

Realtime Problems: Exposing WebSocket Servers Hidden in Plain Sight

By: Erik Elbieh

Security Consultant & Researcher, Palindrome Technologies

November 11, 2021

Abstract

HTTP is the protocol most often associated with web communications, but there is another protocol flying under the radar: the WebSocket protocol. WebSockets offer better real-time communication abilities than HTTP thanks to reduced communications overhead. WebSockets were standardized in 2011 in RFC 6455, but are now widely adopted. WebSockets are often found in chat platforms, financial trading platforms, chatbots, real time mapping applications, real time graphs, and even the Kubernetes and Docker APIs. Compared to the amount of security research on HTTP, WebSockets have received very little attention in the security world. Numerous hurdles have existed for anyone doing security research on WebSockets – discovering WebSocket-capable endpoints, determining the server-side software, and testing if the server is vulnerable. This research attempts to provide partial solutions to all three hurdles that have existed for WebSockets security research by providing a tool suite named STEWS (Security Tool for Enumerating WebSockets), available at <https://github.com/PalindromeLabs/STEWS>. The goal of this research is to draw attention to this underappreciated yet ubiquitous protocol and open the door for many future investigations into WebSockets security.

Introduction

The WebSocket protocol is primarily defined by RFC 6455, first standardized in 2011. In 2015, RFC 7692 defined Compression Extensions for WebSockets, enabling more efficient transfer of payloads. WebSockets offer lower overhead and latency than HTTP, making them an ideal solution for many real-time applications on the web. While HTTP is designed around providing a client with resources stored by the server, the WebSocket protocol is designed around providing a client with frequently updated data. A WebSockets server is often found at the same port number as the primary HTTP server, but this is not always the case.

A WebSocket connection consists of a two-step process. The first step happens over HTTP and is called the WebSocket handshake. A client sends a HTTP “Upgrade” request to the server with WebSocket-specific headers such as “Sec-WebSocket-Version”, “Sec-WebSocket-Key”, and “Upgrade: websocket”. If the server does not support WebSockets, it will return an error to the “Upgrade” request. But if the web server does support WebSockets, a HTTP response will contain a 101 status code with WebSocket-specific headers such as “Sec-WebSocket-Accept” and “Upgrade: websocket”. An example of this HTTP handshake is shown below.

```
> GET / HTTP/1.1
> Host: 127.0.0.1:8085
> User-Agent: curl/7.74.0
> Accept: */*
> Upgrade: websocket
> Sec-WebSocket-Key: dXP3jD9Ipw0B2EmWrMDTEw==
> Sec-WebSocket-Version: 13
> Connection: upgrade
>
```

Figure 1: Example WebSocket request sent from client

```
< HTTP/1.1 101 Switching Protocols
< Upgrade: websocket
< Connection: Upgrade
< Sec-WebSocket-Accept: GLWt4W80gwo6lmX9ZGa314RMRr0=
< X-Powered-By: Ratchet/0.4.3
```

Figure 2: Example WebSocket response sent from server

An important point to mention is that this HTTP request is usually parsed by the WebSocket server, not the main HTTP server. Because WebSocket servers have not been scrutinized as closely, it is possible that known HTTP attacks could see additional mileage when applied to certain WebSocket server implementations.

The second step of the WebSocket connection happens after the client and server complete the handshake process and agree to “upgrade” their HTTP connection. Once agreement has been reached, the client and server switch to using the WebSocket protocol. The WebSocket protocol sits on top of the TCP layer of the networking stack, at the same layer as HTTP. The structure of a WebSocket frame is seen below.

The topic of proxy mishandling of WebSockets was expanded upon by Mikhail Egorov at Hacktivity 2019³. This talk introduced the idea of WebSocket smuggling, which enables reverse proxy bypassing because some reverse proxies do not properly maintain the state of a WebSocket connection. Some reverse proxies will assume a WebSocket connection is established after observing some portion of the HTTP “Upgrade” request, providing an opportunity to send a follow-up HTTP request through the open TCP tunnel to the backend, bypassing reverse proxy blacklists. Further research on the topic of proxy bypassing was done by Jake Miller⁴ and Sean Yeoh⁵, but this follow-up research has focused mostly on using a HTTP/2 “Upgrade” request as an alternative to a WebSockets “Upgrade” request.

In addition to these efforts, a vulnerability specific to WebSockets, Cross-site WebSockets Hijacking (CSWSH), has been perhaps the most common vulnerability found in WebSockets implementations, resulting for the fact that the same-origin policy does not apply to WebSockets⁶. Many bug bounties for CSWSH have been disclosed on sites such as HackerOne, and PortSwigger’s Web Security Academy includes a lab for this vulnerability. Other common WebSocket weaknesses include insecure authentication mechanisms and sensitive data leaks, both of which often result from custom implementation errors. Weaknesses in protocols built on top of WebSockets are also a common target, where IDOR, SQL injection, and other common web vulnerabilities can exist. The websocket-fuzzer tool by Andres Riacho⁷ is designed to fuzz JSON over WebSockets, which is a common data format for APIs that communicate over WebSockets.

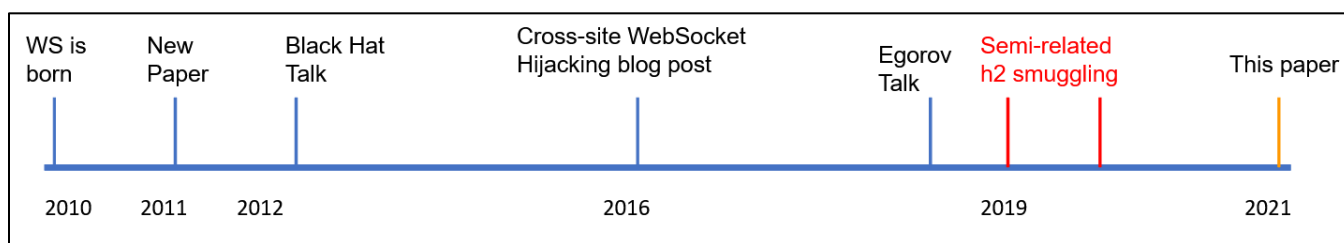


Figure 4: Timeline of related WebSocket research

Despite these past efforts, very little investigation has been done into the security of the WebSocket servers themselves. In fact, the author was only able to find 12 published CVEs for WebSocket server projects over the last decade (see Appendix A). Existing tools lack the support for customized WebSocket tests to be performed, meaning custom tooling must be developed to send low-level WebSocket frames. Even Wireshark lacks full support for certain less common WebSocket frames. While Burp Suite supports the replay of custom WebSocket packets and creation of new WebSocket connections, Burp Suite does not support WebSockets in Burp Suite extensions and currently only allows for data payload values to be modified⁸. Discussions around adding WebSocket support for nmap have occurred⁹, but no visible action has occurred on this topic in the last 5+ years. Newer tools such as nuclei have also shown interest in WebSocket support, but it is unclear if, when, or how this feature will be introduced¹⁰.

³ <https://github.com/0ang3el/websocket-smuggle>

⁴ <https://labs.bishopfox.com/tech-blog/h2c-smuggling-request-smuggling-via-http/2-clear-text-h2c>

⁵ <https://blog.assetnote.io/2021/03/18/h2c-smuggling/>

⁶ <https://christian-schneider.net/CrossSiteWebSocketHijacking.html>

⁷ <https://github.com/andresriacho/websocket-fuzzer>

⁸ <https://forum.portswigger.net/thread/websockets-api-support-c8e1660b9f0ab>

⁹ <https://seclists.org/nmap-dev/2015/q1/134>

¹⁰ <https://github.com/projectdiscovery/nuclei/issues/539>

In summary, while WebSockets have not been completely neglected by the security community, no go-to customizable tool for security testing of WebSockets exists. The past security work hardly mentions the names of any popular WebSocket servers, let alone analyzes them. Additional obstacles include discovering WebSocket servers, understanding how to identify the WebSocket server, testing the WebSocket server for vulnerabilities, and creating a local development environment where the WebSocket servers can be thoroughly tested for vulnerabilities. This paper aims to resolve all of these difficulties in WebSocket server security research. Simultaneous with this paper's publication, the STEWS (Security Tool for Enumerating WebSockets) tool suite will be released to the public. A development playground with WebSocket server examples from popular WebSocket server libraries will also be released, to allow for testing in a lab environment. Lastly, an "Awesome WebSocket Security" GitHub repo will be published, providing a go-to reference summarizing WebSocket security research to date.

WebSockets Discovery

As described in the introduction, the WebSocket connection process consists of 2 main steps: the HTTP handshake followed by the WebSocket protocol communication. Normally this process is initiated inside of a web browser using JavaScript. There are several ways in which a WebSocket server can be identified. All methods considered will be outlined to demonstrate the alternative options and demonstrate why ZGrab2 was chosen as the best scalable option for this research.

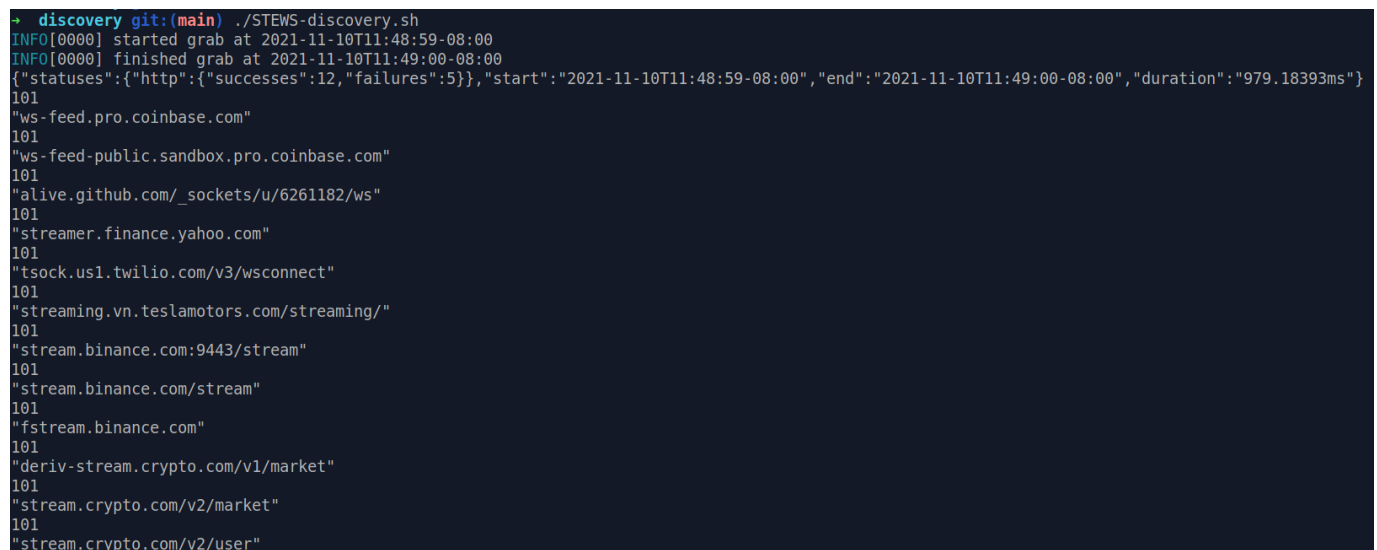
1. If website traffic is passed through a web proxy such as Burp Suite or OWASP ZAP, a filter for HTTP responses with a status code of 101 with specific WebSocket response headers (such as "Upgrade: websocket") can identify WebSocket responses. However, browsing with a web proxy is slow, especially when done manually.
2. One approach to automating the web browsing process is to use a web crawler. These tools can visit and load many webpages without needing user interaction. However, the web crawler must run the webpage JavaScript in order to initiate WebSocket connection and observe a 101 response. Although this process could be automated, loading full webpages and running the associated JavaScript can be time consuming.
3. Similar to the previous approach, it is possible to use a web crawler to visit webpages, but rather than running JavaScript and waiting for a 101 HTTP response status code, the crawler can search the JavaScript files for keywords that identify a WebSocket connection attempt. Simply searching files for keywords can be faster than running JavaScript, but an additional step would be needed to identify the exact URL where the WebSocket server endpoint is located.
4. Perhaps the lowest effort approach to discovering WebSockets is to brute force a WebSocket connection request to a large number of URLs. While many URLs will not support WebSockets, the benefit to this method is that only the HTTP status code is needed to identify the endpoint as a WebSocket-supporting server, so only the HTTP header needs to be examined, not the HTTP body or linked JavaScript files. As will be shown, what this method may lack in precision, it makes up for with speed and ease of use.
5. A hybrid approach, using a combination of the above options, would provide a balance of precision and speed. For instance, if only a single domain is in scope of testing, a web crawler may be a good approach in some situations, but it might also be beneficial to brute force common endpoints for WebSocket servers in case the web crawler does not gain full coverage of the domain. The brute forcing process could use the known paths information gathered from the web crawler to perform a more targeted brute force scan for WebSockets endpoints.

Since this research effort was interested in larger scale discovery of WebSockets, option 4 provided a highly parallelizable approach. However, the tooling for this approach was not available in a prepared package, since WebSockets have not been a prime candidate for security research so far. Several types of tools were considered for this task. Port scanners such as masscan operate at the TCP layer and analyze individual IP addresses. However, since the first handshake portion of a WebSocket connection occurs over HTTP, using a tool such as masscan would require implementing support for HTTP for this specific use case, since the HTTP protocol sits on the layer above TCP. A tool such as Turbo Intruder, a Burp Suite extension, supports high speed HTTP requests but is designed to send many requests to a single host. However, the Turbo Intruder README suggests using ZGrab to send a single request to many hosts¹¹.

ZGrab2 is used by Censys to map the internet. The ZGrab2 project has supported the HTTP protocol for some time, but only in the last few months did a pending pull request appear in the ZGrab2 project implementing support for custom HTTP headers. At the time of writing, the PR has not been merged into the project, but it works quite well. Some additional modifications to ZGrab2 was needed in order to send the proper headers needed for a WebSocket request, such as including an "Origin" header value matching the host being tested. The code for this WebSocket-oriented fork of ZGrab2 is available at <https://github.com/PalindromeLabs/zgrab2>. Below is an example from the STEWS-discovery tool to test for WebSocket endpoints. It relies on the custom Palindrome Technologies ZGrab2 fork and uses a text file of URLs named "known-endpoints.txt":

```
cat ./known-endpoints.txt | shuf | /opt/zgrab2 http --custom-headers-
names='Upgrade,Sec-WebSocket-Key,Sec-WebSocket-Version,Connection' --custom-
headers-values='websocket,dXP3jD9Ipw0B2EmWrMDTEw==,13,Upgrade' --remove-
accept-header --dynamic-origin --use-https --port 443 --max-redirects 10 --
retry-https --cipher-suite portable -t 10 | jq
'.data.http.result.response.status_code,.domain' | grep -A 1 -E --line-
buffered '^101' | tee -a STEWS-discovery-output.txt
```

And example of the output of the STEWS-discovery tool is shown below.



```
+ discovery git:(main) ./STEWS-discovery.sh
INFO[0000] started grab at 2021-11-10T11:48:59-08:00
INFO[0000] finished grab at 2021-11-10T11:49:00-08:00
{"statuses":{"http":{"successes":12,"failures":5},"start":"2021-11-10T11:48:59-08:00","end":"2021-11-10T11:49:00-08:00","duration":"979.18393ms"}
101
"ws-feed.pro.coinbase.com"
101
"ws-feed-public.sandbox.pro.coinbase.com"
101
"alive.github.com/_sockets/u/6261182/ws"
101
"streamer.finance.yahoo.com"
101
"tsock.us1.twilio.com/v3/wsconnect"
101
"streaming.vn.teslamotors.com/streaming/"
101
"stream.binance.com:9443/stream"
101
"stream.binance.com/stream"
101
"fstream.binance.com"
101
"deriv-stream.crypto.com/v1/market"
101
"stream.crypto.com/v2/market"
101
"stream.crypto.com/v2/user"
```

Figure 5: Example of STEWS-discovery tool output

¹¹ <https://github.com/PortSwigger/turbo-intruder>

Testing with ZGrab2 in this manner allowed testing of hundreds of endpoints per second, or thousands of endpoints per second if DNS lookups were performed beforehand with a tool like massdns or zdns. Other attempted approaches, such as combining curl (a simple bash HTTP client) with xargs (a bash parallel process utility) only resulted in a handful of requests per second. However, this scanning method does not come without drawback.

The first drawback was that ZGrab2, like masscan, was originally designed to operate at the TCP layer, so DNS lookups for large datasets can be a bottleneck. This issue was possible to resolve using massdns, a tool that parallelizes the DNS lookup of domain names using multiple DNS resolvers. While using massdns turns the discovery process into a two-step process (first performing the DNS lookups, then running ZGrab2), this additional step improves the overall results in situations where ZGrab2 overwhelms the DNS server it is using.

A second drawback was that ZGrab only tests the URLs provided to it. This means that if only the top-level domain is known, ZGrab2 does not automatically check for a WebSocket endpoint at example.com/ws, or ws.example.com. Instead, these “high probability” URLs must be generated by the user performing testing. For high precision situations where a single domain must be tested in depth, it is preferable to use a crawler or a list of known URLs for the domain, such as the URLs stored by Wayback Machine. However, for the internet-scale testing discussed here, it is sufficient to use wordlists of common URL paths for brute forcing, and this drawback was known when deciding to use this approach. While most existing wordlists contain general HTTP paths, a short custom wordlist was developed by analyzing known WebSocket endpoint URLs. This short wordlist can be seen in the results of Table 1.

A third drawback is that ZGrab2 is still a work in progress, with recent PRs from the last year adding important features. Like masscan, running a highly parallelized scan does not always provide perfect results. Running the same scan with the same wordlist several times can result in a slightly different number of WebSocket servers being discovered (roughly a 1% difference).

An important question for domain-level internet-scale testing is where to get a list of domains. Tools such as Shodan and Censys scan all possible IPv4 addresses, but because a single IP address can host multiple domains, the same technique cannot be used in this situation. While a quick search shows that lists of the top few million websites are easily available, it is possible to get a list of all .com domains registered domains by requesting the .com zone file. These zone files are effectively a superset of all domains known to all DNS servers. Access to these zone files must be requested, which can take some time to be approved. To avoid approval delays, the list of the top 10 million domains was deemed sufficient for this paper’s testing purposes.

Using a short wordlist of common WebSocket URL paths, the ZGrab2-based discovery method was applied on a set of roughly 3 million top domains, with the discovery results found in Table 1. Note that the total number of servers discovered does not necessarily refer to unique domains, as some domains answered a WebSockets request from any path at the domain. These results are the first known study of WebSocket server discovery using a brute force handshake request approach.

Table 1: ZGrab2-based WebSocket server discovery on port 443 across 3 million domain names

URL	Number of WebSocket servers found
domain.com	2281
domain.com/ws	1991
domain.com/ws/v1	1605
domain.com/ws/v2	1606
domain.com/socket.io/?EIO=3&transport=websocket	1389
domain.com/stream	448
domain.com/feed	452
www.domain.com	1582
ws.domain.com	891
stream.domain.com	574
Total	12819

WebSocket Fingerprinting

Fingerprinting any backend web application increases an attacker's knowledge of the system and provides crucial information for the next steps of an attack. With information about the backend application, a persistent adversary can either use a known vulnerability, or if the application is an open-source project (which is the case for all WebSocket servers examined in this paper), attempt to find a new vulnerability. The basic knowledge of what software is running on the backend can inform an adversary about the preferred programming language, potential software stack, and/or development lifecycle of the backend system. The fingerprinting process can be broken down into three steps:

1. Finding identifying features between different WebSocket server implementations
2. Creating a database of known WebSocket server fingerprints
3. Creating a tool to collect fingerprints from a WebSocket server and attempt to match it to the database of known WebSocket server fingerprints

A WebSocket connection involves two steps: the HTTP protocol handshake and the subsequent WebSockets protocol communication. Both the HTTP and WebSockets portions of a WebSocket connection can be used for fingerprinting. In some cases one of the protocols can provide higher signal-to-noise information. For example, if authentication is handled during the HTTP WebSocket handshake, it may not be possible to access the WebSockets protocol communication for fingerprinting without proper credentials. In a different scenario, if a reverse proxy limits the HTTP response that is returned to the client, it might be more useful to rely on the WebSockets protocol communication for fingerprinting. Both approaches were used in this research. The first step in fingerprinting is to understand the identifying features of different WebSocket servers.

The HTTP WebSocket handshake was examined for identifying features first. Since HTTP is a common protocol, there are existing tools that perform fingerprinting using HTTP responses. Perhaps the most popular fingerprinting tool, nmap, uses a variety of fingerprinting methods in HTTP scripts. Among these methods are sending requests to special URL paths, identifying specific response headers or payloads using keywords, and sending specially crafted HTTP requests (with specific headers and body content). wafw00f, a popular web application firewall (WAF) fingerprinting tool, relies primarily on HTTP response header values and HTTP response body text to identify servers¹².

¹² <https://github.com/EnableSecurity/wafw00f>

Given the limited nature of the WebSocket handshake process, not all of the fingerprinting techniques in nmap or wafw00f can be borrowed for the WebSocket handshake. When examining possible options for modifying the HTTP GET request from the client, sending custom headers is the easiest configurable data field. Since the WebSocket handshake request is a GET request, sending body data is not an option. For fingerprinting the HTTP 101 server response, the response generally has no HTTP body meaning only HTTP headers from the server response can be used for WebSocket fingerprinting. In cases where the WebSocket server deployment was not hardened, it is often possible to find a HTTP header in a WebSocket handshake response that identifies the WebSocket server and version number. However, this is the only the most basic fingerprinting scenario, and this identifying header is often removed in secured production deployments. WebSocket fingerprinting can use several special WebSocket headers in the WebSocket server response not available to wafw00f or other standard HTTP fingerprinting tools.

For starters, the accepted values for the “Sec-WebSocket-Version” header can help identify a server. Most servers only allow for version 13, but some allow for a version header of 8, and some servers accept other values. Additionally, a server’s response to specific malformed version values, such as “13\\” or “13\n”, is another indicator. For example, the Java Spring framework WebSockets server accepts a connection with a version of “13\r”, while most other servers do not. Similarly, the npm ws WebSocket server accepts a version number of 8, while most WebSocket servers only allow a version of 13. Note that if the WebSocket server is behind a reverse proxy, some reverse proxies handle the “\n” and “\r” characters differently, impacting the fingerprint that can be acquired.

Two other unique headers in WebSocket handshake requests can allow for additional identification. The WebSocket protocol contains a “Sec-WebSocket-Protocol” header and a “Sec-WebSocket-Extensions” header. Both of these headers allow a client and server to agree upon using specific protocols or extensions to communicate. While some WebSocket servers use custom subprotocol or extension values, there is a list of official reserved values specified¹³. When a WebSocket server supports one of these subprotocols or extensions, the server usually replies with a response header repeating the supported subprotocol or extension. Therefore, by sending subprotocol or extension values one at a time, we can learn what subprotocols and extensions the WebSocket servers supports. One example of this being useful for fingerprinting is that uWebSockets and the npm ws WebSocket servers always repeat back the same “Sec-WebSocket-Protocol” header value provided by the user, rather than only repeating back the header for supported subprotocol values. Most other implementations do not include the “Sec-WebSocket-Protocol” header if they do not support the subprotocol value provided by the client. Similarly, support for WebSocket Compression “Sec-WebSocket-Extensions” values is not found in all WebSocket servers, since the WebSocket Compression RFC was introduced more recently in 2015. If a WebSocket server does support “Sec-WebSocket-Extensions” values, as in the case of uWebSockets, it will respond with supported values in the “Sec-WebSocket-Extensions” header of the HTTP response.

With the first half of the WebSocket connection accounted for (the HTTP protocol portion), we can examine fingerprinting options in the second half of the WebSocket connection, using the WebSocket protocol. Since no public tools have attempted this task before, there are no prior examples to reference. Nevertheless, there are several important parts of the WebSocket frame that can be used to differentiate WebSocket servers. Both the reserved (rsv) bits and opcode fields are handled quite differently in different implementations. One finding from this research, which resulted from fuzzing the values of these fields, is that many WebSocket server implementations return verbose error messages to the user. These error messages are unique, for the most part, and can greatly aid in identifying the WebSocket server. For example, the PHP Ratchet WebSocket library

¹³ <https://www.iana.org/assignments/websocket/websocket.xml#subprotocol-name>

returns the message “Ratchet detected the first frame of a message was a continue” when a WebSocket frame with an opcode value of 0 (the “continue” opcode value) was sent to the server. For the same “continue” opcode value, the Java-WebSocket WebSocket server returns an error message of “WebSocket frame was sent with an unrecognized opCode of [0]”. Because this is the first time that error messages across WebSocket server implementations have been compared, it is unlikely that the verbosity or uniqueness of these WebSocket error messages have been considered problematic in the past. An example of one particular test case, test case 200, from STEWS-fingerprint.py tool is shown in Table 2 below. Test case 200 sets the 1st and 2nd reserved bits to 0 while setting the 3rd reserved bit to 1.

Table 2: STEWS-fingerprint.py test case 200 (set reserved bit 3) server results

WebSocket Server Implementation	STEWS-fingerprint.py Test Case 200 Response
npm ws	<i>No error message</i>
faye	One or more reserved bits are on: reserved1 = 0, reserved2 = 0, reserved3 = 1
Gorilla	unexpected reserved bits 0x10
uWebSockets	<i>No error message</i>
Java Spring Boot	The client frame set the reserved bits to [1] for a message with opCode [2] which was not supported by this endpoint
Python websockets	<i>No error message</i>
Ratchet	Ratchet detected an invalid reserve code
Tornado	<i>No error message</i>

The STEWS-fingerprint.py tool, which is published alongside this paper, provides a fingerprinting implementation to identify WebSocket servers. It breaks down the type of fingerprinting tests into 7 categories, as follows:

- 100-series tests: opcode tests (WebSocket protocol)
- 200-series tests: rsv bit tests (WebSocket protocol)
- 300-series tests: version tests (HTTP protocol)
- 400-series tests: extensions tests (HTTP protocol)
- 500-series tests: subprotocol tests (HTTP protocol)
- 600-series tests: long payload tests (WebSocket protocol)
- 700-series tests: hybi and similar tests (WebSocket protocol)

An example of the output from the STEWS-fingerprint tool is shown below.

```

+ fingerprint git:(main) python3 STEWS-fingerprint.py -3 -n -u 127.0.0.1:8084/echo
Exception while attempting reconnection: Handshake status 400 Bad Request
Exception while trying to connect for version: 7 socket is already closed.
Exception while attempting reconnection: Handshake status 400 Bad Request
Exception while trying to connect for version: 8 socket is already closed.
Exception while attempting reconnection: Handshake status 400 Bad Request
Exception while trying to connect for version: 13; socket is already closed.
Exception while attempting reconnection: Handshake status 400 Bad Request
Exception while trying to connect for version: 13- socket is already closed.
Exception while attempting reconnection: Connection to remote host was lost.
Exception while trying to connect for version: 13
socket is already closed.
Exception while attempting reconnection: Handshake status 400 Bad Request: invalid header value
socket is already closed.connect for version: 13
Exception while attempting reconnection: Handshake status 400 Bad Request
Exception while trying to connect for version: 13\ socket is already closed.
Exception while attempting reconnection: Handshake status 400 Bad Request
Exception while trying to connect for version: 13\ socket is already closed.
=====
Identifying...
=====
List of deltas between detected fingerprint and those in database
[3, 7, 0, 8, 2, 2, 6, 5]
=====
>>>Most likely server: Gorilla -- % match: 100.0
>>>Second most likely server: Java Spring boot, Python websockets -- % match: 81.81818181818181
=====
Most likely server's fingerprint:
{'100': 1, '101': 1, '102': 0, '103': 0, '104': 'continuation after final message frame', '105': 1, '200': 'unexpected reserved bits 0x10', '201': 'unexpected reserved bits 0x10', '202': 'unexpected reserved bits 0x10', '203': 'unexpected reserved bits 0x10', '204': 'unexpected reserved bits 0x10', '205': 'unexpected reserved bits 0x10', '206': 'unexpected reserved bits 0x10', '300': 0, '301': 0, '302': 0, '303': 1, '304': 0, '305': 0, '306': 0, '307': 0, '308': 0, '309': 0, '310': 0, '400': 0, '401': 0, '402': 0, '403': 0, '404': 0, '405': 0, '500': 0, '501': 0, '600': 0, '601': 0, '602': 0, '603': 0, '604': 0, '605': 0, '606': 0, '607': 0, '608': 0, '609': 0, '610': 0, '611': 0, '612': 0, '700': 'Bad Request', '701': 'Bad Request', '702': '400 Bad Request', '703': 'Bad Request', '704': 'Bad Request', '705': '400 Bad Request: missing required Host header'}
=====
Tested server's fingerprint:
{'300': 0, '301': 0, '302': 0, '303': 1, '304': 0, '305': 0, '306': 0, '307': 0, '308': 0, '309': 0, '310': 0}
+ fingerprint git:(main)

```

Figure 6: Example of STEWS-fingerprint tool output

The above tests were found to highlight differences in responses between WebSocket server implementations to help identify the implementation. Other variations in client requests, such as specific masking key values in the WebSocket protocol, did not yield different responses between different WebSocket server implementations. While the STEWS-fingerprint.py tool is the first to enable WebSocket server fingerprinting, it still has areas to be improved. At the time of release, the STEWS fingerprinting tool has support for many of the most common WebSocket servers fingerprint identifiers, but more server fingerprints and test cases should be added to enhance the usability and accuracy of the tool. Additionally, the latest versions of the WebSocket servers were the focus for extracting server fingerprint identifiers, and it is possible that older versions of the same servers do not have the same identifying characteristics.

WebSockets Vulnerability Testing

The final step in most security assessments or attacks is the vulnerability detection or exploitation phase. The first release of the STEWS tool suite only contains vulnerability identification capabilities for public CVEs taken from Appendix A. Readers interested in investigating further can find some additional WebSocket server vulnerabilities without CVE identifiers outlined in old GitHub issues.

Some of the vulnerabilities seen in Appendix A are difficult to identify directly without triggering the issues on the server, such as the large message DoS CVEs (CVE-2018-21035, CVE-2016-10544, and CVE-2016-10542). Due to this, when testing for these CVEs against a WebSocket server, a warning message is provided to the user to warn them of the server-side impact it may have. These vulnerabilities could be identified in a benign manner using other fingerprinting identifiers. For example, opcode error messages and “Sec-WebSocket-Version” values have changed in some WebSocket libraries over time, and while these do not map exactly to the versions vulnerable to the CVEs, they may be the best option available other than attempting an exploit against the server.

The RegEx DoS CVEs, CVE-2020-7662 and CVE-2020-7663, are an easier vulnerability to check for because this type of DoS allows for “analog” variability in the length of the regex input value, rather than a “digital” positive

or negative input value. By submitting multiple “Sec-WebSocket-Extensions” values of different but reasonable lengths, it is possible to observe a difference in response time from the server in vulnerable versions.

A couple of the vulnerabilities in Appendix A, namely CVE-2020-27813 (Gorilla integer overflow) and the weak PRNG for the socket.io socket IDs (CVE-2017-16031) were not deemed straightforward to test for when no visibility exists into the target server. Similarly, the two recent socket.io vulnerabilities related to WebSocket file uploads are relevant for only a small fraction of WebSocket server endpoints, so the check for the vulnerability was not prioritized for the initial release of this tool.

Lastly, one vulnerability not found in Appendix A that STEWS supports and is easy to test for (in the simplest case) is CSWSH. If a server initiates a WebSocket connection both with a matching “Origin” header value and without an “Origin” header, the server is vulnerable to CSWSH. Even in cases where the “Origin” header value is checked, it may be possible to bypass this check by, for instance, testing for an unescaped period in a regex check. While a couple simple “Origin” header validation bypass checks are included in the STEWS tool suite, validation bypasses should either be tested manually or using a more advanced automated method.

Ideas for Future Work

Another area for improvement in future work is to perform internet-scale WebSocket scanning from multiple geographic origins, because scanning from a single geographic origin can overlook and miss some hosts. Areas for future work include building a better wordlist of common WebSocket URL patterns and investigating ports other than 443 for WebSocket servers.

Because most prior internet-scale scanning efforts have identified servers based on IP address, and because ZGrab2 support for custom HTTP headers is a very recent development, the approach used for this research can be used for other future research efforts outside of WebSockets security, such as the discovery of servers supporting HTTP/2 upgrade requests.

This research paves the way for substantial future WebSockets security work in multiple directions. Some of the ideas considered for future work by this author include, but are not limited to:

- a) Create a more extensive list of common WebSocket URL paths and ports other than 443 for WebSockets discovery
- b) Analyze dataset of publicly accessible WebSockets servers (most common implementations, etc.)
- c) Implement WebSocket discovery combining crawling with JavaScript file keyword analysis
- d) Expand WebSocket fingerprinting for more servers and version-specific identifiers
- e) Analyze impact of different reverse proxies on WebSocket fingerprints
- f) Research WebSocket Compression Extensions support (RFC 7692) and implementation errors
- g) Research implementation vulnerabilities in WebSocket subprotocols (STOMP, WAMP, MQTT)
- h) Research lower-level buffer and socket handling by different WebSocket servers
- i) Improve WebSocket proxy bypass (AKA smuggling) methods and attacks
- j) Research feasibility of request splitting and other HTTP vulnerabilities in WebSocket HTTP handshake request
- k) Research server multithreading issues with large numbers of malicious client connections
- l) Research custom non-browser WebSocket client implementation vulnerabilities
- m) Fuzzing of WebSocket servers using invalid WebSocket frames

- n) Using ZGrab2 HTTP domain-level scanning for other security topics (HTTP/2 research, etc.)

Conclusion

WebSockets are an integral part of many modern websites. In the last decade, the WebSockets protocol has found many use cases for low-latency use cases. While WebSocket servers have lurked in the shadows and remained mostly hidden to security researchers in the past, the tools and methods provided by this research enable easier access to the security testing of the WebSockets protocol. This paper illustrates how WebSockets can be discovered through highly parallelized brute force method, how specific WebSockets servers can be identified by fingerprinting methods, and how a WebSocket server can be checked for known vulnerabilities. Many topics within WebSockets security remain open and unexplored, providing green pastures for future exploration. By making it easier to find, compare, and test WebSocket servers, the security community can look forward to an exciting new frontier of exploration with the WebSocket protocol.

Appendix A: Related WebSockets Research Repositories

<https://github.com/PalindromeLabs/awesome-websockets-security>: A collection of CVEs, research, and reference materials related to WebSocket security.

<https://github.com/PalindromeLabs/WebSocket-Playground>: A script to jumpstart nearly a dozen WebSockets servers simultaneously, for testing and comparison purposes.

<https://github.com/PalindromeLabs/STEWs>: Security Tool for Enumerating WebSockets. Provides methods to discovery, fingerprint, and vulnerability scan WebSockets endpoints.

<https://github.com/PalindromeLabs/zgrab2>: Fork of ZGrab2 with support for rapidly testing WebSockets support of endpoints.

Appendix B: WebSocket Server CVEs

Table 3: List of WebSocket server CVEs

CVE ID	Vulnerable package	Related writeup	Vulnerability summary
CVE-2021-42340	Tomcat	Apache mailing list	DoS memory leak
CVE-2020-36406	uWebSockets	Google OSS-Fuzz	Stack buffer overflow
CVE-2021-33880	Python websockets		HTTP basic auth timing attack
CVE-2021-32640	ws	GitHub Advisory	Regex backtracking Denial of Service
CVE-2020-24807	socket.io-file	Auxilium Security	File type restriction bypass
CVE-2020-15779	socket.io-file	Auxilium Security	Path traversal

CVE ID	Vulnerable package	Related writeup	Vulnerability summary
CVE-2020-27813	Gorilla	Auxilium Security	Integer overflow
CVE-2020-11050	Java WebSocket	GitHub advisory	SSL hostname validation not performed
CVE-2020-15134	faye-websocket	GitHub advisory	Lack of TLS certificate validation
CVE-2020-15133	faye-websocket	GitHub advisory	Lack of TLS certificate validation
CVE-2020-7663	Ruby websocket-extensions	Writeup	Regex backtracking Denial of Service
CVE-2020-7662	npm websocket-extensions	Writeup	Regex backtracking Denial of Service
CVE-2018-1000518	Python websockets		DoS via memory exhaustion when decompressing compressed data
CVE-2018-21035	Qt WebSockets	Bug report	Denial of service due large limit on message and frame size
CVE-2017-16031	socket.io	GitHub Issue	Socket IDs use predictable random numbers
CVE-2016-10544	uWebSockets	npm advisory	Denial of service due to large limit on message size
CVE-2016-10542	NodeJS ws	npm advisory	Denial of service due to large limit on message size
None	draft-hixie-thewebsocketprotocol-76	Writeup	

Appendix C: Default Max Payload Sizes

Table 4: Default maximum WebSocket message payload size for different WebSocket servers

Server	Max payload (bits)	Link to evidence
Faye	67108863	https://github.com/faye/websocket-driver-node#driver-api https://github.com/faye/websocket-driver-node/commit/f15b331a3459d30800a8b9781da5e2a7b3982160
Npm ws	104857600	https://github.com/websockets/ws/blob/114de9e33668075f0af88dc440f1ebd813161e72/lib/websocket-server.js#L30
Gorilla	No limit!	https://github.com/gorilla/websocket/search?q=maxMessageSize
uWebSockets	16777216	https://github.com/uNetworking/uWebSockets/blob/9d8e37d8a44ddb3916ceaac1db9a70a025b04d0/misc/main.cpp#L37
Python websockets	1048576	https://github.com/aagustin/websockets/blob/3dd672332308b94221f4acabfae15d519459e5a9/src/websockets/connection.py#L69

Appendix D: WebSocket Server Implementations

Table 5: Popular WebSockets server implementations

Name	Language	Repository	GitHub Stars (as of Nov 2021)
ws	JS	https://github.com/websockets/ws	17,200
Gorilla	Go	https://github.com/gorilla/websocket	15,700
uWebSockets	C++	https://github.com/uNetworking/uWebSockets	13,300
Java-WebSocket	Java	https://github.com/TooTallNate/Java-WebSocket	8,500
Cowboy	Erlang	https://github.com/ninenines/cowboy	6,500
Ratchet	PHP	https://github.com/ratchetphp/Ratchet	5,600
warp	Rust	https://github.com/seanmonstar/warp	5,500
WebSocket++	C++	https://github.com/zaphoyd/websocketpp	5,100
websocket-sharp	C#	https://github.com/sta/websocket-sharp	4,400
ws	Go	https://github.com/gobwas/ws	4,200
websockets	Python	https://github.com/aaugustin/websockets	3,700
libwebsockets	C	https://github.com/warmcat/libwebsockets	3,200

References

1. <http://www.adambarth.com/papers/2011/huang-chen-barth-rescorla-jackson.pdf>
2. https://www.youtube.com/watch?v=-ALjHUqSz_Y
3. <https://github.com/0ang3el/websocket-smuggle>
4. <https://labs.bishopfox.com/tech-blog/h2c-smuggling-request-smuggling-via-http/2-clear-text-h2c>
5. <https://blog.assetnote.io/2021/03/18/h2c-smuggling/>
6. <https://christian-schneider.net/CrossSiteWebSocketHijacking.html>
7. <https://github.com/andresriancho/websocket-fuzzer>
8. <https://forum.portswigger.net/thread/websockets-api-support-c8e1660b9f0ab>
9. <https://seclists.org/nmap-dev/2015/q1/134>
10. <https://github.com/projectdiscovery/nuclei/issues/539>
11. <https://github.com/PortSwigger/turbo-intruder>
12. <https://github.com/EnableSecurity/wafw00f>
13. <https://www.iana.org/assignments/websocket/websocket.xml#subprotocol-name>