WebSocket Endpoint Vulnerability Analysis

Sushma E
Dept. of CSE
PES University
Bengaluru, Karnataka
sushmae@pes.edu

Tejas Mysore Harish
Dept. of CSE
PES University
Bengaluru, Karnataka
tejasharish1234@gmail.com

Puskar Mishra
Dept. of CSE
PES University
Bengaluru, Karnataka
mishrapuskar2004@gmail.com

***Abstract*—WebSocket is a technology that allows a client and a server to communicate in real-time, using a single TCP connection. WebSockets are a significantly faster means of communication compared to HTTP, enabling fast real-time communication. However, this increase in communication speed comes at the cost of minimal security. This lack of security can lead to exposure of sensitive data, unauthorized access, and many more problems. This paper discusses the detection and testing of WebSockets in real-world web applications. WebSocket endpoints of real-world websites are obtained through the process of crawling, and then, test attacks are run on the WebSocket endpoints, including tests on handshaking, session management, origin and authentication checks, subprotocols, encryption, Denial of Service (DOS), resource management, and cross-origin attacks, and protocol fuzzing using different payloads. By running these tests, we obtain a complete and comprehensive report on all vulnerabilities for a given WebSocket.**

***Index Terms*—WebSocket, Vulnerabilities, Security, Real websites, WS/WSS**

I. Introduction

A. What is a WebSocket?

WebSocket is a full-duplex protocol for communication, as standardized by IETF in RFC 6455 [1]. The system operates on a single, long-lived TCP connection, enabling real-time, bidirectional communication between servers and clients. Rather than using a request-response cycle for each data exchange, WebSocket provides an open TCP channel through which messages can be sent and received at any moment, without delay. Thus, it is suitable for time-critical web applications, such as live chat services, collaborative software, online game sites, and financial data feeds [2].

The initiation of a WebSocket connection is via an HTTP request. The client upgrades to WebSocket by sending a connection upgrade request header. After the channel is initiated, it persists until both ends terminate it explicitly. The upgraded method enhances HTTP's intrinsic security features, but introduces some new issues which can be exploited without adequate security precautions [3], [4].

B. WebSocket vs. HTTP

Although both WebSocket and HTTP utilize TCP and standard ports (such as 80 or 443), their working models differ profoundly. HTTP is a rigid request-response-based model in which all communication begins at the client and terminates at the server. In contrast, WebSocket offers event-based communication, and both endpoints can transmit messages at any time once the connection has been established [5], [6].

HTTP is a stateless protocol that is enhanced by its well-defined security features, including CORS, CSRF, and robust caching and authentication mechanisms. WebSockets, on the other hand, are stateful and focus more on instant communication, lacking such controls by default. While dynamic applications benefit from their performance, their flexibility requires developers to manually implement security best practices.

C. Why WebSocket is Vulnerable?

The lack of strict validation and access control measures can weaken the security of WebSockets for developers. Since a WebSocket connection remains continually open, an attacker can use it to gain unauthorized access to the server and exchange messages with it without requiring further authentication [7]. Popular vulnerabilities include:

- Lack of validation of the Origin header or its absence.
- Acceptance of unauthenticated or expired sessions.
- Ineffective message framing and payload inspection.

These vulnerabilities can be exploited through a wide range of attacks, including session hijacking, denial-of-service (DoS) attacks, and data exfiltration, which can cause serious problems for both the client and server.

D. Motivation and Contribution

Although there is a rise in the use of WebSockets in contemporary web designs, security research and tooling for their use are still largely constrained to HTTP. Inadequate assessment of WebSocket endpoints by existing vulnerability scanners and application firewall models tends to create blind spots in enterprise threat modelling. Additionally, most developers rely on the channel's security following a successful WebSocket handshake to prevent potential abuse by attackers. The paper provides a methodical vulnerability assessment of publicly exposed WebSocket endpoints. In particular, we craft and run WebSocket-specific tests that detect typical implementation weaknesses across classes, including: handshake validation, origin enforcement, authentication controls, message handling, and protocol abuse. The results reveal widespread vulnerabilities, emphasizing the need for heightened developer awareness and enhanced tool support. It is to be noted that these tests are being conducted only on publicly available WebSockets of real-time websites that can be accessed without API keys, and there is no intention to cause harm or damage to the servers of these websites.

II. Problem Statement and Objectives

Real-time capabilities, such as live chats, stock updates, multiplayer gaming, and collaborative editing, are made possible by WebSocket communication in modern web applications. WebSocket connections differ from traditional HTTP traffic in that they are stateful, long-lived, and often dynamically created through client-side JavaScript. Unfortunately, the security of WebSocket endpoints is often overlooked during application development and security audits.

The majority of present vulnerability detectors and web application firewalls (WAFs) concentrate on HTTP traffic and lack significant support for analysing the WebSocket protocol. Consequently, numerous security vulnerabilities, including broken authentication and origin bypasses, as well as incorrect frame handling and denial-of-service vectors, often remain undiscovered in production settings. This is an unfortunate reality [8], [9].

This research aims to develop an automated medium for testing WebSocket security with great precision on real websites. The system is designed to:

- Crawl WebSocket endpoints by simulating the browser's behaviour.
- Identify and implement 90 protocol-compliant attack scenarios from 9 vulnerability classes.
- Use heatmaps, bar charts, and detailed summaries to report results.

Combining browser automation, intelligent crawling, and a comprehensive analysis of WebSockets, this research aims to provide an understanding of the various vulnerabilities in modern WebSocket infrastructure, with the goal of secure management in the near future.

III. Implementation

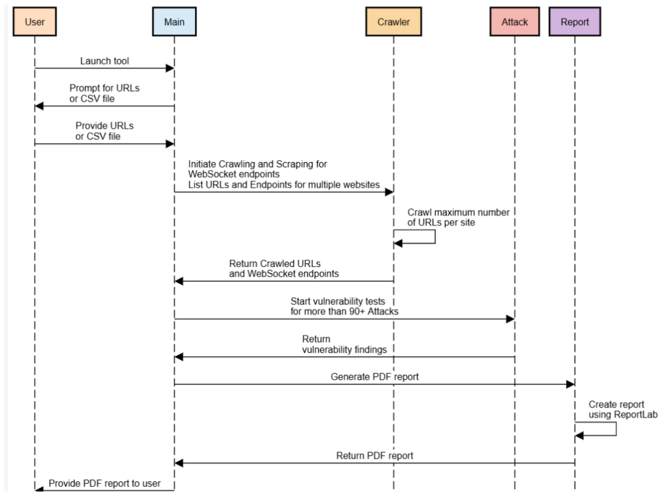
User
Main
Crawler
Attack
Report
Launch tool
Prompt for URLs or CSV file
Provide URLs or CSV file
Initiate Crawling and Scraping for WebSocket endpoints List URLs and Endpoints for multiple websites
Crawl maximum number of URLs per site
Return Crawled URLs and WebSocket endpoints
Start vulnerability tests for more than 90+ Attacks
Return vulnerability findings
Generate PDF report
Create report using ReportLab
Return PDF report
Provide PDF report to user

Fig. 1. Sequence Diagram of WebSocket Vulnerability Testing Tool

Explanation for sequence diagram

Fig. 1 shows the sequence diagram which illustrates the workflow of the tool, beginning via the main file. The user can either manually enter a website URL or upload a CSV file containing multiple target websites. Once the input is received, the main module begins execution of the process of crawling.

The crawler module uses the Playwright library to obtain WebSocket endpoints. The crawler extracts these WebSocket endpoints, along with all scanned URLs and relevant metadata, and returns them to the main module. The process through which this is done is explained in detail in Stage 1. If no WebSockets are found, the tool falls back on user-provided endpoints or extracts alternatives from the uploaded CSV file.

The discovered WebSocket URLs are then forwarded to the attack module for security evaluation. This module performs over 90 distinct vulnerability tests targeting common WebSocket security flaws. These include malformed handshake requests, missing or expired authentication tokens, spoofed origin headers, improper session handling, and more.

Finally, all test results are passed to the report generator module. This module compiles the findings into a structured, visually informative PDF report. The report includes graphs such as bar charts, pie charts, and heatmaps, which illustrate various findings in a pictorial manner for a quick understanding on problems, but also proceeds to go in depth, explaining the issues in detail.

Stage 1: Finding the WebSocket Endpoints

A. Browser Automation with Playwright: Microsoft's Playwright, an open-source browser automation library, is utilized for real-time page analysis [10]. Through APIs that emulate user interactions, JavaScript programming, and network traffic interception, Playwright provides complete control over browser behaviour through Chromium, a headless browser (it does not have a GUI).

All HTTP and WebSocket traffic, including dynamically generated resources and delayed WebSocket initializations triggered by AJAX or script evaluation, is monitored during web browsing.

Playwright employs various methods of stealth, such as:

- Randomization of user-agent strings.
- The fabrication of browser fingerprinting APIs.
- Disabling automation detection flags.

With these features, the crawler can bypass bot detection mechanisms used by many modern websites, continuing to provide complete endpoint discovery. Through Playwright, the system can:

- Load complex web applications.
- Interact with dynamic elements.
- Observe WebSocket handshake requests.
- Obtain real-time WebSocket URLs that are hidden or deferred.

B. Web Crawler and Scraper: The system is designed to crawl websites to identify resources that are accessible through programming methods. The crawler utilises browser automation to simulate real user interactions, rather than relying solely on static HTML. By rendering web content dynamically, it can uncover resources that static analysis might overlook.

In addition, a scraping module has been integrated to extract valuable information from both the page and network responses. This includes embedded JavaScript, JSON payloads, inline HTML, and similar data, all of which enhance the system's capability for thorough endpoint discovery. By combining crawling and scraping, the system creates a comprehensive map of possible communication channels that are subjected to security analysis [11].

C. Crawling Procedure: The endpoint discovery process is characterised as follows:

1) The system initiates Chromium browser with Playwright and generates an ensemble of randomly selected user agents for stealth.
2) The system enables navigation and monitoring by loading the page for each target URL and listening to both incoming and outgoing network traffic.
3) JavaScript files, AJAX calls, API responses, and observed traffic are used to extract patterns from WebSocket endpoints through regular expressions. The scheme used to identify a WebSocket URL is `ws` or `wss`.
4) The validation process involves validating discovered endpoints and discarding paths that are not navigable, such as fonts or images. The remaining routes are redirected for further exploration within the domain, subject to the specified request and depth limits.
5) The crawler stops when all paths have been explored or when domain-specific limits are met.
6) The system returns a more refined set of WebSocket URLs for every domain.
7) These WebSockets are then put through a test to remove all junk that may have been entered during the process of crawling, and also remove the duplicate URLs.

We choose a maximum of three representative endpoints per website for testing, taking into account activity and uniqueness.

By using this pipeline, it is possible to identify and capture WebSocket endpoints that are dynamically generated, delayed, or hidden, which are typically not visible to traditional crawlers.

Stage 2: Executing Vulnerability Tests on Endpoints

Our approach involved executing 90 protocol-compliant test attacks to rigorously test the security of discovered WebSocket endpoints, as per RFC 6455, the official specification for the WebSocket protocol. Fig. 2 shows the distribution of all the attacks performed in this module. Nine different types of vulnerability are present, each associated with a unique phase of WebSocket interaction or protocol handling. The classification system ensures that our tool covers a wide range of aspects while also providing sufficient depth in each category.

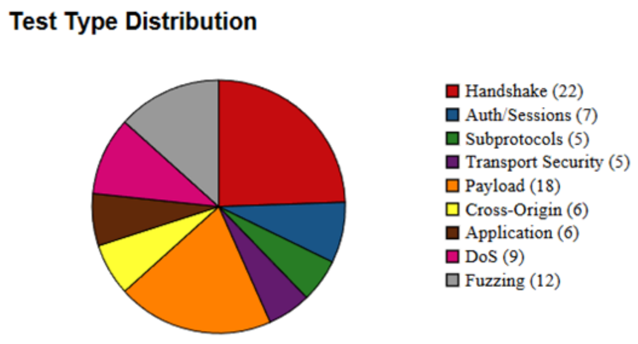

Fig. 2. Test Distribution

A. Handshake and Protocol Validation: When the WebSocket protocol is to be activated, it begins with a handshake to upgrade the HTTP/1.1 connection. This is of utmost importance as it establishes the boundary between stateless HTTP and the persistent, stateful, full-duplex communication, that is, the WebSocket.

Twenty-two different vulnerabilities are tested (#1–22) in this category [6], [8]. By modifying or eliminating crucial fields in the handshake request, each test endeavours to establish whether the server adheres strictly to the WebSocket upgrade specifications outlined in RFC 6455.

A typical handshake request:

```
GET <path> HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: <base64 encoded key>
Sec-WebSocket-Version: 13
```

A dependable server responds:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: <key>
```

Tests include:

- An invalid port, a non-WS scheme, and an HTTP/0.9 Handshake check to evaluate how servers handle protocol-level transgressions.
- Manipulation of headers by omitting the Sec-WebSocket-Key, duplicating the Sec-WebSocket-Key, sending the wrong upgrade header, removing host headers, and faking the host to check if servers are vulnerable to downgrade, spoofing, or misrouting.
- Encoding anomalies like Unicode URL, Long URL Path, and Case-Sensitive Headers to evaluate the robustness of URL parsing and header normalization.

Accepting non-compliant handshake requests, improper header validation, or insecure fallback mechanisms that undermine the integrity of protocol negotiation can lead to potential downgrade attacks [7].

B. Authentication and Session Management Tests: This test category (#23-29) comprises seven specific tests that investigate the security measures employed by servers to manage user identity, session validation, and token access control over persistent WebSocket connections. In contrast to stateless HTTP, which mandates authentication per request, WebSocket requires a one-time handshake followed by continuous communication, creating potential risks if access control checks are not performed during the connection time.

This group simulates real-life scenarios, such as:

- No Session Cookie and Missing Authentication: Attempting unauthenticated connections to evaluate whether the server enforces login or session validation before upgrade [9], [12].
- Expired Cookie, Fake Token, and Stale Session Reconnect: Using outdated or forged credentials to test resilience against token replay and session fixation attacks.
- HTTP Session Reuse: Reusing an HTTP-authenticated session without re-validating the origin or token freshness.
- Cross-Site Cookie Hijack: Initiating connections from cross-origin contexts to exploit improperly scoped or unsecured session cookies.

Exploiting these weaknesses would result in:

- Private resources being accessed without permission.
- Session hijacking and impersonation.
- Privilege escalation.

These tests are necessary to ensure that the server maintains strict isolation between user sessions, origins, and scopes while ensuring per-connection validation is rigorous.

C. Subprotocol and Extension Handling Tests: Through the use of WebSocket, clients can request subprotocols (such as MQTT or STOMP) and extensions (like permessage-deflate) that alter communication semantics or compress the data. The category comprises five tests (#30–34) that aim at detecting improper negotiation, injection, or fallback behaviour in subprotocol and extension handling.

Test scenarios:

- Invalid Subprotocol and Unaccepted Subprotocol: Requesting unsupported or arbitrary protocol strings to determine whether the server fails closed or defaults insecurely.
- Conflicting Subprotocols: Sending multiple mutually exclusive protocols to test negotiation logic.
- Fake Extension and Conflicting Extensions: Declaring bogus or contradictory extensions to identify parser confusion, invalid negotiation, or resource overcommitment.

These attacks demonstrate if the server is at risk of:

- Protocol Downgrade Attacks: The attacker employs an inadequate or non-functional protocol.
- Unexpected Code Paths: The invalidated extension behaviour leads to unexpected code paths.
- Mismanagement of buffers as a result of incomplete or unsuccessful negotiations.

The correct execution must ensure the validity of all protocol and extension declarations, while also rejecting connections that fail during the negotiation process.

D. Transport Security and Encryption Tests: The implementation of secure WebSocket connections (wss://) over TLS, utilizing strong cryptographic primitives and providing downgrade resistance, is necessary for ensuring their confidentiality and integrity. The server's robustness to transport-layer security is evaluated in this category, which comprises five tests (#35-39).

The following vectors are examined:

- TLS Downgrade, HTTP/1.0 Downgrade, and Spoofed Connection Header: Attempting to establish connections with lower protocol versions or malformed upgrade headers to simulate downgrade attempts.
- Weak TLS Ciphers: Forcing the use of insecure or deprecated cipher suites during TLS negotiation.
- Certificate Mismatch: Initiating wss:// connections with invalid, mismatched, or self-signed certificates to test server-side certificate validation and client behaviour.

The application could be vulnerable to the following security flaws:

- Man-in-the-middle (MITM) attacks.
- Session hijacking, via interception or manipulation of traffic.
- Replay attacks that occur if encryption is implemented improperly or if it is disabled.

Well-configured servers must reject any deviation from strong TLS configuration, prevent plain-text upgrades from HTTP/1.0, and fail the connection when certificate or header anomalies are detected.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Fig. 3. WebSocket Frame Format

E. Payload Handling and Fragmentation Tests: One of the most important part of the WebSocket protocol is the framing of messages, as all messages are sent as data frames structured in the format illustrated in Fig. 3 [13]. The purpose of this test class is to determine if the server applies correct frame parsing and rejects payloads that are malformed, fragmented, or improperly masked, using 18 test cases (#40–57).

Test categories:

- Opcode Validation:
  - Sending frames with undefined or illegal opcodes is prohibited using the Undefined Opcode and the reserved Opcode.
  - Using binary as text, text as binary to exchange data types and create errors in interpretation.
- Framing and Fragmentation:
  - Zero-length fragmentation, invalid payload length, negative payload lengths, mismatched payload, and oversized control frame testing based on the implementation of fragment rules, size limits, internal tracking, etc.
  - Checking server performance during connection teardown, no connection, long closed reason, and invalid close code.
- Masking Violations:
  - To get around the masking requirement on client-to-server frames, one can use an invalid masking key to unmask a client's frame.
- Encoding and Control Frames:
  - Validating encoding and rejecting non-standard characters in textual payloads through the use of null bytes in texts.
  - The use of reserved bits to determine if servers are incorrectly supporting undocumented features or extensions can be detected by checking for invalid RSV bits.

These tests help detect:

- Vulnerabilities in parsing that could cause memory corruption or logic bypass.
- The desynchronisation which can occur due to improper fragmentation or interleaving of frames.
- Security breaches that occur due to the rules for masking or encoding that may not be strictly enforced.

When connections are encountered with incorrect or non-compliant frames, the server should immediately disconnect and record these attempts for additional scrutiny.

F. Cross-Origin and Mixed Content Tests: WebSocket applications based on browsers are vulnerable to cross-origin attacks, where the same-origin policy is a vital security measure. Attacks #58–63 are a set of six targeted tests that simulate cross-origin contexts, mixed security levels, and origin spoofing.

Test cases:

- Missing CORS Headers: Verifying if the server discloses sensitive content to unauthorized origins [6].
- Cross-Origin Iframe and PostMessage Abuse: Emulating attacks from embedded frames or malicious scripts.
- Mixed Content: Testing insecure (ws://) WebSocket connections initiated from secure (https://) web contexts.
- Missing Origin Check and Spoofed URL: Omitting or forging the Origin header to test origin-based access controls [14].

Threats:

- Cross-Site WebSocket Hijacking (CSWSH) [15], [16].
- The transmission of data to unapproved domains.
- Clickjacking and iframe abuse.
- Browsers' trust issues in HTTPS scenarios.

Our framework verifies that WebSocket endpoints perform a proper check on both the Origin and Referrer headers and reject any cross-origin attempts that are not explicitly whitelisted.

G. Application-Layer Logic and Misconfiguration Tests: This category emphasizes on exploits at the application level, rather than focusing on just basic protocol-level bugs. There are six attack cases (#64–69) that discuss inadequate sanitization, insecure headers, and overexposed API design patterns.

Key scenarios:

- Error Message Leak and Server Disclosure: Inducing errors to determine if internal server paths, debug logs, or stack traces are exposed.
- URL Path Traversal: Simulating directory escape attempts such as `../../etc/passwd`.
- Invalid Content-Type and Missing Security Headers: Checking for lax content negotiation and absent headers like Content-Security-Policy or X-Frame-Options.
- Query Parameter Flood: Sending excessive query parameters to evaluate request handling limits.

Lack of input validation, verbal error reporting, or internal routing logic is a common cause of these vulnerabilities. The implementation of a secure WebSocket involves fail-safe error handling and sanitized responses, particularly when protocol upgrades bridge the client's direct input to backend logic.

H. DoS and Resource Management Tests: To resist resource usage by malicious actors, WebSocket servers must be kept to a minimum. We implement nine focused tests (#70–78) to test the server's ability to handle denial-of-service (DoS) conditions and resource mismanagement [17].

Tests include:

- Connection Flood and Max Connections: Stressing concurrent connections beyond expected thresholds.
- Oversized Message, Large Payload Resource Leak, and High Compression Ratio: Sending excessive or highly compressible data to evaluate memory usage.
- Idle Timeout Abuse and No Timeout Policy: Holding idle connections open indefinitely.
- TCP Half-Open Resource Leak: Simulating clients that initiate but never complete handshakes, causing resource starvation.
- No Compression Negotiation: Forcing compression even when the server has not agreed to it

The tests replicate practical DoS vectors, such as Slowloris attacks, compression bomb amplification, and connection pool exhaustion. These tests are not static. The failure of servers to disconnect idle or non-compliant clients can lead to poor performance and full-service outages.

I. Protocol Fuzzing Tests: By transmitting syntactically or semantically malformed WebSocket frames, protocol fuzzing is an advanced technique for identifying zero-day vulnerabilities, memory safety issues, and unexpected behaviours. This suite comprises 12 specialized fuzzing tests (#79-90) that examine control-plane and application-layer attack surfaces, and are described in Table 1 [18], [19].

Fuzz Test	**Description**
Malformed JSON	Incomplete or corrupted JSON string
XSS Attempt	HTML injection with `<script>` tag
Large Payload for DoS	JSON body with 1 million `A` characters
Invalid Binary Frame	Arbitrary invalid binary payload
Command Injection Simulation	Payload attempting `whoami` execution
SQL Injection Simulation	SQL logic bypass pattern (`' OR '1'='1`)
Expression Evaluation	Template-based payload: `${7*7}`
Null Bytes in JSON	JSON with embedded `\0` characters
Unicode Characters	Payload containing emoji: `[emoji symbols]`
Oversized DoS Message	2 million-character JSON message
Path Traversal Simulation	`{"path": "/../../etc/passwd"}`
PostMessage Abuse	Script injection into `window.postMessage()`

These assessments can aid in identifying:

- Unhandled exceptions, segmentation faults or crashes affecting the parser.
- Business logic bypass upon failure of input validation.
- Encoding or escaping defects that lead to XSS, command injection, or data leakage.

All fuzzing inputs are subject to WebSocket framing rules, meaning that crashes arise due to payload logic and not protocol violations. A server's ability to handle malformed inputs is a reliable indicator of its readiness for production.

IV. Equations and Workflow Logic

This section outlines the internal logic and fundamental components of our WebSocket vulnerability analysis framework. The processes of crawling, scraping, attack execution, as well as result aggregation, is performed using defined sets of functions and control structures [20].

A. Core Sets

Let the initial seed URL be denoted by:

$$U_0 = \text{Starting URL}$$

While crawling the target site, the following sets are generated:

- C: Set of all successfully crawled URLs.
- D: Set of all discovered URLs, including visited and queued.
- W: Set of all WebSocket URLs found in D.

WebSocket endpoints are identified as:

$$W = \{u \in D \mid u \text{ starts with “ws://” or “wss://”}\}$$

B. API Scraping Logic

Embedded URLs are extracted by parsing JSON responses from API calls. Let R_i be the i-th response string. The regex pattern applied is:

$$\texttt{Regex} = (\texttt{https?}|\texttt{wss?})\texttt{://[\textbackslash s "\textbackslash '}]$$

Filtered URLs matching the above pattern are added to D for further crawling.

C. Recursive Crawling Conditions

Let Q be the crawling queue. For each URL $u \in Q$:

- $\text{depth}(u)$: Depth of URL u from seed U_0.
- max_depth: Maximum allowed depth.
- max_requests: Cap on crawl requests.

The recursive crawl condition is:

$$\forall u \in Q : (u \notin C \land \text{depth}(u) \leq \text{max\_depth}) \Rightarrow \text{crawl}(u)$$

D. Filtering Conditions

Let F be the set of static file extensions:

$$F = \{.js, .css, .png, .jpg, .gif, .woff, .svg, \dots\}$$

Let $\text{ext}(u)$ denote the file extension of URL u. Then:

$$u \in Q \Rightarrow \text{crawl}(u) \iff \text{ext}(u) \notin F$$

E. Final WebSocket Endpoint Collection

After completing the crawl, the WebSocket set W is redefined as:

$$W = \{u \in D \mid \text{scheme}(u) \in \{\texttt{"ws"}, \texttt{"wss"}\}\}$$

This set W becomes the input to the vulnerability engine.

F. WebSocket Vulnerability Testing Logic

Let T be the set of test functions. Each test function $t \in T$ operates on endpoint $w \in W$. The set of vulnerabilities V is defined as:

$$V = \bigcup_{w \in W} \bigcup_{t \in T} t(w)$$

Where:

- W: WebSocket endpoint set.
- T: Test function set.
- V: Collected vulnerability reports.
- $t(w)$: Result of test function t applied to w, which may include header manipulation, payload injection, or origin spoofing.

G. Workflow Controller Logic

From collection to reporting, each domain u is processed and its results are represented by:

$$R[u] = \{C_u, W_u, V_u, T_u\}$$

Where:

- C_u: Crawled URLs from domain u.
- W_u: WebSocket endpoints found.
- V_u: Vulnerabilities detected.
- T_u: Total scan duration for u.

V. Results and Analysis

We have developed an automated framework that can identify real-world flaws in WebSocket implementations, as demonstrated in this project. Upon crawling target domains and extracting live WebSocket endpoints, our engine performed 90 vulnerability tests, which included protocol validation, session management flaws, improper subprotocol handling, transport security weaknesses, and misconfiguration of the application layer.

Specified payloads, including malformed handshake headers, expired authentication tokens, over-sized binary messages, and protocol fuzzing mutations, were applied to each endpoint. This was done with great precision. The following is a heatmap and a bar chart, presented to illustrate the security posture of each site and identify common trends in vulnerabilities [4].

A. WebSocket Vulnerability Heatmap

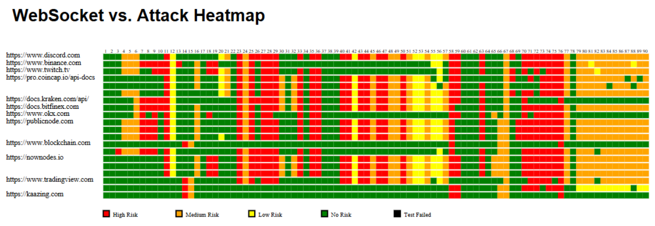

Fig. 4. Heatmap of Website vs Test Attacks

Fig. 4 is the attack heatmap that displays the vulnerability of each domain scanned across all test categories. In our test suite, we recorded 90 attacks, with each column representing a distinct attack. On analysis of the heatmap, we can draw a couple of insights. Firstly, we notice there are four distinct blocks of issues. These blocks can be attributed to four categories, namely authentication, payload handling, DOS, and fuzzing. Secondly, numerous attacks appear to occur in about 90% of the WebSockets tested, and these are critical attacks that need to be addressed promptly. Thirdly, improper authentication of incoming clients is a crucial aspect, and it is observed that most servers do not adhere to the protocol standard. Fourthly, it is noted that the DOS attacks, which are meant to stress the server by sending requests that are too large to be processed, are not being rejected. This implies that the server is accepting message requests of huge sizes, and the acceptance of these requests means that the server can be easily overwhelmed, reducing its efficiency. Lastly, it is worth noting that certain websites, such as Blockchain.com and Tradingview.com, appear to have robust security measures, which is reassuring to users.

B. Vulnerability Distribution by Type

Fig. 5 displays the vulnerability categories against the total number of vulnerabilities found for each category. There are nine classes of attacks, and this bar chart shows how many vulnerabilities were found from testing all the endpoints. On reviewing the graph, it is evident that payload handling is the most significant vulnerability class. Upon detailed review, it is observed that while the website does respond with messages such as bad request, these messages are at the application layer not the protocol layer. The server is still allowing communication when, in fact, the server must close the connection immediately to prevent such attacks from affecting either server or client. This is a recurring theme with the protocol fuzzing attack set as well [19].

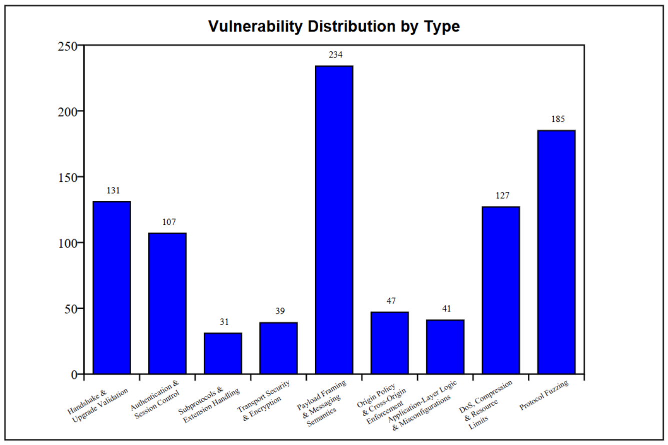

Fig. 5. Vulnerability Category vs Number of Vulnerabilities

C. Severity Distribution & Risk Posture

The visual analysis of results demonstrates that:

- A large number of tests are considered high-severity, particularly those resulting from unauthenticated access and denial-of-service attacks. These categories need to be addressed rather urgently.
- Less exploitable bugs, primarily resulting from failures to comply with protocol regulations for frame handling established by the IETF, are classified under the medium-severity category. However, they can still be significant.
- Although not immediately exploitable, low-risk issues indicate the presence of outdated or non-strict implementations and highlight areas where standards compliance is erratic.

D. In Depth Analysis

The complete review of all WebSockets of each website can be found in the Report directory of the following repository: (GitHub Repository Link)

VI. Use Case

This testing tool is a relatively simple tool to operate. No prior knowledge is needed to utilize it. It runs entirely on Python and only requires a couple of Python modules to be installed for successful execution. The tool is compatible with real-world web applications and complies with RFC 6455 [1]. It features a simple command-line interface that allows users to enter website names. If the user wishes to perform the test for multiple websites, they can provide the input in CSV form. Given the website name, it is mainly able to successfully identify multiple WebSocket endpoints through crawling. It provides exhaustive test coverage, which is custom-built for WebSocket endpoints, handling various domains. Our tool also provides a simple, easy-to-read PDF report that offers analysis for each website tested [20]. It is limited due to its beginner-friendly approach, as it does not provide integration with browser cookies and also does not delve deeply into complex problems such as CSWH, which are challenging to implement without integrating with external tools. Table 2 compares our tool against tools in the industry, and our tool stands out to be a better candidate for WebSocket analysis.

TABLE I
COMPARISON OF WEBSOCKET SECURITY TESTING TOOLS

Metrics	**Our Tool**	**Burp Suite Pro**	**OWASP ZAP**	**WS-Attacker**
Accessibility	Open Source	Closed Source	Partially Open	Limited Open
Input Methods	CSV + User Input	User Input	User Input	One URL Limit
Crawling	Exhaustive with Full JS Support	Partial JS Support	Limited Crawling	No Crawling
WebSocket Test Coverage	90 Tests	Basic Tests	Basic Tests	10 Attacks Only
Report Output	PDF + Charts	Manual Review	HTML Format	None Available

VII. CONCLUSION

This paper offers a comprehensive and automated system for identifying security flaws in WebSocket implementations across real-world web platforms. Using Playwright's headless browser automation, the system can search for active WebSocket endpoints. Once it has identified endpoints, the tool conducts a range of 90 targeted tests, classified into nine vulnerability classes. Various attack vectors are emulated, and the endpoint's security measures are evaluated by examining the server's response to the attack, or lack thereof. A PDF report is generated, providing a comprehensive analysis of the tested WebSockets.

WebSocket security is often overlooked by developers, who prioritize instant communication and application-level handling over following protocol standards. There appears to be a lack of attention to the proper security of WebSockets, and this study aims to address that. [6], [9]

Our research revealed important vulnerabilities in WebSocket security. Simple misconfigurations or failure to comply with RFC standards were also observed in well-known domains. These are fundamental aspects that need to be addressed to prevent any major security issues from arising.

In summary, the tool offers the following:

- A scalable, reproducible, and automated approach to testing WebSocket security.
- A visual understanding of potentially vulnerable domains.
- Key insights for developers, security teams, and researchers to enhance the safety of real-time applications.

REFERENCES

[1] I. Fette and A. Melnikov, "The WebSocket Protocol," IETF, RFC 6455, Dec. 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6455

[2] P. Murley, Z. Ma, J. Mason, M.Bailey, and A.Kharraz, "WebSocket Adoption and the Landscape of the Real-Time Web," in *Proc. 2021 Web Conf. (WWW)*, 2021. [Online]. Available: https://faculty.cc.gatech.edu/~mbailey/publications/www21_websocket.pdf

[3] N. Nadeem, Y. Raza, A. Sajid, H. Razzaq, and S. Vidanagamachchi, "Review Analysis of WebSocket Security: Case Study," *IETI Transactions on Data Analysis and Forecasting (iTDAF)*, 2024. [Online]. Available: https://doi.org/10.3991/itdaf.v2i2.51015

[4] S. Arora, "WebSockets and Real-Time Communication," *Insights2Techinfo*. [Online]. Available: https://insights2techinfo.com/websockets-and-real-time-communication/

[5] Y. Fu and M. García-Valls, "Security aspects of full-duplex web interactions and WebSockets," *2023 20th ACS/IEEE Int. Conf. on Computer Systems and Applications (AICCSA)*, Giza, Egypt, 2023, pp. 1–8, doi: 10.1109/AICCSA59173.2023.10479302. [Online]. Available: https://ieeexplore.ieee.org/document/10479302

[6] H. Kuosmanen, "Security Testing of WebSockets," B.S. thesis, Jyväskylä Univ. of Applied Sciences, 2016. [Online]. Available: https://janet.finna.fi/Record/theseus_jamk.10024_113390?sid=5087676388

[7] BrightSec, "WebSocket Security: Top 8 Vulnerabilities and How to Solve Them," 2021. [Online]. Available: https://brightsec.com/blog/websocket-security-top-vulnerabilities

[8] K. S. Greasley and P. G. Lubbers, "WebSocket Security," in *The Definitive Guide to HTML5 WebSocket*, Berkeley, CA: Apress, 2013, pp. 129–149. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4302-4741-8_7

[9] S. Ghasemshirazi and P. Heydarabadi, "Exploring the attack surface of WebSocket," *arXiv preprint* arXiv:2104.05324, Apr. 2021. [Online]. Available: https://arxiv.org/abs/2104.05324

[10] Microsoft, "Playwright: Fast and reliable end-to-end testing for modern web apps," GitHub, 2023. [Online]. Available: https://playwright.dev/

[11] R. Koch, *On WebSockets in penetration testing* [Diploma Thesis], Technische Universität Wien, 2013. [Online]. Available: https://resolver.obvsg.at/urn:nbn:at:at-ubtuw:1-58898

[12] B. Pandey and P. Kumar, "Security risks and best practices for securing WebSocket communications," *Int. J. Novel Res. Dev. (IJNRD)*, vol. 9, no. 7, pp. C753–C755, Jul. 2024. [Online]. Available: https://ijnrd.org/papers/IJNRD2407274.pdf

[13] K. Peacock, "WebSocket Framing, Masking, Fragmentation, and More," openmymind.net, Jul. 2013. [Online]. Available: https://www.openmymind.net/WebSocket-Framing-Masking-Fragmentation-and-More/

[14] MITRE, "CWE-1385: Missing Origin Validation in WebSockets," Common Weakness Enumeration, 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/1385.html

[15] PortSwigger Web Security Academy, "Cross-site WebSocket Hijacking (CSWSH)," [Online]. Available: https://portswigger.net/web-security/websockets/cross-site-websocket-hijacking

[16] J. Somerville, "Cross-site WebSocket Hijacking (CSWSH)," Praetorian Labs Blog, May 7, 2019. [Online]. Available: https://www.praetorian.com/blog/cross-site-websocket-hijacking-cswsh

[17] J. Erkkilä, "Websocket security analysis," Aalto Univ. Sch. Sci., 2012. [Online]. Available: https://juerkkil.iki.fi/files/websocket2012.pdf

[18] I. P. A. Dharmaadi, E. Athanasopoulos, and F. Turkmen, "Fuzzing Frameworks for Server-Side Web Applications: A Survey," arXiv preprint arXiv:2406.03208, Jun. 2024. [Online]. Available: https://arxiv.org/abs/2406.03208

[19] X. Zhang et al., "A Survey on the Development of Network Protocol Fuzzing Techniques," *Electronics*, vol. 12, no. 13, p.2904, 2023. [Online]. Available: https://doi.org/10.3390/electronics12132904

[20] T. Wirasingha and N. Ruwan Dissanayake, "A Survey of WebSocket Development Techniques and Technologies," in *Proc. Professional Integration for Secure Nation*, Sep. 2016. [Online]. Available: https://www.researchgate.net/publication/309462392_A_Survey_of_WebSocket_Development_Techniques_and_Technologies