WebSocket Insecurities Paper

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. These vulnerabilities can lead to sensitive data exposure, unauthorised 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 crawling and then, test attacks are run on the WebSocket endpoints, including tests on origin and authentication checks, protocol fuzzing using different payloads, handshaking, fragmentation, session management, subprotocols, encryption, Denial of Service (DOS), resource management, and cross-origin attacks. By running these tests, we obtain a complete and comprehensive report on all vulnerabilities for a given WebSocket.

Keywords

WebSocket, Vulnerabilities, WS/ WSS, Real-time websites

Introduction

What is a WebSocket?

A WebSocket is a communication protocol that enables persistent, full-duplex communication over a single TCP connection between a client and a server. Standard HTTP websites send a new request each time they want to update content on the page. This is done by clicking a link or button, or refreshing the page. WebSocket enables the server and client to exchange messages instantly, without requiring a connection to be established each time. This makes it perfect for things like live chat apps, online games, stock tickers and live notifications, which require constant communication. WebSocket starts as a regular web request and then upgrades to a faster connection that stays open. But because of this special setup, it bypasses many standard web security checks, and if not done carefully, it can be a problem. It can lead to serious security problems.

Websocket vs HTTP

WebSocket and HTTP are both communication protocols used on the web, but they work differently. HTTP follows a request-response model, where the client must ask for data and then wait for the server to respond, making it suitable for basic websites and APIs. In contrast, WebSocket creates a persistent connection that stays open, allowing both the client and server to send data to each other at any time. This makes WebSockets ideal for real-time applications like chat, gaming, or live notifications. While HTTP is simpler and more secure by default, WebSocket is faster for ongoing communication but requires extra care to secure, as it skips many built-in protections of HTTP.

Websocket vulnerabilities

WebSockets are vulnerable because, unlike HTTP, it does not come with built-in protections like origin checks, CSRF tokens, or strict authentication rules. Once the connection is established, it stays open, which means if an attacker manages to connect, they can continuously send or receive data. Common mistakes like missing origin header validation, accepting connections without proper authentication, or trusting data without sanitising it can lead to serious issues like data theft, session hijacking, or even remote code execution. These risks arise mainly from developers not applying extra security measures, assuming WebSocket is safe by default, when it’s not.

Crawling

What is a crawler

A crawler is a tool that automatically navigates through websites to discover all accessible resources, such as links, API endpoints, and WebSocket connections. In this report, the crawler simulates a real user’s browser using Playwright to load pages, interact with dynamic content, and capture WebSocket URLs. Along with crawling, we use scraping to extract valuable data from responses—like JavaScript, JSON, and HTML—so that we can detect embedded URLs and WebSocket endpoints that are not directly visible on the page. Together, crawling and scraping help build a set of target communication pathways, which are then analysed for vulnerabilities.

How the crawler works

Playwright uses Chromium browser, which is a headless browser(No GUI), to act like a real user. It creates many user agents which are used to visit and analyse a website. It masks and switches between these user agents to avoid bot detectors. Playwright visits the specified URL and analyses the site, understanding its traffic and behaviour. It captures all possible HTTP requests and responses during browsing, including AJAX and WebSocket handshakes. It also parses the APIs to extract hidden URLs and WebSocket addresses using regular expressions. A WebSocket URL is identified as it starts with ws:// or wss://. All the newly discovered paths are added to the crawling queue for further exploration. The crawler eliminates all non-navigable paths, such as images or fonts, and continues crawling till it reaches the domain or depth limit, or crawls through all paths of a website, and finally returns the set of all WebSocket endpoints. To simplify the testing process =, we will use only 5 WebSocket endpoints from each website.

Equations and Logic used in the Crawler:

1. Core Sets

U₀ is the starting URL. Then the crawler analyses the site, and stores all the visited URLs in set C (crawled URLs) & all discovered links in D (discovered URLs). From set D, we extract set W, the set of all WebSocket URLs, which includes only those links that begin with "ws://" or "wss://". This is written as :

               W={u∈D∣u starts with "ws://" or "wss://"}

This set is used for vulnerability testing.

2. Recursive Crawling

The newly discovered URL u is placed into a queue Q for recursive exploration. Two conditions need to be met. The URL u is within the maximum allowed recursion depth and the upper bound on total pages to visit. This can be written as :

∀u∈Q : if (u∈/C)∧(depth(u)≤max\_depth)⇒crawl(u)

3. Filtering Conditions

Let F be the set of file extensions to skip:

               F={.js,.css,.png,.jpg,.gif,.woff,.svg,…}

Then, for any URL u:

               u∈Q⇒crawl(u) iff ext(u)∈/F

This prevents unnecessary crawling.

4. API Scraping

API endpoints return JSON-formatted text. Let Rᵢ be a JSON response body, and extract URLs using regex r:

               r = \texttt{(https?|wss?)://[^\s\"']+}

Then the new set of extracted URLs :

These are added to D and possibly re-queued.

5. Final WebSocket collection

After the crawling ends W={u∈D∣u.scheme∈{ws,wss}}  This set w is passed to the WebSocket attack engine for further testing.

Attacks

1. HTTP and Handshake Tests:

A WebSocket connection is obtained by upgrading the current HTTP/HTTPS connection. The following is the request message to the server:

req = (

  f"GET {path} HTTP/1.1\r\n"

  f"Host: {host}\r\n"

  "Upgrade: websocket\r\n"

  "Connection: Upgrade\r\n"

  f"Sec-WebSocket-Key: {key}\r\n"

  "Sec-WebSocket-Version: 13\r\n"

  "\r\n"

)

The response message from the server is as follows:

HTTP/1.1 101 Switching Protocols

Date: <date><time>

Connection: upgrade

Sec-WebSocket-Accept: <key>

Upgrade: websocket

…

The attacks under this class talk about modifying or omitting some lines from the request message to check if the server will still respond with the acceptance message and allow for the switching of protocols. If it does allow for switching, then it means the server is vulnerable to attacks which bypass protocol rules, spoof handshakes, manipulate the upgrade process or establish unauthorised connections.

2. Payload Handling and Fragmentation Tests:

Once the connection is established, the next class of tests are executed. In these tests, we cover vulnerabilities during data frame transmission, mainly looking at opcodes, control frames, and fragmentation. We try to send illegal frames of different forms and see if the server returns a valid output. This category helps test the server's resilience against inputs that could disrupt communication, crash the server, or cause data leakage.

 3. Authentication and Session Management Tests:

This class assesses whether the server securely handles user identity, token validity, cookie usage, and cross-origin protections. We try to send fake headers containing incorrect information and check whether the server responds with a valid response. These tests check whether the server ensures that sessions cannot be hijacked, reused maliciously, or initiated without proper authorisation.

4. Subprotocol and Extension Handling Tests

Subprotocols and extensions allow WebSocket connections additional functionality beyond basic messaging. This class checks whether the server safely handles such functions without introducing vulnerabilities or errors. Secure handling ensures that only recognised and supported subprotocols or extensions are accepted.

5. Transport Security and Encryption Tests

Transport security ensures that WebSocket communications remain confidential and secure over the network. This class verifies the integrity of the underlying TLS configuration, checks for strong encryption standards, and ensures resistance to protocol downgrades or spoofing. We also evaluate how well the server handles malformed or unexpected protocol interactions. A secure transport layer is vital to defending against man-in-the-middle attacks, unauthorised interception, and ensuring end-to-end trust in real-time applications.

6. DoS and Resource Management Tests

This category focuses on how well the server handles resource allocation and connection management under abnormal or malicious usage patterns. It tests for susceptibility to denial-of-service (DoS) attacks by evaluating how the server deals with excessive connections, message sizes, timeouts, and compression abuse.

A secure WebSocket server must efficiently manage memory, CPU, and bandwidth usage, ensuring that a single client or set of clients cannot exhaust resources and degrade service availability for others. These tests help ensure the server handles edge cases and enforces proper thresholds to maintain scalability and uptime.

7. Cross-Origin and Mixed Content Tests

Cross-origin and mixed content vulnerabilities arise when WebSocket servers fail to enforce strict boundaries between different web origins. This category ensures that browsers and servers respect origin policies, especially when dealing with content embedded in iframes, insecure WebSocket schemes, or scripts communicating across domains.

When improperly configured, origin-based attacks can lead to unauthorised access, data leakage, or browser trust violations. Strong origin validation is crucial to maintaining the same-origin policy and ensuring secure, context-aware communication in browser-based WebSocket applications.

8. Application-Layer Tests

This final category targets weaknesses in the server's error handling and response behaviour, beyond transport or protocol-level concerns. It includes tests that identify information leakage, improper security headers, path traversal attempts, and abuse of query parameters or application routes.

These vulnerabilities often reveal implementation flaws that could assist attackers in bypassing controls or exploiting misconfigured logic. A secure application must sanitise inputs, restrict error disclosures, and protect internal server architecture from exposure through the WebSocket interface.

Analysis and Results

Run the test for the 10 popular websites that make use of WebSockets

Heatmap of 10 websites and all vulnerability categories

2 pie charts: % of vulnerabilities, % of websites(which website ends up getting the most)

Use Case

There are certain industry tools that exist for the analysis of websockets.

1. OWASP Zed Attack Proxy(ZAP)

According to OWASP, the OWASP ZAP tool is one of the world’s most popular free security tools and is actively maintained by hundreds of international volunteers. It can help you automatically find security vulnerabilities in web applications while developing and testing applications. OWASP ZAP provides support for viewing, intercepting and modifying WebSocket messages on the fly and afterwards. OWASP ZAP is written in Java. OWASP provides installation packages for OS X, Windows and Linux with a preliminary requirement that the Java Runtime Environment (JRE) is installed.

1. Burp Suite

Burp Suite is a commercial product developed by PortSwigger for web application security testing. According to PortSwigger, Burp Suite is an integrated platform for performing security testing of web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application's attack surface to finding and exploiting security vulnerabilities. Burp Suite has provided support for viewing, intercepting and modifying WebSocket messages. However, these functionalities are only part of the professional edition.

Our tool is quite simple to use compared to the above-mentioned tools. No prior knowledge is needed to utilise our tool. It requires a couple of Python modules to be installed for successful execution. The tool is compatible with real-world web applications and is compliant with RFC6455. It uses a simple command-line interface for 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 largely 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 provides analysis on each website that is tested. The tool is limited due to its beginner-friendly approach, as it does not provide integration with browser cookies, and also doesn’t dive deep into complex problems such as XSS and CSWH, which are difficult to implement within Python.

Conclusion

References

RFC 6455

<https://datatracker.ietf.org/doc/html/rfc6455>

Types of vulnerabilities:

1️⃣ Handshake & HTTP Request Tests

<https://core.ac.uk/download/pdf/45601062.pdf>

2️⃣ Payload Handling & Fragmentation

<https://www.openmymind.net/WebSocket-Framing-Masking-Fragmentation-and-More/>

3️⃣ Authentication & Session Management

<https://arxiv.org/pdf/2104.05324>

4️⃣ Subprotocol & Extension Handling

[https://www.cyberchief.ai/2025/05/securing-websockets.html](https://www.cyberchief.ai/2025/05/securing-websockets.html?utm_source=chatgpt.com)

5️⃣ Security & Encryption

<https://www.ijnrd.org/papers/IJNRD2407274.pdf>

6️⃣ Denial of Service (DoS) & Resource Management

<https://cqr.company/web-vulnerabilities/denial-of-service-dos-via-websockets/>

8️⃣ Other Server Vulnerabilities

<https://arxiv.org/abs/1409.3367>

**Papers used at the start**

<https://www.researchgate.net/publication/384105299_Review_Analysis_of_Web_Socket_Security_Case_Study>

<https://juerkkil.iki.fi/files/websocket2012.pdf>

<https://oulurepo.oulu.fi/bitstream/handle/10024/6116/nbnfioulu-201603081281.pdf?sequence=1>

<https://www.ijsr.net/getabstract.php?paperid=SUB152856>

<https://www.praetorian.com/blog/meshcentral-cross-site-websocket-hijacking-vulnerability/>

<https://ieeexplore.ieee.org/document/10479302>