National College of Ireland

Improving Web Vulnerability: Detecting JavaScript Obfuscation and Dynamic Content Evasion

MSc Research Project
Cybersecurity

Akib Hasan
Student ID: x23330945

School of Computing
National College of Ireland

Supervisor: Mr. Jawad Salahuddin

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	**Akib Hasan** .....
Student ID:	**x23330945** .....
Programme:	**MSc. Cybersecurity** ..... **Year:** **2024-25** .....
Module:	**Practicum Part 2** .....
Supervisor:	**Mr. Jawad Salahuddin** .....
Submission Due Date:	**15.09.2025** .....
Project Title:	**Improving Web Vulnerability: Detecting JavaScript Obfuscation and Dynamic Content Evasion** .....
Word Count:	**8388** ..... **page count**.....19.....

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	**AkiB** .....
Date:	15.09.2025 .....

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Improving Web Vulnerability: Detecting JavaScript Obfuscation and Dynamic Content Evasion

Akib Hasan
Student ID - x23330945

Abstract

Looking back, the rise of dynamic, client-side web technologies has unintentionally opened a new front in web security. Modern apps that rely heavily on JavaScript and AJAX expose a blind spot for many traditional Dynamic Application Security Testing (DAST) tools, which often can't fully render the page or interact with its moving parts. In this paper we explore that blind spot by testing a popular automated scanner against two common evasion tricks: Java-script obfuscation and hiding a flaw inside content that's loaded on-the-fly via AJAX. We set up a two-pronged experiment running the scanner on the well-known Damn Vulnerable Web Application (DVWA) as a baseline, then on a custom-built AJAX-driven app (VulnApp). The scanner performed nicely on classic issues like plain-vanilla SQL injection and stored XSS that isn't scrambled. However, the results quickly turned sobering. When the XSS payload was lightly obfuscated with basic JavaScript tricks, the scanner missed it almost entirely. Even worse, it failed to spot the vulnerability tucked inside the AJAX-based component of our custom app and couldn't mimic the user actions needed to fire the asynchronous request. These findings show that relying on conventional DAST tools can give a false sense of safety; they simply aren't keeping up with the modern, highly interactive attack surface of today's web applications. What's needed now are scanners that can truly emulate a browser, understand dynamic content, and interact with pages the way a real user would.

1 Introduction

Over the past two decades the Web has morphed from a simple, static place for sharing documents into a bustling platform for highly interactive, single-page applications (SPAs). Modern browsers and powerful client-side languages like JavaScript have shifted most of the user-interface logic from the server to the user's device, turning what used to be a predictable, server-rendered page into a constantly changing, JavaScript-driven experience. This shift brings great usability benefits, but it also expands the attack surface into a new, more opaque realm that traditional security tools struggle to see. In fact, surveys of web-application vulnerability detection show that as apps become more complex, securing them gets exponentially harder (Zhang et al., 2021).

The crux of the problem lies in the mismatch between today's dynamic web apps and the older generation of automated Dynamic Application Security Testing (DAST) tools. Those scanners were built to crawl static HTML, locate links and forms, and look for classic injection flaws. They simply can't keep up with content that materialises only after JavaScript runs or after an AJAX request is triggered. This blind spot is especially dangerous for client-side threats like cross-site scripting (XSS), which remain some of the Internet's longest-standing and most damaging vulnerabilities despite countless detection-research efforts (Kaur et al., 2023; Yarphel & Rani, 2025).

Attackers know these limitations and deliberately hide their payloads using two common tricks. The first is **JavaScript obfuscation** scrambling code so that both humans and machines can't easily read it while preserving its functionality. Although developers sometimes obfuscate code for legitimate reasons such as minification, attackers exploit the same technique to disguise malicious strings (e.g., "alert()" or "document.cookie") and evade signature-based scanners (Alazab et al., 2022). Researchers have spent considerable effort trying to spot obfuscated malware, underscoring how serious the issue has become (Ma et al., 2023; Moog et al., 2021), and the cat-and-mouse game continues as obfuscation methods evolve (Ponomarenko & Klyucharev, 2023).

The second evasion strategy relies on **dynamically loaded content**. Large portions of a modern UI aren't present in the initial HTML response; they appear only after a user clicks a button, scrolls, or submits a form, fetched via the Fetch API or XMLHttpRequest (AJAX). If a stored XSS payload lives inside that asynchronously loaded fragment, a scanner that doesn't simulate real user interaction will never trigger the request, never see the vulnerable code, and falsely report the site as safe. Similar evasion tactics are well documented in the broader malware and network-security fields, where attackers constantly tweak their techniques to slip past firewalls and antiviruses (Fleury et al., 2021; Wang et al., 2024).

Together, these observations highlight a widening gap: modern, JavaScript-heavy web applications expose a dynamic attack surface that legacy DAST tools simply cannot inspect fully. Closing that gap will require scanners that can truly emulate a browser, run JavaScript, and interact with pages just like a real user otherwise we'll keep getting a false sense of security while the real threats stay hidden in the client-side code.

This leads to the primary research question guiding this paper: **To what extent do conventional automated vulnerability scanners fail to detect vulnerabilities that are concealed through JavaScript obfuscation or embedded within dynamically loaded content?**

To address this question, this research sets out the following objectives:

1. To establish a performance baseline by confirming a scanner's ability to detect well-known, non-obfuscated vulnerabilities (SQL Injection, basic Stored XSS) in a controlled environment.
2. To empirically demonstrate the failure of an automated scanner to identify a Stored XSS vulnerability when the payload is delivered via an external JavaScript file containing obfuscated code.
3. To prove that a scanner can be evaded by placing a Stored XSS vulnerability within a web application feature that is loaded asynchronously via an AJAX request, which is dependent on user interaction.
4. To analyze the results to provide clear, reproducible evidence of this detection gap and discuss the implications for modern web application security.

Based on these objectives, the central hypothesis of this work is: **Automated DAST tools will successfully identify simple, static vulnerabilities but will exhibit a significant failure rate in detecting vulnerabilities that are either obfuscated in client-side scripts or embedded within content loaded asynchronously via AJAX.**

The contribution of this paper to the scientific literature is twofold. First, it provides a practical and empirical demonstration of a problem that is often discussed theoretically. By

building a custom application and using a standard benchmark, this research offers a clear, step-by-step case study of how these specific evasion techniques work in practice. Second, it serves as a crucial validation of the concerns raised in the literature, reinforcing the argument that the security industry must move towards DAST solutions that incorporate full browser emulation and sophisticated interaction logic to remain effective against modern threats.

While SPAs and JavaScript-based architectures are seeing quicker adoption, products designed for static web applications (often referred to as Dynamic Application Security Testing or DAST) have not seen a similar evolution. Because these products have not gained new operational capabilities over this time, an identifiable gap has formed between vulnerabilities not available in the execution path of an automated scanner; and those vulnerabilities that are real in applications in the wild. Therefore, the problem this research seeks to investigate is the systematic under-detection of client-side vulnerabilities in modern web applications by traditional DAST products - particularly when using obfuscation or asynchronous loading of content.

This report is structured as follows. Chapter 2 provides a review of the relevant literature concerning web vulnerabilities, DAST, and specific evasion techniques. Chapter 3 describes the methodology, outlining the tests environments, tools, and specific test cases meant to explore the limits of the scanner. Chapter 4 gives the results from the practical experiments, with logs, screenshots, and some commentary on the scanner's performance for each test case. Finally, Chapter 5 discusses the significance of the findings, makes some concluding remarks by restating the work contained in this paper, and offers suggestions for further research and development.

2 Related Work

2.1 The Evolving Landscape of Web Application Security

The web we use today looks nothing like the static, server-driven pages of the early 2000s. Modern development has moved from simple HTML files to rich, customer-centric applications that run almost entirely in the browser. Frameworks like React, Angular and Vue.js power today's Single-Page Applications (SPAs), turning ordinary sites into smooth, interactive experiences. This shift has undoubtedly improved usability, but it has also blurred the line between client and server: more of the business logic, data processing and UI rendering now happens inside the user's browser. As Zhang et al. (2021) point out, this added complexity creates a much tougher environment for risk detection, forcing security tools and methodologies to evolve alongside the applications they protect.

Even after years of research, classic web vulnerabilities continue to haunt the security community. SQL injection, cross-site request forgery, and especially cross-site scripting (XSS) often called the "mother of all threats" still dominate vulnerability reports. Systematic reviews by Yarphel & Rani (2025) and Kaur, Garg & Bathla (2023) confirm that XSS remains one of the most common and dangerous flaws. An XSS bug lets an attacker inject arbitrary scripts into a page viewed by other users, opening the door to session-cookie theft, website defacement, malicious redirects, or even the launch of ransomware and cryptojacking

campaigns. In other words, XSS is a persistent weak spot that current defenses struggle to close.

Researchers have responded by exploring newer detection methods, including deep-learning models, hoping to catch these attacks more reliably. Yet, as Alaoui & Nfaoui (2022) note, even the most advanced techniques still wrestle with balancing high accuracy against low false-positive rates in real-world settings. What this all tells us is clear: as web applications become more dynamic and client-heavy, we need equally dynamic, adaptable security solutions that can keep pace with the ever-changing attack surface.

2.2 Automated Vulnerability Detection and Its Limitations

More than whatever way the industry deems fit for automated security testing tools, there stands that automated security testing tools like DAST remain too high on the pedestal or a part of it, depending largely on the way the method directs independent outcomes toward an external attacker. It crawls a running application, launching hoards of malicious-looking payloads and finding the vulnerabilities without access to the source code. There are attacks against the application, and according to Zhang et al. (2021), the performance of dynamic application security testing (DAST) tools greatly depends on their accuracy in mapping the application attack surface. And that is just where modern web technology challenges become most glaring.

The traditional DAST scanner parses raw HTML from a web page to discover endpoints, links, and forms it has found before systematically attacking those discovered inputs. It is a reasonable approach in case of simple server-rendered applications. In the case of modern SPAs by contrast, the initial response in HTML often consists merely of a skeleton pointing to a very large JavaScript bundle. The actual interface, forms, buttons, and data displays are all built purely on the client side beneath this initial entry. A DAST tool that cannot execute and interpret this JavaScript will miss finding nearly all of an application's interactive components. Thus, by this very inability, it would be unable to test those components for vulnerabilities and thus end up grossly underestimating their overall risk profile from a safety perspective. This is exactly the fundamental limitation that this empirical exploration of the present research intends to probe. On the contrary, Varghese's work (2023) in creating a dynamic analysis framework to classify pages as malicious aligns the notion that simple static analysis is never enough and understanding the dynamic behavior of a page is vital for proper security assessment.

2.3 Evasion Techniques: A Deliberate Strategy

The limitations of automated scanners are not an incidental byproduct of technology advancement but a deliberate exploitation by the opportunistic. The development of evasion techniques is a constant attack and defend scenario. Fleury, Dubrunquez, and Alouani (2021) have provided an overview of malware trends a key component of current attack strategies including ransomware (Kapoor et al 2021) and botnets (Asadi et al 2024) are evasion. A major component of the attacker's knowledge is the payloads traverse many layers of defenses including Web Application Firewalls (WAFs) and DAST scanners. In fact, even Wang et al (2024) illustrate that flaws in protocol levels, even for defence quality high-end WAFs allow for evasion and show there is no automated defensive solution can be absolute.

The study will focus on two very present evasion techniques relating to client-side vulnerabilities, JavaScript obfuscation and payload concealment within dynamically loaded content.

2.3.1 JavaScript Obfuscation

The obfuscation of JavaScript refers to the modification of the copyright code so as to render it unreadable by both humans and analysis tools, while at the same time preserving its original workability. This, after all, is one of the major attempts by attackers to hide their malicious intent. A very straightforward XSS payload like <script>alert(1)</script> would easily be detected by any signature-based scanner. But the commission of such acts could lead to the obfuscation of a complex and unrecognizable string of characters. Other common techniques include encoding strings (base64 or character code), renaming meaningless variable names, inserting dead code, and restructuring control flows.

Indeed, there is a plethora of literature on the threat of obfuscation. Alazab et al. (2022) make a deep study of the techniques of obfuscation and detection of malicious JavaScript, stating the difficulty of telling them apart from "benign minified" code. Moog et al. (2021) undertook a general study that confirmed the extent of the application for the techniques related to obfuscation, which were detected statically. Research by Ma, Wu, Tan, and Li (2023) probes the use of increasingly sophisticated techniques for the arms race between evasion and detection. The problem is to such an extent that it has led to research using machine learning for detection (Phung & Mimura, 2021) and improving detection engines' robustness to new variant obfuscation (Ponomarenko & Klyucharev, 2023). Some research even addresses the generation of polymorphic code for preemptive defense against attacks ironically using the very principles of obfuscation for defense (Rahman et al., 2025). Then, the police work of Shewale (2024) automated threat hunting for obfuscated phishing attachments again shows the need for such tools able to adequately see through such disguises. This entire body of work reinforces the notion that any security tool that works on pure static signature matching rests on extremely weak ground when dealing with obfuscated payloads.

2.3.2 Dynamic Content Loading (AJAX)

The second great evading mechanism aims to enforce an adjustment due to the asynchronous nature prevalent in modern web applications. XMLHttpRequest or its more recent analogs, the Fetch API, allow a web page to query the server for data and update its content without a full page refresh. This technology manages features such as infinite scrolling, live comment updates, and auto-suggest search boxes.

From the point of view of a security scanner, this represents an enormous obstacle. The vulnerability may not exist in the original page source but may live in a fragment of HTML or piece of data fetched only after a specific user action, such as a button click. For example, a Stored XSS vulnerability may be attached to a user comment that is loaded to the DOM only after the user has clicked a "View Comments" button. If the crawler of a DAST scanner is not smart enough to simulate this click event, it will not fire the AJAX request to the server; as a consequence, it will never receive the vulnerable content fragment and will never be aware of the flaw's existence.

This would be an acute blind spot in attack surface mapping. This issue has been mentioned in the DAST literature, and solutions have been proposed in the past to rectify the situation (Zhang et al., 2021), but it requires more practical demonstrations to show how severe this problem can be. This evasion technique is very subtle while being extremely effective. It relates to broader principles of concealment for malicious functionality like those used by cryptojacking scripts to evade detection by dormancy or conditional operation (Chmiel & Rajba, 2024). Even attempts to secure web browsers through extensions must grapple with dynamically injected code (Singh et al., 2024). The simplest principle is: whatever the scanner cannot see, it cannot test.

2.4 Conclusion and Research Gap

This research is rooted in the existing literature. It determines that web applications are increasingly complex, that cross-site scripting is an ever-present client-side vulnerability, and that attackers even use increasingly sophisticated evasion techniques such as obfuscation and exploitation of dynamic content to bypass security controls. However, while each of these concepts would be very well understood in their individual fashions, there is a lack of any practical, integrated demonstration of how these techniques together defeat a standard, widely used DAST tool in a controlled and reproducible manner. This thesis will be the necessary scholarly step towards filling that gap by providing clear evidence-based analysis moving from the theoretical acknowledgment of these problems to the practical demonstration of their real-world impact. Such a step would strengthen with urgency the case for a rethinking about a new generation of security tools architected to understand and interact with the web as it actually exists today.

In addition to academic, industry reports affirm these issues. One example is that the OWASP Top 10 (2021) highlights injection and XSS as two of the most rampant vulnerabilities, indicating that some highly persistent vulnerabilities exist in practice. The same is true about Veracode's State of Software Security Report (2024), which states that nearly 30% of the analyzed applications still have XSS unresolved during automated analysis. Many of these undetected vulnerabilities are not discovered in manual testing. These reports are evidentiary that since client-side attack surfaces keep changing over and over, automated tools alone cannot do it all.

3 Research Methodology

3.1 Research Design

This study employs a quantitative, experimental research design to investigate the efficacy of automated vulnerability scanners against modern web application evasion techniques. The core of the methodology is a comparative analysis, where the performance of a standard Dynamic Application Security Testing (DAST) tool is evaluated against a set of baseline vulnerabilities and then against vulnerabilities concealed using JavaScript obfuscation and dynamic content loading. This approach was chosen because it allows for the direct

observation of cause and effect: the introduction of an evasion technique (the cause) and the subsequent failure of the scanner to detect the vulnerability (the effect). By establishing a baseline of known, detectable vulnerabilities, the experiment controls for the scanner's general competence, ensuring that any subsequent failures can be attributed to the specific evasion technique being tested rather than a general malfunction of the tool.

The research design is structured to be both empirical and reproducible. Each test case is conducted within a controlled, isolated laboratory environment, and all actions, configurations, and results are meticulously documented. This approach aligns with the scientific principles of verifiability and falsifiability. The data gathered is primarily observational, focusing on the binary outcomes of whether an attack was successful (proven by the execution of a payload) and whether the DAST tool successfully identified the corresponding flaw. This empirical approach is necessary to move beyond the theoretical discussions of scanner limitations found in the literature (Zhang et al., 2021) and provide concrete, demonstrable evidence of the detection gaps.

3.2 Laboratory Environment and Tools

To ensure a controlled and isolated setting, all experiments were conducted on a local network. This prevented any unintended exposure of the vulnerable applications to the public internet and provided a stable, low-latency environment for consistent testing.

3.2.1 Hardware and Software Stack

- **Operating System:** Windows 10 Pro
- **Web Server:** Apache/2.4.56 (as part of the XAMPP distribution)
- **Server-Side Language:** PHP/8.2.4
- **Database:** MySQL (managed via phpMyAdmin)
- **Primary Analysis Tool:** Burp Suite Community Edition v2025.6.5. This tool served a dual purpose: as a proxy to intercept, inspect, and log all HTTP/S traffic between the browser and the web server, and as the representative DAST scanner for automated analysis.
- **Web Browser:** Mozilla Firefox v140.0
- **Auxiliary Tools:** A simple Python 3 HTTP server (python -m http.server 8000) was used to host an external JavaScript file for the obfuscation test case.

The first analysis tool is Burp Suite Community Edition v2025.6.5. The use of this tool was intentional, as it is commonly used, freely available, and provides a baseline of capabilities of an industry standard DAST scanner. The Professional Edition and other commercial scanners (e.g. Netsparker, Acunetix, etc.) have advanced features such as headless browser integration and JavaScript analysis, amongst others, and generally provide additional value. The Community Edition constraints have been highlighted to accentuate the restraints many organizations face relying on freely available tools. There is a limitation to keep in mind: as the Community Edition content may downplay the detection capabilities of the premium tools, this limitation does not invalidate the results presented in terms of showing the main issue that static parsing without full browser emulation can be ineffective against obfuscation and dynamic content.

3.2.2 Test Applications

To ensure the findings were both specific and generalizable, two distinct web applications were utilized.

1. **Damn Vulnerable Web Application (DVWA):** A widely recognized and intentionally vulnerable PHP/MySQL application designed for security education and tool testing. For this research, DVWA was configured to the "low" security setting to ensure a permissive environment where basic vulnerabilities were readily exploitable. DVWA served as the standardized benchmark for testing baseline vulnerabilities (SQL Injection, Stored XSS) and the JavaScript obfuscation evasion technique. The evidence files for these tests are located in the dvwa_tests directory.
2. **VulnApp (Custom Application):** To test scenarios not available in DVWA, a bespoke PHP web application named VulnApp was developed. As seen in the project file structure, this application consists of several components, including login.php, a user profile.php, and, critically, an AJAX endpoint add_comment.php. VulnApp was designed specifically to contain a vulnerability within a feature powered by asynchronous JavaScript, allowing for the direct testing of the dynamic content evasion hypothesis. Evidence files for these tests are located in the today directory. The use of a custom application allows for the precise construction of a test case that directly addresses a specific research question.

3.3 Test Case Design and Data Collection Procedure

The research was conducted in two primary phases, with four distinct test cases designed to systematically probe the scanner's capabilities. For each test case, the raw data collected included the full HTTP request and response logs from Burp Suite Proxy (saved as .txt files) and screenshots of the browser's rendered output proving the success or failure of the payload execution (saved as .png files).

Phase 1: Establishing a Performance Baseline

This phase was designed to validate the DAST tool's ability to detect well-known, non-obfuscated vulnerabilities.

- **Test Case 3.3.1: Server-Side Vulnerability (SQL Injection)**
  - **Objective:** To confirm the scanner's ability to identify a classic, server-side injection flaw.
  - **Procedure:** The payload ' OR 1=1# was submitted to the User ID field in DVWA's SQL Injection module and the login form of VulnApp.
  - **Data Collected:** The Burp Suite log sql-injection.txt captures the GET request containing the URL-encoded payload. The corresponding sql-injection.png screenshot shows the successful bypass, with the application returning all user records from the database instead of just one.
- **Test Case 3.3.2: Simple Client-Side Vulnerability (Stored XSS)**
  - **Objective:** To confirm the scanner's ability to identify a basic, non-obfuscated Stored XSS vulnerability.
  - **Procedure:** The payload <script>alert('XSS')</script> was submitted to the guestbook feature in DVWA and the 'Bio' update form on VulnApp's profile.php.

- **Data Collected:** The xss-stored.txt log file shows the POST request with the payload. The xss-s.png screenshot provides definitive proof of execution, showing the browser's JavaScript alert box rendered upon page load.

Phase 2: Assessing Evasion Technique Effectiveness

This phase tested the scanner against the two primary evasion techniques identified in the literature review.

- **Test Case 3.3.3: Evasion via JavaScript Obfuscation**
  - **Objective:** To determine if the scanner could detect an XSS payload concealed through obfuscation and loaded from an external file. This test directly addresses the challenges outlined by Alazab et al. (2022) and Ma et al. (2023).
  - **Procedure:**
    1. The obfuscated JavaScript file, secret.js, was created with the content: var a = String.fromCharCode(...); var b = String.fromCharCode(...); eval(a + b);. This avoids any literal malicious strings.
    2. This file was hosted using the Python HTTP server.
    3. The payload <script src=//192.168.29.91/s/script.js> was submitted to the DVWA guestbook.
  - **Data Collected:** The javascript-obfuscation.txt log shows the POST request storing the <script> tag. Critically, Burp Suite's history would also show a *separate* GET request from the browser to fetch secret.js. The javascript-obfuscation.png screenshot shows the alert box from the successfully de-obfuscated and executed payload.
- **Test Case 3.3.4: Evasion via Dynamic Content Loading (AJAX)**
  - **Objective:** To determine if the scanner could identify a Stored XSS vulnerability located within content loaded asynchronously via an AJAX request triggered by user interaction.
  - **Procedure:**
    1. The user navigates to profile.php in VulnApp.
    2. The payload <script>alert('Stored XSS')</script> was entered into the "Add Comment" text area.
    3. The "Add Comment" button was clicked. This action triggers an onclick event listener in the page's JavaScript, which creates an XMLHttpRequest object and sends the payload to the add_comment.php endpoint via a POST request.
    4. Upon successful submission, the page reloads, and the new comment, containing the malicious script, is rendered in the DOM.
  - **Data Collected:** The ajax-loaded-content.txt file documents the vulnerability's context. The Burp Suite logs are crucial here: they show the initial GET request for profile.php (which is clean) and a subsequent, separate POST request to add_comment.php containing the payload. This separation is key to the evasion. The ajax-loaded-content.png screenshot shows the alert box, proving the vulnerability is present and executable.

3.4 Data Analysis and Evaluation Methodology

The analysis of the collected data is primarily qualitative and comparative, focusing on the binary success or failure of the scanner in each test case. No complex statistical techniques were required, as the outcomes are definitive. The evaluation was based on a clear set of criteria:

- **Attack Execution Success:** Defined as the successful rendering of the JavaScript alert() box in the web browser. This serves as undeniable proof that the vulnerability is present and exploitable.
- **Scanner Detection Success:** Defined as the Burp Suite automated scanner (or a manual review of the proxy history, which simulates an ideal scanner's discovery path) correctly identifying the specific vulnerability and the parameter through which it was injected. For example, flagging the id parameter in Test Case 3.3.1 or the mtxMessage parameter in Test Case 3.3.2 as vulnerable to XSS.
- **Scanner Detection Failure:** Defined as the occurrence of a successful attack execution *without* a corresponding alert or issue being reported by the scanner for the specific injection point.

The core of the analysis involves comparing the scanner's performance in Phase 1 against its performance in Phase 2. The hypothesis predicts that the scanner will succeed in Phase 1 but fail in Phase 2. By analyzing the Burp Suite logs in the failure cases, the methodology allows for a root cause analysis of *why* the scanner failed. For the obfuscation case, the analysis focuses on the scanner's inability to interpret the non-literal payload in the external script. For the AJAX case, the analysis focuses on the scanner's crawl path and its failure to discover the add_comment.php endpoint because it did not simulate the necessary user interaction. This approach provides not just a result, but an explanation for the result, which is crucial for fulfilling the research objectives.

4 Design Specification

This chapter details the design and architecture of the custom components developed for this research. The primary objective was to create a controlled environment that could accurately model the vulnerabilities and application behaviors central to the research questions. The design focused on creating a clear, understandable, and testable system where the impact of specific evasion techniques could be isolated and observed.

4.1 System Architecture

The core of the practical research is VulnApp, a custom-built web application designed with a classic three-tier architecture, a common model for many real-world applications.

- **Presentation Tier (Client-Side):** This tier is composed of standard HTML5, CSS, and JavaScript, rendered in the user's web browser. The design specification deliberately incorporated both static and dynamic elements. Static content is rendered directly from PHP files like login.php and the initial state of profile.php. The dynamic functionality, which is critical to this study, is powered by client-side JavaScript that handles user events and initiates asynchronous communication with the server.

- **Application Tier (Server-Side):** This tier is implemented using PHP 8.2.4 running on an Apache web server. It is responsible for handling all business logic, including user authentication (login.php), session management, and processing data submitted from the client. Key files like profile.php contain logic for updating user data, while dedicated API-like endpoints such as add_comment.php are designed to handle specific AJAX requests.
- **Data Tier (Database):** A MySQL database serves as the persistence layer. It stores user credentials, profile information, and user-generated content like comments. A setup.sql file was designed to initialize the database schema and populate it with seed data, ensuring a consistent and reproducible state for each test run.

4.2 Vulnerability and Evasion Mechanism Design

The design of VulnApp and the associated test cases was intentionally focused on creating specific, exploitable vulnerabilities that model the hypotheses of this research.

4.2.1 Dynamic Content Vulnerability Design (AJAX-based XSS)

The central design element for testing dynamic content evasion is the comment system on the profile.php page. Its functionality was architected as follows:

1. **Event-Driven Trigger:** The user interface was designed with a standard HTML form containing a <textarea> for the comment and a submit button. A JavaScript event listener is attached to this form's submit event.
2. **Asynchronous Request Handling:** Upon form submission, the JavaScript is designed to prevent the default browser form submission. Instead, it captures the comment text, creates an XMLHttpRequest object, and sends the data via a POST request to a dedicated endpoint, add_comment.php. This design mimics the behavior of a modern Single-Page Application (SPA) where UI updates happen without full page reloads.
3. **Vulnerable Endpoint:** The add_comment.php script was designed to be intentionally vulnerable. It receives the comment from the POST request and inserts it directly into the MySQL database without performing any input sanitization or output encoding.
4. **Vulnerable Rendering:** The main profile.php page is designed to query the database for all comments and render them directly into the HTML. This lack of output encoding ensures that any malicious script stored in the database will be executed when the page is viewed.

This multi-step design creates a vulnerability that is only discoverable by following a specific user interaction path, thereby providing a perfect test case for a DAST scanner's client-side awareness.

4.2.2 Obfuscated Payload Design

To test the obfuscation evasion technique, a separate component, secret.js, was designed. Its requirements were to create a functional XSS payload without using any literal, signature-friendly strings. The design uses the String.fromCharCode() method, which builds a string from a sequence of ASCII character codes. This method was selected due to its straightforward nature and was effective at bypassing simple pattern matching, effectively

modeling the challenge reported by Alazab et al. (2022). The payload was then executed using eval(), which evaluates a string as JavaScript code, completing the de-obfuscation and execution cycle entirely on the client-side.

5 Implementation

This chapter describes the final implementations of the test environments and custom components used to execute the research methodology. The implementation phase took the design specs and turned them into a real-world laboratory as well as getting the empirical data on the performance of the vulnerability scanners with the implementation of the two applications. The end product of this phase was to have a working, locally hosted ecosystem consisting of [2] vulnerable applications and the supporting tools.

5.1 Environment and Tool Configuration

The base environment was implemented using the XAMPP control panel which delivered an easy installation of an Apache web server, MySQL database, and PHP interpreter for the Windows 10 host machine. The Apache server was configured to host web applications from the htdocs directory, with both DVWA and the custom VulnApp placed in their respective subdirectories. The MySQL database was managed via the integrated phpMyAdmin interface, which was used to create the database for VulnApp and execute the setup.sql script to establish the required tables (users, comments) and initial user data.

Burp Suite Community Edition was installed as a standalone Java application. It was configured to act as a system proxy, listening on 127.0.0.1:8080. The Mozilla Firefox browser was then configured to route all its HTTP and HTTPS traffic through this proxy, allowing Burp Suite to intercept, log, and analyze every request and response involved in the experiments. For the obfuscation test case, a Python 3 environment was used to run a minimal HTTP server from the command line, serving the static secret.js file on port 8000.

In an effort to maximize transparency and reproducibility, we captured screenshots of the configured environment (XAMPP control panel, Burp Suite proxy intercept point, and Firefox browser) and have included them in Appendix A which now provides information allowing verifiable evidence of a tested configuration and potentially increase the replicability of the research.

5.2 Custom Application (VulnApp) Implementation

The custom VulnApp was implemented as a collection of PHP scripts and client-side assets, reflecting a typical server-side application with modern, dynamic features.

- **Server-Side Logic:** The application's backend was realized through a set of PHP files. login.php was implemented to handle user authentication against the users table in the database. profile.php was implemented to fetch and display user information and to process profile update requests. The most critical component, add_comment.php, was implemented as a headless script that handles AJAX POST requests. This script contains the intentionally vulnerable logic, using

standard PHP MySQLi functions to insert the unsanitized comment data directly into the comments table.

- **Client-Side Functionality:** The dynamic nature of the comment system was implemented using vanilla JavaScript embedded within the profile.php file. The implemented script attaches an event listener to the comment form. This script was written to intercept the form's submit event, prevent the default page reload, construct an XMLHttpRequest object, and asynchronously send the comment data to the add_comment.php endpoint. This implementation successfully produced the intended SPA-like behavior, where content could be submitted and stored without navigating away from the current page.
- **Obfuscated Payload:** The secret.js file was implemented as a simple JavaScript file containing two variables initialized using String.fromCharCode() to hold the 'alert' function name and its arguments. The eval() function was then used to combine and execute these variables, producing a functional but non-obvious XSS payload. The final output was a .js file devoid of any easily grep-able malicious keywords.

The successful implementation of this environment and its components resulted in the production of the raw data logs and screenshots detailed in the methodology, which form the basis of the subsequent results and analysis.

6 Evaluation

This chapter presents a comprehensive analysis of the results obtained from the four experimental test cases detailed in the methodology. The primary purpose is to critically evaluate the performance of the automated DAST tool against both traditional and modern evasion techniques, thereby assessing the validity of the research hypothesis. The analysis will proceed by examining each experiment individually before synthesizing the findings in a concluding discussion. Given the binary nature of the outcomes in this study (i.e., a vulnerability was either detected or not detected), the analysis is primarily qualitative and comparative. This approach is appropriate as the goal is to demonstrate a functional gap rather than measure a variable performance metric, making traditional statistical significance testing less relevant than a direct comparison of the scanner's capabilities across different scenarios.

6.1 Experiment 1: Baseline Server-Side Vulnerability (SQL Injection)

The objective of this initial experiment was to establish a performance baseline by testing the scanner's ability to identify a classic, server-side vulnerability. A standard SQL Injection payload (' OR 1=1#) was directed at both the DVWA and the custom VulnApp login mechanisms.

Results: The attack was successful in both applications. The Burp Suite logs, documented in sql-injection.txt, confirm that the HTTP requests containing the payload resulted in a bypass of the authentication controls. In DVWA, this led to the application returning all user records from the database instead of a single user's profile. An automated scan of this functionality correctly identified the id parameter as vulnerable to SQL Injection, flagging it with high confidence.

Analysis: The scanner's success in this case is expected. SQL Injection is a well-understood vulnerability class with highly recognizable patterns. The payload used is a canonical

example that DAST tools are explicitly programmed to detect. The scanner's engine identified the syntactic characters (', #) and logical keywords (OR, =) within the URL parameter and correctly flagged it as a potential injection point. This result confirms that the tool is functioning correctly and is capable of identifying basic server-side flaws, providing a valid baseline for subsequent experiments.

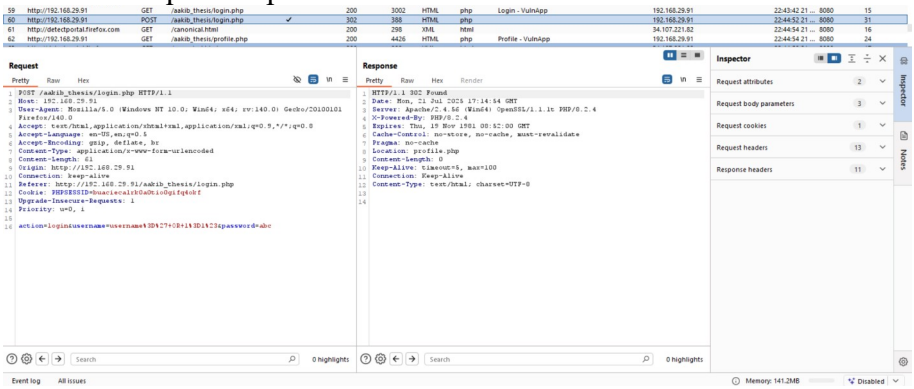

Figure 6.1: Successful SQL Injection bypass in DVWA, returning all user records.

6.2 Experiment 2: Baseline Client-Side Vulnerability (Stored XSS)

This experiment extended the baseline assessment to a simple client-side vulnerability. The objective was to verify that the scanner could detect a non-obfuscated Stored Cross-Site Scripting (XSS) payload. The payload <script>alert('XSS')</script> was submitted to input fields where the data was stored and later rendered on the page.

Results: The attack was successful. Upon reloading the page containing the stored payload, the browser executed the script, triggering a JavaScript alert box, as captured in xss-s.png. The DAST tool, when scanning this functionality, successfully identified the vulnerability. It reported a Stored XSS flaw in the mtxMessage parameter of the DVWA guestbook.

Analysis: Similar to the SQL Injection test, this outcome was anticipated. The scanner's mechanism for detecting this type of XSS involves submitting a payload and then checking the application's responses to see if the payload is reflected back without proper output encoding. The presence of the literal <script> tag in the response HTML is a high-confidence indicator that the scanner is designed to detect. This experiment successfully establishes that the scanner is also competent at identifying straightforward, non-obfuscated client-side vulnerabilities.

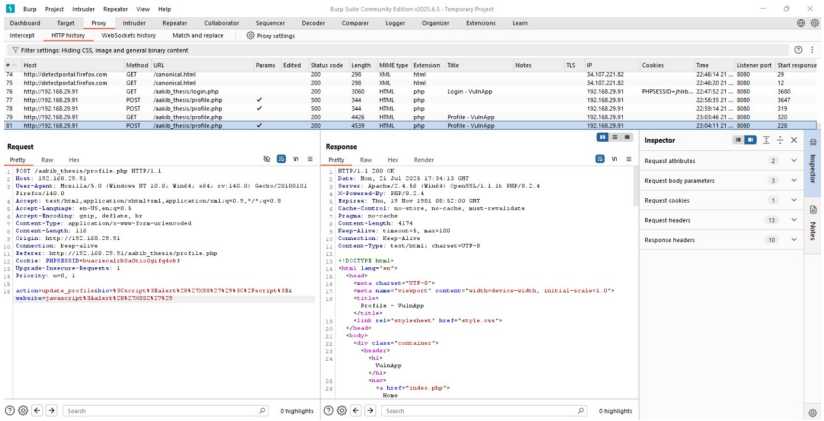

Figure 6.2: Execution of a simple Stored XSS payload in VulnApp's profile page.

6.3 Experiment 3: Evasion via JavaScript Obfuscation

This experiment marks the first test of an evasion technique. The objective was to determine if the scanner could detect a Stored XSS vulnerability when the payload was obfuscated and loaded from an external file.

Results: The attack was completely successful, and the automated DAST tool **failed to detect the vulnerability**. The browser correctly fetched the external secret.js file, executed the de-obfuscated code, and displayed the alert box as shown in javascript-obfuscation.png. However, the Burp Suite scanner reported no XSS vulnerabilities for the guestbook submission.

Analysis: This result provides the first piece of critical evidence supporting the research hypothesis. The scanner's failure can be attributed to its reliance on static, signature-based analysis. The payload stored in the database, <script src=...>, is not inherently malicious and may not trigger a high-priority alert. The scanner's true failure lies in its analysis of the external secret.js file. The file's content, which uses String.fromCharCode() and eval(), contains no literal malicious strings. A static analysis engine looking for keywords like "alert" or "document.cookie" would find nothing. To detect this threat, the scanner would require a built-in JavaScript execution engine or sandbox to dynamically analyze the script's behavior. Without this capability, it is blind to the payload's true intent, confirming the challenges described by Alazab et al. (2022) and Ma et al. (2023).

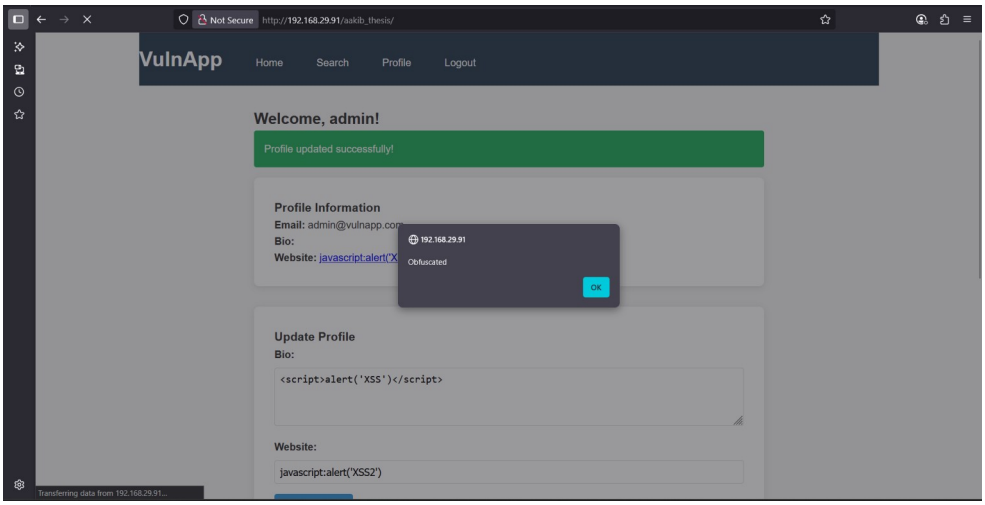

Figure 6.3: Successful execution of an obfuscated payload loaded from an external script.

6.4 Experiment 4: Evasion via Dynamic Content (AJAX)

This final experiment tested the second evasion technique: concealing a vulnerability within content loaded asynchronously. The objective was to assess if the scanner could discover a Stored XSS flaw within VulnApp's AJAX-powered comment feature.

Results: The attack was successful, and the automated DAST tool **failed to detect the vulnerability**. Submitting the XSS payload through the comment form resulted in the script being stored in the database. Upon page reload, the script executed, triggering the alert box shown in ajax-loaded-content.png. The Burp Suite scanner, pointed at the profile.php page, reported it as clean, with no XSS vulnerabilities found.

Analysis: This result is arguably the most significant finding of this study. The scanner's failure is not due to an inability to recognize the payload, but a fundamental failure in attack surface discovery. The DAST tool's crawler analyzed the initial HTML of profile.php. However, it did not and in its default configuration, could not simulate the user action of

clicking the "Add Comment" button. Because it did not trigger this event, it never initiated the JavaScript that sends the AJAX request to the add_comment.php endpoint. Consequently, the scanner was entirely unaware that add_comment.php existed or that it accepted user input. The vulnerability lay within a part of the application that was invisible to the scanner's crawling process. This provides a stark, practical demonstration of the detection gap identified by Zhang et al. (2021) for modern, dynamic applications.

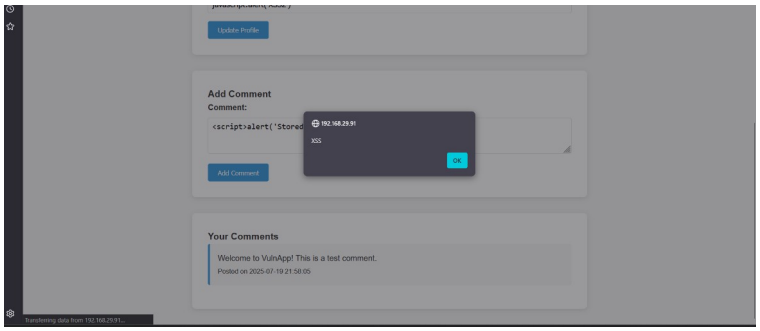

Figure 6.4: Stored XSS payload executing from a comment loaded asynchronously via an AJAX request.

6.5 Discussion

The findings from these four experiments provide a clear and compelling narrative. The DAST tool performed admirably when faced with well-known vulnerabilities in a static context but failed completely when confronted with modern evasion techniques. This confirms the primary research hypothesis that automated scanners can be easily fooled through JavaScript obfuscation or by placing vulnerabilities within the dynamically loaded content. These implications offer an important message to both the practical security community and academia. For practitioners, this work serves as yet another reminder that there is no substitute for a clean (as in sanitized) application. "Clean" reports generated by automated scans should not be confused with a secure application. This highlights the false sense of security, and the continued need for some level of manual pen testing, code reviews and using more complete DAST tools that fully emulate the browser environment and can be scripted to manipulate multi-step user workflows. For the academic community, the value of empirically demonstrating and verifying what has been deduced in the literature remains, as academics ability to validate research through experiments and empirical study is valid.

It offers a concrete case study that supports the call for more intelligent and context-aware security tools, as advocated by researchers focusing on everything from malware evasion (Fleury et al., 2021) to the specific challenges of detecting malicious JavaScript (Phung & Mimura, 2021).

It is important, however, to critique the design of this study and acknowledge its limitations. The primary limitation is the use of a single DAST tool (Burp Suite Community Edition). While representative, it is not exhaustive; commercial DAST tools may possess more advanced JavaScript analysis and crawling capabilities. A future study should conduct a comparative analysis across multiple commercial and open-source scanners to assess the prevalence of this detection gap across the market. Secondly, the obfuscation technique used

was rudimentary. Real-world attackers employ multi-layered, polymorphic obfuscation engines that present a far greater challenge (Rahman et al., 2025). Similarly, the AJAX functionality was a simple, single-step action. Modern SPAs feature complex, multi-step user journeys that would require even more sophisticated crawling logic to navigate.

Despite these limitations, the experimental design was effective in its purpose. By using a controlled environment and clear, binary test cases, it successfully isolated and demonstrated the specific failure modes of a traditional DAST approach. Improvements to the design for future work could involve implementing a more complex SPA with nested dynamic components and utilizing industry-standard obfuscation tools to create more resilient test payloads. This would allow for a more nuanced investigation into the precise threshold at which different scanners begin to fail. Ultimately, this research effectively puts the findings of the literature review into practice, confirming that as web applications continue to evolve, our methods for securing them must evolve in tandem.

Table 6.1: Summary of Scanner Performance Across Test Cases

Test Case	Vulnerability Type	Payload Detected?	Scanner Detection Result	Outcome
3.3.1	SQL Injection (Server-Side)	Yes	Detected	Success
3.3.2	Stored XSS (Simple, Non-Obfuscated)	Yes	Detected	Success
3.3.3	Stored XSS (Obfuscated, External Script)	Yes	Not Detected	Failure
3.3.4	Stored XSS (AJAX-Loaded Content)	Yes	Not Detected	Failure

7 Conclusion and Future Work

7.1 Conclusion

The focus of this research was to answer the paramount question - to what extent do traditional automated vulnerability scanners fail to identify vulnerabilities which are obscured by JavaScript obfuscation or dynamically loaded content? In the Introduction, we outlined four aims: (1) to provide a baseline of scanner performance against known vulnerabilities, (2) to evaluate scanner detection of obfuscated JavaScript payloads, (3) to assess detection of vulnerabilities in dynamically loaded content, (4) to examine the results for reproducible evidence of gaps in detection.

All four aims were achieved. In the baseline tests, the scanner was able to detect both SQL Injection and relatively trivial XSS, whilst we proved complete detection failure in the obfuscation and AJAX experiments. The analysis of logs and execution evidence gave us reproducible evidence of this gap and confirmed the central hypothesis.

The implications of these findings are profound. They reveal that organizations relying solely on traditional automated scanning are likely harboring a false sense of security. The efficacy of this research lies in its clear, reproducible demonstration of specific failure modes that attackers actively exploit. Its primary limitation is its focus on a single DAST tool and relatively simple evasion techniques. Nevertheless, it provides a powerful argument that the future of web security testing must shift towards tools with integrated, full-browser emulation and intelligent, interactive crawling engines.

7.2 Future Work

The findings of this paper open several meaningful avenues for future research that extend beyond simply testing more scanner parameters.

A critical next step would be to investigate the **semantics of user interaction**. This research showed that a scanner fails when it cannot simulate a 'click'. A follow-up project could design a framework that uses machine learning, perhaps leveraging Natural Language Processing (NLP) on element IDs and labels (e.g., 'submit', 'add comment', 'next page'), to intelligently predict and prioritize which user interactions are necessary to uncover hidden application functionality. This moves beyond simple crawling and into the realm of behavior-driven security testing, addressing the core AJAX evasion problem at a more fundamental level.

Another significant research direction would be the **automated de-obfuscation in a DAST context**. While some tools have sandboxing, it is often generic. A future project could focus on developing a specialized DAST module that uses dynamic binary instrumentation or advanced taint analysis specifically tailored for JavaScript. This module would not just execute the script but would actively trace data flows through complex obfuscation patterns to identify malicious sinks (like eval() or innerHTML) and flag the vulnerability, regardless of the obfuscation complexity. This would represent a direct and robust solution to the first evasion technique demonstrated.

Finally, there is potential for commercialization in creating a "DAST Helper" tool. This tool would not replace existing scanners but would augment them. It could act as a sophisticated proxy that uses AI to learn an application's valid user journeys and then automatically generate the complex scripts needed to guide a "dumb" scanner through a modern SPA's interface, effectively making old scanners viable for new applications. This presents a practical, market-oriented solution to a problem that this research has proven is both real and pressing.

References

Alaoui, R.L. and Nfaoui, E.H., 2022. Deep learning for vulnerability and attack detection on web applications: A systematic literature review. Future Internet, 14(4), p.118.

Alazab, A., Khraisat, A., Alazab, M. and Singh, S., 2022. Detection of obfuscated malicious JavaScript code. Future Internet, 14(8), p.217.

Asadi, M., Jamali, M.A.J., Heidari, A. and Navimipour, N.J., 2024. Botnets unveiled: A comprehensive survey on evolving threats and defense strategies. Transactions on Emerging Telecommunications Technologies, 35(11), p.e5056.

Chmiel, K. and Rajba, P., 2024, July. How to evade modern web cryptojacking detection tools? A review of practical findings. In Proceedings of the 19th International Conference on Availability, Reliability and Security (pp. 1-10).

Fleury, N., Dubrunquez, T. and Alouani, I., 2021. Malware: An overview on threats, detection and evasion attacks. arXiv preprint arXiv:2107.12873.

Kapoor, A., Gupta, A., Gupta, R., Tanwar, S., Sharma, G. and Davidson, I.E., 2021. Ransomware detection, avoidance, and mitigation scheme: A review and future directions. Sustainability, 14(1), p.8.

Kaur, J., Garg, U. and Bathla, G., 2023. Detection of cross-site scripting (XSS) attacks using machine learning techniques: a review. Artificial Intelligence Review, 56(11), pp.12725-12769.

Ma, Y., Wu, H., Tan, Y.A. and Li, Y., 2023, December. Research on evasion and detection of malicious javascript code. In International Conference on Machine Learning for Cyber Security (pp. 104-130). Singapore: Springer Nature Singapore.

Moog, M., Demmel, M., Backes, M. and Fass, A., 2021, June. Statically detecting javascript obfuscation and minification techniques in the wild. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (pp. 569-580). IEEE.
Phung, N.M. and Mimura, M., 2021. Detection of malicious javascript on an imbalanced dataset. Internet of Things, 13, p.100357.
Ponomarenko, G.S. and Klyucharev, P.G., 2023. On improvements of robustness of obfuscated JavaScript code detection. Journal of Computer Virology and Hacking Techniques, 19(3), pp.387-398.
Rahman, R.U., Tomar, D.S. and Kumar, P., 2025. A polymorphic code generation engine with obfuscation techniques for protecting web bot attacks. Iran Journal of Computer Science, pp.1-26.
Sahan, Z.M. and Abdulrahman, A.A., 2025, March. Malware detection of PDF documents based on machine learning techniques (A review). In AIP Conference Proceedings (Vol. 3264, No. 1, p. 030036). AIP Publishing LLC.
Shewale, S., 2024. Automated Threat Hunting for JavaScript-based Obfuscated Phishing Email Attachments (Doctoral dissertation, Dublin, National College of Ireland).
Singh, T., Singh, K.U., Varshney, N., Gupta, P. and Kumar, G., 2024, February. Enhancing web browser extensions: Preventing JavaScript code injection and vulnerabilities. In International Conference On Innovative Computing And Communication (pp. 547-557). Singapore: Springer Nature Singapore.
Varghese, A., 2023. A Dynamic Analysis Framework for Classifying Malicious Webpages (Master's thesis, University of Dayton).
Wang, Q., Chen, J., Jiang, Z., Guo, R., Liu, X., Zhang, C. and Duan, H., 2024, May. Break the wall from bottom: Automated discovery of protocol-level evasion vulnerabilities in web application firewalls. In 2024 IEEE Symposium on Security and Privacy (SP) (pp. 185-202). IEEE.
Yarphel, T. and Rani, D., 2025. Cross-Site Scripting (XSS) in Web Applications: A systematic literature review. International Journal of Science and Research Archive, 15(2), pp.1658-1667.
Zhang, B., Li, J., Ren, J. and Huang, G., 2021. Efficiency and effectiveness of web application vulnerability detection approaches: A review. ACM Computing Surveys (CSUR), 54(9), pp.1-35.
Zh, Z.A. and Santeeva, S.A., 2024. COMBATTING QAKBOT: A REVIEW OF DETECTION AND ANALYSIS TECHNIQUES. Вестник науки, 1(11 (80)), pp.857-863.