert in DOM, not any random or untrusted string. However, this is not supported in every browser, requires updating legacy code and requires extra efforts in implementation.

Subresource Integrity (SRI) is also a browser feature that protects external scripts. The browser uses a cryptographic hash to verify if external files have been tampered with or not. If scripts are modified, then the browser does not execute the external script. Its main goal is to make third-party scripts safer, but it is not the direct solution for stored XSS.

Apart from this, sandboxed iframes, strict MIME type checking, and HTTPS-like secure context also improve security. They reduce security risks, but they are not a replacement for input validation or output encoding.

There is a limitation to browser-based defences that they will work properly only when they are properly configured. If CSP is too permissive, then attacks will not be blocked, and if policies are too strict, then legitimate features or normal behaviour can break. In these cases, developers compromise policy for compatibility while compromising security. That is why browser-based protections only work effectively when they are used alongside secure coding practices, proper testing and continuous policy monitoring [18].

2.5 Stage-Wise View of XSS Exploitation

To understand XSS (Cross-Site Scripting), four stages have been used: persistence, execution, egress, and replay.

Persistence: It refers to how an attacker's input becomes a permanent or temporary part of the application. For example, in Stored XSS, the payload gets stored in the database and is not stored in the case of DOM-based and reflected XSS, but comes in the processing flow of the page. This stage mainly depends on input validation, sanitisation, and safe storage [19].

Execution: This stage does not treat malicious input as normal text but as an active script. If output encoding is not properly configured, then there are chances that the payload might execute. To prevent this, CSP uses proper encoding and DOM APIs.

Egress: In this stage, the running script tries to send data, such as cookies, tokens, or form data, from the victim's browser. The browser restricts cross-origin requests but does not prevent outbound requests completely. To reduce these risks, HttpOnly cookies, strict CSP rules [17, 20], and careful handling of sensitive data are used.

Replay: This is the last stage where the attacker uses stolen data, such as a session token, to impersonate the victim or take unauthorized access. To prevent this, mechanisms such as session expiry, token binding, server-side validation, anomaly detection, and re-authentication for sensitive actions are used.Thisfinal stage shows why XSS is not only a scripting problem but also an authentication and session-management problem. A complete defence must therefore address all stages rather than relying on one control.

This technical background establishes the foundation for the remaining chapters.

The literature review examines how existing work addresses XSS detection and mitigation. The methodology then applies a controlled experiment to observe how an XSS payload progresses through the stages described here. The results and conclusion use this background to evaluate why layered defences are necessary in modern web applications.

CHAPTER 3

LITERATURE REVIEW

One of the more common and serious security flaws in web applications is the cross-site scripting (XSS) vulnerability, widely studied in prior XSS taxonomies and defence surveys, caused by the increased use of client-side scripting, dynamic content and user-controllable data. Despite considerable advances in application security, XSS continues because it takes advantage of the trust between the application and the browser in order to run malicious scripts in an application's trusted environment. XSS vulnerabilities arise when applications don't validate or encode untrusted user-provided inputs and reflect them to the browser, allowing scripts to be executed. These scripts are then executed with the privileges of the application, making it difficult for the browser to determine the difference between trusted and untrusted code. This enables an attacker to change the way the application behaves or steal data, or to perform any action on behalf of the user. The threat posed by XSS attacks is even greater when they are used to steal session tokens, a risk linked to practical XSS exploitation and cookie/session abuse [2, 5].

This is where the script is exploited to steal and transmit the session token (e.g. cookie) to a remote system. Through experiments, we've demonstrated that this data can be used to reuse these session tokens to steal and reopen a session without a password and essentially break the account. Web security controls help address these threats by introducing two types of protection: cookie-based and browser-based. The first type of mitigation is based on cookies, which seeks to safeguard the session token by using attributes such as HttpOnly, Secure and SameSite, which are discussed in cookie and browser-control studies [13, 14]. These restrict direct access to cookies and prevent exposure to other domains. But recent research has demonstrated that these do not stop XSS but mitigate it after it has happened, in particular by restricting information available to the attacker. Conversely, browser security approaches to prevent XSS try to prevent scripts from running on the browser by enforcing policies. CSP limits scripts from where they can be downloaded and prevents unsafe inline scripts. Previous studies worked on retrofit and implementation approaches of CSP in existing applications [17]. Trusted Types addresses the assignments of unsafe DOM and prevents the injection of dangerous content directly into the DOM [6]. Apart from this, techniques such as Trusted Types and Subresource Integrity (SRI) make browser security mechanisms stronger because they regulate DOM access and verify the authenticity of external resources. These techniques provide better prevention, but can be limited by the ease of configuration, legacy of existing code and often have not been adopted in practice.

There is a lot of work on detection, classification and various prevention approaches

for XSS, but comparatively less about the ”big picture”. The literature tends to focus on individual facets of XSS, and not the relationship between the evolution of XSS from injection to the outcome of session hijacking. So, there is a need for a structured analysis of XSS exploits. This approach, which breaks the attacks into different stages (such as injection, execution, data exfiltration, and session replay), allows measuring the effectiveness of various defence mechanisms on each individual stage. This not only helps comprehend the mind of the attacker but also identifies the stage of attack where security approaches can be introduced. This forms the foundation of the work in this paper that combines proof-of-concept with analysis to understand the exploitation and the defence of XSS in the current web domain.

Table 3.1. Comparative Analysis of Defence Mechanisms in Existing Literature

Study Focus	**Defence Type**	**Strength**	**Limitation**
Detection-based approaches [21]	Input filtering	Prevents injection at input points	Does not stop payload execution after bypass
Cookie-based protection [13]	HttpOnly and SameSite attributes	Protects session data during egress	Does not prevent XSS execution
Browser-based mechanisms [17]	CSP and Trusted Types	Restricts script execution in the browser	Requires careful configuration and adoption
Hybrid approaches [22]	Multi-layer defence	Covers multiple stages of the attack chain	Adds implementation and maintenance overhead

Table 3.2. Key Prior Studies Relevant to the Proposed Stage-Wise XSS Analysis

Key Reference	**Contribution to This Study**	**Remaining Limitation**
Gupta and Gupta [1]	Provides a classification of XSS attacks and core defence mechanisms, forming the baseline for understanding stored, reflected, and DOM-based XSS.	The discussion is broad and does not fully trace a payload through egress and replay.
Mahmoud et al. [8]	Compares XSS detection and defensive techniques, supporting the selection of filtering, validation, and mitigation controls for review.	The comparison gives limited attention to what happens after a payload successfully executes.

Key Reference	Contribution to This Study	Remaining Limitation
Grossman [2]	Explains practical XSS exploitation patterns and how scripts can abuse authenticated browser contexts.	Modern cookie attributes and browser policy controls require additional evaluation.
Lekies [11]	Gives detailed coverage of client-side XSS exploitation, detection, mitigation, and prevention.	The work is comprehensive, but this thesis narrows the focus to a controlled session-hijacking flow.
Kaur et al. [21]	Reviews machine-learning approaches for XSS detection and shows the value of automated early identification.	Detection models do not by themselves explain defence effectiveness after execution.
Sharma and Yadav [19]	Summarises detection and prevention gaps for scripting attacks, supporting the need for a gap-focused analysis.	The review is not organised around persistence, execution, egress, and replay stages.
Drakonakis et al. [12]	Demonstrates why cookies and session material are high-value targets in web authentication weaknesses.	Cookie auditing does not remove the XSS condition that enables script execution.
Khodayari and Pellegrino [13]	Analyses SameSite cookie behaviour and its effectiveness in limiting cross-site session abuse.	SameSite helps only in specific request contexts and does not block same-origin XSS execution.
Tyler and Nunes [14]	Examines browser-side controls for protecting cookies from malicious client-side access.	Browser cookie controls reduce exposure but cannot replace safe rendering and output encoding.
Fazzini et al. [17]	Shows how CSP can be retrofitted to reduce script execution risks in existing applications.	CSP is sensitive to configuration quality and can be weakened by compatibility requirements.
Wang et al. [6]	Provides evidence that Trusted Types can prevent DOM-based XSS in production framework settings.	Adoption requires changes to unsafe DOM patterns and may be difficult in legacy code.
Rodriguez-Galan et al. [20]	Supports the egress stage by showing how cookie-collector behaviour can reveal XSS-driven exfiltration.	The approach emphasises detection of leakage rather than complete prevention.
Pizzolante et al. [23]	Highlights the value of server-side forensic analysis for understanding and documenting XSS incidents.	Forensic visibility helps after compromise but must be paired with preventive controls.

Key Reference	Contribution to This Study	Remaining Limitation
Tang et al. [22]	Presents a proxy-based XSS protection framework, supporting the need for layered defensive architecture.	Proxy-based defences may introduce deployment and maintenance complexity.
Niakanlahiji and Jafarian [24]	Introduces moving-target defence as an adaptive way to reduce XSS exploit reliability.	Adaptive defences are useful but may be harder to implement than standard controls such as encoding, CSP, and cookie attributes.

3.1 Research Gap

Many research studies explain the XSS security mechanism in different ways, but do not clearly explain how the XSS payload progresses from storage to execution, data theft and session replay. Because of all this, it is difficult to understand at which stage which defence mechanism is effective. This work is to fill the gap by using a stage-wise model, which does a comparison of cookie-based and browser-based approaches in the XSS exploitation chain.

Main studies explain XSS from the perspective of coding weakness or browser weakness, but do not connect both properly. Stored XSS generally starts from application-level mistakes, like missing validation or unsafe output reading. It shows real impact inside the browser when the injected script executes with the permission of a trusted web application.

One more important gap is that there are very few practical ways in which cookie-based and browser-based comparisons have taken place. HttpOnly and SameSite cookie attributes mainly reduce the risk of data theft and session replay, whereas CSP and Trusted Types, like browser policies, restrict or prevent script execution. In existing literature, this distinction is not explained properly, especially for students, developers and researchers who need to decide which protection to use at which stage.

There are many challenges in practical deployment. In a real web application, sometimes due to issues in security policies, such as incomplete, compatibility, or overly permissive policies, they are disabled [25, 18]. In a legacy codebase, trusted and untrusted contents are mixed, due to which it is difficult to apply strict browser policies. Because of these issues, this thesis focuses on stage-wise evaluation so that it is easy to clearly understand the practical role, intervention point and limitations in the defence mechanism.

Table 3.3. Research Gaps Identified from Existing Literature

Research Area	**Observed Gap**	**How This Study Addresses the Gap**
XSS detection and filtering	Prior work often focuses on identifying malicious input, but gives less attention to post-injection exploitation stages.	The study analyses persistence, execution, egress, and replay as connected stages of one attack chain.
Cookie-based protection	Cookie attributes are commonly evaluated as session-protection controls rather than as part of a complete XSS defence model.	The study examines how HttpOnly, Secure, and SameSite affect egress and replay stages.
Browser-based controls	Browser policies such as CSP and Trusted Types are effective but often discussed without practical stage-wise comparison.	The study compares browser controls with cookie controls at each stage of exploitation.
Integrated defence strategy	Existing literature gives limited practical guidance on combining multiple mechanisms against multi-stage XSS attacks.	The study proposes a defence-in-depth view that maps controls to different points in the XSS life cycle.

CHAPTER 4

METHODOLOGY

This study used an implementation-based controlled experimental methodology [26] to examine how Cross-Site Scripting (XSS) exploits work and to evaluate the strengths and limitations of different XSS defence mechanisms. The paper is focused on the analysis of the evolution of a stored XSS attack over time [27], and how this vulnerability could ultimately be exploited to conduct a session hijack attack [2]. This work is not based on large datasets or statistical modelling; instead, an implemented prototype environment that mimics real-world attack conditions [27] is used in a practical application that is based on a web application. The goal is to see the entire attack life cycle and test the response of various security measures to adversarial activities throughout the process.

The implementation consists of a lightweight client-server web application with a simple backend for session handling, cookie management, and request processing, together with a browser-based frontend built using standard web technologies such as HTML, CSS, and JavaScript. The vulnerable baseline implementation simulates a real-world case in which user inputs are saved and rendered without sanitisation [28, 11], enabling some malicious and injected script to be executed. The baseline version of the implemented system is intentionally kept vulnerable so that the exploitation process can be observed clearly. This is to simulate the role of an attacker and a victim, and to create two instances of the browser to simulate a true-to-life interaction [29] between an attacker and a victim.

The XSS attack is viewed in terms of a structured four-phase process of Persistence, Execution, Egress and Replay [11, 20]. During the persistence stage, the bad input is kept in the application [1]. In the execution stage, the stored content gets executed when it is shown in the browser. The egress stage involved the removal of sensitive data (e.g., session cookies) via client-side script and delivery to an external endpoint [20]. Last but not least, during the replay phase, the information collected during the session is used to impersonate a victim and gain access. Each stage is monitored and recorded to gain insight into vulnerability's progression to successful session attacks.

The system implemented is then progressively enhanced by adding common defenses such as output encoding, HttpOnly cookie attributes and the Content Security Policy (CSP) [17, 6] to evaluate their effectiveness. The effectiveness of each mitigation is then measured again through a repeated attack [19] to quantify the effectiveness during different stages of a potential attack. Based on this comparative approach, it is possible to identify where each defence mechanism ends the attack chain. The implementation is tested in a modern web browser, such as Google Chrome or Mozilla Firefox, by using developer tools in order to keep track of network requests [23] and

browser cookies, as well as script execution. External request logging [23] is used to capture data exfiltrated through a simple HTTP listener or shared webhook service. Thus, the whole attack can be authenticated with evidence [26]. This is an implementation-based approach that offers repeatable steps and stages to analysing XSS attacks and evaluating defenses. It goes beyond theoretical vulnerabilities to attack behavior by conducting a vulnerability analysis and showing how different security controls affect each stage of an XSS attack.

4.1 Setup and Threat Model

The implemented setup is used to study XSS attacks and their progression toward session hijacking. The prototype is based on a client-server web application in which a client-side interface communicates with a simple web server that sets a session cookie. It implements a simple user-administration interface that stores submitted input and displays it in an administrator view. In the baseline version, the application intentionally omits sanitisation and output encoding to establish the vulnerable attack scenario. This allows for the injection of scripts into the application, thus simulating an insecure real application. The security model assumes an attacker who can inject arbitrary input into the application [8, 9], but was not able to control the inner workings of the server and/or the browser itself. The adversary's objective is to exploit XSS to inject scripts into the victim's browser, steal session data and perform arbitrary actions. The victim, who is often an administrator or privileged user, interacts with the injected code by touching inputs in the interface. We evaluate the attack in two parts: first in the unprotected version of the system and second in the protected system, where the system uses protection mechanisms such as HttpOnly cookies and Content Security Policy (CSP) [13, 17]. This allows us to compare the system-protected and unprotected behaviours.

4.2 Stage-wise XSS Model

To better understand the exploitation of XSS, the analysis of an attack is broken down into four ordered steps: persistence, execution, egress, and replay. This breakdown helps understand how an exploit progresses from the initial injection to full compromise of the user's session, and allows for assessment of preventative measures throughout the lifecycle of an attack [19, 21].

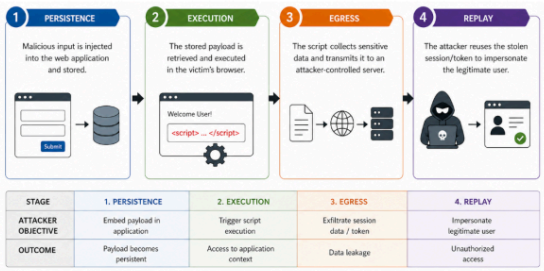

Figure 4.2. Stage-wise model of XSS exploitation

Persistence: The persistence stage is the first stage of the attack, involving the injection of malicious input and storage in the system. This usually happens when input from users is taken into the system without appropriate encoding and rendering in the interfaces.

These circumstances enable the malicious script to become embedded in the content of the application, especially in the case of stored input. This allows the malicious script to be transmitted to other affected users via the vulnerable interface, thus facilitating other attacks.

Execution: During the execution phase [11], the injected script is rendered and executed in the victim's browser. Because the script is displayed as a component of the application, it shares the same origin and security privileges as other scripts.

This allows the script to access client resources and interact with the application code like legitimate scripts. The runtime phase is key, as it enables the attacker to perform various actions like accessing data and modifying the application's behaviour.

Egress: The egress phase is responsible for transferring information from the victim's browser to a remote server under the attacker's control. In this stage, the code executes, gathers data (such as session IDs) and sends requests.

While browser security policies restrict access to responses [3] originating from other domains, they have no effect on the requests. As a result, the attacker can steal the encoded data without directly accessing the response, making this stage crucial to the attack's success.

Replay: During the replay phase, the attacker reuses stolen session data to impersonate the victim and gain unauthorized access.

Table 4.1. Stage-wise Model of XSS Exploitation

Stage	Description	Attacker Objective	Outcome
Persistence	Injection and storage of malicious input	Embed payload in application	Payload becomes persistent
Execution	Rendering and execution in browser	Trigger script execution	Access to application context
Egress	Transmission of sensitive data	Exfiltrate session/token	Data leakage
Replay	Reuse of stolen session	Impersonate user	Unauthorized access

4.3 Defence Mechanism Evaluation

To evaluate the implemented defence strategy, security mechanisms are added step by step to the prototype system. The assessment consists of both cookie-based security and browser-generated security and looks at the point at which each protection/defence stops the attack chain. First, the payload is executed against the vulnerable baseline implementation without protective controls. The same payload is then reused after enabling each security feature individually and in combination, allowing the effect of each implementation change to be compared.

The following protection attributes are also used in connection with cookies: HttpOnly, Secure and SameSite [13, 14]. Minimises direct session theft in the egress by restricting access to session cookies in the presence of JavaScript using HttpOnly. Secure ensures that cookies are only sent via encrypted HTTP/HTTPS connections, and SameSite also means that cookie transfer in cross-origin contexts are limited. Content Security Policy (CSP) is one of the mechanisms available in the browser that can be used to restrict inline script execution, and to control script sources that are trusted. Effectiveness of each of these mechanisms is discussed based on whether it prevents data exfiltration, or whether it blocks script execution, or whether it prevents successful session replay, or whether it prevents payload storage. The comparative evaluation assists in determining where their strengths and practical weaknesses lie when using each defence mechanism in the context of each state of the XSS attack model.

4.4 Data collection and analysis.

Analysis is performed while the implemented attack flow runs in the browser environment [29]. Browser developer tools are used to monitor JavaScript execution, inspecting cookies, analysing changes to the DOM and capturing network traffic generated during the attack execution. Outbound attack traffic is tracked using external request listeners and webhook endpoints to determine if information like session identifiers is able to be exfiltrated during the egress [20].

Observations are collected from the implemented prototype, and qualitative analysis compares the behaviour of the protected and unprotected versions at each attack phase. For the purposes of the evaluation, the focus is whether each defence mecha-

nism completely blocks a particular attack action, partially blocks it, or does not block it at all. This stage-wise comparison shows the impact of security measures on the transmission of vulnerabilities through the life cycle of an XSS attack. The results are also analysed to gain insights into practical implementation challenges, configuration restrictions and potential effectiveness of multi-layered defence strategies [22, 24] in the context of web application security in the modern era.

CHAPTER 5

RESULTS AND DISCUSSION

In order to replicate a real-world scenario of stored XSS to result in hijacking of the session [1], the evaluation of Cross-Site Scripting (XSS) was carried out using the implemented controlled web application prototype. The results illustrate that for the user to enter information that is stored and subsequently displayed in the application without a proper sanitisation mechanism, malicious scripts can be injected and executed in the user's browser context [11]. In the baseline implementation, where no security controls were enabled, there was a possibility for the payload injected in the system to persist in the system [1, 27], run when the page loaded, and interact with the Document Object Model (DOM) [6] when browsing the page in an authenticated session. The fact that this is confirmed proves that supplying malicious data can enable complete exploitation of the vulnerability, if there is one, in and of itself.

It was noted during the execution phase that the malicious script is executed under the privileges of the regular application. Without adequate cookie safeguards, this enables attackers to reach certain information found on the client's browser, such as cookies for sessions [12]. The experiment also demonstrated that session keys could be sent to a separate script [20] that can be controlled by an attacker and placed outside the project, confirming that exfiltrations are possible via JavaScript-based techniques. With the acquired session token, it was subsequently used again in another browser session to hijack the victim [5], which was deemed successful to achieve a full session hijacking without the use of authentication credentials. This makes sure it is not only a script injection problem, but also a direct attack on the user authentication system.

As each defensive mechanism was implemented and enabled, a clear change was observed in the attack behaviour. By delivering the output in the correct encoding [8], WebMAM was effectively able to neutralise the attack by not turning injected scripts into JavaScript code. Again, if the script is fired, the lack of JavaScript access to session cookies due to the utilisation of the HttpOnly attribute of a cookie [13, 14] did not allow for information to be collected. In addition, the Content Security Policy (CSP) [17] proved to be a very effective preventive measure as it provides a way to block the ability to execute inline script [17] on the web page as well as limit the number of external services being called that were not authorised by them, effectively cutting at various points of the attack chain of events according to the level of enforcement. The results show that none of these defence methods is able to defend the whole XSS lifecycle, but rather against only part of the lifecycle [19].

The comparative analysis points out that the action of the successful attack is mostly mitigated with cookie-based mechanisms, while with browser-based mechanisms, it is mostly prevented from being executed or exploited in terms of data-flow or

not. Cookie-based controls (such as HttpOnly, Secure, and SameSite) [13] were able to restrict the exposure of the cookies but did not prevent script execution. CSP and browser policies, on the other hand [17], offer a higher level of preventive assurances, but they need to be configured with care to prevent functionality problems. However, their effectiveness in some tests was seriously compromised through misconfiguration and/or overly permissive policies [25, 18], a common issue encountered in production deployments.

The findings support the notion that XSS attacks can pursue a predictable life cycle (persistence, execution, egress, and replay phases) [11] and that specific security mechanisms are oriented at different phases of this life cycle. The study shows that it's actually not possible to get rid of XSS threats using a single protection mechanism [22]. To mitigate the risks and advantages of XSS attacks in contemporary web applications, a multi-layered defense-in-depth strategy [22, 24] is the best possible strategy, which includes proper coding practices, cookie attributes, and browser policies. The result also indicates that the effectiveness of a control should not be judged only by whether the final attack succeeds or fails [19]. A mechanism may fail to stop the initial injection but may still reduce the impact by blocking cookie access or preventing data transfer. Similarly, a mechanism may stop script execution but may not solve the underlying storage of unsafe input. Therefore, the results are best interpreted as a chain of partial controls rather than as a single pass-or-fail security outcome.

The implemented evaluation further shows that runtime visibility during testing is important. Browser developer tools, cookie inspection, and request monitoring made it possible to identify the exact transition from one stage to another. This helped distinguish between payload persistence, script execution, outward communication, and session reuse [20]. Such observation is useful for security testing because it shows where evidence of compromise appears and where mitigation should be applied. The practical implication of the implementation is that developers should not rely only on vulnerability scanners [30], but should also validate how the application behaves during runtime under realistic authenticated user conditions.

5.1 Attack Analysis

An XSS attack is broken down in stages, from entry to exploitation, and they are captured using a stage-wise analysis. In the persistence phase, whenever data isn't validated, untrusted data can be stored and then presented in the application [1]. This is because it allows an environment where untrusted data can be injected into the application, allowing such an attack to occur. During the execution phase the injected script is interpreted as integral part of the application itself. It runs in the same origin as the web application [3, 7] and it can access protected resources and take action with the privileges of the web application. This step is very important as further interaction with the application environment is allowed. The egress stage shows how it can be possible to send very sensitive information from the victim's browser. Outbound requests can still be made even when the browser security settings are set to normal, and encoded information (like a session identifier) can be sent without needing to access the response that the outbound request is a part of [20]. This illustrates one of the vulnerabilities in

browser security, as blocking the transfer of data is more difficult than controlling access to data. During the replay phase, the captured session data is reused to play back the session. By injecting the session identifier into the new environment [5, 16], unauthenticated access can be obtained. This verifies that XSS can be used to directly result in a session hijacking attack if proper safeguards have not been implemented. In summary, the analysis supports the claim that XSS is multi-step and that different stages [8, 21] lead to a successful attack. The steps of the attack demonstrate the necessity to adopt several security measures rather than just one. The analysis of stages allows us to look at the development of XSS attacks in a systematic manner from injection to session hijacking. In the persistence stage, failure to validate input results in persisting the malicious input to be displayed by the application. It creates a condition of injecting untrusted input into the output of the application and has the potential to spread. The execution stage involves rendering the payload by the client's browser within the application. The script code originates from the same place, so we have access to the client-side resources and can run as an application. This is a key stage as it provides access to the environment. The egress stage demonstrates how data can be stolen from the victim's browser to a remote location. Despite the conventional security settings in the browser, requests are still possible, which allows encoded data (e.g. a session ID) to be exfiltrated without actually receiving the response. This highlights the challenge in browser security, which is the more complex nature of controlling data flow, rather than access. In the second stage, the session data is reused to replay a session. The stolen session ID can be used to get access without needing to authenticate. If not prevented, it indicates that XSS can lead to session hijacking. In sum, the study reveals that XSS is not a one-stage attack, but a multi-stage vulnerability, where each stage is a critical part in the attack. This flow through multiple stages shows that multiple levels of security are needed to combat the threat.

5.2 Defence Evaluation

The analysis of defence mechanisms across the attack stages highlights their different capabilities. Defence mechanisms anchored in cookies primarily impact the latter stages of the attack, namely, access to data and egress. Controls like HttpOnly that obstruct direct script-access to cookies [14] make the egress stage of data exfiltration more difficult. But they don't prevent scripts from being executed and hence, do not tackle the core of the vulnerability. Prevention becomes stronger with browser controls, preventing the attack in earlier stages. CSP is an important factor in preventing the execution of scripts [17] through its controls over valid sources and unsafe inline scripts. It can severely restrict the execution phase and thus prevent access to the data. While both types of defence mechanisms are effective, they have shortcomings. Most cookie-based mechanisms are not effective by themselves, as session identifiers get automatically attached. While more robust, browser-based mechanisms are often limited by configuration difficulties, compatibility problems and limited integration into current systems. Our research suggests that we need multiple mechanisms for effective mitigation of XSS attacks. A defensive approach is needed, which involves the use of multiple mechanisms to combat different stages of the attack. Preventing the attack

in the first place, and perhaps more importantly, limiting its impact can considerably diminish the risk of XSS attacks. From the evaluation, input validation and output encoding are most valuable before or during rendering because they reduce the possibility that hostile input will become executable code. HttpOnly cookies are valuable after execution because they reduce the ability of a script to directly read session identifiers, but they cannot stop the script from changing the page or performing actions as the user. SameSite and Secure attributes add further protection [13] in specific transport and cross-site conditions, but their benefit depends on the attack context. CSP provides stronger control over execution, although it becomes effective only when policies are strict and consistently maintained. Therefore, the strongest result is obtained when these mechanisms are combined rather than deployed independently [22, 24].

Table 5.1. Effectiveness of Defence Mechanisms Across XSS Attack Stages

Defence Mechanism	**Persistence**	**Execution**	**Egress**	**Replay**	**Limitation**
Input Validation	High	Limited	None	None	Bypass possible
Output Encoding	High	High	None	None	Context-dependent
HttpOnly Cookie	None	None	High	Moderate	No execution control [14]
SameSite Cookie	None	None	Moderate	Moderate	Limited scope
CSP	None	High	Moderate	None	Misconfiguration risk [25]
Trusted Types [6]	None	High	None	None	Limited adoption

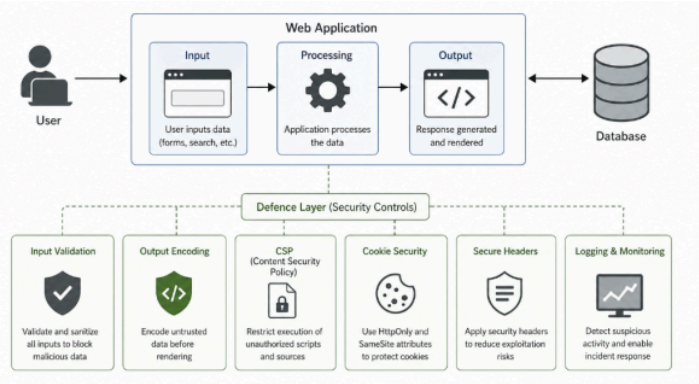

Figure 5.1. Layered defence architecture for mitigating XSS attacks

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

This research systematically analyses XSS by dividing the attack into different stages: persistence, execution, egress and replay [11, 20]. The study shows that XSS is not a simple one-step attack, but it is a multi-stage attack where each stage helps to complete malicious activity. Research also highlights that if there is no proper security protection, then XSS creates serious security issues like session hijacking [2, 5]. This happens because injected scripts execute inside the browser's security model and access sensitive information, and can send the same, due to which an attacker can impersonate the victim or gain access to the victim's browser. The evolution of defence strategies shows that cookie-based security reduces the impact of attacks by protecting session identifiers [13, 14]. Browser-based strategies provide more preventive protection because they control the execution of malicious scripts [17, 6]. But if these mechanisms are used individually, the impact will be limited because they will not be able to cover every aspect. Overall, this study shows how to effectively mitigate the XSS attack using a multi-layered defence approach. In this approach, different security mechanisms are used at different stages to reduce the impact of application vulnerabilities [22, 24]. This work can be extended to study XSS attacks in more realistic environments, such as in modern web frameworks and other large-scale distributed systems. The impact of more advanced browser protections and enhanced policy settings can also be studied to enhance prevention efforts. We can also explore automation of security policies and build protection directly into the software development process to minimize configuration errors. Similarly, research on new client-side attack vectors and exploitation trends may help to better understand future threats. Detailed studies of monitoring, anomaly detection, and adaptive security are useful in preventing and detecting attacks. By including performance and user experience in the evaluation, it helps to design a scalable security mechanism.

A major contribution of this work is to present XSS as a complete lifecycle rather than an input-validation failure. By dividing the attack into persistence, execution, egress and replay stages, the thesis shows at which stage different security controls work and which protection is effective on which stage and not effective on others [11, 20]. A stage-wise approach explains that neither cookie-based nor browser-based defence is a substitute for the other. Cookie control reduces the damage done by exposed information, whereas browser policies and secure rendering practices help to prevent the execution of malicious scripts [13, 14, 17, 6].

This study concludes that for XSS defence, careful configuration and testing are very important. CSP policies provide strong protection only when they are written specifically according to the application, and even after the change, the application

policies are regularly reviewed [18]. Just like this, cookie attributes provide session protection, but do not fix insecure rendering logic. Due to this, secure coding practices, output encoding, strict session handling and browser security should be used in a combined manner [22, 24]. This layered approach reduces both the probability of successful exploitation and the severity of impact if one control fails.

Future work may extend this research by testing the same stage-wise model on modern single-page applications, microservice-based systems, and applications that use third-party scripts or browser extensions. Further studies can also compare different CSP configurations, Trusted Types deployment strategies, automated XSS scanners, and runtime monitoring tools. Another useful direction is to evaluate usability and performance impacts of strict browser security policies, because overly strict policies are often weakened during deployment. A broader implementation setup with multiple browsers, frameworks, and authentication mechanisms would provide deeper insight into how XSS defences behave in real production-like environments.

Goyal

ORIGINALITY REPORT

6%	5%	1%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Delhi Technological University Student Paper	2%
2	dspace.dtu.ac.in:8080 Internet Source	2%
3	sinergiejournal.eu Internet Source	<1%
4	Maura A. Deek, Fadi P. Deek, Wei Yao. "From Code to Cloud - Developing Web Applications", Auerbach Publications, 2026 Publication	<1%
5	thesesjournal.com Internet Source	<1%
6	Submitted to MIT-ADT University Student Paper	<1%
7	psaspb.upm.edu.my Internet Source	<1%
8	Submitted to Florida Atlantic University Student Paper	<1%
	ogma.in	

9	Internet Source	<1%
10	softwarepatternslexicon.com Internet Source	<1%
11	cdn.chools.in Internet Source	<1%
12	cybersecuritytraining.tech Internet Source	<1%
13	ijetrm.com Internet Source	<1%
14	test.c-sharpcorner.com Internet Source	<1%
15	betanet.net Internet Source	<1%
16	core.ac.uk Internet Source	<1%
17	rifqimulyawan.com Internet Source	<1%
18	Arvind Dagur, Karan Singh, Pawan Singh Mehra, Dhirendra Kumar Shukla. "Artificial Intelligence, Blockchain, Computing and Security", CRC Press, 2023 Publication	<1%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		