Automated Fuzzing Framework for Vulnerability Detection in Complex Systems

Martim Luís Delgado Ribeiro
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Automated Fuzzing Framework for Vulnerability Detection in Complex Systems

Martim Luís Delgado Ribeiro

Dissertation carried out as part of the Master in Information Security

Department of Computer Science

2025

Supervisor
João Miguel Maia Soares de Resende,
Assistant Professor,
Faculty of Sciences of the University of Porto

Acknowledgements

I would like to thank Professor João Resende for his guidance, support, and for the opportunities he provided throughout this journey.

I'm also grateful to James Kettle for his openness and for taking the time to answer my questions. Thanks as well to Andreas Happe and Erik Elbieh for their availability and help.

A big thanks to everyone involved in xSTF for the shared knowledge and spirit of mutual help.

Thank you to Hugo for believing in me and for the valuable ideas shared along the way.

Lastly, heartfelt thanks to my family for their constant support and encouragement. And thank you, Margarida, for your presence and for being by my side.

This work is funded by Component 5 - Capitalization and Business Innovation, integrated in the Resilience Dimension of the Recovery and Resilience Plan within the scope of the Recovery and Resilience Mechanism (MRR) of the European Union (EU), framed in the Next Generation EU, for the period 2021 - 2026, within project HOSKY, with reference 2024.07347.IACDC.

Resumo

O aumento de aplicações web em tempo real estabeleceu o protocolo WebSocket como uma tecnologia de comunicação fundamental. No entanto, a sua natureza assíncrona e ”stateful”apresenta desafios de segurança que as metodologias de teste tradicionais, baseadas em HTTP, não conseguem abordar. O panorama atual das ferramentas de segurança para WebSockets é dominado por abordagens manuais ou semi-automatizadas, que não têm sofisticação suficiente para detetar de forma dinâmica e eficiente vulnerabilidades do lado do servidor, deixando uma superfície de ataque por explorar.

Esta dissertação aborda esta lacuna através do desenvolvimento de uma framework de fuzzing automatizada para deteção de vulnerabilidades do lado do servidor em várias implementações de WebSocket. A principal contribuição consiste na melhoria do extensão Backslash Powered Scanner (BPS) do Burp Suite, originalmente destinada a HTTP, para o domínio dos WebSockets. A solução envolveu a introdução de uma abordagem inovadora para o fuzzing de mensagens WebSocket, desde o estabelecimento da ligação até à recolha de respostas. A lógica de deteção principal do scanner foi adaptada com novos critérios de comparação ajustados ao protocolo, o que permite uma análise detalhada do comportamento do servidor sem depender da semântica tradicional de HTTP.

A eficácia da framework foi validada em dois ambientes distintos: a aplicação Damn Vulnerable Web Sockets (DVWS) e um laboratório personalizado com uma implementação complexa de Socket.IO. A avaliação demonstrou que a framework identificou com sucesso e de forma automática uma série de vulnerabilidades de alto impacto do lado do servidor, incluindo SQL Injection, Server-Side Template Injection, PHP Code Injection, e File Inclusion. Em testes comparativos, a framework superou consistentemente as ferramentas existentes, que exigiam uma intervenção manual significativa e falhavam frequentemente em cenários com handshakes complexos.

Esta investigação fornece uma ferramenta prática e robusta que preenche uma lacuna na segurança de aplicações web modernas. Ao adaptar com sucesso a lógica de análise do BPS ao protocolo WebSocket, este trabalho oferece uma solução eficaz para profissionais de segurança e contribui com melhorias valiosas e open-source para a comunidade.

Palavras-chave: WebSocket Security, Fuzzing, Automated Vulnerability Detection, Burp Suite, Backslash Powered Scanner, Socket.IO, Server-Side Vulnerabilities

Abstract

The increase of real-time web applications has established the WebSocket protocol as a fundamental communication technology. However, its asynchronous and stateful nature presents security challenges that traditional HTTP-based testing methodologies fail to address. The current landscape of WebSocket security tools is dominated by manual or semi-automated approaches, which lack the sophistication to dynamically and efficiently detect server-side vulnerabilities, leaving an attack surface underexplored.

This thesis addresses this gap by developing an automated fuzzing framework for detecting server-side vulnerabilities within various WebSocket implementations. The primary contribution is the extension of the Backslash Powered Scanner (BPS) Burp Suite extension, originally for HTTP, to the WebSocket domain. The solution involved introducing a novel approach to how to fuzz WebSocket messages, from connection establishment to response collection. The scanner's core detection logic was adapted with new, protocol-specific comparison metrics, enabling nuanced analysis of server behavior without reliance on traditional HTTP semantics.

The framework's efficacy was validated in two distinct environments: the Damn Vulnerable Web Sockets (DVWS) application and a custom-built laboratory featuring a complex Socket.IO implementation. The evaluation demonstrated that the framework successfully and automatically identified a range of high-impact server-side vulnerabilities, including SQL Injection, Server-Side Template Injection, PHP Code Injection, and File Inclusion. In comparative tests, the framework consistently outperformed existing tools, which required significant manual intervention and often failed in scenarios involving complex handshakes.

This research delivers a practical and robust tool that bridges a gap in modern web application security. By successfully adapting BPS scanning engine to the WebSocket protocol, this work provides an effective solution for security professionals and contributes valuable, open-source enhancements to the community.

Keywords: WebSocket Security, Fuzzing, Automated Vulnerability Detection, Burp Suite, Backslash Powered Scanner, Socket.IO, Server-Side Vulnerabilities

Table of Contents

List of Figures

Chapter 1

Introduction

In an era where real-time communication is critical for a large number of applications, the WebSocket protocol has emerged as a cornerstone technology. Since its standardization in 2011 [1], WebSocket has been increasingly adopted by enabling persistent, bidirectional connections between clients and servers [2]. Unlike traditional HTTP, which follows a request-response model, WebSocket facilitates low-latency data exchange with minimal protocol overhead. This fundamental shift in design has made it an attractive choice for applications requiring continuous updates and real-time interactions, such as collaborative tools, online gaming, financial trading platforms, and IoT device management systems.

Despite its widespread adoption and technical advantages, WebSocket communication can introduce new security challenges. As noted by PortSwigger, “In principle, practically any web security vulnerability might arise in relation to WebSockets” [3]. But the asynchronous, stateful nature of WebSocket communication complicates traditional security testing approaches, which are primarily designed for HTTP’s request-response model. This discrepancy has resulted in a notable gap in the tools and methodologies available for testing WebSocket implementations, leaving many real-time applications vulnerable to attacks.

The current landscape of WebSocket security testing remains underdeveloped. While tools for testing traditional web applications have evolved and matured over time, those designed specifically for WebSocket security are limited in both number and capability [4]. The dynamic nature of WebSocket communications, characterized by bidirectional, asynchronous messaging, presents unique challenges for automated security testing. These challenges are further exacerbated when binary formats or custom implementations are

used, making it difficult to intercept, manipulate, and consistently test messages. As a result, researchers and security professionals often resort to time-consuming manual testing, which is not scalable.

The relatively low number of publicly disclosed vulnerabilities in WebSocket contexts may not accurately reflect the true security posture of WebSocket-based applications. Instead, it could be indicative of the limitations in current security testing methodologies. Existing tools, which are primarily focused on HTTP traffic, fail to adequately assess vulnerabilities specific to WebSocket implementations. This gap in effective automated testing tools means that critical vulnerabilities may remain undetected until they are exploited in production environments, posing significant risks to both users and organizations.

This situation underscores the urgent need for more advanced, automated security testing solutions tailored to WebSocket implementations. The development of such tools is essential not only for improving the security of WebSocket-based applications but also for advancing the understanding of WebSocket vulnerabilities.

1.1 Motivation

Traditional fuzzing techniques, while effective for HTTP endpoints, fail to address the complexities of WebSocket communications, which involve stateful and persistent connections. As the adoption of real-time applications relying on WebSocket protocols increases, these limitations have become more pronounced, making traditional methods inadequate for effectively securing such systems. WebSockets, with their long-lived connections and bidirectional communication, present a unique challenge for security testing tools.

Current tools for WebSocket security assessment are predominantly manual or semi-automated, and present a lack of features, requiring significant time and expertise to identify vulnerabilities [5–9]. This method is both resource-intensive and not scalable, making it difficult to keep pace with the rapid development cycles of modern web applications.

The Backslash Powered Scanner (BPS) [10], a Burp Suite extension, has demonstrated considerable effectiveness in fuzzing traditional HTTP-based applications, identifying vulnerabilities through its powerful and flexible testing capabilities. However, its current implementation does not extend to WebSocket communications, leaving a notable gap in its functionality. Enhancing Backslash to support WebSocket fuzzing could significantly

expand its utility, enabling it to assess a broader range of real-time applications and contribute to the evolution of automated security testing methodologies.

1.2 Proposed Solution

The work presented in this thesis aims to develop a comprehensive WebSocket fuzzing capability within the BPS framework, addressing the current limitations in automated security testing of WebSocket-enabled applications. We intend to create a solution that can effectively handle the stateful nature of WebSocket connections while maintaining the robust fuzzing capabilities that BPS is known for. The development of this enhanced functionality would benefit both security researchers and application developers by providing automated, thorough security assessment capabilities for WebSocket implementations.

Our solution will build upon BPS's existing architecture while introducing specialized components for WebSocket handling, message mutation, and state tracking.

1.2.1 Objectives

The main goals that we pretend to reach with our proposal are:

- **Goal 1 -** Develop a WebSocket security testing procedure that aligns with best practices in offensive security and penetration testing methodologies.
- **Goal 2 -** Establish an evaluation methodology applicable to various WebSocket implementations, ensuring compatibility across different use cases.
- **Goal 3 -** Implement a modular and maintainable automated WebSocket security assessment framework to dynamically identify vulnerabilities, designed to allow for seamless updates as WebSocket technology evolves.

1.3 Contributions

As part of this work, a pull request was submitted to the BPS [10], introducing support for WebSocket fuzzing. This contribution was reviewed, accepted, and successfully merged into the original project, making WebSocket fuzzing an integrated feature of the tool.

Additionally, another pull request was submitted to the BPS [10] to address a bug identified during testing. This issue was related to a function injection probe that incorrectly caused a

valid payload to fail, leading to false negatives in PHP injection scenarios. The submission was successfully merged into the project.

Finally, a contribution was made to Damn Vulnerable Web Sockets (DVWS) [11] through a pull request that introduced SSTI and PHP injection vulnerabilities to the application. This submission was also sucessfully merged into the project.

1.4 Outline

This thesis is structured as follows: Chapter 2 provides the background on WebSocket technology and on server-side vulnerabilities. It also introduces the BPS and explores related work on WebSocket security testing. Chapter 3 details the system design of our proposal and Chapter 4 presents the implementation details and approaches used on our system. Chapter 5 evaluates the system's effectiveness and, in Chapter 6, we lay down some ideas for future work and we mention some final remarks.

Chapter 2

State of The Art

The adoption of the WebSocket protocol for real-time web applications has introduced significant security challenges that traditional, HTTP-based testing methodologies cannot address. The stateful and asynchronous nature of WebSockets complicates automated vulnerability detection, and the current landscape of security tools is dominated by manual or semi-automated approaches that struggle with complex implementations. This gap leaves a critical attack surface underexplored.

To address this limitation, our research extends the Backslash Powered Scanner (BPS) to provide automated security testing for WebSocket communications. This chapter establishes the foundation for our work by examining the WebSocket protocol, common server-side vulnerabilities, and the core architecture of the BPS. It then reviews the existing landscape of WebSocket security testing to contextualize our contributions and highlight the need for a more robust, automated solution.

2.1 WebSocket protocol

2.1.1 History

WebSockets were created to address the inefficiencies of traditional HTTP-based communication methods, especially for real-time, bidirectional communication. Prior to WebSockets, developers used workarounds to simulate real-time updates, that essentially abused the HTTP protocol. Two examples of this were polling, which required the client to repeatedly request data from the server and creating unnecessary load, and long polling, which kept a connection open until new data was available, but still required frequent reconnections [12].

Initially developed by the World Wide Web Consortium (W3C) and the WHATWG group as part of the HTML5 standard, the WebSocket protocol was later outsourced from HTML5 for more flexibility and faster progress. In February 2010, development was moved under the Internet Engineering Task Force (IETF). After several drafts and revisions, including addressing a security vulnerability related to cache poisoning attacks, the IETF published the finished WebSocket protocol as RFC 6455 [1] in December 2011.

2.1.2 Handshake

WebSocket communication begins with a client sending a special HTTP request to the server, signaling its intent to upgrade the connection to use the WebSocket protocol. This request is initiated by a browser or another WebSocket-enabled client and contains a set of specific HTTP headers. These headers are essential for establishing a WebSocket connection and ensuring that the client and server can securely and efficiently transition from HTTP to WebSocket.

The WebSocket handshake, as this initial exchange is called, starts with the client requesting an upgrade to the WebSocket protocol through the `Upgrade` and `Connection` headers, which are both set to `websocket` and `Upgrade`, respectively. Additionally, the client includes the `Sec-WebSocket-Key`, a randomly generated 16-byte nonce, encoded in Base64. This nonce plays a critical role in the challenge-response process that ensures the security of the WebSocket connection. The client also specifies the WebSocket protocol version it supports using the `Sec-WebSocket-Version` header, with version 13 being the latest as of RFC 6455 [1].

When the server receives the request, it performs several validation checks. First, the server ensures that the client has included a valid `Sec-WebSocket-Key`. Then, it concatenates the client's `Sec-WebSocket-Key` with a fixed *GUID* string (defined in RFC 6455 [1]) and calculates a *SHA-1* hash of the concatenated value. The server encodes this hash using Base64 and returns it in the `Sec-WebSocket-Accept` header. This response verifies that the server is willing to upgrade the connection and confirms that it is the correct server that the client intended to connect with [13].

If the server is unable to accept the upgrade for any reason, such as an unsupported WebSocket version or a mismatch in the keys, it sends an appropriate HTTP error code,

typically a `400` series response, and terminates the connection. In contrast, if the handshake is successful, the server responds with an HTTP `101` status code, indicating that the protocol is being switched to WebSocket, and the connection is upgraded.

At this point, both the client and server can freely exchange data over the open connection, using the WebSocket protocol for real-time, bidirectional communication. Data can be sent in either binary or UTF-8 encoded text format, and both the client and server are free to send messages at any time without needing to re-establish the connection or wait for a request/response cycle, as is required in traditional HTTP communication [14].

While the WebSocket connection could technically be terminated by simply closing the underlying TCP connection, the WebSocket specification mandates an in-band closing handshake. This ensures that the connection is properly closed and is only officially considered as so when either the client or the server sends a closing handshake, signaling that they no longer wish to continue communication. [15]

In terms of connection addressing, the WebSocket protocol defines two types of Uniform Resource Identifiers (URIs) for establishing connections: `ws://` and `wss://`. These URIs are similar to the standard HTTP and HTTPS protocols but are specifically designed for WebSocket communication.

- `ws://`: is used for non-encrypted connections. When a WebSocket client initiates a connection using the `ws://` URI scheme, it establishes an unencrypted WebSocket connection with the server.
- `wss://`: on the other hand, is used for encrypted WebSocket connections. This scheme employs TLS (Transport Layer Security) to encrypt the communication between the client and the server, providing a secure channel for transmitting data. It is the WebSocket equivalent of `https://` in the HTTP world, ensuring that the communication is protected against eavesdropping and tampering.

An example of the handshake request can be seen on *Figure 2.1*.

2.1.3 Frames

With the connection open, client and server can exchange messages at any time. More specifically, a WebSocket message is composed of one or more frames. The WebSocket frame structure is a binary syntax that includes several key components which each play

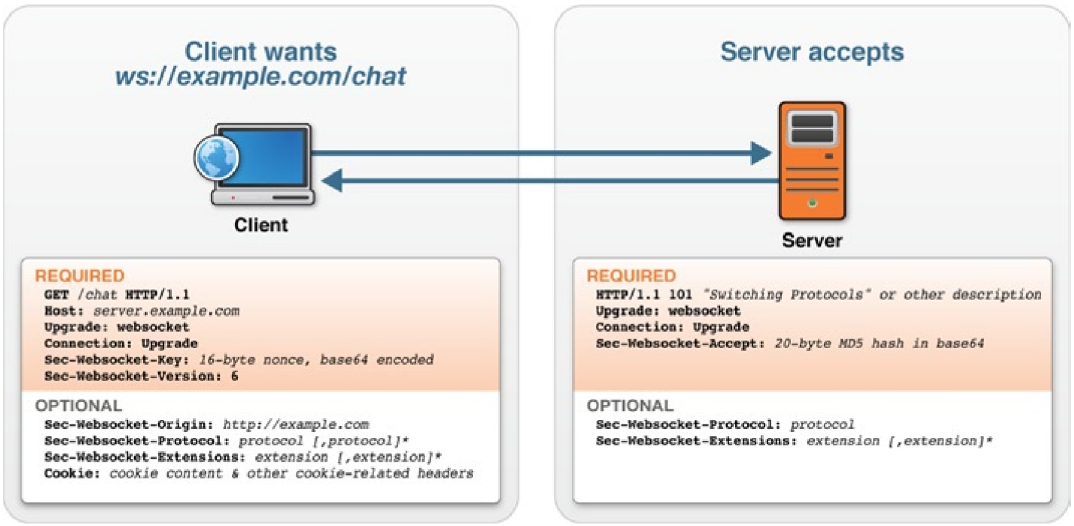


Figure 2.1: Example of a WebSocket opening handshake. Source: [16]

a significant role in how the WebSocket protocol operates. Understanding this frame structure is essential for debugging and ensuring correct WebSocket message delivery. The frame's components encapsulate the data to be transmitted between the client and the server, including metadata such as the payload length, whether the message is fragmented, whether the payload is masked, and more. They can be seen on *Figure* 2.2.

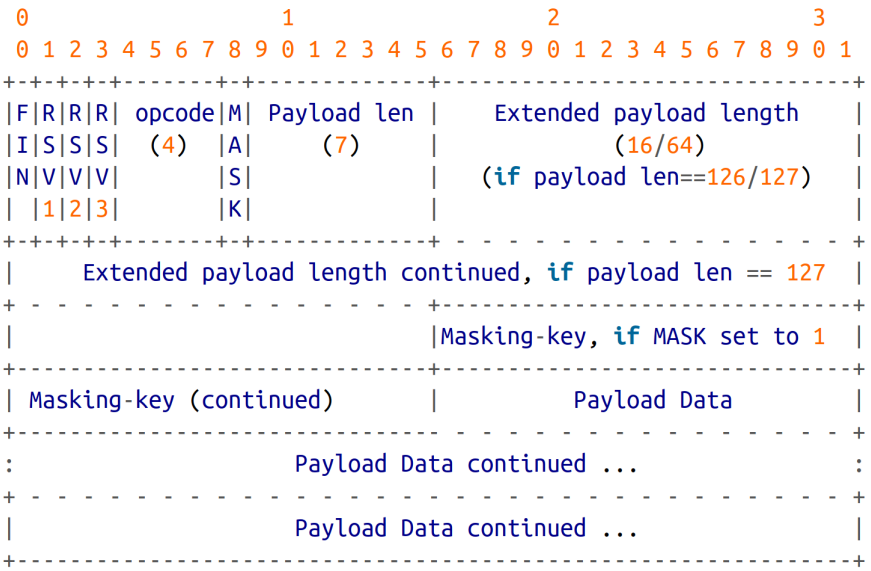


Figure 2.2: Text representation of a WebSocket frame. Source: [1]

The first element of the WebSocket frame is the `FIN` bit. This bit indicates whether the frame is the final frame in the message. If the `FIN` bit is set to 1, the frame is the last part of the message, signaling that no further frames will follow. If it is set to 0, the frame is a continuation, and more frames will follow. The `Opcode` follows the `FIN` bit, occupying the next four bits of the frame. This field determines the type of data being sent. For example,

an opcode of `0x01` signifies that the frame contains a text message, while an opcode of `0x02` indicates binary data. Other opcodes include `0x08` for connection close, `0x09` for ping, and `0x0a` for pong.

`RSV1`, `RSV2`, and `RSV3` are reserved bits. They must be 0 unless an extension was negotiated during the opening handshake that defines non-zero values [17]. See Subsection 2.1.4 for more details.

The `Mask` bit indicates whether the frame is masked. By default, the WebSocket protocol requires that all frames sent from the client to the server be masked. The masking is used to avoid certain cross-protocol attacks, such as cache poisoning, and is not intended for security purposes in the conventional sense (i.e., to prevent eavesdropping), and as such, there is no need to mask messages from server to client, since the client is not vulnerable to these attacks. When the `Mask` bit is set, the frame is followed by a masking key, which is a 4-byte value used to obfuscate the payload data. The masking process involves performing an XOR operation on the payload using the masking key to prevent the payload from being easily read [18].

The `Payload length` field encodes the length of the payload data, which is the actual message content. The length is encoded in the last 7 bits of the second byte of the frame header. If the payload length is less than 126 bytes, the length is simply encoded using these 7 bits. If the payload is longer than 125 bytes, the length field takes additional bytes to encode the full length. For payloads between 126 and 65535 bytes, 2 additional bytes are used, and for payloads larger than 65535 bytes, 8 additional bytes are used. This length encoding ensures the WebSocket protocol can efficiently handle both small and large messages [19].

The Extended Payload length field is used when the length of the payload exceeds the normal 125 bytes and the length needs to be encoded across multiple bytes. This field uses 2 or 8 additional bytes to encode the length, depending on whether the payload size is between 126-65535 or larger than 65535 bytes, respectively.

The `Payload Data` follows the length field and contains the actual data being sent in the frame. This can be either binary data or UTF-8 encoded text. If the frame contains binary data, it is transmitted directly as is. If the frame contains text, it is encoded in UTF-8 to ensure compatibility with a wide range of systems and to avoid issues with different encodings. The payload may also contain extension data if WebSocket extensions are

in use. Extensions provide additional functionality to the WebSocket protocol, such as multiplexing, message compression, or message fragmentation.

WebSocket messages can be fragmented into multiple frames if necessary. This fragmentation is controlled by the `FIN` bit and the `opcode`. If the message is fragmented, the `opcode` for continuation frames is `0x00`, and the `FIN|` bit is set to `0` for all frames except the last one. The last frame in the sequence will have the `FIN` bit set to `1`, indicating the end of the message. In cases of message multiplexing or when a large message needs to be split into smaller parts, fragmentation allows for efficient data transfer by sending smaller pieces of data over multiple frames. Importantly, a WebSocket message can only contain one type of data, either text or binary, and cannot mix the two within the same message [17].

The frame format is essential to the WebSocket protocol as it provides the necessary structure to send, receive, and interpret data across the connection. While higher-level WebSocket APIs abstract the frame details, a deep understanding of the frame format allows developers to troubleshoot, optimize, and ensure the reliability of WebSocket communication. Key elements such as the `FIN` bit, opcode, masking, and payload length help ensure correct message delivery and proper handling of large or fragmented messages.

2.1.4 Subprotocols and Extensions

Subprotocols allow higher-level application protocols to be layered on top of the WebSocket Protocol, facilitating structured communication between a client and a server. The WebSocket Protocol itself remains unchanged, serving as a transport layer upon which different subprotocols define their own message formats, sequencing, and handling logic. The negotiation of a subprotocol occurs during the handshake phase via the `Sec-WebSocket-Protocol` header, where the client proposes one or more subprotocols, and the server selects one if it supports any of them. If the server does not support the proposed subprotocols, it will either return null or exclude the Sec-WebSocket-Protocol header in its response [17].

Subprotocols can be classified into three broad categories: registered protocols, open protocols, and custom protocols. Registered protocols are formally recognized and listed by the IANA WebSocket Subprotocol Name Registry. Open protocols include widely adopted messaging standards like MQTT and STOMP, which facilitate structured message exchange over WebSockets. Custom protocols, on the other hand, are typically

application-specific and often use domain names as identifiers to avoid conflicts. Regardless of their category, subprotocols enable WebSocket applications to enforce rules on message structure and exchange semantics, ensuring compatibility between communicating entities [13].

Unlike subprotocols, WebSocket extensions modify the fundamental behavior of the WebSocket Protocol itself. Extensions allow for the introduction of new opcodes, framing structures, and advanced transmission capabilities such as message compression and multiplexing. Similar to subprotocols, extensions are negotiated during the handshake using the `Sec-WebSocket-Extensions` header. The client can propose multiple extensions, and the server may accept any subset that it supports. However, all negotiated extensions must be explicitly acknowledged in the server's response to be used during the session.

An example of a WebSocket extension is `x-google-mux`, an experimental extension aimed at enabling multiplexing, allowing multiple logical WebSocket connections to share a single physical connection. Unlike subprotocols, extensions require browser and server support at a lower level, making them harder to deploy since they depend on implementation-specific adoption [13].

While a client can negotiate multiple extensions in a single request, only one subprotocol can be selected per connection. This summarizes the different roles that subprotocols and extensions play within WebSocket communication. Subprotocols define high-level communication standards, whereas extensions modify the fundamental data transmission mechanisms.

2.1.5 API

The WebSocket API enables real-time, full-duplex communication between clients and servers over a single, long-lived connection.

A WebSocket server can be implemented in any language that supports Berkeley sockets (a standard API for network communication). The server listens for incoming WebSocket connections over a standard TCP socket. After the initial handshake, the server and client can freely exchange messages asynchronously [17].

On the client side, a WebSocket connection is established using the `WebSocket` constructor [17]:

```
const socket = new WebSocket('wss://example.org');
socket.onopen = function(event) {
    console.log('Connection established');
};
```

Listing 2.1: Creating a WebSocket connection

WebSocket communication follows an event-driven model. The key events in the WebSocket API include:

- **open**: Triggered when the connection is successfully established.
- **message**: Fired when the client receives a message from the server.
- **error**: Occurs when an unexpected error is encountered.
- **close**: Triggered when the connection is closed, either intentionally or due to an error.

Messages sent via WebSockets can be text or binary data. The `send()` method is used to transmit data:

```
socket.send(JSON.stringify({message: 'Hello, server!'}));
```

Listing 2.2: Sending a message via WebSocket

To close a WebSocket connection, the `close()` method is called:

```
socket.close(1000, 'Normal closure');
```

Listing 2.3: Closing a WebSocket connection

WebSocket connections have various states, accessible via the `readyState` property:

- **0 (CONNECTING)**: The connection is being established.
- **1 (OPEN)**: The connection is open and ready for data exchange.
- **2 (CLOSING)**: The connection is in the process of closing.
- **3 (CLOSED)**: The connection has been closed.

Unlike HTTP requests, there is no need for continuous re-establishment of connections.

2.2 Server-side vulnerabilities

Web application vulnerabilities can be separated into two main families based on where the security flaw exists and runs: client-side or server-side. Client-side vulnerabilities, like Cross-Site Scripting (XSS), execute in a user's web browser. An attacker might exploit such a flaw to steal a user's session cookies or alter the content of a webpage for that specific user. While damaging, the impact is often limited to the individual whose client has been compromised [20]. Server-side vulnerabilities, however, are fundamentally different. They represent security holes in the application's backend infrastructure (the web server, the application logic, and the databases that support it). When an attacker successfully exploits a server-side flaw, they are not just attacking a single user, they are attacking the application's core. This can lead to the theft of the entire user database, files, unauthorized control over the application's features, or even a full compromise of the server itself, affecting every user and the integrity of the service as a whole [21].

The root cause of the vast majority of server-side vulnerabilities is the improper handling of user input. A web application is built on a fundamental trust boundary between the client and the server. The server must operate on the principle that any data arriving from a client is untrustworthy and potentially malicious. Many security flaws appear when developers neglect this principle and implicitly trust the input they receive. When user-supplied data is not correctly validated, sanitized, or escaped, an attacker can craft specialized inputs that the application might misinterpret. Instead of treating the input as simple data, the server-side components may execute it as a database command, a system instruction, or a line of code, giving the attacker unintended power [20].

Injection flaws are the most direct consequence of this broken trust. For instance, SQL Injection (SQLi) occurs when an attacker provides input that alters the structure of a SQL query. If an application builds a query by simply joining strings together, like `SELECT * FROM users WHERE username = ' + userInput + ';`, an attacker can provide input like `' OR '1'='1` to change the query's logic and retrieve all user records. A broader category is Code Injection, where user input is passed directly to a function that executes it, such as PHP's `eval()` function. This effectively gives the attacker a channel to run their own code on the server [21]. A more recent variant is Server-Side Template Injection (SSTI), which targets the template engines used to generate dynamic web pages. If user input containing template syntax is processed without sanitization, an attacker can execute code and take control of the server [22]. Another critical injection-style flaw is File

Inclusion. This allows an attacker to manipulate the application into reading or even executing files from the server's filesystem, which can expose sensitive configuration files or lead to code execution.

However, server-side vulnerabilities extend beyond injection. Flaws can also exist in the application's core logic, particularly in how it manages access control. Authentication is the process of verifying who a user is, while authorization is the process of checking what an authenticated user is allowed to do. Vulnerabilities in these mechanisms can allow an attacker to bypass login forms entirely or access administrative functions that should be restricted. Another dangerous class of vulnerability is Insecure Deserialization. Modern applications often transmit complex data by serializing it, which means converting data objects into a string or byte stream for transmission. When this data is received, it is deserialized back into an object. If an attacker can manipulate this serialized object, the deserialization process on the server could be exploited to instantiate unintended classes or execute arbitrary code. These logic and data-processing flaws are often more subtle than injection vulnerabilities but can be just as devastating [23]. The unifying thread among them all is the server's failure to enforce strict rules on data and user actions, creating opportunities for exploitation of the system.

2.3 Backslash Powered Scanner

The BPS [10] is a Burp Suite extension that distinguishes itself from traditional vulnerability scanners by adopting an iterative, behavior-based approach inspired by manual testing techniques.

Unlike others that rely on predefined, technology-specific payloads, this scanner utilizes a set of generic payloads containing special characters (e.g. backslashes, quotes, braces) to assess how the server processes and responds to these inputs. It begins by modifying the input and evaluating the response against predefined metrics, such as HTTP status codes, the total number of new lines in the response, or the frequency of various keywords. If a deviation is detected, the scanner investigates further. Otherwise, the input is disregarded. In essence, rather than explicitly searching for known vulnerabilities, the scanner focuses on identifying anomalous or unexpected behavior, which may indicate potential security flaws.

This approach enables the detection of both known and previously unknown injection vulnerabilities. Known vulnerabilities refer to issues that have been identified, documented,

and often mitigated, while unknown vulnerabilities represent flaws yet to be discovered. The extension is primarily designed to identify server-side injection vulnerabilities, where malicious input is processed by the server, as opposed to client-side vulnerabilities that impact the user's browser or device. Its strength lies in its ability to highlight interesting or anomalous behavior, even in the absence of a fully identifiable vulnerability. By isolating the small subset of inputs that trigger unusual responses, it facilitates and allows for a deeper manual investigation, potentially leading to the discovery of a security issue.

The scanner can be launched from any intercepted message within Burp Suite, most commonly via the Proxy or Repeater components. It offers several configuration options to tailor its behavior to specific use cases, including the types of tests to be performed, the number of confirmations required to validate an anomaly, and other customizable parameters. From a more technical perspective, the scanner relies on three primary techniques:

- **Transformation Scan** – Detects how input is altered or escaped by the server.
- **Diffing Scan** – Analyzes discrepancies between server responses and baseline behavior using paired probes, in order to identify deviations that may indicate improper input handling.
- **Interference Scan** – Examines concurrency-related issues such as race conditions, cross-request contamination, and other forms of request interference.

This work deliberately excludes the Interference Scan, prioritizing the establishment of a robust foundation for WebSocket fuzzing. Race conditions in this context remain even less explored than server-side vulnerabilities, but this implementation has been designed to accommodate the future integration of this and other techniques with minimal effort. The remaining scan types are explained in more detail in sections 2.3.2 and 2.3.3.

But before exploring the scanning mechanisms, it is essential to acknowledge that some of the code utilized by the BPS originates from a repository called *BulkScan* [24]. In essence, *BulkScan* is designed to queue and execute multiple scans concurrently across numerous endpoints or applications, within Burp Suite.

This is relevant to our work because BPS relies heavily on these classes, which are hard-coded for HTTP and tailored to its logic. To add WebSocket support, it is crucial to understand the specific functions of these classes and determine the necessary modifications.

To clarify this, we will first provide an overview of these classes.

2.3.1 Integration with *BulkScan*

This section provides a high-level overview of the key classes inherited from *BulkScan* that are integral to the BPS operation.

The *Probe* class plays a central role in the scanner by defining and managing the test cases used during fuzzing. Each probe represents a specific type of input pattern designed to explore how the target application handles potentially malicious or malformed data. Its main function is to generate payloads that aim to trigger unusual or vulnerable behavior in the server. These payloads can vary in structure and intent. For instance, they may include characters that disrupt normal execution or bypass input validation mechanisms.

The class also allows for flexible configuration of how these payloads are inserted into messages, supporting different injection strategies. Additionally, it includes options to increase variability and resilience in testing, such as randomizing injection points or avoiding caching issues. Once a probe is executed, the associated results are categorized based on how the application responded. This classification helps the scanner distinguish between inputs that likely triggered something significant and those that did not. In summary, it encapsulates the logic for crafting, deploying, and assessing test inputs.

The *QuantitativeMeasurements* class is responsible for analyzing numerical characteristics of server responses, such as response lengths or timing-related data. Its main purpose is to support the detection of patterns or anomalies by comparing these values across different attack attempts. By aggregating and organizing quantitative metrics, this class helps the scanner identify subtle behavioral variations that may indicate a vulnerability, particularly in performance-sensitive or context-dependent scenarios.

The *Attack* class then organizes and analyzes the outcome of a specific fuzzing attempt. Its main role is to track how the server responds to a given test payload and to identify any patterns or anomalies that might indicate a vulnerability. It does this by collecting various characteristics from the server's responses, such as timing, consistency, and content features, and creating a kind of "fingerprint" that summarizes the observed behavior. By comparing these fingerprints across different payloads, the class helps detect deviations that may suggest improper input handling or exploitable behavior. Additionally, it supports comparison between multiple attacks, making it easier to identify subtle changes in server responses that might otherwise go unnoticed. In doing so, it provides a structured and

comprehensive way to evaluate the effectiveness and impact of each payload.

The *FastResponseVariations* class is designed to detect changes in server responses during fuzzing. Its main purpose is to determine whether specific response attributes, such as length, structure, or formatting, remain consistent or vary when different payloads are sent. By monitoring these characteristics, the class helps identify patterns that suggest stable behavior, as well as anomalies that could point to a vulnerability or unexpected server logic. This distinction between consistent (invariant) and inconsistent (variant) features enables the scanner to focus on inputs that produce meaningful differences, reducing noise and improving efficiency. In essence, this class provides a lightweight and effective way to assess how a server reacts to different inputs.

Lastly, the *PayloadInjector* class is central to the fuzzing process, as it handles the task of inserting test payloads into requests and orchestrating the execution of these attacks. Its primary role is to generate variations of requests that include the crafted payloads and send them to the target application to test for vulnerabilities. This class ensures that payloads are correctly placed within requests and that different attack scenarios are systematically explored. It also manages the process of verifying whether a payload has triggered any notable changes in the application's behavior, helping to confirm the validity and significance of any detected anomalies. In short, the *PayloadInjector* class automates the critical step of transforming theoretical test cases into actual attacks.

2.3.2 Transformation Scan

With the BulkScan classes established as the foundation of the fuzzing process, we can now introduce the specific scanning mechanisms.

A high-level representation of the tool's workflow, applicable to both scanning techniques, is illustrated in Figure 2.3.

Starting with the transformation scan, it is a technique focused on understanding how an application processes special characters and escape sequences within user input. The core idea is to send various payloads that contain these special characters and observe the application's response. The goal is to identify any anomalies or unexpected behaviors in the way the server handles these characters, as this can indicate potential vulnerabilities in how the application processes user input.

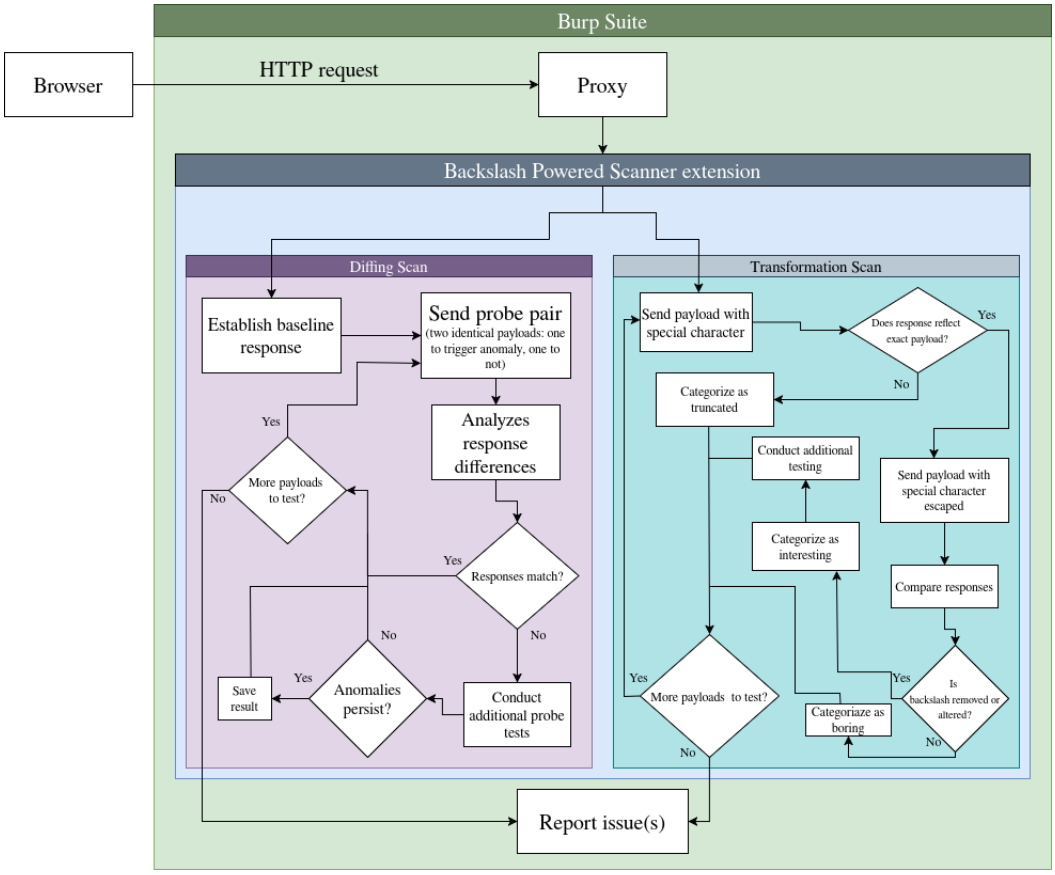


Figure 2.3: High-level architecture of the Backslash Powered Scanner

At a high level, the scanner starts by sending different payloads that include special characters. This step is important to see how the application initially handles these inputs. After the first round of testing, the scanner sends the same characters again but this time with an escape sequence (e.g., a backslash) to observe how the server handles the escaped version of the payload. Any differences in the handling of these inputs are analyzed, with the goal of identifying suspicious behavior that could indicate a vulnerability. These findings are then classified into “boring” or “interesting”. These classifications help prioritize which issues might require further investigation.

For instance, if the payload `\zz` is sent and the server responds with the same input `\zz`, it indicates that the server is treating this input normally without any special transformation. However, if a payload like `\{` is sent and the server responds with `{`, it suggests that the `{` character is being interpreted differently, potentially indicating that the application treats it as a special character or operator. If such behavior is detected, the scanner continues by identifying which characters trigger such transformations. If an escape sequence (such as the backslash) affects the behavior of specific characters, it is flagged as a ”suspicious transformation.” These characters are then marked for further analysis, which can help in

detecting vulnerabilities related to improper input handling.

The transformation scan is carried out through the *TransformationScan* class, which is specifically designed to identify and classify input transformations within HTTP responses. The class operates through a series of methods, most notably the *recordHandling()* and *classifyHandling()* methods, which serve different purposes in the scanning process.

In the *recordHandling()* method, the scanner generates a payload using random left and right anchors and sends it to the server using the *makeHttpRequest()* method. The response is then analyzed to check how the injected payload is reflected in the server's response. If the injected payload is not fully reflected or does not appear at all, it is categorized as ”Truncated” or ”Reflection disappeared,” respectively.

The *classifyHandling()* method is responsible for categorizing the different transformations that occur based on how the server handles the backslash character (`\\`). The handling of the backslash is critical because it can have special meanings in various programming languages and environments. The server's treatment of the backslash is analyzed to see if it's consumed by the application (i.e., removed or altered in some way). If the backslash is handled in a trivial way (i.e., not altered or treated as a special character) the transformation is classified as ”boring.” If it's consumed by the server, or if it results in a non-trivial transformation, it's considered ”interesting.”

The *findTransformationIssues()* method is the core of the scanning process. It starts by testing the server with a basic attack to see if backslashes are consumed in any way. The method then probes the server with various other payloads, which include encoded characters (such as `101` or `x41`, and special characters like quotes (`"`) or curly braces (`{`)). Based on how the server responds to these payloads, the transformations are classified into either ”interesting” or ”boring.” If any transformations are deemed ”interesting,” follow-up payloads are tested to gather more detailed information and uncover deeper insights into how the server processes these inputs.

Once the scanning process is complete, the results are compiled into an *InputTransformation* object. This object provides detailed information about which transformations were classified as interesting or boring, along with the original request and the insertion point where the input was tested.

2.3.3 Diffing Scan

To minimize the number of payloads sent while maximizing efficiency, James Kettle introduced the concept of ”probe pairs” in BPS. Each pair consists of two nearly identical payloads: one designed to trigger an anomaly and another that should not. This method allows the scanner to infer how inputs are processed by the application while avoiding unnecessary network traffic. The concept is best explained in Kettle's own words:

> First, we identify the normal response of the application by sending a probe containing random alphanumeric characters. This will be referred to as the 'base' response. If a probe containing `'` consistently gets a response that's different from the base, we can infer that the `'` character has a special significance to the application. This may not indicate a vulnerability - the application might just be rejecting inputs containing `'`. Once again, we can use backslashes to escape our predicament. If the application responds to probes containing `\'` in the same way as random alphanumeric probes, we can infer that the anomalous response to `'` is a caused by a failure to escape the character. [10]

This logic is illustrated in Figure 2.4, where the scanner's decision-making process is visualized. If an injected payload produces a response different from the baseline, a secondary probe is sent to confirm whether the anomaly results from improper input handling or merely a defensive rejection.

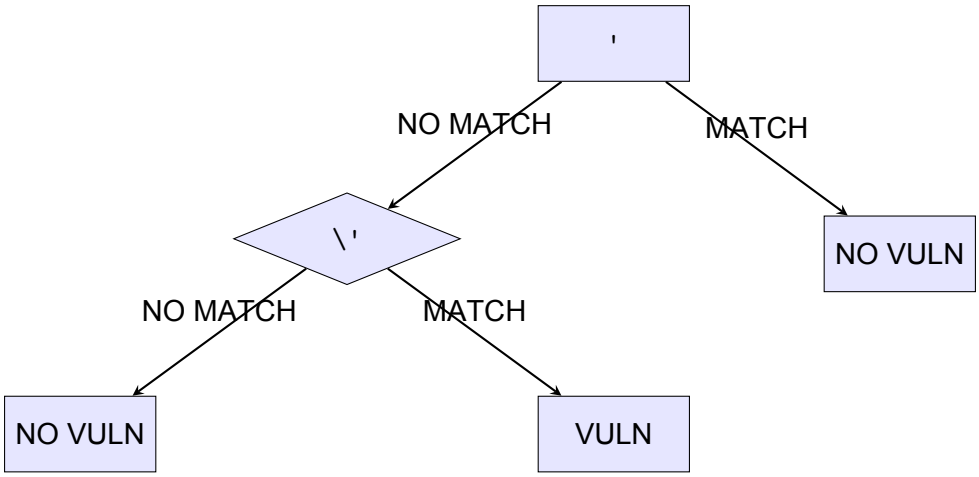


Figure 2.4: Diffing scan logic

To determine whether two responses differ meaningfully, the scanner treats responses as collections of attributes (e.g., status code, word count, content type). A difference is considered significant if:

- It is consistently different between the first and second probe.
- It remains stable across repeated tests.

Using this method, the scanner can infer processing contexts by asking structured questions, such as:

- Am I inside a single-quoted string?
- Am I in a numeric context?
- Am I being processed within a JSON value?
- Can I invoke a generic function?

These questions are tested using corresponding probe pairs, as demonstrated in Table 2.1.

Question	**Prob pair**
Am I in a single-quoted string?	`z'\z vs. z'\' z`
Am I in a numeric context?	`x/0 vs. x/1`
Am I in a file path?	`./../x vs. ././x`
Am I a function invocation?	`sprintfg vs. sprintf`
Am I in a JSON value?	`"a"," vs. "a":"`
How can I concatenate?	`z"z"z vs. z"||"z`
Can I call a generic function?	`"+abz(1)+" vs. "+abs(1)+"`

Table 2.1: Probe pair examples for structured scanning

With this methodology established, the *DiffingScan* class is responsible for implementing these scanning techniques. It extends the *ParamScan* class, which provides a framework for both passive and active scanning.

The core functionality is divided into three key methods:

- ***exploreAvailableFunctions***: Generates and tests probe pairs to detect function injection vulnerabilities.

- ***tryIncrementAttack***: Iteratively modifies numeric inputs to uncover sequential vulnerabilities.
- ***findReflectionIssues***: Analyzes reflection-based vulnerabilities by injecting various syntax elements.

The *exploreAvailableFunctions* method takes in parameters such as a *PayloadInjector*, an *Attack*, and optional prefix/suffix configurations. It constructs test payloads for different execution contexts, such as SQL, Ruby, and Python, generating both valid and invalid function calls (e.g., `abs(1)` vs. `abz(1)`). If a valid function executes while an invalid one does not, the scanner infers function injection potential.

The *tryIncrementAttack* method focuses on numerical vulnerabilities, sending payloads that slightly modify values (e.g., testing `x/0` vs. `x/1`). This approach can detect issues like divide-by-zero errors or application logic flaws.

The most critical method, *findReflectionIssues*, begins by running an interference scan. It then creates a *PayloadInjector*, which will be used for injecting different test values into the request. The first step in understanding how the application responds is to retrieve the base value (i.e. the original input found at the insertion point) so that it can be used for comparison.

To create a reference point, the method sets up a ”soft baseline,” which represents a normal request with minimal modifications. If the global setting ”*ignore baseresponse*” is enabled, this baseline is built using an empty *Attack* that contains only the base value itself. Otherwise, it includes the actual server response, allowing for a more accurate comparison. This soft baseline helps determine whether changes to the input lead to meaningful differences in how the application reacts.

Before performing more complex tests, the method runs a quick initial check using a crude fuzzing attack. This involves injecting a string with various special characters to see if the response changes in any noticeable way. If the response remains too similar to the baseline, it suggests that the input field may not process user input in a way that is useful for further testing, so the method stops early to avoid unnecessary work. However, if the response does change, it indicates that the application might be reacting to special characters, making it worthwhile to continue testing.

To further refine the analysis, the method then establishes a ”hard baseline” by sending an empty payload. If the response to this empty *Attack* is significantly different from the crude fuzz test, it suggests that the application is sensitive to certain input variations. This provides a stronger foundation for detecting syntax-related vulnerabilities.

Once the baseline comparisons are complete, the method begins systematically injecting different types of test payloads. These include escape sequences and special characters to check for syntax errors, concatenation operators to see how input is processed within expressions, and function injection attempts to determine if the application executes code dynamically. Additionally, JSON-specific payloads are used to test whether the input is interpreted as structured data, and context-aware tests are performed to check for behaviors related to technologies like MongoDB, SQL, or template engines. If any of these probes reveal inconsistent behavior, the method continues refining its approach, adjusting the payloads and testing additional variations to confirm whether a vulnerability exists.

If a clear discrepancy is detected, such as a change in response structure, an error message, or unexpected behavior, the method compiles the results and reports a potential security issue.

2.4 Review of WebSocket Security Tools

This section will review the existing work on WebSocket security testing and compare them with our proposal.

Initial work on WebSocket security was presented by *M. Shema et al.*, who were among the first to highlight that “WebSockets still fall victim to ‘old’ threats” [25]. However, this talk occurred during the early stages of the protocol’s adoption, a time when widespread usage was limited. As a result, the discussion around security concerns was relatively underdeveloped, with much of the focus on traditional threats such as Man-in-the-Middle (MITM) attacks and Denial-of-Service (DoS) vulnerabilities. The lack of awareness of WebSocket traffic by security devices (firewalls, IDS and IPS), was also mentioned as a factor that could allow malicious traffic to go undetected.

The first mainstream security testing framework to incorporate WebSocket testing was Zed Attack Proxy (ZAP). The work developed by *R. Koch* [15] enabled testers to not only inspect WebSocket communication but also modify outgoing message payloads. This

breakthrough also allowed for the fuzzing of messages using a list file; however, the results still required manual analysis. One notable observation in Koch's work was the difficulty of automating vulnerability detection in WebSockets:

> "With WebSockets vulnerability detection is not as easy as with HTTP. When you send a HTTP request there is always a response, where you can easily detect changed output, or slow response times. With WebSockets, communication is bi-directional and asynchronous. There is no request/response pattern, although a sub-protocol in use may define such. As a result automated detection is fairly hard and has to be done manually. However, tools can aid the search for vulnerabilities. The approach to take is similar to stored XSS attacks, where there is often no indication of a vulnerability. While the attack payload is stored in a database, it may appear on any page of the website.
>
> If there exists a request-response pattern in the WebSocket communication, you can measure the response time after sending crafted requests. Like with Blind SQL-injection attacks, one could send values that delay the execution of database or file system queries. If the execution slowed down, a hidden vulnerability may be detected." [15].

H. Kuosmanen [26] developed a tool aimed at testing common security vulnerabilities in WebSocket implementations. The tool provides functionality to establish and manage WebSocket connections, manipulate request headers, send and receive messages, and maintain active connections using ping/pong frames. It also supports logging WebSocket communication, interacting with the user via input prompts, and working with HTTP proxies. However, the tool does not incorporate automated vulnerability detection or validation mechanisms. Instead, it serves as a manual testing utility, requiring users to craft and analyze WebSocket messages, a process that, as noted by the author, demands expertise in the WebSocket protocol. This manual approach also makes it cumbersome for testing applications using layered protocols like Socket.IO, which require a specific sequence of handshake messages before fuzzing can begin. Despite this limitation, *Kuosmanen* briefly discusses server-side vulnerabilities in his work. However, he does not provide an in-depth analysis, merely acknowledging their existence and suggesting that they can be identified through fuzzing messages with varied payloads and analyzing response errors. He further notes that exhaustively testing all possible payloads for server-side vulnerabilities is inefficient.

A. Riancho [5] developed a lightweight WebSocket fuzzer that has been widely cited in related research. The tool provides a simple method for fuzzing WebSocket messages, requiring users to manually supply a wordlist for the process, or use the default one. While it also includes functionality to analyze results, this analysis can be somewhat complex. Still, it is also possible to manually inspect the results. Additionally, the fuzzer supports sending a prerequisite message before each payload, which is a step towards handling stateful interactions. However, it is primarily designed for applications that use JSON as the serialization format. While it can be adapted for other formats or more complex, multi-step handshakes, this requires custom scripting, making it less adaptable out-of-the-box for protocols like Socket.IO.

Several years later, *A. Hauser* [27] introduced a similar tool with comparable functionality. Like its predecessor, it enables testers to supply a list of payloads, along with cookies and authentication headers, for WebSocket fuzzing. Additionally, it supports the inclusion of prerequisite messages before each payload to simulate real-world communication flows. However, response analysis remains a manual process, relying on a web proxy for inspection. Furthermore, the tool is primarily designed for JSON-based WebSocket messages and is not inherently equipped to handle the specific handshake and messaging sequences of more structured subprotocols.

In 2019, *M. Fowl et al.* [28] addressed the same problem as this work, but proposed a different approach. Their solution involved translating WebSocket messages into HTTP requests and vice versa, effectively creating a fuzzing harness. This technique allowed traditional web application security tools, such as SQLMap, to be used directly for WebSocket-heavy applications. Nevertheless, the approach is limited by its dependence on HTTP translation, which may fail for WebSocket interactions without direct HTTP equivalents. For instance, it is unsuitable for protocols like Socket.IO that require a specific handshake and messaging sequence that has no direct HTTP counterpart. Additionally, its effectiveness is constrained by the capabilities of the underlying tools; for instance, when used with SQLMap, it is restricted to detecting SQL injection vulnerabilities. Automation is likewise dependent on the underlying tool, requiring users to configure and coordinate multiple tools if they wish to test for a broader range of vulnerabilities.

A. Happe [6] tackled the same challenge from a different angle, developing a lightweight tool based on the Kitty fuzzing framework. Kitty, which primarily supports mutation-based fuzzing, generates input by mutating existing valid data structures rather than sending

purely random or arbitrary payloads. In Happe's implementation, these mutated WebSocket messages were used to probe for vulnerabilities. To mitigate issues with persistent connections, the tool reopened the WebSocket connection after each message, avoiding disconnect problems at the cost of reduced performance. Since he had access to server-side logs during testing, his tool omitted built-in logging or response monitoring mechanisms.

A significant advancement in WebSocket security research was made by *E. Elbieh* [4], who noted that WebSockets had received minimal attention in the security domain. His research aimed to address key challenges, including the discovery of WebSocket-capable endpoints, the identification of server-side software, and the assessment of server vulnerabilities. While the first two contributions improved WebSocket reconnaissance, the vulnerability assessment was limited to identifying known Common Vulnerabilities and Exposures (CVEs) and was not designed for fuzzing complex application logic within stateful protocols.

Researchers at Doyensec, following an engagement with a client whose web application relied heavily on WebSockets, developed a tool with an interactive REPL interface that is user-friendly and easily automated [7]. While designed for penetration testing, its functionality is largely limited to sending and receiving WebSocket messages interactively, customizing headers and ping/pong messages, and supporting plug-ins for automating complex interaction scenarios. Nonetheless, it introduces some interesting features, such as displaying message opcodes and injecting fake ping messages.

In 2023, *E. Ward* introduced SocketSleuth [8], the first Burp Suite extension specifically designed for WebSocket testing. This tool emerged due to the limitations of existing WebSocket testing tools and the lack of automation in Burp Suite, such as Intruder functionality and match-and-replace rules. Ward's work introduced key improvements, including an optimized WebSocket history, an Intruder-style fuzzer, and match-and-replace rules, enhancing WebSocket security testing. Although it can leverage an existing, proxied connection to bypass complex handshake requirements, its utility can be hampered by connection instability during fuzzing. Like other tools, SocketSleuth still requires manual analysis of the results, which can limit its efficiency in large-scale or repetitive testing scenarios, especially for complex protocols.

Around the same time, PortSwigger released WebSocket Turbo Intruder, a Burp Suite extension for fuzzing WebSocket messages using custom Python scripts [9]. It allows

testers to modify attack logic and automate message sending, providing more flexibility for WebSocket security assessments. Its scriptable nature allows it to handle complex protocols like Socket.IO, but it places the entire burden of implementing the protocol-specific logic and performing the response analysis on the user. Despite its advantages, this tool also shares the limitation of requiring manual analysis of the results.

Importantly, while several existing tools explore WebSocket fuzzing, none of them are explicitly designed to detect server-side vulnerabilities in an automated way. Our work specifically addresses this gap by implementing a server-side vulnerability detection framework that is integrated into Burp Suite.

Tool/Author	Fuzzing	Integration	Payload Gen.	Prerequisite messages	Response Analysis	Key Feature
R. Koch [15]	Yes	ZAP	Manual	N/A	Manual	Basic WebSocket inspection
H. Kuosmanen [26]	No	Standalone	N/A	N/A	N/A	Message exchange only
A. Riancho [5]	Limited (JSON)	Standalone	Manual	Yes	Manual	Lightweight JSON fuzzer
A. Hauser [27]	Limited (JSON)	Standalone	Manual	Yes	Manual	Lightweight JSON fuzzer
M. Fowl et al. [28]	Indirect (via HTTP)	Standalone	Depends on external tools	No	Automated	WebSocket to HTTP translation
A. Happe [6]	Yes	Standalone	Automated	No	Manual	Uses Kitty framework
E. Elbieh [4]	No	Standalone	N/A	N/A	Limited (known CVEs)	Reconnaissance-focused
Doyensec [7]	Yes (via scripting)	Standalone	Manual	No	Manual	Interactive REPL
E. Ward [8]	Yes	Burp Suite	Manual	No	Manual	Match & replace rules
Port Swigger	Yes	Burp Suite	Manual	No	Manual	Custom Python fuzzing
Our Solution	**Yes**	**Burp Suite**	**Automated**	**Yes**	**Automated**	**Automation**

Table 2.2: Comparison of WebSocket Security Testing Tools

Table 2.2 summarizes the results of our review of existing WebSocket security testing tools, providing a direct comparison with our proposed solution. As illustrated, most existing tools lack automation in either payload generation, response analysis, or both, placing a significant manual burden on testers. None of the reviewed tools incorporate WebSocket-specific metrics for response analysis, which limits their ability to fully capture the unique characteristics of WebSocket communication. They also tend to lack features tailored to real-world WebSocket scenarios and often rely on HTTP-centric fuzzing approaches, failing to treat WebSocket traffic as a distinct protocol. Another key limitation is the absence of true stateful analysis, making it impossible to reliably correlate received responses with the exact sent messages that triggered them, which is useful to understand which payloads caused changes. Our solution addresses these gaps by automating both fuzzing and response analysis, integrating seamlessly into Burp Suite, the industry-standard security testing platform, and focusing specifically on detecting server-side vulnerabilities in WebSocket applications with the protocol-specific precision they require.

Chapter 3

System Architecture

The security assessment of WebSocket-based applications remains significantly underdeveloped compared to traditional HTTP-based testing. Existing methodologies and tools predominantly focus on request-response paradigms, which do not adequately accommodate the complexities introduced by WebSocket communication. Unlike HTTP, WebSockets establish persistent, bidirectional connections, necessitating fundamentally different techniques for effective vulnerability detection.

Traditional web security tools, including automated scanners, are not suited for WebSocket testing due to their reliance on deterministic response patterns and fixed insertion points. This methodological gap increases the likelihood of undetected vulnerabilities in modern real-time applications. To address this limitation, we propose an automated WebSocket fuzzing framework designed to detect vulnerabilities dynamically and efficiently.

As previously discussed in Section 1.2, the framework is designed to meet the following four core objectives:

- Develop a WebSocket security testing procedure that aligns with best practices in offensive security and penetration testing methodologies.
- Establish an evaluation methodology applicable to various WebSocket implementations, ensuring compatibility across different use cases.
- Implement a modular and maintainable automated WebSocket security assessment framework to dynamically identify vulnerabilities, designed to allow for seamless updates as WebSocket technology evolves.

To address this, our proposed architecture extends the BPS to incorporate support for WebSocket communication. This integration required addressing two key challenges:

first, adapting the scanner's HTTP-centric fuzzing mechanisms to accommodate the persistent and asynchronous nature of WebSocket connections; and second, redefining how responses are analyzed in the absence of traditional HTTP semantics such as status codes or headers. To meet these challenges, the architecture introduces new logic for WebSocket message management while also enhancing the existing vulnerability detection capabilities to operate effectively in a WebSocket context.

At a high level, the architecture comprises two primary functional units:

- **WebSocket Message Handling**: This unit is responsible for establishing WebSocket connections, sending messages, and capturing responses. It replaces the traditional HTTP request-response model with a WebSocket-specific approach, ensuring that the asynchronous and stateful nature of WebSocket communication is properly handled.
- **Vulnerability Detection**: This unit leverages the existing BPS framework to analyze the responses captured by the WebSocket Message Handler. It applies the same fuzzing techniques used for HTTP-based vulnerabilities but adapts them to the WebSocket context. This includes identifying anomalies in response behavior, detecting potential injection vulnerabilities, and classifying the severity of detected issues.

Figure 3.1 illustrates the modified BPS architecture, highlighting in yellow the components modified or added to support WebSocket functionality.

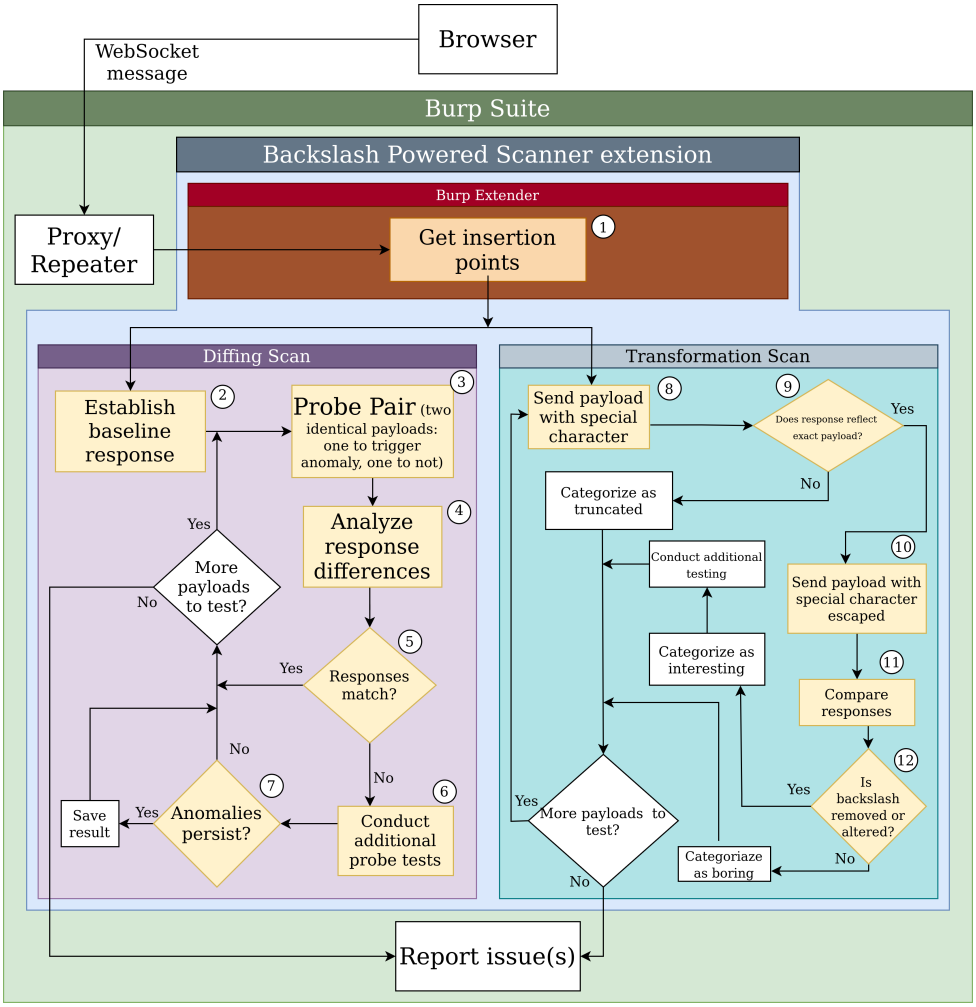


Figure 3.1: Modified components of the Backslash Powered Scanner architecture

3.1 WebSocket Message Handling

The WebSocket Message Handler is a core component of our system, responsible for managing the WebSocket communication flow. Unlike HTTP, which follows a clear request-response model where a single request triggers a corresponding response, WebSocket communication is bidirectional and asynchronous. This introduces significant challenges for security testing, as responses may not arrive in a predictable order or may vary depending on the application's state.

The need to develop this custom handler arose from limitations in Burp Suite's Montoya API for WebSockets. While Burp's HTTP API is mature and feature-rich, its WebSocket support is relatively recent and lacks equivalent functionality for comprehensive message handling and analysis.

As discussed in Chapter 2.4, there is currently no universally standardized method for fuzzing WebSocket messages. At a fundamental level, however, the objective remains

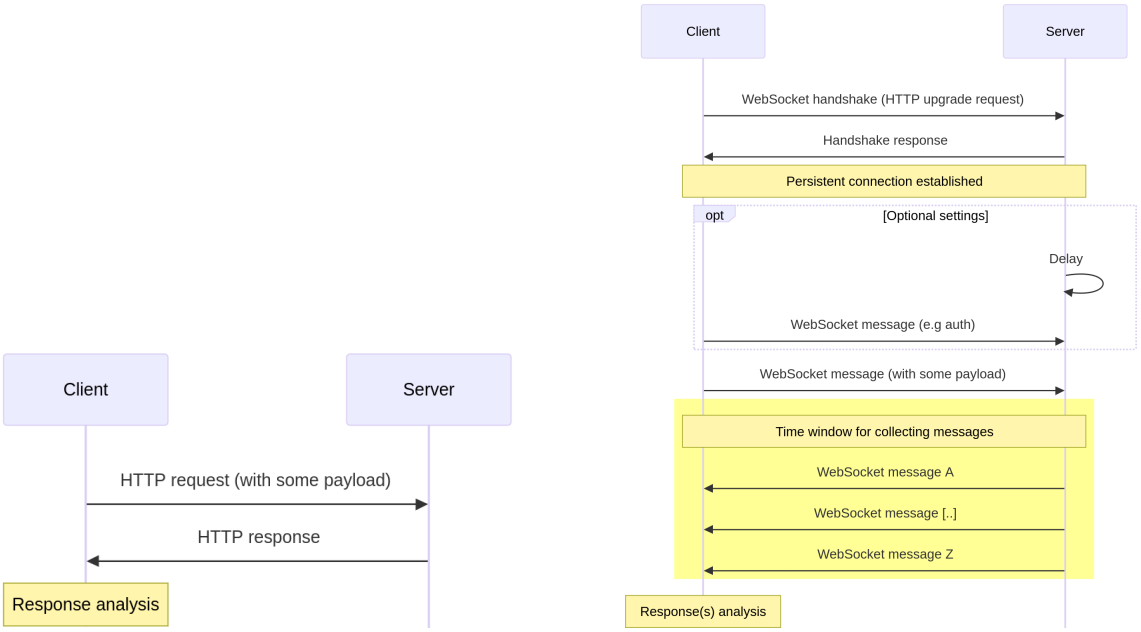
consistent: send a message, observe the server's response, and analyze its behavior by comparing it with other responses. To address this, we implemented a novel approach for fuzzing WebSocket messages by listening for incoming responses during a predefined timeout period and capturing all received messages along with their metadata, such as format and timestamps.

This process raised several design questions, which we resolved with a focus on our specific use case. Rather than overcomplicating the implementation with excessive features, we concentrated on building a solid and extensible foundation for WebSocket fuzzing that can be enhanced with additional capabilities in the future.

To address these challenges, the WebSocket Message Handler performs the following tasks:

- **Connection Management**: The handler establishes a WebSocket connection with the target server using the HTTP upgrade request. This allows for the use of cookies and other authentication mechanisms that may be necessary for the connection. A new connection is opened for each test, to avoid side effects from earlier payloads that might affect the application's state.
- **Payload Sending**: With a connection established, the handler sends a WebSocket message with a given payload. If needed, it is able to wait and/or send prerequisite messages before the actual payload (e.g. for authentication).
- **Response Capture**: The handler captures all responses received from the server within a configurable time window. This ensures that all relevant messages are recorded for analysis, even if they arrive asynchronously.
- **Message Storage**: The handler stores the captured responses in a structured format, including metadata such as message type (text or binary) and response time. This structured approach simplifies the correlation of messages and enables more accurate behavioral analysis.

Figure 3.2 provides a clearer illustration of our approach to capture WebSocket messages. The same illustration is shown, for comparison, for HTTP, which highlights the added complexity involved in fuzzing WebSocket traffic.



(a) HTTP **(b)** WebSockets

Figure 3.2: Sending a payload and analyzing the response in HTTP and our approach to WebSockets

3.2 Vulnerability Detection

The Vulnerability Detection unit leverages the existing BPS framework to fuzz the target and analyze the responses, through the WebSocket Message Handler. The BPS framework is known for its powerful fuzzing capabilities, which are designed to detect both known and unknown vulnerabilities in web applications. However, these capabilities were originally designed for HTTP-based communication and needed to be adapted for WebSocket.

The key modifications made to the BPS framework include:

- **Response Analysis**: The BPS framework analyzes responses based on metrics such as response length, status code, and headers. Since WebSocket responses do not have these traditional HTTP attributes, we introduced new metrics tailored to WebSocket communication, such as message count, message types (text or binary), and message lengths.
- **Scans**: This component incorporates the original scan logic and flow of the BPS framework, including transformation scans and diffing scans, but adapts them to operate effectively over WebSocket communication.

Chapter 4

Implementation

As outlined in Chapter 3, the system architecture was designed to support flexible, automated WebSocket fuzzing. This chapter describes the implementation process, which involved building several components from scratch and adapting existing ones to meet the unique requirements of WebSocket-based communication. The project was implemented in Java to maintain compatibility with the original BPS, which is also Java-based.

4.1 WebSocket Message Handling

WebSocket communication differs substantially from traditional HTTP due to its bidirectional and asynchronous nature. Since the original BPS was tailored for synchronous HTTP request-response workflows, adapting it for WebSocket testing required introducing a dedicated abstraction layer. This was particularly important given the current limitations of Burp Suite's Montoya API for handling WebSocket interactions, as previously discussed in Section 3.1.

To address this, we implemented a custom component referred to as the *WebSocket Handler*. This wrapper encapsulates the full lifecycle of a WebSocket interaction within the fuzzing workflow, managing connection establishment, payload injection, message timing, and structured response collection.

4.1.1 WebSocket Handler Wrapper

A critical aspect of this workflow is payload injection, which first required solving the fundamental challenge of defining where to inject test data within a WebSocket message.

Unlike HTTP, Burp Suite's Montoya API does not provide a direct equivalent to the *ScannerInsertionPoint* interface for WebSocket messages, leaving the location of payload injection ambiguous. To support automated and flexible fuzzing, we designed a custom injection strategy. When a message is identified as using JSON or *Socket.IO* format, the scanner automatically parses its structure and fuzzes individual fields. For unstructured messages, the tool searches for a special marker pattern (`FU...ZZ`) and injects payloads at the specified location. If neither condition applies, the entire message is fuzzed as a single unit. This strategy is visualized in Box 1 of the architecture diagram (Figure 3.1).

With the payload injection points established, the handler then manages the full lifecycle of the message exchange, from sending the payload to meticulously tracking all incoming server responses.

Internally, the WebSocket Handler maintains a traceable record of all received responses, capturing details such as message type, content, and timing. This enables the tool to correlate sent payloads with observed responses, which is essential for detecting server-side anomalies in asynchronous communication environments.

To enhance flexibility and support real-world WebSocket implementations, the wrapper includes configurable options for advanced scenarios. It is possible to configure the desired time window for listening to incoming messages. It also allows the insertion of delays and prerequisite messages before the main payload is sent, functionality that is essential for compatibility with protocols such as *Socket.IO*, where message sequencing and session initialization are required for valid interactions. The configuration interface for these options is shown in Figure 4.1.

Figure 4.1: User-configurable options for response wait time, delay before payload, and prerequisite messages.

Overall, the WebSocket Handler acts as a protocol-aware bridge between the scanner and the target application. It is utilized throughout the fuzzing workflow, as illustrated in the architecture diagram (Figure 3.1). Specifically, it is involved in Box 2 to establish a behavioral baseline, and in Boxes 3, 6, 8, and 10 to manage payload delivery, response tracking, and result storage.

Implementing this complex message handling required careful consideration of how to integrate our code with the existing BPS framework.

During development, we considered whether to fully integrate our WebSocket-related functionality into the original Backslash Powered Scanner (BPS) codebase or to isolate it in separate modules. Although we demonstrated that a seamless integration was technically feasible, and even discussed the possibility with the original author, we ultimately chose to encapsulate all WebSocket-related logic in dedicated classes. This decision was made to avoid potential future conflicts, preserve the original HTTP functionality, and keep our modifications modular and maintainable.

To achieve this, we created parallel implementations of the core classes used in the original framework. This approach allowed us to mirror key behaviors while replacing HTTP-specific methods with WebSocket equivalents. In most cases, the necessary changes were straightforward, involving the substitution of HTTP calls with our custom WebSocket handler. Nonetheless, we conducted a thorough review of the entire codebase to ensure compatibility.

Some adjustments to the original logic were necessary, particularly in areas where WebSocket behavior diverges from HTTP. For instance, unlike HTTP's strict request-response model, WebSocket messages can yield multiple asynchronous responses. This required modifying parts of the analysis and tracking logic to accommodate such cases. Additionally, since the original code relied heavily on Burp Suite's Montoya API for HTTP, and no direct equivalents exist for many of these methods in the WebSocket API, we had to reimplement several features from scratch to support WebSocket functionality effectively.

4.2 Vulnerability Detection

WebSocket communication differs substantially from traditional HTTP, not only in the way messages are sent and received, but also in how responses must be interpreted and compared. These differences necessitate adjustments in both the metrics used for response comparison and the methodology for injecting payloads during fuzzing.

To accommodate these protocol-specific requirements, we developed a custom component referred to as the Vulnerability Detection Wrapper. This module encapsulates both the comparison metrics tailored for WebSocket communication and the adapted scanning logic originally implemented in the BPS.

4.2.1 Vulnerabilty Detection Wrapper

Effective vulnerability detection requires robust and context-aware comparison metrics. In the case of HTTP, responses are typically evaluated using well-defined attributes such as status codes, headers, and body content. However, these attributes are absent in WebSocket communication, which is instead characterized by asynchronous, stateful, and by being minimalistic in design, where a WebSocket message consists almost entirely of raw payload data, with virtually no accompanying metadata.

To address this, we introduced a set of metrics specifically designed for analyzing WebSocket responses:

- **Message Count**: The total number of messages received within a predefined time window after sending a payload.
- **Message Types**: The sequence of message types (e.g., text or binary) observed in the response stream.
- **Message Lengths**: The individual length of each received message.
- **Whitespace Frequency**: The number of space characters in each message.
- **HTML Tag Count**: The number of HTML tags present in each message.

Due to the current limitations of Burp Suite's WebSocket API, lower-level attributes such as opcodes are not yet accessible. These could offer additional insight into message semantics and behavior, and may be incorporated in future iterations of the framework as the API evolves. Importantly, the current architecture is modular and designed for extensibility, making the integration of new metrics straightforward.

As illustrated in the architecture diagram (Figure 3.1), these metrics are utilized at various stages of the system, specifically in Boxes 4, 5, 7, 11, and 12, where captured responses are evaluated and compared.

These WebSocket-specific metrics provide the analytical foundation for the core fuzzing strategies employed by the framework.

The core vulnerability detection capabilities rely on two primary fuzzing strategies: the *Transformation Scan* and the *Diffing Scan*. These techniques mirror the logic of their

HTTP-based counterparts in BPS but are adapted for the unique context of WebSocket communication.

To support this adaptation, we conducted a thorough review of the original implementation and introduced WebSocket-specific versions of all relevant classes. In most cases, the required modifications were relatively straightforward, involving structural substitutions rather than complete redesigns. Scans that are not applicable or meaningful in the context of WebSocket traffic, such as HTTP parameter pollution, were removed. However, the framework remains extensible, allowing for their reintroduction or the inclusion of new scan types as needed.

A key consideration in adapting the scans was the possibility of receiving multiple, asynchronous responses for a single payload. As such, parts of the analysis logic were modified to account for the complexity of correlating multiple messages with a single input.

Furthermore, due to the prevalence of structured message formats such as JSON and Socket.IO in WebSocket-based applications, our system includes automated support for escaping payloads when injected into such formats. This ensures that the message structure remains valid, thereby avoiding unintended parsing errors and improving the accuracy of the fuzzing process. This behavior aligns with the automated injection logic described in Section 4.1.1.

Lastly, the adaptation of these scans was guided by the same modular approach used for the WebSocket Handler Wrapper, oulined in Section 4.1.1, to ensure seamless compatibility with the original codebase.

Chapter 5

Evaluation

This chapter presents a comprehensive evaluation of the proposed automated fuzzing framework for WebSocket security. The primary objective of this evaluation is to assess the framework's efficacy in detecting various server-side vulnerabilities within WebSocket communications, particularly its ability to handle complex interaction patterns inherent in protocols like Socket.IO. To achieve this, the framework was tested against two distinct laboratory environments: the Damn Vulnerable Web Sockets (DVWS) application and a custom-developed Socket.IO-based lab.

The evaluation methodology focuses on demonstrating the framework's capability to identify well-understood vulnerability classes, showcasing its adaptability to different WebSocket implementations, and highlighting its advantages over existing tools, especially in scenarios requiring intricate handshake procedures and message sequencing. This chapter will detail the rationale for selecting specific vulnerabilities, describe the test environments, and present the results of the framework's application.

5.1 Vulnerability Selection and Rationale

The vulnerabilities chosen for evaluation - SQL Injection (SQLi), Server-Side Template Injection (SSTI), PHP Code Injection, and File Inclusion - are prevalent, high-impact server-side flaws, frequently featured in security benchmarks. Their selection was also guided by the fact that these vulnerability classes were already accounted for in the original Backslash Powered Scanner's logic and algorithms. A key objective was to validate the extended framework's ability to detect these specific, critical vulnerabilities within WebSocket contexts without altering the core detection engine, thereby testing the integrity and adaptability of the original tool's capabilities when applied to this new communication protocol.

- **SQL Injection (SQLi)**: A vulnerability where attackers inject malicious SQL statements into input fields, enabling unauthorized access to or manipulation of a database.
- **Server-Side Template Injection (SSTI)**: Occurs when untrusted user inputs are rendered in server-side templates, allowing attackers to execute arbitrary code on the server.
- **PHP Code Injection**: A flaw where unfiltered user input is passed directly into PHP functions like `eval()` or `include()`, letting attackers execute arbitrary PHP code.
- **File Inclusion**: An attacker tricks the application into including unintended files, which can lead to code execution or information disclosure.

The DVWS environment, together with the Socket.IO lab, provides test cases for these vulnerabilities, allowing for a baseline assessment of the tool's effectiveness in a controlled WebSocket setting.

5.2 Test Environment: Damn Vulnerable Web Sockets (DVWS)

To establish a performance baseline, the framework was first evaluated against Damn Vulnerable Web Sockets (DVWS), a deliberately insecure web application designed to help security professionals test WebSocket vulnerabilities in a controlled environment. To ensure a comprehensive evaluation of the framework's capabilities, the existing DVWS application was enhanced with two additional vulnerabilities, Server-Side Template Injection and PHP Code Injection, which were contributed back to the original DVWS project.

The following subsections detail the framework's performance against each test case and provide a comparative analysis against other security tools, highlighting the differences in workflow, automation, and detection capabilities.

5.2.1 Test Case 1: SQL Injection

- **Framework Performance**: DVWS provides an error-based SQL Injection vulnerability in a login page that accepts credentials via a JSON message. The framework was used to test this endpoint by simply initiating a scan on the message. Its built-in parser automatically identified the JSON structure and fuzzed the individual *username* and *password* fields with the payloads JSON escaped. After a few minutes, the tool successfully reported two distinct MySQL injection vulnerabilities, one for

each field, confirming the framework's effectiveness in automated fuzzing of structured WebSocket messages. The detection in the *auth_pass* field can be seen in Figure 5.7.

- **Comparative Analysis**: In contrast, other tools required significant manual intervention. With Socket Sleuth and WebSocket Turbo Intruder, detecting the vulnerability was possible but not automated. The process involved manually selecting payload lists and defining insertion points. While payloads could be sent, neither tool automatically flagged the vulnerability, requiring the user to manually analyze response logs to identify database errors. Furthermore, these tools were often only successful if the payload was inserted into a specific field (e.g., the password field), limiting their discovery potential. Riancho's WebSocket Fuzzer, after considerable setup effort, could also trigger the vulnerability but similarly relied on manual log analysis for detection. The WebSocket Harness approach, when paired with a dedicated tool like SQLmap, successfully identified the SQL injection, demonstrating the effectiveness of tool-chaining for specific vulnerability classes but lacking the all-in-one capability of the proposed framework. Finally, the fuzzer by A. Happe was unsuccessful in detecting this vulnerability.

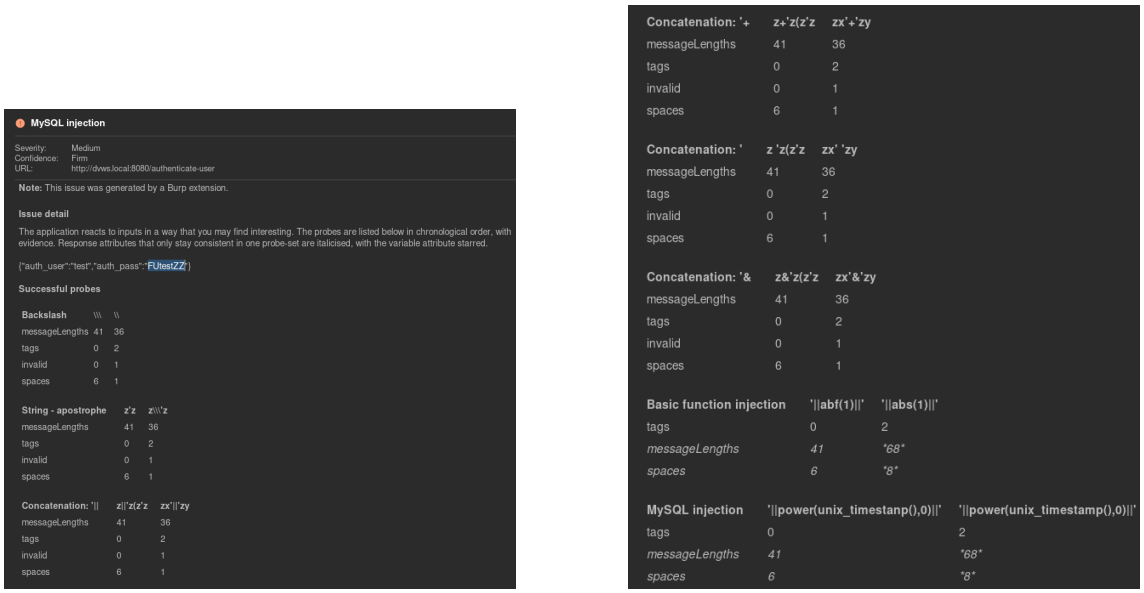


Figure 5.1: Detection of SQL Injection

5.2.2 Test Case 2: Server-Side Template Injection (SSTI)

- **Framework Performance**: An SSTI vulnerability using the Twig template engine was added to DVWS. The vulnerable page accepts a name as input and reflects it in

a greeting. Our framework was launched against a standard message containing a name. The scan results did not explicitly identify SSTI, but instead reported anomalous behavior, noting that input containing curly braces (`{}`) was handled differently by the server (Figure 5.2). Since curly braces are fundamental to many template engines, this finding serves as a strong indicator of a potential SSTI vulnerability, guiding the security tester toward a high-value area for minimal manual confirmation. This demonstrates the framework's power to identify suspicious behavior even when a definitive vulnerability signature is not matched.

- **Comparative Analysis**: Manual tools like Socket Sleuth, WebSocket Turbo Intruder, and Riancho's Fuzzer were able to trigger the SSTI vulnerability when supplied with an appropriate payload list. However, all of them failed to provide any automated notification of success. Detection was entirely dependent on the tester's manual inspection of logs to find evidence of template evaluation. The WebSocket Harness approach paired with SSTImap was, once again, effective and provided a rapid and accurate detection. The script by A. Happe was not successful. This comparison highlights a key advantage of the proposed framework: its ability to flag "interesting" behavior. When a concrete vulnerability can't be proved, the scanner can still provide valuable insight to the tester.

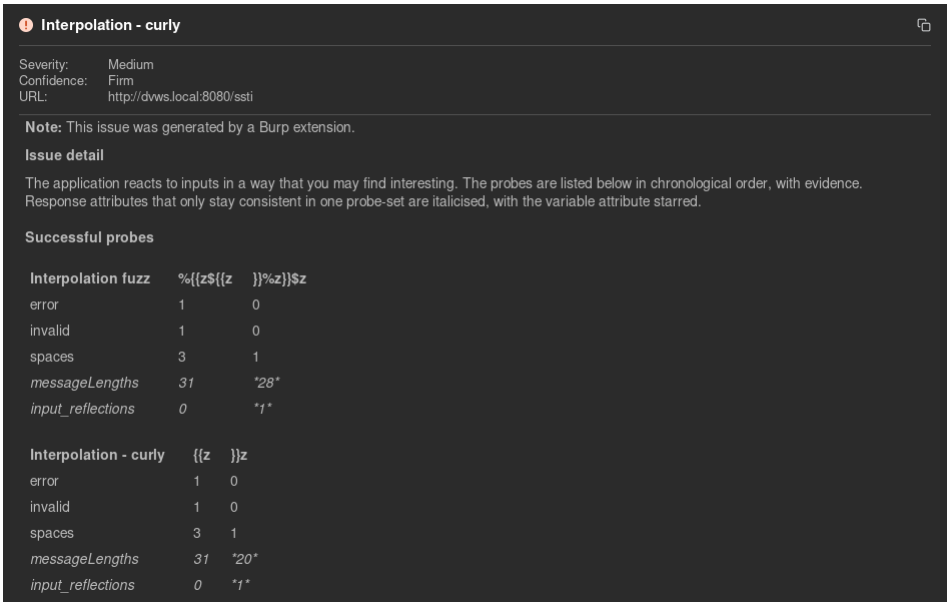


Figure 5.2: Detection of Anomalous Behavior Suggesting SSTI

5.2.3 Test Case 3: PHP Code Injection

- **Framework Performance**: A PHP Code Injection vulnerability was introduced via a page that accepts a mathematical operation and computes the result using the `eval()` function. When the framework was used to fuzz a basic mathematical expression (e.g., `7+7`), it correctly identified a comment injection (Figure 5.3), signaling that the server was processing input in an unusual way. Recognizing that a single number is also a valid input, a scan was launched against the input `7`. In this case, the framework successfully identified a conclusive PHP code injection vulnerability (Figure 5.4). This difference in detection stems from the original fuzzer's logic, which treats `7+7` as a string but processes `7` as a number, subjecting it to more intensive numerical analysis. This demonstrates that the nuanced detection logic of the original scanner was successfully preserved and applied in the WebSocket context.

- **Comparative Analysis**: Similar to the previous test cases, other tools could facilitate the discovery of this vulnerability but could not automate it. Manual analysis of logs in Socket Sleuth, Turbo Intruder, and Riancho's fuzzer revealed that the application did respond to PHP code injection payloads, but the tools themselves provided no automated detection of this behavior. The WebSocket Harness approach and the fuzzer by A. Happe were not effective in this scenario. The proposed framework stands out by not only detecting the vulnerability but also by demonstrating nuanced analysis that differentiates between input types (string vs. numeric) to uncover deeper flaws.

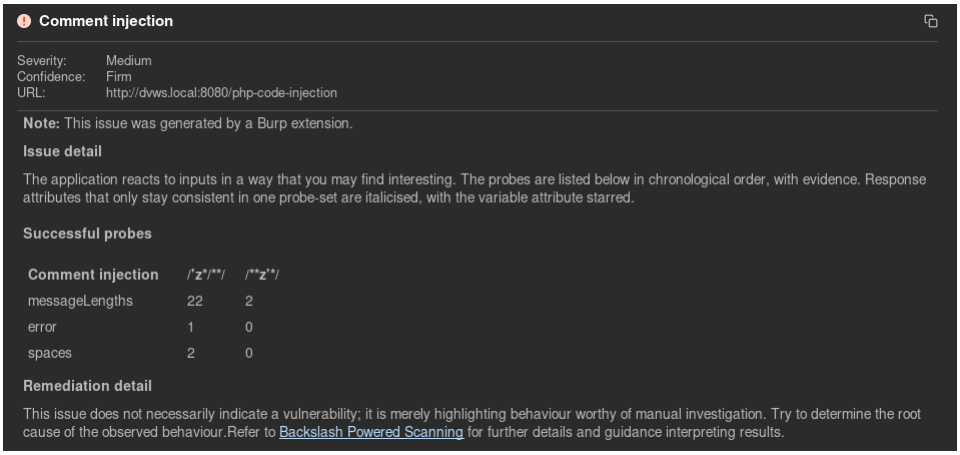


Figure 5.3: Detection of Anomalous Behavior

PHP injection

Severity: Medium
Confidence: Firm
URL: http://dvws.local:8080/php-code-injection

Note: This issue was generated by a Burp extension.

Issue detail

The application reacts to inputs in a way that you may find interesting. The probes are listed below in chronological order, with evidence. Response attributes that only stay consistent in one probe-set are italicised, with the variable attribute starred.

Successful probes

Divide by 0	/0	-0
messageLengths	22	1
error	1	0
spaces	2	0

Divide by expression	/(2-2)	/(1*1)
messageLengths	22	1
error	1	0
spaces	2	0

Basic function injection	/abf(1)	/abs(1)
messageLengths	22	1
error	1	0
spaces	2	0

PHP injection	/(int)phpversiom(),0)	/pow((int)phpversion(),0)
messageLengths	22	1
error	1	0
spaces	2	0

Comment injection	/*z*/**/	/**z**/
messageLengths	22	1
error	1	0
spaces	2	0

Remediation detail

This issue does not necessarily indicate a vulnerability; it is merely highlighting behaviour worthy of manual investigation. Try to determine the root cause of the observed behaviour.Refer to Backslash Powered Scanning for further details and guidance interpreting results.

Figure 5.4: Detection of PHP Injection

5.2.4 Test Case 4: File Inclusion

- **Framework Performance**: The final DVWS test case involved a File Inclusion vulnerability, where a user's selection loads a corresponding text file. The request message includes the path, such as `pages/games.txt`. To test this, the filename was replaced with the `FUZZ` marker, instructing the tool where to inject its payloads. The framework successfully detected a file path manipulation vulnerability (Figure 5.5), confirming its ability to handle user-guided fuzzing with insertion point markers.

- **Comparative Analysis**: With a custom wordlist and manual analysis, it was possible to identify the vulnerability using Socket Sleuth, WebSocket Turbo Intruder, and Riancho's fuzzer. However, these tools are highly dependent on the quality of the provided wordlist. For instance, if the wordlist did not test for the correct directory traversal depth (e.g., it tested with one less `../`), the vulnerability would be missed. The proposed framework's analysis, by contrast, is based on behavioral differences and is less reliant on having a ”perfect” payload, thus reducing the chances of missing such a vulnerability. The fuzzer by A. Happe was again unsuccessful and caused the WebSocket server to crash when a payload with a null byte was sent.

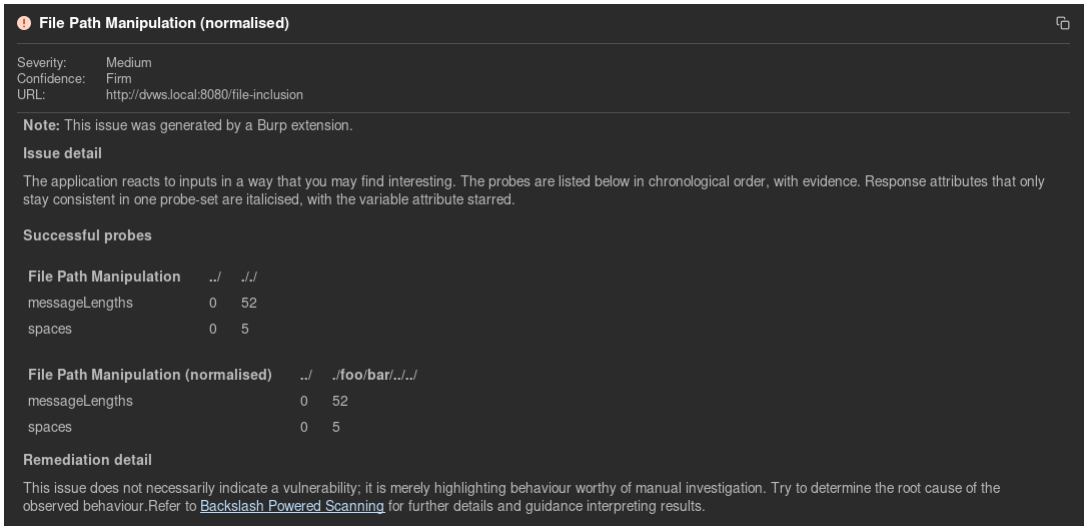


Figure 5.5: Detection of File Path Manipulation

5.3 Advanced Scenario: Custom Socket.IO Laboratory

To further evaluate the framework's capabilities, particularly its proficiency in handling more complex WebSocket interactions and subprotocols, a custom laboratory environment utilizing Socket.IO was developed.

Socket.IO is a library that enables real-time, bidirectional, and event-driven communication between web clients and servers. It builds upon the WebSocket protocol by providing a more feature-rich layer, including capabilities like automatic reconnections and broadcasting. The communication typically begins with an HTTP-based handshake that attempts to upgrade to a WebSocket connection, establishing a persistent channel. This abstraction simplifies the development of interactive applications.

Following the initial transport-level connection and handshake, the client initiates a connection to a Socket.IO namespace by sending a `40` packet (Engine.IO message type `4`, Socket.IO CONNECT type `0`). The server then acknowledges the successful establishment of this namespace connection by responding with its own `40` packet, which includes the session identifier (sid) for that namespace. Once this namespace connection is confirmed, both client and server can exchange custom event-based messages using the `42` packet type (Engine.IO message `4`, Socket.IO EVENT `2`). This packet typically contains a JSON array, with the first element being the event name and subsequent elements representing the data payload, facilitating structured and flexible real-time interactions.

5.3.1 Laboratory Setup and Rationale

The custom Socket.IO laboratory was designed to simulate a common use case: a login mechanism susceptible to SQL Injection. The rationale for creating this specific lab was twofold:

- **Complexity Demonstration**: Standard WebSocket fuzzers often struggle with protocols like Socket.IO that impose a specific sequence of initial messages or a handshake procedure before application-level data can be exchanged. This lab requires the testing tool to manage such a sequence.
- **Full Power Showcase**: This environment allows for the demonstration of the framework's advanced features, such as configuring prerequisite messages, initial delays, and specific listener timeouts, which are crucial for successfully interacting with and fuzzing Socket.IO applications.

5.3.2 Detecting SQL Injection in a Socket.IO Environment

The proposed framework was configured to interact with the custom Socket.IO lab and test its login functionality for SQL Injection vulnerabilities within the `username` and `password` fields. The process was as follows:

1. **Configuration for Socket.IO Interaction**: The necessary configurations, including initial delay, prerequisite message ("`40`"), and message listener timeout, were set within the framework's interface to align with the Socket.IO protocol requirements (as depicted in Figure 5.6).
2. **Fuzzing Process**: With the handshake parameters configured, the framework proceeded to fuzz the `username` and `password` fields of the login request message.
3. **Results and Detection**: After a period of scanning, the framework successfully identified SQL Injection vulnerabilities in both the username and password fields. The detection was based on anomalous responses from the server, indicative of database errors or altered application behavior consistent with successful SQL injection. These findings are illustrated in Figure 5.7a and Figure 5.7b, which show the reported vulnerabilities by the tool.

Figure 5.6: Configuration for Socket.IO Interaction

(a) Username field

(b) Password field

Figure 5.7: Detection of SQL Injection

This successful detection in a Socket.IO environment underscores the framework's adaptability and its capability to handle non-trivial WebSocket communication patterns, a significant advancement over many existing tools.

5.3.3 Comparative Analysis with Existing Tools

The custom Socket.IO laboratory presents a scenario where many existing WebSocket security testing tools, as reviewed in Chapter 2.4, would likely encounter significant difficulties or fail to detect the SQL injection vulnerability. The general reasons for this anticipated failure include the inability to optionally send prerequisite messages or having control over delaying the payload, and no clear way to match the sent payload with the messages received. A more detailed analysis of specific tools reveals the following:

- Socket Sleuth [8]:
  - Process: In Socket Sleuth, a message from the history is selected and sent to its Intruder-like component. Payload positions are manually selected, payload lists are manually provided, and an active WebSocket connection must be chosen for the attack.
  - Socket.IO Interaction: The ability to reuse an existing connection initially seems advantageous, as it bypasses the need for explicit delays or prerequisite message configuration for the Socket.IO handshake to be satisfied (assuming the connection was established correctly by the browser and captured). Payloads can be sent.
  - Limitations: A significant issue observed was that the WebSocket connection tended to drop after approximately five payloads were sent. Consequently, the majority of payloads in a larger list would have no effect. Even if the initial,

successful payloads were the correct ones to trigger a vulnerability, Socket Sleuth does not provide clear warnings or automated detection. The user is required to manually sift through a list of all sent and received messages to identify server errors or indications of a successful payload, a time-consuming and error-prone process.

- WebSocket Turbo Intruder [9]:
  - Process: This tool requires Python scripting for attack configuration, offering more flexibility for custom scenarios.
  - Socket.IO Interaction: For the Socket.IO lab, a Python script was developed to handle the prerequisite "40" message. After this, a payload list was added, and the attack was initiated. All messages, including the prerequisite and subsequent fuzz payloads, were correctly sent.
  - Limitations: Similar to Socket Sleuth, WebSocket Turbo Intruder necessitates manual analysis of all messages and responses. The user must look for errors or signs of payload success. Correlating specific sent messages with asynchronous responses can be challenging due to the unordered nature of WebSocket traffic, although scripting might offer ways to improve this correlation. The burden of detection and analysis remains squarely on the user.
- WebSocket Fuzzer (Andres Riancho) [5]:
  - Process: This tool, not having been updated for some time, required considerable effort to resolve requirement and compatibility issues, including code changes, before it could be run.
  - Socket.IO Interaction: After the necessary fixes, the tool was configured according to its instructions. It was capable of sending payloads in compliance with the Socket.IO protocol, including handling prerequisite messages if configured. The tool includes a default wordlist, which can be modified.
  - Limitations: Despite successfully sending payloads, the primary drawback was the need for manual log checking and response analysis. While the tool features an "analyze output" function, it was found to be of limited utility in this scenario. A positive aspect was that messages and their corresponding responses were generally well-correlated, making manual analysis slightly less ambiguous than with other tools, but the lack of automated detection remains a significant hurdle.

- WebSocket Harness [28]:
  - Limitations: This approach, which relies on translating WebSocket interactions into HTTP requests for traditional web application scanners, proved unsuitable for the Socket.IO lab. The harness did not provide a mechanism to handle the specific handshake and prerequisite messaging required by Socket.IO. As such, it was unable to establish a valid communication channel to effectively fuzz the application logic.

In summary, while some tools could be coaxed into sending payloads in the Socket.IO scenario, often with significant manual effort or scripting, none offered the automated detection and an easy handling of complex handshake protocols that the proposed framework provides. The manual analysis required by these tools, coupled with issues like connection instability or the inability to manage protocol-specific interactions, severely limits their effectiveness and efficiency in such environments. Therefore, the successful and automated identification of SQL injection in this Socket.IO lab by the proposed framework highlights its superior capability in navigating and testing complex, real-world WebSocket implementations.

5.4 Summary of Evaluation Findings

The evaluation conducted across both the standardized DVWS environment and the advanced Socket.IO laboratory comprehensively validates the effectiveness and versatility of the proposed framework. The results demonstrate that the core vulnerability detection logic of the Backslash Powered Scanner was successfully adapted for the WebSocket protocol, enabling the identification of a range of critical server-side flaws.

The framework consistently outperformed existing tools by providing automated detection, which significantly reduces the manual effort and potential for human error inherent in log analysis. Its ability to not only identify definitive vulnerabilities like SQL Injection but also to flag ”interesting” anomalous behavior, as seen in the SSTI test case, provides actionable intelligence that is invaluable for security testers. Furthermore, the successful testing of the custom Socket.IO lab highlights the framework’s most significant contribution: its capacity to handle complex, stateful communication protocols that require specific handshake sequences. This capability is largely absent in other evaluated tools, which either fail entirely or require considerable manual scripting and intervention.

Table 5.1 provides a consolidated overview of the comparative performance of the evaluated tools against the test cases.

Tool/Framework	DVWS SQLi	DVWS SSTI	DVWS PHP Inj.	DVWS File Incl.	Socket.IO SQLi
Proposed Framework	✓	✓	✓	✓	✓
Socket Sleuth	~	~	~	~	~
WebSocket Turbo Intruder	~	~	~	~	~
Riancho's Fuzzer	~	~	~	~	~
WebSocket Harness	✓	✓	✗	✗	✗
A. Happe's Fuzzer	✗	✗	✗	~	✗

Table 5.1: Comparative Performance of WebSocket Security Testing Tools

Legend:

- ✓: Automated Detection / Strong Indication
- ~: Manual Detection Possible
- ✗: Failed / Not Applicable

In conclusion, the evaluation confirms that the framework meets its primary design objectives. It provides an automated, adaptable, and robust solution for WebSocket vulnerability detection that effectively addresses the limitations of existing tools, particularly in the context of modern, complex real-time applications.

Chapter 6

Conclusion

The proliferation of real-time applications has cemented the WebSocket protocol as a solid component of modern web infrastructure. However, the security tools and methodologies for this technology have not kept pace with its adoption. The stateful, asynchronous, and bidirectional nature of WebSockets presents significant challenges for traditional security testing approaches, which are overwhelmingly designed for the stateless request-response model of HTTP. This gap leaves a vast number of applications potentially exposed to vulnerabilities that standard scanners cannot detect.

This thesis confronted these challenges by extending the powerful Backslash Powered Scanner to the WebSocket domain. Our work aimed to create an automated fuzzing framework capable of dynamically identifying server-side vulnerabilities in WebSocket communications without requiring manual intervention, bridging a gap in the current security testing landscape.

6.1 Research Summary

Our research began with a thorough analysis of the WebSocket protocol and the existing landscape of security testing tools. This initial investigation confirmed that most available tools are manual or semi-automated, lack sophisticated detection mechanisms, and struggle with the protocol's inherent complexities, especially when dealing with subprotocols like Socket.IO. We identified a clear need for an automated fuzzer that could understand and adapt to the nuances of WebSocket interactions.

With these requirements defined, we designed and implemented an extension for the Backslash Powered Scanner. The core of our work involved developing a custom WebSocket Message Handler to manage the entire communication lifecycle, from connection

establishment and state management to payload injection and asynchronous response collection. We also engineered a Vulnerability Detection Wrapper with WebSocket-specific comparison metrics (e.g., message count, types, and lengths) to analyze server behavior effectively. The framework was built to automatically parse structured formats like JSON and handle custom insertion points, ensuring broad applicability. The code developed as part of this thesis was successfully submitted to, and merged into, the original Backslash Powered Scanner project, validating the quality and utility of the implementation

To validate our solution, we conducted a rigorous evaluation using two distinct environments: the deliberately insecure Damn Vulnerable Web Sockets (DVWS) application, which we enhanced with new vulnerabilities, and a custom-built Socket.IO laboratory. The framework successfully and automatically identified a range of server-side vulnerabilities, including SQL Injection, Server-Side Template Injection (by flagging anomalous behavior), PHP Code Injection, and File Path Manipulation. Most notably, it succeeded in the complex Socket.IO environment, a scenario where other compared tools failed due to the complex protocol interaction.

6.2 Current Limitations

Despite the framework's success, several limitations must be acknowledged. These originate from the inherent complexity of real-world WebSocket implementations and the current state of supporting APIs.

One primary limitation is the difficulty in analyzing proprietary or non-standard message formats. For instance, some applications may send JSON-formatted messages but receive responses in a compressed or obfuscated binary format (e.g. Discord), making automated analysis exceedingly difficult. Similarly, implementations that require a unique signature or timestamp for every message (e.g. Binance) pose a significant challenge to automated fuzzing. While our framework handles common formats, binary messages also remain a largely unsolved problem and are not currently supported.

From a functional standpoint, the current implementation creates a new WebSocket connection for each test to ensure a clean state and avoid cross-payload interference. However, it lacks the ability to reuse an existing connection. This feature would be beneficial in scenarios where maintaining a persistent session is crucial. Furthermore, the framework currently initiates scans from an intercepted upgrade request. It does not yet support

opening a WebSocket connection directly from a `ws://` or `wss://` URL, which would enhance its flexibility as a standalone testing tool.

Finally, the response analysis metrics, while effective, are constrained by the information exposed by Burp Suite's API. Deeper insights could be gained by analyzing lower-level protocol details, such as WebSocket opcodes, which are not yet accessible through the current API.

6.3 Future Work

The limitations identified present clear and promising directions for future development. Addressing these points would significantly enhance the framework's power, flexibility, and applicability.

First, expanding the analytical capabilities to handle more diverse data formats is a high priority. Future work should focus on integrating parsers and decompilers for common binary formats and developing heuristics to identify patterns in custom protocols. This would broaden the framework's utility against the heterogeneous landscape of real-world WebSocket applications.

Next, several key functional enhancements are planned. Implementing the ability to reuse existing WebSocket connections would improve efficiency and enable the testing of applications with complex state-dependent logic. Complementing this, we intend to add functionality to initiate connections directly from a WebSocket URL. This would untether the tool from the proxy workflow and allow for more direct, scriptable security assessments.

The refinement of response analysis metrics is another crucial area for improvement. As Burp Suite's API evolves, we plan to incorporate lower-level protocol attributes, such as opcodes and fragmentation details, into the detection engine. This would enable the framework to spot more subtle anomalies and uncover new classes of vulnerabilities.

Finally, the true test of a security tool is its application in the wild. Continued development will be guided by user feedback and real-world use cases. As the tool is adopted by more security professionals, their insights will be invaluable for identifying new challenges and prioritizing future enhancements, ensuring the framework remains a relevant and effective solution for WebSocket security.

6.4 Conclusions

This thesis set out to address the significant gap in automated security testing for WebSocket-based applications. The traditional tools and methodologies, rooted in the HTTP paradigm, are ill-equipped to handle the dynamic and stateful nature of WebSocket communication, leaving a critical attack surface underexplored.

Our primary contribution is an automated fuzzing framework, built upon the Backslash Powered Scanner, that is specifically designed to detect server-side vulnerabilities in WebSocket implementations. By creating a robust WebSocket handler and tailoring the scanner's powerful behavioral analysis engine to this new context, we have developed a tool that successfully identifies high-impact vulnerabilities with minimal user intervention. Our solution proved its superiority over existing tools, particularly in its ability to navigate complex, stateful protocols like Socket.IO, which are a common stumbling block for other fuzzers.

By successfully adapting a sophisticated scanning engine to the WebSocket protocol, this work not only provides a practical tool for security professionals but also lays a foundation for future research in this domain. The successful merger of this work into the Backslash Powered Scanner project serves as a testament to its practical value and contribution to the security community.

References

[1] I. Fette and A. Melnikov, “The WebSocket Protocol,” Dec. 2011, RFC 6455. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6455 [Cited on pages 1, 6, and 8.]

[2] P. Murley, Z. Ma, J. Mason, M. Bailey, and A. Kharraz, “Websocket adoption and the landscape of the real-time web,” in *Proceedings of the Web Conference 2021*, ser. WWW ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1192–1203. [Online]. Available: https://doi.org/10.1145/3442381.3450063 [Cited on page 1.]

[3] PortSwigger, “Testing for websockets security vulnerabilities,” 2024. [Online]. Available: https://portswigger.net/web-security/websockets#websockets-security-vulnerabilities [Cited on page 1.]

[4] E. Elbieh, “We’re not in http anymore: Investigating websocket server security,” in *OWASP Global AppSec USA 2021*, 2021, conference presentation. [Online]. Available: https://www.youtube.com/watch?v=bMFP71UAbPo [Cited on pages 1, 26, and 27.]

[5] A. Riancho, “Websocket fuzzer,” GitHub repository, 2018. [Online]. Available: https://github.com/andresriancho/websocket-fuzzer [Cited on pages 2, 25, 27, and 47.]

[6] A. Happe, “To fuzz a websocket,” May 2019, blog post. [Online]. Available: https://snikt.net/blog/2019/05/22/to-fuzz-a-websocket/ [Cited on pages 25 and 27.]

[7] A. Konstantinov, “Streamlining websocket pentesting with wsrepl,” Doyensec Blog, Jul. 2023. [Online]. Available: https://blog.doyensec.com/2023/07/18/streamlining-websocket-pentesting-with-wsrepl.html [Cited on pages 26 and 27.]

[8] E. Ward, “Realtime communications, realtime risks,” 44CON Information Security Conference, Aug. 2023, conference presentation. [Online]. Available: https://www.youtube.com/watch?v=JMEuxEW3e2k [Cited on pages 26, 27, and 46.]

[9] P. Swigger, “Websocket turbo intruder,” GitHub repository, 2023. [Online]. Available: https://github.com/PortSwigger/websocket-turbo-intruder [Cited on pages 2, 26, and 47.]

[10] J. Kettle, “Backslash powered scanning: Hunting unknown vulnerability classes,” 2016. [Online]. Available: https://portswigger.net/research/backslash-powered-scanning-hunting-unknown-vulnerability-classes [Cited on pages 2, 3, 14, and 20.]

[11] A. Jayaraj, “Owasp damn vulnerable web sockets,” 2023. [Online]. Available: https://owasp.org/www-project-damn-vulnerable-web-sockets/ [Cited on page 4.]

[12] V. Pimenteombar and B. G. Nickerson, “Communicating and displaying real-time data with websocket,” *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, July-August 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6197172 [Cited on page 5.]

[13] A. Lombardi, *WebSocket: Lightweight Client-Server Communications*. O’Reilly Media, Sep. 2015, 1st edition. [Cited on pages 6 and 11.]

[14] S. Ghasemshirazi and P. Heydarabadi, “Exploring the attack surface of websocket,” *arXiv preprint arXiv:2104.05324*, 2021. [Online]. Available: https://arxiv.org/abs/2104.05324 [Cited on page 7.]

[15] R. Koch, “On websockets in penetration testing,” Master’s Thesis, Technische Universität Wien, 2013. [Online]. Available: https://repositum.tuwien.at/retrieve/21955 [Cited on pages 7, 23, 24, and 27.]

[16] V. Wang, F. Salim, and P. Moskovits, *The Definitive Guide to HTML5 WebSocket*, 1st ed. Apress, 2013. [Cited on page 8.]

[17] A. Diaconu, *The WebSocket Handbook*. Ably, 2002. [Cited on pages 9, 10, and 11.]

[18] L.-S. Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson, “Talking to yourself for fun and profit,” in *Proceedings of the Web 2.0 Security Privacy (W2SP) Conference*. IEEE, Jul. 2011. [Online]. Available: https://www.adambarth.com/experimental/websocket.pdf [Cited on page 9.]

[19] D. Skvorc, M. Horvat, and S. Srbljić, “Performance evaluation of WebSocket protocol for implementation of full-duplex web streams,” in *2014 37th International Convention on Information and Communication Technology, Electronics and*

Microelectronics (MIPRO). IEEE, 2014, pp. 1003–1008. [Online]. Available: https://doi.org/10.1109/MIPRO.2014.6859715 [Cited on page 9.]

[20] X. Li and Y. Xue, "A survey on server-side approaches to securing web applications," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–29, 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2541315 [Cited on page 13.]

[21] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*, 2nd ed. John Wiley & Sons, Sep. 2011. [Cited on page 13.]

[22] L. Pisu, D. Maiorca, and G. Giacinto, "A survey of the overlooked dangers of template engines," *arXiv preprint arXiv:2405.01118*, 2024. [Online]. Available: https://arxiv.org/abs/2405.01118 [Cited on page 13.]

[23] GIAC, "Fear of the unknown: A metanalysis of insecure object deserialization vulnerabilities," Global Information Assurance Certification (GIAC), GIAC Research Paper GXPN-1746, 2019, paper ID: 159864. [Online]. Available: https://www.giac.org/paper/gxpn/1746/fear-unknown-metanalysis-insecure-object-deserialization-vulnerabilities/159864 [Cited on page 14.]

[24] J. Kettle, "bulkscan," Github repository, 2023. [Online]. Available: https://github.com/albinowax/bulkScan [Cited on page 15.]

[25] M. Shema, S. Shekyan, and V. Toukharian, "Hacking with websockets," Presentation at Black Hat USA 2012, 2012. [Online]. Available: https://media.blackhat.com/bh-us-12/Briefings/Shekyan/BH_US_12_Shekyan_Toukharian_Hacking_Websocket_Slides.pdf [Cited on page 23.]

[26] H. Kuosmanen, "Security testing of websockets," Master's thesis, JAMK University of Applied Sciences, 2016. [Online]. Available: https://core.ac.uk/download/pdf/45601062.pdf [Cited on pages 24 and 27.]

[27] A. Hauser, "Websocket fuzzing - development of a fuzzer," Apr. 2023, blog post. [Online]. Available: https://www.scip.ch/en/?labs.20230420 [Cited on pages 25 and 27.]

[28] M. Fowl and N. Defoe, "Old tools, new tricks: Hacking websockets," 2019, conference presentation. [Online]. Available: https://www.irongeek.com/i.php?page=videos/derbycon9/stable-35-old-tools-

new-tricks-hacking-websockets-michael-fowl-nick-defoe [Cited on pages 25, 27, and 48.]