**I. INTRODUCTION**

**A. What is WebSocket?**

WebSocket is a full-duplex communication protocol standardized by the IETF as RFC 6455. It operates over a single, long-lived TCP connection, enabling real-time bidirectional communication between a client and a server. Unlike traditional HTTP, where each data exchange requires a new request-response cycle, WebSocket maintains an open channel that allows messages to be sent and received asynchronously at any time. This makes it highly suitable for time-sensitive web applications such as live chat systems, collaborative tools, online gaming platforms, and financial data feeds.

A WebSocket connection begins as a standard HTTP request and then performs a protocol upgrade via the Upgrade and Connection headers. Once established, the channel persists until explicitly closed by either endpoint. While this upgrade mechanism enhances performance and responsiveness, it also circumvents many of the built-in security features of HTTP, thereby introducing new attack surfaces that can be exploited if not properly secured.

**B. WebSocket vs. HTTP**

Though both WebSocket and HTTP operate over TCP and often use the same ports (e.g., 80 or 443), their operational models differ significantly. HTTP adheres to a strict request-response paradigm, wherein the client initiates all interactions and the server responds accordingly. In contrast, WebSocket enables event-driven communication, where either party may initiate message transmission at any point after the connection is established.

HTTP is inherently stateless and benefits from well-established security measures such as CORS (Cross-Origin Resource Sharing), CSRF (Cross-Site Request Forgery) protection, and strong caching and authentication mechanisms. WebSocket, on the other hand, does not enforce these controls by default. Although it offers performance advantages for dynamic applications, its flexibility necessitates that developers implement security best practices manually.

**C. Why is WebSocket Vulnerable?**

WebSockets can become vulnerable when developers neglect to implement rigorous validation and access control mechanisms. The persistent nature of a WebSocket connection implies that once an attacker gains access, they can interact with the server continuously without additional authentication checks. Common misconfigurations include:

* Missing or weak origin header validation,
* Acceptance of unauthenticated or expired sessions,
* Absence of subprotocol negotiation,
* Insufficient message framing and payload inspection.

Such weaknesses can lead to a wide array of attacks, including session hijacking, denial of service (DoS), data exfiltration, unauthorized actions, and even remote code execution in extreme cases. Moreover, because WebSocket traffic may bypass traditional web security layers (e.g., web application firewalls), it becomes harder to detect and mitigate malicious activity post-deployment.

**D. Motivation and Contribution**

Despite the increasing adoption of WebSocket in modern web architectures, security research and tooling around its implementation remain relatively limited compared to HTTP. Existing vulnerability scanners and application firewalls often do not comprehensively assess WebSocket endpoints, leading to potential blind spots in enterprise threat models. Furthermore, many developers assume that once a WebSocket handshake is successful, the channel is inherently trusted—an assumption that attackers can exploit.

This paper presents a systematic vulnerability analysis of publicly accessible WebSocket endpoints. We design and execute a series of targeted tests to identify common implementation flaws across various categories such as handshake validation, origin enforcement, authentication controls, message handling, and protocol misuse. Our results highlight widespread security lapses and underscore the urgent need for better developer awareness and tooling support. The contributions of this work are threefold:

1. We present a structured taxonomy of WebSocket vulnerabilities based on empirical testing.
2. We develop an automated framework capable of scanning and identifying insecure WebSocket behaviors.
3. We provide actionable recommendations to enhance the security posture of WebSocket-based applications.

**II. PROBLEM STATEMENT AND OBJECTIVES**

Modern web applications increasingly rely on WebSocket communication to support real-time features such as live chats, stock updates, multiplayer gaming, and collaborative editing. Unlike traditional HTTP traffic, WebSocket connections are long-lived, stateful, and often dynamically created through client-side JavaScript. Unfortunately, the security posture of WebSocket endpoints is frequently neglected during application development and security auditing.

Most existing vulnerability scanners and web application firewalls (WAFs) focus on HTTP traffic and offer limited or no support for analyzing the WebSocket protocol. As a result, a significant number of security flaws—ranging from broken authentication and origin bypasses to malformed frame handling and denial-of-service vectors—go undetected in production environments.

**Objectives**

The primary goal of this research is to develop an automated, extensible framework that enables robust testing of WebSocket security across real-world websites. The system is designed to:

* Discover dynamically generated WebSocket endpoints using simulated browser behavior.
* Parse embedded scripts, JSON responses, and network requests to extract hidden URLs.
* Perform intelligent crawling and recursive exploration using a headless browser.
* Execute 90+ protocol-compliant attack scenarios drawn from 9 vulnerability classes.
* Automatically classify and report findings using heatmaps, bar charts, and detailed summaries.

By combining browser automation, intelligent crawling, and protocol-aware fuzzing, the proposed solution provides a scalable and precise security analysis method tailored to modern WebSocket infrastructures.

**III. IMPLEMENTATION**

Now, let’s refine your **Implementation section** to be more IEEE-conference appropriate. I’ve rewritten and expanded each subsection for clarity, formality, and technical completeness, while preserving your key ideas.

**A. Web Crawler and Scraper**

The first component of our system is a crawler that programmatically explores a website to discover reachable resources. Unlike traditional crawlers that operate on static HTML, our crawler leverages browser automation to simulate real user interaction. It dynamically renders JavaScript-heavy web content, thereby exposing resources that are otherwise inaccessible through static analysis.

To support comprehensive endpoint discovery, we incorporate a scraping module that extracts useful artifacts from the page and network responses—such as embedded JavaScript, JSON payloads, and inline HTML. These artifacts are parsed using pattern-matching techniques (e.g., regular expressions) to identify candidate WebSocket endpoints that may not be actively used at the time of page load.

By combining crawling and scraping, the system constructs a complete map of potential communication pathways, which are subsequently analyzed for security vulnerabilities.

**B. Browser Automation with Playwright**

To facilitate real-time page analysis, we employ Playwright—an open-source browser automation library developed by Microsoft. Playwright supports Chromium, Firefox, and WebKit, and allows full control over browser behavior through APIs that emulate user interactions, execute JavaScript, and intercept network traffic.

In our implementation, Playwright operates in headless mode to simulate a typical browser session. It monitors all HTTP and WebSocket traffic during browsing, including dynamically generated resources and delayed WebSocket initializations triggered via AJAX or script evaluation.

Playwright also incorporates stealth techniques such as:

* Randomization of user-agent strings,
* Spoofing of browser fingerprinting APIs,
* Disabling of automation detection flags.

These features enable the crawler to bypass bot detection mechanisms deployed by many modern websites, thereby ensuring uninterrupted endpoint discovery.

Through Playwright, the system can:

* Load complex web applications,
* Interact with dynamic elements,
* Observe WebSocket handshake requests,
* Capture hidden or deferred WebSocket URLs in real-time.

**C. Workflow Overview**

The endpoint discovery process proceeds as follows:

1. **Initialization:** The system launches a headless Chromium browser using Playwright and generates a set of randomized user agents for stealth.
2. **Navigation and Monitoring:** For each given target URL, the system loads the page and listens to all outgoing and incoming network traffic, including AJAX calls and WebSocket connections.
3. **Pattern Extraction:** WebSocket endpoints are extracted from observed traffic, as well as from parsed API responses and JavaScript files using regular expressions. WebSocket URLs are identified by the presence of ws:// or wss:// schemes.
4. **URL Filtering and Queuing:** Discovered WebSocket endpoints are validated and non-navigable paths (e.g., fonts, images) are discarded. Remaining paths are enqueued for further exploration within domain and depth limits.
5. **Termination Criteria:** The crawler halts when all paths have been explored, or when domain-specific thresholds (e.g., depth or request limits) are reached.
6. **Output Set:** The system returns a refined set of WebSocket URLs for each domain. For downstream testing, we limit this to a maximum of **three representative endpoints per website**, selected based on activity and uniqueness.

This pipeline ensures that dynamically generated, delayed, or hidden WebSocket endpoints—often missed by traditional crawlers—are discovered and captured for security analysis.

**IV. VULNERABILITY CLASSIFICATION AND TESTING**

To rigorously assess the security of discovered WebSocket endpoints, we implemented a structured suite of **90 protocol-compliant attack tests**, developed in accordance with **RFC 6455**, the official WebSocket specification. These test cases are classified into **nine vulnerability categories**, each corresponding to a different phase of WebSocket interaction or protocol handling.

Each class contains a set of carefully designed test cases that simulate attack conditions—such as malformed handshake headers, expired tokens, spoofed origins, fragmented payloads, and invalid opcodes—to evaluate how resilient the server is to misbehavior. These tests examine whether the server:

* Adheres strictly to the WebSocket protocol,
* Gracefully handles unexpected input,
* Rejects suspicious payloads,
* Or exhibits exploitable behavior like memory errors, crashes, or unintended access.

**Test Coverage Overview**

**Fig. 1 – Test Type Distribution**  
As illustrated in Fig. 1, a significant number of test cases are concentrated in the **Handshake (22 tests)** and **Payload Framing (18 tests)** categories, reflecting the complexity and sensitivity of the connection initiation and data exchange phases. **Protocol Fuzzing (12 tests)** and **DoS Simulations (9 tests)** also receive considerable emphasis, aimed at detecting stability and resilience issues at scale.

This classification ensures that our tool offers **broad, layered coverage**—from protocol compliance and security posture to robustness under adversarial conditions.

**A. Handshake and HTTP Upgrade Tests**

The WebSocket protocol initiates with a standard **HTTP/1.1 upgrade handshake**, where the client requests a protocol switch and the server responds with confirmation. This process is critical, as it defines the boundary between stateless HTTP and persistent full-duplex WebSocket communication.

In our project, we evaluated **22 distinct handshake-related vulnerabilities (Attacks #1–22)**. Each test manipulates the handshake request by altering or omitting key fields to assess whether the server strictly follows the WebSocket upgrade specifications defined in **RFC 6455**.

A standard handshake includes headers such as:

GET /chat HTTP/1.1

Host: example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==

Sec-WebSocket-Version: 13

The expected response from a compliant server is:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=

Our tests include:

* Invalid Port, Non-WS Scheme, and HTTP/0.9 Handshake to evaluate how servers handle protocol-level violations.
* Header manipulation like Omit Sec-WebSocket-Key, Duplicate Sec-WebSocket-Key, Wrong Upgrade Header, Missing Host Header, and Fake Host Header to check if servers are vulnerable to downgrade, spoofing, or misrouting.
* Encoding anomalies such as Unicode URL, Long URL Path, and Case-Sensitive Headers to assess robustness of URL parsing and header normalization.

Acceptance of non-compliant handshake requests signals possible **downgrade attacks**, **improper header validation**, or **insecure fallback mechanisms**, all of which compromise protocol negotiation integrity. These tests are foundational to ensuring that the server enters WebSocket mode **only under strict conformance** to protocol expectations.

**B. Authentication and Session Management Tests**

This category comprises **7 targeted tests (Attacks #23–29)** that evaluate how securely a server handles **user identity, session validation**, and **token-based access control** over persistent WebSocket connections.

Unlike stateless HTTP, where authentication is enforced per request, WebSocket relies on a one-time handshake followed by continuous communication, creating risks if access control checks are bypassed at connection time or during prolonged sessions.

The tests in this group simulate real-world scenarios, including:

* No Session Cookie and Missing Authentication: Attempting unauthenticated connections to evaluate whether the server enforces login or session validation before upgrade.
* 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 revalidating the origin or token freshness.
* Cross-Site Cookie Hijack: Initiating connections from cross-origin contexts to exploit improperly scoped or unsecured session cookies.

Successful exploitation of these flaws could allow:

* **Unauthorized access** to private resources,
* **Session hijacking** and impersonation,
* **Privilege escalation** across user roles or tenants.

These tests are essential in determining whether the server performs **strict per-connection validation** and maintains isolation between user sessions, origins, and scopes.

**C. Subprotocol and Extension Handling Tests**

WebSocket allows clients to request **subprotocols** (like MQTT or STOMP) and **extensions** (e.g., permessage-deflate) that modify communication semantics or compression behavior. This category includes **5 tests (Attacks #30–34)** targeting improper negotiation, injection, or fallback behavior in subprotocol and extension handling.

Test cases include:

* 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, unvalidated negotiation, or resource overcommitment.

These attacks reveal if the server is vulnerable to:

* **Protocol downgrade attacks**, where the attacker forces fallback to a weak or null protocol,
* **Unexpected code paths**, due to unvalidated extension behavior,
* **Buffer mismanagement**, arising from partial or failed negotiations.

Proper implementation should strictly validate all protocol and extension declarations and reject connections that fail negotiation.

**D. Transport Security and Encryption Tests**

To ensure confidentiality and integrity, secure WebSocket connections (wss://) must be established over **TLS**, configured with strong cryptographic primitives and downgrade resistance. This class includes **5 tests (Attacks #35–39)** that assess whether the server enforces robust transport-layer security.

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 behavior.

Security flaws in this layer could expose the application to:

* **Man-in-the-middle (MITM) attacks**, especially on public networks,
* **Session hijacking**, via interception or manipulation of traffic,
* **Replay attacks**, if encryption is improperly implemented or disabled.

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

**E. Payload Handling and Fragmentation Tests**

Payload framing is central to the WebSocket protocol, as all messages are transmitted as **data frames** structured per the format shown in **Fig. 2**. This class comprises **18 test cases (Attacks #40–57)** designed to validate whether the server enforces correct frame parsing and rejects malformed, fragmented, or improperly masked payloads.

Key test scenarios include:

* **Opcode Validation**:
  + Undefined Opcode, Reserved Opcode: Sending frames with illegal or undefined opcodes.
  + Binary as Text, Text as Binary: Swapping data types to induce interpretation errors.
* **Framing and Fragmentation**:
  + Zero-Length Fragment, Invalid Payload Length, Negative Payload Length, Mismatched Payload, Oversized Control Frame: Testing adherence to fragmentation rules, max size limits, and internal state tracking.
  + Early Close Frame, No Close Frame, Long Close Reason, Invalid Close Code: Assessing how servers handle connection teardown and state finalization.
* **Masking Violations**:
  + Invalid Masking Key, Unmasked Client Frame: Attempting to bypass the masking requirement on client-to-server frames.
* **Encoding and Control Frames**:
  + Non-UTF-8 Text, Null Bytes in Text: Verifying encoding validation and rejection of non-standard characters in textual payloads.
  + Invalid RSV Bits: Setting reserved bits to detect whether servers incorrectly support undocumented features or extensions.

These tests help detect:

* **Parsing vulnerabilities** that could lead to DoS, memory corruption, or logic bypass,
* **Desynchronization** due to improper fragmentation or interleaved frames,
* **Security bypasses**, when masking or encoding rules are not strictly enforced.

Correct behavior requires that the server immediately terminate connections on encountering malformed or non-compliant frames and log these attempts for further analysis.

**F. Cross-Origin and Mixed Content Tests**

Cross-origin vulnerabilities are especially dangerous in browser-based WebSocket applications, where **same-origin policy** is a critical security control. This category includes **6 targeted tests (Attacks #58–63)** that simulate cross-origin contexts, mixed security levels, and origin spoofing.

Examples include:

* Missing CORS Headers: Verifying if the server discloses sensitive content to unauthorized origins.
* 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.

Failure to enforce origin validation can lead to:

* **Cross-Site WebSocket Hijacking (CSWSH)**,
* **Data leakage** to unauthorized domains,
* **Clickjacking and iframe abuse**,
* **Browser trust violations** in HTTPS contexts.

Our framework verifies whether WebSocket endpoints appropriately inspect the Origin and Referer headers during the handshake phase and deny any cross-origin attempts not explicitly whitelisted.

**G. Application-Layer Logic and Misconfiguration Tests**

This category shifts focus from protocol-level flaws to **application logic vulnerabilities** and misconfigured server behaviors. We developed **6 attack cases (Attacks #64–69)** to uncover poor sanitization, insecure headers, and overexposed API design patterns.

Key tests include:

* 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.

These vulnerabilities often originate from **lack of input validation**, **overly verbose error reporting**, or **exposed internal routing logic**. A secure WebSocket implementation must ensure **fail-safe error handling** and **sanitized responses**, especially when protocol upgrades bridge the client directly to backend logic.

**H. DoS and Resource Management Tests**

WebSocket servers must remain robust against malicious attempts to exhaust resources. Our framework includes **9 focused tests (Attacks #70–78)** that evaluate the server's **resilience to denial-of-service (DoS)** conditions and resource mismanagement.

Scenarios 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 hasn't agreed to it.

These tests mimic real-world DoS vectors such as **slowloris attacks**, **amplification via compression bombs**, and **connection pool exhaustion**. Servers that fail to disconnect idle or non-compliant clients risk degraded performance and full service outages.

**I. Protocol Fuzzing Tests**

Protocol fuzzing is an advanced technique used to detect **zero-day vulnerabilities**, **memory safety issues**, and **unexpected behaviors** by sending syntactically or semantically malformed WebSocket frames. Our suite includes **12 custom fuzzing tests (Attacks #79–90)**, which explore both control-plane and application-layer attack surfaces.

The payloads used span various malicious inputs:

| **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; ls |
| 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: 🚀🌟💥 |
| Oversized DoS Message | 2 million-character JSON message |
| Path Traversal Simulation | {"path": "/../../etc/passwd"} |
| PostMessage Abuse | Script injection into window.postMessage() |

These tests are useful for uncovering:

* **Parser crashes**, segmentation faults, or unhandled exceptions,
* **Business logic bypasses** when input validation fails,
* **Encoding or escaping flaws** that result in XSS, command injection, or data leakage.

All fuzzing inputs comply with WebSocket framing rules, ensuring that crashes are due to payload logic rather than protocol format violations. The server's **resilience to malformed inputs** is a strong indicator of its readiness for production use.

**IV. Equations and Workflow Logic**

This section formalizes the internal logic and key components of our WebSocket vulnerability analysis framework. The system performs crawling, scraping, attack execution, and result aggregation through a series of well-defined sets, functions, and control structures.

**A. Core Sets**

Let the initial seed URL be denoted by:

U0=Starting URL

As the crawler navigates the target site, it populates the following sets:

* C: Set of all successfully **crawled URLs**
* D: Set of all **discovered URLs**, which includes both visited and queued URLs
* W: Set of all **WebSocket URLs** found in D

WebSocket endpoints are identified as:

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

This set W is passed to the vulnerability engine for dynamic testing.

**B. API Scraping Logic**

During crawling, API endpoints that return **JSON responses** are parsed to extract embedded URLs. Let Ri​ be the response body of the ith API call.

A regular expression is used to extract potential HTTP or WebSocket URLs from within Ri​:

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

The resulting URLs are added to D and optionally requeued for crawling, if they pass the filtering conditions.

**C. Recursive Crawling Conditions**

Let Q be the **crawling queue**. For each URL u∈ Q, define:

* depth(u): depth of URL u from seed U0​
* max\_depth: maximum allowed depth
* max\_requests: global upper bound on crawl requests

The recursive crawl condition becomes:

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

**D. Filtering Conditions**

Let F denote the set of file extensions to be ignored during crawling:

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

Let ext(u) denote the extension of URL u. Then, the filtering logic is:

u∈Q⇒crawl(u)⟺ext(u)∈/F

This ensures efficient crawling by avoiding static or non-navigational assets.

**E. Final WebSocket Endpoint Collection**

After crawling completes, the finalized WebSocket set W is redefined as:

W={u∈D∣scheme(u)∈{"ws","wss"}}

This set W becomes the input to the **WebSocket vulnerability engine**.

**F. WebSocket Vulnerability Testing Logic**

Each WebSocket endpoint w∈W is tested using a predefined set of test functions T. Each test function t∈T returns either a vulnerability report or a "safe" verdict. The cumulative set of all detected vulnerabilities is:

V=(w∈W)⋃(​t∈T)⋃​t(w)

Where:

* W: WebSocket endpoint set
* T: Test function set
* V: Set of vulnerability reports v∈V

Each function t(w) performs a specific attack vector (e.g., header manipulation, fuzzed payloads, origin bypass) and logs the result.

**G. Workflow Controller Logic**

The system’s controller manages the full workflow—from URL collection to final result generation. For each scanned site u∈U, we define the scan result structure:

R[u]={Cu​,Wu​,Vu​,Tu​}

Where:

* Cu​: Crawled URLs for site u
* Wu​: WebSocket URLs found
* Vu​: Vulnerabilities detected
* Tu​: Total scan duration

**VI. Results and Analysis**

This project demonstrates the effectiveness of our automated framework in identifying vulnerabilities in real-world WebSocket implementations. After crawling target domains and extracting live WebSocket endpoints, our engine executed **90 vulnerability tests**—including protocol validation, session management flaws, improper subprotocol handling, transport security weaknesses, and application-layer misconfigurations.

Each endpoint was subjected to **targeted payloads**—ranging from malformed handshake headers and expired authentication tokens to oversized binary messages and protocol fuzzing mutations. The results were then visualized using heatmaps and bar charts to highlight the security posture of each site and identify common classes of vulnerabilities.

**A. WebSocket Vulnerability Heatmap**

**Fig. 3** presents a **WebSocket vs. Attack Heatmap** that showcases the vulnerability exposure of each scanned domain across all test categories. Each **row represents a domain**, while each **column corresponds to a specific attack** from our test suite (totaling 90).

**Insights:**

* Domains such as pixivsketch.com, tradingview.com, and discord.com showed consistent exposure to **high and medium-risk vulnerabilities**, especially in **payload fragmentation**, **authentication tests**, and **origin validation**.
* Some sites (e.g., coincap.io, binance.com) had a higher percentage of 🟩 (low-risk or fully hardened) blocks, suggesting mature WebSocket implementations.
* The **horizontal red bands** indicate systemic vulnerabilities, i.e., a particular test that failed across many sites—these include "Missing Origin Check", "Unmasked Client Frame", and certain "Fuzzing" tests.
* The heatmap serves as a powerful comparative tool—enabling rapid identification of **weakest sites**, **test categories with widespread failures**, and the **variability of WebSocket security maturity** across real deployments.

**B. Vulnerability Distribution by Type**

**Fig. 4** provides a **quantitative overview** of vulnerabilities grouped by their respective test categories. The bar chart categorizes all 90 attacks into 9 classes and shows the **total number of vulnerabilities detected per class** across all endpoints.

**Key Observations:**

* The **Payload Framing & Messaging Semantics** category leads by a wide margin, indicating a lack of robust validation of WebSocket frame structures. This includes acceptance of malformed opcodes, oversized control frames, or unmasked payloads, which are often overlooked in production deployments.
* **Protocol Fuzzing** ranked second, validating the importance of sending unpredictable or malformed inputs to uncover undocumented crashes, misbehaviors, or parsing bugs.
* **Handshake and Upgrade Failures** (131 detections) confirm that many servers are lenient toward malformed or incomplete handshake headers, such as missing Upgrade or Sec-WebSocket-Key headers—violating RFC 6455 compliance.
* Notably, **Authentication & Session Control** accounted for 107 findings, exposing weak token handling, session reuse flaws, or missing authentication requirements during upgrade.
* **Subprotocol Handling**, **Transport Security**, and **Application Logic** showed lower—but still concerning—amounts of vulnerability. These may often be masked by custom application logic rather than explicit protocol handling, but their exploitation can still lead to privilege escalation or data leakage.
* **DoS-related attacks** (127 detections) confirm the danger of uncontrolled memory, timeout, or message buffering issues under stress testing.

**C. Severity Distribution & Risk Posture**

The visual breakdown of findings reveals that:

* A **significant majority** of issues fall under the **high-severity category**, especially those related to unauthenticated access, missing origin checks, and malformed input acceptance.
* The **medium-severity category** captures less exploitable—but still impactful—bugs such as improper compression negotiation or error message leaks.
* **Low-risk issues**, though not immediately exploitable, suggest the presence of outdated or non-strict implementations and highlight areas where standards compliance is lax.

**D. Summary**

These results reinforce several important insights:

* **WebSocket security is often an afterthought**—developers rely on initial handshake validation but ignore frame-level robustness, origin control, and proper cleanup mechanisms.
* The presence of **high-severity issues across even popular domains** like discord.com and tradingview.com underscores the **real-world relevance** of our scanner and the need for automated analysis.
* Visualizations such as the heatmap and bar chart are **effective tools for risk triaging**, helping organizations prioritize patching efforts by attack class or domain.

**VII. Use Case: Comparison with Industry Tools**

While several established tools exist for analyzing WebSocket communication, most are either targeted at professional pentesters or require significant manual configuration. Below, we compare two widely used solutions and contextualize our tool in terms of usability, scope, and accessibility.

**1. OWASP Zed Attack Proxy (ZAP)**

OWASP ZAP is one of the most popular open-source security tools, maintained by a global community of contributors. It is widely used for dynamic application security testing (DAST), offering features such as automated scanning, passive analysis, and intercepting proxies. ZAP includes support for **intercepting, viewing, and modifying WebSocket traffic**, allowing testers to manually analyze protocol messages. However, ZAP is primarily GUI-based and requires a moderate learning curve to configure for WebSocket-specific workflows.

ZAP is written in Java and is platform-independent, with installers available for Windows, macOS, and Linux (provided the Java Runtime Environment is available). While powerful, its WebSocket testing features are less automated and may require scripting or extension development for deeper protocol-level testing.

**2. Burp Suite**

Burp Suite, developed by PortSwigger, is a commercial-grade security testing platform used by penetration testers and security professionals. It offers an integrated suite of tools for **reconnaissance, scanning, manipulation, and exploitation** of web applications. Burp Suite supports detailed WebSocket analysis, including message interception, replay, and tampering. However, these features are limited to the **Professional Edition**, which is a paid version.

Burp is exceptionally powerful for deep, manual exploration and is ideal for expert users, but it lacks the plug-and-play nature that casual users or automation-focused researchers might prefer.

**3. Our Tool: Lightweight and Fully Automated**

Our WebSocket vulnerability analysis tool is designed with **automation, accessibility, and customizability** in mind. Unlike ZAP or Burp, which require domain expertise or paid licenses, our tool:

* Requires **no prior security knowledge** to operate.
* Is implemented in **Python** and runs via a simple **command-line interface**.
* Accepts **single website URLs or CSV lists** for batch testing.
* Performs **90+ protocol-compliant tests** aligned with **RFC6455**, covering a broad spectrum of vulnerabilities.
* Uses **Playwright** to dynamically crawl JavaScript-heavy websites and discover WebSocket endpoints generated at runtime.
* Generates easy-to-understand **PDF reports** featuring vulnerability heatmaps, bar graphs, and detailed summaries per target.
* Is suitable for educational use, security research, CI pipelines, and pre-production testing.

While the tool intentionally sacrifices deep integration with browser sessions (e.g., cookies, localStorage) and advanced payload crafting (e.g., full XSS/CSWH simulations), this trade-off enables a **lightweight and user-friendly solution** ideal for rapid scanning and protocol-level validation of WebSocket services.

**VIII. Conclusion**

This work presents a comprehensive and automated framework for detecting security vulnerabilities in WebSocket implementations across real-world websites. By leveraging **headless browser automation (Playwright)**, the system is capable of crawling dynamic, JavaScript-rich pages to uncover **active WebSocket endpoints**—including those embedded deep within API responses or runtime-generated JavaScript.

Upon discovering endpoints, the tool executes a **diverse set of 90 targeted tests** grouped into nine vulnerability classes. These tests cover:

* **Handshake manipulation**
* **Authentication bypass**
* **Subprotocol/extension misuse**
* **Transport security**
* **Payload structure violations**
* **Cross-origin policy enforcement**
* **Application-level logic errors**
* **DoS scenarios**
* **Protocol fuzzing**

Crafted payloads simulate a variety of attack vectors, such as **invalid opcodes, expired tokens, unmasked frames, oversized messages**, and **malicious JSON inputs**. Each test observes how the server responds—whether it accepts, rejects, ignores, or crashes—offering insight into the **robustness of the endpoint’s security controls**.

The findings are compiled into a structured PDF report, visualized using:

* A **heatmap** showing per-domain vulnerabilities across the attack set.
* A **bar chart** representing the distribution of vulnerabilities by type.
* A **pie chart** categorizing attack classes, providing clear visibility into the protocol’s attack surface.

Our results highlight widespread gaps in WebSocket security, particularly around **payload framing**, **fuzzing resistance**, and **origin enforcement**. Even well-known domains were observed to be vulnerable to **basic misconfigurations** or **non-compliance with RFC standards**.

In summary, the tool offers:

* A **scalable, reproducible**, and **automated method** to evaluate WebSocket security.
* Valuable insights for **developers, security teams**, and **researchers** seeking to build safer real-time applications.
* A foundation for future enhancements, such as support for browser session integration, custom attack scripting, and CI/CD pipeline deployment.

By bridging the gap between accessibility and depth, this project contributes a **practical solution** for securing the often-overlooked WebSocket layer of modern web infrastructure.

I. INTRODUCTION.  
A. What's WebSocket?  
IETF has standardized WebSocket, a full-duplex communication protocol, as RFC 6455.2.15. The system functions on a single, long-lasting TCP connection, enabling real-time bidirectional communication between symantic clients and servers. Instead of requiring another request-responses cycle for every data exchange, WebSocket has an open channel that allows messages to be sent and received at any time without delay. Hence its effectiveness for time-sensitive web applications, such as live chat systems, collaborative tools, online gaming platforms and financial data feeds.  
The beginning of a WebSocket connection is as normal HTTP request, and then it upgrades to its own protocol with the Upgrade and Connection headers. Once the channel is established, it continues until both ends explicitly close the end. i.e. The upgraded method improves HTTP' underlying security features, but it also introduces new attack vector that can be exploited without proper security measures.  
B. WebSocket vs. HTTP.  
Although WebSocket and HTTP use TCP and share common ports (e.g, 80 or 443), their operational models differ significantly. HTTP is based on strict request-response, with all interactions starting from the client and ending at the server. WebSocket, on the other hand, allows for event-based communication, allowing either party to send messages at any time after the connection is established.  
HTTP is a stateless protocol that benefits from its well-established security features, including CORS, CSRF, and robust caching and authentication mechanisms. WebSocket does not have these controls as a default. Although dynamic applications benefit from its performance, its flexibility means developers must manually implement security best practices.  
C. Why is WebSocket Vulnerable?  
The absence of rigorous validation and access control mechanisms can compromise the security of WebSockets for developers. Because a WebSocket connection is always connected, an attacker can use it to gain entry to the server and communicate with it without any further authentication. Common misconfigurations include:  
A lack of validation by an origin header or its absence,  
The acceptance of unauthenticated or expired sessions.  
• Absence of subprotocol negotiation,  
Inadequate message framing and payload inspection.  
Such vulnerabilities can lead to a variety of attacks, including session hijacking, denial of service (DoS), data exfiltration, unauthorized actions, and even remote code execution in extreme cases. Additionally, WebSocket traffic can bypass conventional web security layers like firewalls, making it more challenging to detect and mitigate malicious activity after deployment.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
D. Motivation and Contribution.  
Despite the increasing use of WebSocket in modern web architectures, security research and tooling around its implementation are still limited to HTTP. Why is this so? Insufficient evaluation of WebSocket endpoints by current vulnerability scanner/app firewall models often leads to the creation of blind spots in enterprise threat modeling. In addition, many developers assume that the channel is dependable after a successful WebSocket handshake to prevent possible abuse by attackers. This assumption is also true.  
The paper presents a systematic vulnerability analysis of publicly accessible WebSocket endpoints.' Specifically, we design and execute targeted tests that identify common implementation flaws across various categories including: handshake validation, origin enforcement; authentication controls; message handling; and protocol misuse. The outcomes highlight prevalent vulnerabilities and emphasize the necessity for heightened developer awareness and tooling support. It has contributed threefold to this work:.  
1. We provide a systematic classification of WebSocket vulnerabilities through empirical research.\_  
2. Our framework is designed to automatically detect and flag insecure WebSocket operations.  
3. Our approach involves making practical suggestions to enhance the security of WebSocket-based applications.  
II. PROBLEM STATEMENT AND OBJECTIVES.  
Real-time capabilities like live chats, stock updates, multiplayer gaming, and collaborative editing are made possible by WebSocket communication in modern web applications. WebSocket connections are not like traditional HTTP traffic as 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/or security auditing.  
The majority of present vulnerability detectors and web application firewalls (WAFs) concentrate on HTTP traffic and lack significant support for analyzing the WebSocket protocol. Consequently, numerous security vulnerabilities, including broken authentication and origin bypasses, as well as mishandled frame handling and denial-of-service vectors are not discovered in production settings. This is an unfortunate reality.  
Objectives.  
This research aims to develop an automated, cross-platform platform for testing WebSocket security with great precision on real websites. The system is designed to:  
Create WebSocket endpoints that function dynamically by simulating the browser's behavior.  
Identify hidden URLs by parsing embedded scripts, JSON responses, and network requests.  
Utilize a browser-free environment and engage in intelligent crawling with recursive browsing.  
Identify and implement 90+ protocol-compliant attack scenarios from 9 vulnerability classes.  
Use heatmaps, bar charts, and detailed summaries to automatically sort and report results.;  
Combining automation of the browser, intelligent crawling and protocol aware fuzzing provides an adaptable and scaled-up approach to secure management of modern WebSocket infrastructures.[B].  
III. IMPLEMENTATION.  
We can now enhance our Implementation section to be more suitable for IEEE conferences. To maintain your core concepts, I've rewritten and expanded each subsection for clarity, formality or technical intricacy.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
A. Web Crawler and Scraper.  
We have a component in our system that crawls the website and uses it to find resources that are accessible through programming. It is the first part of this system. We use our crawler's browser automation to simulate real user interaction, rather than relying on static HTML. The web content is rendered dynamically, allowing for the discovery of resources that would otherwise be unavailable through static analysis.  
We add a scraping module that extracts useful artifacts from the page and network responses, such as embedded JavaScript or JSON payloads; inline HTML, and other similar data to support comprehensive endpoint discovery. Through the use of pattern-matching techniques, such as regular expressions, candidate WebSocket endpoint matching is filtered for potential non-use during page load parsing.  
By combining crawling and scraping, the system creates a comprehensive map of possible communication channels that are subjected to security analysis.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
B. Browser Automation with Playwright.  
Microsoft's Playwright, an open-source browser automation library, is utilized by us for real-time page analysis. Through APIs that emulate user interactions, JavaScript programming (such as Google Translate), and network traffic intercepting, Playwright provides full control over browser behavior through Chromium.  
Our implementation of Playwright uses headless mode to simulate a typical browser session. A. All HTTP and WebSocket traffic, including dynamically generated resources and delayed WebSocket initializations triggered by AJAX or script evaluation, are 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 and continue to provide complete endpoint discovery.  
Through Playwright, the system can:  
• Load complex web applications,  
• Interact with dynamic elements,  
• Observe WebSocket handshake requests,  
Take real-time WebSocket URLs that are hidden or deferred.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
C. Workflow Overview.  
The endpoint discovery process is characterized as follows:  
1. Set up: The system initiates a headless 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, such as AJAX calls and WebSocket connections.  
3. JavaScript files, API responses, and observed traffic are used to extract patterns from WebSocket endpoints through regular expressions. Ws:// or a similar scheme known as 'wos@' is used to identify WebSocket URLs.  
4. The validation process involves validating discovered WebSocket endpoints and discarding paths that are not navigable, such as fonts or images. The remaining routes are redirected for further exploration within the domain and at depth.  
5. The crawler stops when all paths have been explored or when domain-specific limits are met, such as depth or request limits.  
6. A more refined set of WebSocket URLs for every domain is returned by the system.. We choose a maximum of three representative endpoints per website for downstream 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 for security purposes.  
IV. VULNERABILITY CLASSIFICATION AND TESTING.  
Our approach involved executing 90 protocol-compliant attack tests to rigorously test the security of discovered WebSocket endpoints, as per RFC 6455, the official specification for Web socketing. Nine different types of vulnerability are present in these test cases, each associated with a unique phase of WebSocket interaction or protocol handling.  
To assess the server's ability to withstand attack conditions such as invalid opcodes, fragmented payload types, spoofed origins and malformed handshake headers on an encrypted keypad, and expired tokens that are not present in all attacks is provided in each class' carefully designed test cases. The server's performance is evaluated by these tests:  
Ignores the strict WebSocket protocol.  
• Gracefully handles unexpected input,  
• Rejects suspicious payloads,  
Has exploitable behavior such as memory errors, crashes or unintended access.....  
Test Coverage Overview.  
Fig. 1. As illustrated in Fig. 1, A significant amount of attention is given to Protocol Fuzzing (12 tests) and DoS Simulations (9 test), which are designed to detect stability and resilience issues at large scale.  
The classification system ensures that our tool covers a wide range of aspects, including protocol compliance, security posture, and adversarial resistance.  
A. Handshake and HTTP Upgrade Tests.  
When the WebSocket protocol is activated, it begins with a standard handshake to update HTTP/1.1, where the client requests and then accepts the request. It is of utmost importance as it establishes the boundary between stateless HTTP and persistent full-duplex WebSocket communication.  
In our project we tested 22 different vulnerabilities associated with handshakes (Attitudes #1–22). By modifying or eliminating crucial fields in the handshake request, each test endeavors to establish whether the server adheres strictly to the WebSocket upgrade specifications outlined in RFC 6455.  
A typical handshake may involve headers like:  
GET /chat HTTP/1.1.  
Host: example.com.  
Upgrade: websocket.  
Connection: Upgrade.  
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==  
Sec-WebSocket-Version: 13.  
A dependable server will probably respond as follows:  
HTTP/1.1. 101. Switching Protocols.  
Upgrade: websocket.  
Connection: Upgrade.  
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=  
Our tests include:  
An invalid port caused by Non-WS scheme, and HTTP/0.9.  
Handshake utilized to evaluate the way servers handle protocol-level transgressions.  
To determine if servers are susceptible to downgrade, spoofing, or misrouting, one can manipulate headers like Omit Sec-WebSocket-Key, Duplicate Sec–WebSocket­Key; Wrong Upgrade Header; Missing Host Headers; and Fake Host under Headers.  
Enumerating 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 can lead to potential downgrade attacks, improper header validation (assignment of contract terms), or insecure fallback mechanisms that undermine the integrity of protocol negotiation.

B. Authentication and Session Management Tests This category includes seven targeted tests (Attacks #23-29) that assess how securely a server handles user identity, session validation, and token-based access control over persistent WebSocket connections. Unlike stateless HTTP, where authentication applies to each request, WebSocket uses a one-time handshake followed by ongoing communication. This setup creates risks if access control checks are ignored during the connection or for extended sessions. The tests in this group mimic real-world scenarios, such as: • No Session Cookie and Missing Authentication: Attempting unauthenticated connections to see if the server enforces login or session validation before upgrading. • Expired Cookie, Fake Token, and Stale Session Reconnect: Using outdated or forged credentials to test how well it resists token replay and session fixation attacks. • HTTP Session Reuse: Reusing an HTTP-authenticated session without revalidating the origin or token freshness. • Cross-Site Cookie Hijack: Starting connections from cross-origin contexts to exploit improperly scoped or unsecured session cookies. Exploiting these weaknesses could lead to: • Unauthorized access to private resources, • Session hijacking and impersonation, • Privilege escalation across user roles or tenants. These tests are critical for determining if the server performs strict per-connection validation and keeps user sessions, origins, and scopes separate. C. Subprotocol and Extension Handling Tests WebSocket allows clients to request subprotocols (like MQTT or STOMP) and extensions (like permessage-deflate) that change communication behavior or compression. This category includes five tests (Attacks #30-34) aimed at improper negotiation, injection, or fallback behavior in subprotocol and extension handling. Test cases involve: • Invalid Subprotocol and Unaccepted Subprotocol: Requesting unsupported or arbitrary protocol strings to check if the server fails securely or defaults to an insecure option. • Conflicting Subprotocols: Sending multiple mutually exclusive protocols to test negotiation logic. • Fake Extension and Conflicting Extensions: Declaring bogus or contradictory extensions to reveal parser confusion, unvalidated negotiation, or resource overcommitment. These attacks show if the server is exposed to: • Protocol downgrade attacks, where the attacker forces a fallback to a weak or null protocol, • Unexpected code paths due to unvalidated extension behavior, • Buffer mismanagement from partial or failed negotiations. Proper implementation must strictly validate all protocol and extension requests and reject connections that do not pass negotiation. D. Transport Security and Encryption Tests To guarantee confidentiality and integrity, secure WebSocket connections (wss://) must use TLS, set up with strong cryptographic methods and resistance to downgrading. This category consists of five tests (Attacks #35-39) to evaluate if the server enforces solid transport-layer security. The following vectors are examined: • TLS Downgrade, HTTP/1.0 Downgrade, and Spoofed Connection Header: Trying to create connections with lower protocol versions or bad upgrade headers to simulate downgrade attempts. • Weak TLS Ciphers: Forcing the use of insecure or out-of-date cipher suites during TLS negotiations. • Certificate Mismatch: Initiating wss:// connections with invalid, mismatched, or self-signed certificates to test how well the server validates certificates and the behavior of clients. Security flaws in this area could expose the application to: • Man-in-the-middle (MITM) attacks, especially on public networks, • Session hijacking through interception or manipulation of traffic, • Replay attacks if encryption is incorrectly implemented or turned off. Well-configured servers must reject any deviations from strong TLS settings, prevent plaintext upgrades from HTTP/1.0, and fail the connection when they find certificate or header problems. E. Payload Handling and Fragmentation Tests Payload framing is central to the WebSocket protocol, as all messages are sent as data frames structured according to the format outlined in Fig. 2. This category includes 18 test cases (Attacks #40-57) designed to check if the server enforces correct frame parsing and rejects malformed, fragmented, or improperly masked payloads. Key test scenarios include: • Opcode Validation: ◦ Undefined Opcode, Reserved Opcode: Sending frames with illegal or undefined opcodes. ◦ Binary as Text, Text as Binary: Swapping data types to cause interpretation errors. • Framing and Fragmentation: ◦ Zero-Length Fragment, Invalid Payload Length, Negative Payload Length, Mismatched Payload, Oversized Control Frame: Testing for adherence to fragmentation rules, maximum size limits, and internal state tracking. ◦ Early Close Frame, No Close Frame, Long Close Reason, Invalid Close Code: Evaluating how servers manage connection teardown and final state. • Masking Violations: ◦ Invalid Masking Key, Unmasked Client Frame: Trying to bypass the masking requirement on client-to-server frames. • Encoding and Control Frames: ◦ Non-UTF-8 Text, Null Bytes in Text: Verifying encoding validation and rejection of non-standard characters in textual payloads. ◦ Invalid RSV Bits: Setting reserved bits to see if servers incorrectly support undocumented features or extensions. These tests help find: • Parsing vulnerabilities that could lead to denial of service, memory corruption, or logic bypass, • Desynchronization due to improper fragmentation or interleaved frames, • Security bypasses when masking or encoding rules are not enforced. Correct behavior requires that the server immediately terminate connections when it encounters malformed or non-compliant frames and log these attempts for further analysis. F. Cross-Origin and Mixed Content Tests Cross-origin vulnerabilities are particularly dangerous in browser-based WebSocket applications, where same-origin policy acts as a key security measure. This category includes six targeted tests (Attacks #58-63) that simulate cross-origin contexts, mixed security levels, and origin spoofing. Examples include: • Missing CORS Headers: Checking if the server exposes sensitive content to unauthorized origins. • Cross-Origin Iframe and PostMessage Abuse: Mimicking 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. Failing to enforce origin validation can result in: • Cross-Site WebSocket Hijacking (CSWSH), • Data leakage to unauthorized domains, • Clickjacking and iframe abuse, • Browser trust violations in HTTPS contexts. Our framework checks whether WebSocket endpoints properly inspect the Origin and Referer headers during the handshake phase and deny any cross-origin attempts not explicitly allowed. G. Application-Layer Logic and Misconfiguration Tests This category shifts focus from protocol-level issues to vulnerabilities in application logic and misconfigured server behaviors. We developed six attack cases (Attacks #64-69) to uncover poor sanitization, insecure headers, and overexposed API design patterns. Key tests include: • Error Message Leak and Server Disclosure: Causing errors to see if internal server paths, debug logs, or stack traces are exposed. • URL Path Traversal: Simulating directory escape attempts like ../../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 too many query parameters to test request handling limits. These vulnerabilities often arise from inadequate input validation, overly detailed error reporting, or exposed internal routing logic. A secure WebSocket implementation must ensure safe error handling and sanitized responses, especially when protocol upgrades connect the client directly to backend logic. H. DoS and Resource Management Tests WebSocket servers must be resilient against malicious efforts to deplete resources. Our framework features nine focused tests (Attacks #70-78) that assess the server's ability to handle denial-of-service (DoS) scenarios and resource mismanagement. Scenarios include: • Connection Flood and Max Connections: Stressing concurrent connections beyond expected limits. • 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: Keeping idle connections open indefinitely. • TCP Half-Open Resource Leak: Simulating clients that begin but never finish handshakes, leading to resource starvation. • No Compression Negotiation: Forcing compression even when the server has not agreed to it. These tests simulate real-world DoS methods like slowloris attacks, amplification through compression bombs, and connection pool exhaustion. Servers that fail to disconnect idle or non-compliant clients risk reduced performance and complete service outages. I. Protocol Fuzzing Tests Protocol fuzzing is a technique used to find zero-day vulnerabilities, memory safety issues, and unexpected behaviors by sending syntactically or semantically malformed WebSocket frames. Our suite includes 12 custom fuzzing tests (Attacks #79-90) that explore both control-plane and application-layer attack surfaces.

These tests help reveal: - Parser crashes, segmentation faults, or unhandled exceptions, - Business logic bypasses when input validation fails, - Encoding or escaping flaws that lead to XSS, command injection, or data leakage. All fuzzing inputs follow WebSocket framing rules. This way, crashes occur because of payload logic, not protocol format violations. A server's ability to handle malformed inputs indicates its readiness for production use. IV. Equations and Workflow Logic This section outlines the internal logic and key parts of our WebSocket vulnerability analysis framework. The system crawls, scrapes, executes attacks, and aggregates results using clearly defined sets, functions, and control structures. A. Core Sets Let the initial seed URL be represented as: U0 = Starting URL As the crawler explores the target site, it fills the following sets: - C: Set of all successfully crawled URLs - D: Set of all discovered URLs, including both visited and queued URLs - W: Set of all WebSocket URLs found in D WebSocket endpoints are identified as: W = {u ∈ D | u starts with "ws://" or "wss://"} This set W is sent to the vulnerability engine for dynamic testing. B. API Scraping Logic During crawling, the system parses API endpoints that return JSON responses to extract embedded URLs. Let Ri be the response body of the ith API call. A regular expression extracts potential HTTP or WebSocket URLs from within Ri: regex = r'(https?|wss?)://[^\s"\']+' The resulting URLs are added to D and may be requeued for crawling if they meet the filtering conditions. C. Recursive Crawling Conditions Let Q be the crawling queue. For each URL u ∈ Q, define: - depth(u): depth of URL u from seed U0 - max\_depth: maximum allowed depth - max\_requests: global upper limit on crawl requests The recursive crawl condition is: ∀u ∈ Q: (u ∈ /C) ∧ (depth(u) ≤ max\_depth) ⇒ crawl(u) D. Filtering Conditions Let F represent the set of file extensions to ignore during crawling: F = {.js, .css, .png, .jpg, .gif, .woff, .svg,…} Let ext(u) denote the extension of URL u. The filtering logic is: u ∈ Q ⇒ crawl(u) ⟺ ext(u) ∈ /F This ensures efficient crawling by avoiding static or non-navigational assets. E. Final WebSocket Endpoint Collection After crawling, the finalized WebSocket set W is redefined as: W = {u ∈ D | scheme(u) ∈ {"ws", "wss"}} This set W serves as the input for the WebSocket vulnerability engine. F. WebSocket Vulnerability Testing Logic Each WebSocket endpoint w ∈ W is tested using a predefined set of test functions T. Each test function t ∈ T returns either a vulnerability report or a "safe" verdict. The cumulative set of detected vulnerabilities is: V = (w ∈ W) ∪ (t ∈ T) ∪ t(w) Where: - W: WebSocket endpoint set - T: Test function set - V: Set of vulnerability reports v ∈ V Each function t(w) performs a specific attack vector (e.g., header manipulation, fuzzed payloads, origin bypass) and logs the result. G. Workflow Controller Logic The system’s controller manages the entire workflow, from URL collection to final result generation. For each scanned site u ∈ U, we define the scan result structure: R[u] = {Cu, Wu, Vu, Tu} Where: - Cu: Crawled URLs for site u - Wu: WebSocket URLs found - Vu: Vulnerabilities detected - Tu: Total scan duration VI. Results and Analysis This project shows how effective our automated framework is at finding vulnerabilities in real-world WebSocket implementations. After crawling target domains and extracting live WebSocket endpoints, our engine ran 90 vulnerability tests, including protocol validation, session management flaws, improper subprotocol handling, transport security weaknesses, and application-layer misconfigurations. Each endpoint underwent targeted payloads—from malformed handshake headers and expired authentication tokens to oversized binary messages and protocol fuzzing mutations. The results were visualized with heatmaps and bar charts, highlighting the security stance of each site and identifying common classes of vulnerabilities. A. WebSocket Vulnerability Heatmap Fig. 3 shows a WebSocket vs. Attack Heatmap displaying the vulnerability exposure of each scanned domain across all test categories. Each row represents a domain, while each column corresponds to a specific attack from our test set (totaling 90). Insights: - Domains like pixivsketch.com, tradingview.com, and discord.com showed consistent exposure to high and medium-risk vulnerabilities, particularly in payload fragmentation, authentication tests, and origin validation. - Some sites (e.g., coincap.io, binance.com) had a higher percentage of low-risk or fully hardened blocks, suggesting mature WebSocket implementations. - The horizontal red bands indicate systemic vulnerabilities, meaning a specific test that failed across many sites—these include "Missing Origin Check," "Unmasked Client Frame," and certain "Fuzzing" tests. - The heatmap serves as a powerful tool for comparison, enabling quick identification of the weakest sites, test categories with widespread failures, and the variability of WebSocket security maturity across real deployments. B. Vulnerability Distribution by Type Fig. 4 offers a quantitative overview of vulnerabilities grouped by test categories. The bar chart organizes all 90 attacks into nine classes and shows the total number of vulnerabilities detected per class across all endpoints. Key Observations: - The Payload Framing & Messaging Semantics category leads significantly, highlighting a lack of strong validation of WebSocket frame structures. This includes acceptance of malformed opcodes, oversized control frames, or unmasked payloads, often overlooked in production deployments. - Protocol Fuzzing ranked second, confirming the need for sending unpredictable or malformed inputs to reveal undocumented crashes, misbehaviors, or parsing bugs. - Handshake and Upgrade Failures (131 detections) show that many servers are lenient toward malformed or incomplete handshake headers, such as missing Upgrade or Sec-WebSocket-Key headers, violating RFC compliance. - Notably, Authentication & Session Control accounted for 107 findings, revealing weak token handling, session reuse flaws, or missing authentication requirements during upgrade. - Subprotocol Handling, Transport Security, and Application Logic showed lower—but still concerning—amounts of vulnerability. These may often be obscured by custom application logic rather than explicit protocol handling, though their exploitation can still lead to privilege escalation or data leakage. - DoS-related attacks (127 detections) confirm the risks of uncontrolled memory, timeout, or message buffering issues under stress testing. C. Severity Distribution & Risk Posture The visual breakdown of findings shows that: - A significant majority of issues fall under the high-severity category, especially those tied to unauthenticated access, missing origin checks, and malformed input acceptance. - The medium-severity category captures less exploitable—but still impactful—bugs, such as improper compression negotiation or error message leaks. - Low-risk issues, while not immediately exploitable, suggest outdated or non-strict implementations and highlight areas where standards compliance is lacking. D. Summary These results reinforce several important points: - WebSocket security is often neglected, as developers depend on initial handshake validation but ignore frame-level robustness, origin control, and proper cleanup mechanisms. - The presence of high-severity issues across even well-known domains like discord.com and tradingview.com underscores the real-world relevance of our scanner and the need for automated analysis. - Visualizations such as the heatmap and bar chart are effective tools for risk prioritization, helping organizations focus on patching efforts by attack class or domain. VII. Use Case: Comparison with Industry Tools While there are several established tools for analyzing WebSocket communication, most target professional pentesters or require significant manual setup. Below, we compare two widely used solutions and explain how our tool fits in terms of usability, scope, and accessibility. 1. OWASP Zed Attack Proxy (ZAP) OWASP ZAP is one of the most popular open-source security tools, maintained by a global community of contributors. It is widely used for dynamic application security testing, offering features such as automated scanning, passive analysis, and intercepting proxies. ZAP supports intercepting, viewing, and modifying WebSocket traffic, allowing testers to manually analyze protocol messages. However, ZAP is mainly GUI-based and has a moderate learning curve for WebSocket-specific workflows. ZAP is written in Java and works on multiple platforms, with installers available for Windows, macOS, and Linux (as long as the Java Runtime Environment is available). Although powerful, its WebSocket testing features are less automated and may require scripting or extension development for more profound protocol-level testing. 2. Burp Suite Burp Suite, created by PortSwigger, is a commercial-grade security testing platform used by penetration testers and security professionals. It offers an integrated suite of tools for reconnaissance, scanning, manipulation, and exploitation of web applications. Burp Suite supports detailed WebSocket analysis, including message interception, replay, and tampering. However, these features are limited to the Professional Edition, which is a paid version. Burp is highly effective for deep, manual exploration and is ideal for expert users, but it lacks the plug-and-play nature that casual users or automation-focused researchers might want. 3. Our Tool: Lightweight and Fully Automated Our WebSocket vulnerability analysis tool prioritizes automation, accessibility, and customizability. Unlike ZAP or Burp, which demand domain expertise or cost licenses, our tool: - Requires no prior security knowledge to operate. - Is built in Python and runs via a straightforward command-line interface. - Accepts single website URLs or CSV lists for batch testing. - Conducts over 90 protocol-compliant tests aligned with RFC standards, covering a wide array of vulnerabilities. - Uses Playwright to dynamically crawl JavaScript-heavy websites and uncover WebSocket endpoints generated at runtime. - Produces easy-to-understand PDF reports featuring vulnerability heatmaps, bar graphs, and detailed summaries for each target. - Is suitable for educational use, security research, CI pipelines, and pre-production testing. While the tool intentionally sacrifices deep integration with browser sessions (such as cookies or localStorage) and advanced payload crafting (like full XSS/CSRF simulations), this trade-off allows for a lightweight and user-friendly solution, making rapid scanning and protocol-level validation of WebSocket services easy. VIII. Conclusion This work presents a thorough and automated framework for detecting security vulnerabilities in WebSocket implementations across real-world websites. By using headless browser automation, the system can crawl dynamic, JavaScript-heavy pages to uncover active WebSocket endpoints, including those hidden within API responses or runtime-generated JavaScript. After discovering endpoints, the tool runs a variety of 90 targeted tests grouped into nine vulnerability classes. These tests cover: - Handshake manipulation - Authentication bypass - Subprotocol/extension misuse - Transport security - Payload structure violations - Cross-origin policy enforcement - Application-level logic errors - DoS scenarios - Protocol fuzzing Crafted payloads simulate various attack vectors, such as invalid opcodes, expired tokens, unmasked frames, oversized messages, and malicious JSON inputs. Each test tracks the server's response—whether it accepts, rejects, ignores, or crashes—providing insight into the endpoint’s security controls. The findings compile into a structured PDF report visualized with: - A heatmap showing vulnerabilities per domain across the attack set. - A bar chart illustrating vulnerability distribution by type. - A pie chart categorizing attack classes, offering clear visibility into the protocol’s attack surface. Our results highlight extensive gaps in WebSocket security, particularly concerning payload framing, fuzzing resistance, and origin enforcement. Even well-known domains demonstrated vulnerabilities to basic misconfigurations or failures to comply with RFC standards. In summary, the tool provides: - A scalable, reproducible method for evaluating WebSocket security. - Valuable insights for developers, security teams, and researchers aiming to build safer real-time applications. - A foundation for future enhancements, including support for browser session integration, custom attack scripting, and CI/CD pipeline deployment. By bridging the gap between accessibility and depth, this project contributes a practical solution for securing the often-overlooked WebSocket layer of modern web infrastructure.