



**jamk.fi**

# **Security Testing of WebSockets**

Harri Kuosmanen

Master's thesis

May 2016

Technology, communication and transport

Cyber Security

Master's Degree Programme in Information Technology

**Jyväskylän ammattikorkeakoulu**

JAMK University of Applied Sciences

Author(s) Kuosmanen, Harri	Type of publication Master's thesis	Date May 2016 Language of publication: English
	Number of pages 77	Permission for web publication: x
Title of publication <b>Security Testing of WebSockets</b>		
Degree programme Master's Degree Programme in Information Technology		
Supervisor(s) Salmikangas, Esa		
Assigned by Silverskin Information Security		
Abstract <p>The current state of WebSocket is not an exception when it comes to security issues in traditional web applications. There are vulnerabilities that commonly exist in WebSocket implementations that any attacker could potentially exploit.</p> <p>In order to build a protection to avoid these vulnerabilities it needs to be understood what the vulnerabilities are and how they are discovered, how can they be exploited, and how can they be defended against.</p> <p>Many of the commonly known web application vulnerabilities can be linked to ones found in WebSocket implementations. However, some of them slightly vary from the ones previously known because of the WebSocket protocol level implementation.</p> <p>Commonly available WebSocket security testing tools are not mature enough for comprehensive security testing compared to traditional web application security testing. This is why a tool needs to be developed for the testing.</p> <p>The tool was developed with the intend of being able to test all the common security vulnerabilities found from WebSocket implementations in an understandable way. The tool is not automated, but can be used as long as the tester have the necessary expertise in WebSocket implementations and overall web application security testing.</p> <p>Research was done as a design research utilizing quantitative methodologies.</p>		
Keywords/tags ( <a href="#">subjects</a> ) http, websocket, penetration testing, security, security testing, python, programming		
Miscellaneous		

Tekijä(t) Kuosmanen, Harri	Julkaisun laji Opinnäytetyö	Päivämäärä 18.05.2016
	Sivumäärä 77	Julkaisun kieli Englanti
		Verkojulkaisulupa myönnetty: x
Työn nimi <b>Security Testing of WebSockets</b>		
Koulutusohjelma Master's Degree Programme in Information Technology		
Työn ohjaaja(t) Esa Salmikangas		
Toimeksiantaja(t) Silverskin Information Security		
Tiivistelmä <p>WebSocketin tietoturvan nykytila ei ole poikkeus verratessa sitä tietoturvan tilaan muissa perinteisissä web-sovelluksissa. WebSocket toteutuksista on yleisesti havaittavissa haavoittuvuuksia, joita hyökkääjä kykenee potentiaalisesti hyväksikäyttämään.</p> <p>Suojataksaan WebSocket-toteutuksen yleisesti olemassa olevilta haavoittuvuuksilta toteutuksen ylläpitäjän tulee ymmärtää, millaisia haavoittuvuuksia on olemassa, kuinka ne voidaan havaita, kuinka niitä vasten voidaan hyökätä ja kuinka niitä vastaan tulisi suojautua.</p> <p>Useat WebSocket-toteutuksista löytyvät haavoittuvuudet voidaan yhdistää haavoittuvuuksiin, joita yleisesti havaitaan myös perinteisistä web-sovelluksista. On kuitenkin ymmärrettävä, että osa haavoittuvuuksista on hieman erilaisia johtuen WebSocket-protokollan toteutuksesta.</p> <p>Yleisesti saatavilla olevat tietoturvatestaustyökalut eivät ole tarpeeksi kehittyneitä kokonaisvaltaiseen WebSocket-toteutuksen tietoturvatestaukseen, kun verrataan perinteiseen web-sovelluksen tietoturvatestaukseen. Täten tutkimuksen pääasiallisena tavoitteena olikin luoda oma työkalu testausta varten.</p> <p>Työkalu toteutettiin siitä syystä, että olisi yksi työkalu, jolla kyettäisiin testaamaan kaikki yleisesti havaittavat haavoittuvuudet WebSocket-toteutuksista. Työkalun käyttö kuitenkin vaatii testaajalta osaamista WebSocket-toteutuksista ja perinteisestä web-sovellusten tietoturvatestaamisesta.</p> <p>Tutkimustyö suoritettiin kehittämistutkimuksena käyttäen laadullisia tutkimusmenetelmiä.</p>		
Avainsanat ( <a href="#">asiasanat</a> ) http, websocket, tietoturvatestausta, tietoturva, python, ohjelmointi		
Muut tiedot		

## Content

1	Introduction.....	7
1.1	Research Background .....	7
1.2	Commissioner .....	8
1.3	Problem Definition .....	8
1.4	Research Methodology .....	9
1.5	Literary Overview .....	9
2	WebSocket Protocol.....	11
2.1	History of WebSockets .....	11
2.2	Bidirectional Communication in Web Applications .....	11
2.3	WebSocket Benefits .....	13
2.4	WebSocket Browser Support .....	13
2.5	Relationship to TCP and HTTP .....	14
3	WebSocket Communication.....	15
3.1	Overview.....	15
3.2	WebSocket Uniform Resource Identifiers.....	16
3.3	Opening Handshake .....	17
3.4	WebSocket Frame .....	19
3.4.1	WebSocket Frame Format.....	20
3.4.2	Fragmentation .....	22
3.5	Closing Handshake.....	23
3.6	WebSocket Subprotocols .....	24
3.7	WebSocket Extensions .....	25
4	WebSocket Security Issues.....	26
4.1	State of the WebSocket Security.....	26
4.2	Authentication.....	27

	2
4.3	Authorization .....29
4.4	Cross-Domain Requesting .....30
4.5	Traffic Encryption .....32
4.6	Input Validation .....33
4.6.1	Description.....33
4.6.2	Server-Side Injection Attacks.....34
4.6.3	Client-Side Injection Attacks.....35
4.6.4	Protecting Against Input Validation Related Attacks .....36
4.7	Resource Exhaustion .....36
5	Manual Testing of WebSocket Security .....38
5.1	WebSocket Security Testing Tools .....38
5.1.1	Burp Suite .....38
5.1.2	OWASP Zed Attack Proxy.....39
5.1.3	Other Tools .....40
5.2	Manual Testing .....42
5.2.1	Proxy Tool .....42
5.2.2	Authentication .....43
5.2.3	Authorization .....46
5.2.4	Cross-Domain Requesting.....48
5.2.5	Traffic Encryption.....50
5.2.6	Input Validation .....51
5.2.7	Resource Exhaustion.....53
6	Customized Tool for WebSockets Security Testing.....57
6.1	Preliminary Analysis .....57
6.2	Design .....58
6.2.1	User Interface .....58
6.2.2	Functions.....59

6.2.3	Logic Flow of the Tool.....	60
6.3	Implementation.....	61
6.3.1	Wsdump Tool.....	61
6.3.2	Programming the Tool.....	61
6.4	Testing .....	64
6.5	Usage .....	65
7	Results .....	69
8	Conclusion .....	70
9	Future Research .....	72
	References.....	73
	Appendices .....	75
	Appendice 1. simple-web-socket-client – websocket.html.....	75

## Figures

Figure 1. WebSocket frame format.....	20
Figure 2. The form-based authentication used in a WebSocket service. ....	28
Figure 3. Attacks against authorization levels.....	29
Figure 4. Cross-Site WebSocket Hijacking attack. ....	31
Figure 5. Man-in-the-Middle attack. ....	32
Figure 6. Example attacks against the client and the server. ....	33
Figure 7. Burp Suite's WebSocket history user interface.....	39
Figure 8. OWASP Zed Attack Proxy's WebSocket user interface. ....	40
Figure 9. WebSocket traffic view in Google Chrome's developer tools.....	41
Figure 10. Websocket.org Echo Test tool. ....	41
Figure 11. Principle of the proxy server functionality.....	42
Figure 12. Sent to Repeater functionality in Burp Suite. ....	44
Figure 13. Nullbyte as the cookie's name/value pair's value in Burp Suite Repeater. ....	45
Figure 14. Access to Resend functionality in OWASP Zed Attack Proxy. ....	47
Figure 15. Resend functionality in OWASP Zed Attack Proxy. ....	48
Figure 16. Origin modified WebSocket opening handshake request in Burp Repeater. .....	49
Figure 17. Configure target details view in Burp Repeater.....	51
Figure 18. WebSocket message resend feature in OWASP Zed Attack Proxy.....	52
Figure 19. The Origin manipulation with Burp Suite.....	54
Figure 20. WebSocket server's log when multiple connections are opened by one WebSocket client from the same source. ....	56
Figure 21. Logic flow chart of the tool. ....	60
Figure 22. Banner shown during the tool launch.....	66
Figure 23. Origin verification.....	66
Figure 24. WebSocket opening handshake on the user interface.....	67
Figure 25. WebSocket messaging with another client.....	67
Figure 26. Log file from the tool.....	68
Figure 27. Failed WebSocket connection establishment.....	68

## Tables

Table 1. The WebSocket protocol support on modern browsers (supported since version). .....	14
Table 2. List of possible WebSocket headers during the opening handshake. (Lombardi 2015, 115.) .....	16
Table 3. Explanation of every property in the opening handshake request. (Fielding and others 2014, 6.7; Lombardi 2015, 115.) .....	18
Table 4. Explanation of every property in the opening handshake response. (Fielding and others 2014, 6.7; Lombardi 2015, 115.) .....	19
Table 5. The pre-defined close frame status codes in the WebSocket protocol. (Lombardi 2015, 120.) .....	23
Table 6. Close frame status code ranges in the WebSocket protocol. (Lombardi 2015, 121.) .....	24



## Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BEAST	Browser Exploit Against SSL/TLS
CSWSH	Cross-Site WebSocket Hijacking
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTP/1.1	Hypertext Transfer Protocol version 1.1
HTTP/2	Hypertext Transfer Protocol version 2
HTTPS	Hypertext Transfer Protocol over Transport Layer Security
IETF	Internet Engineering Task Force
JRE	Java Runtime Environment
LDAP	Lightweight Directory Access Protocol
MitM	Man-in-the-Middle
OWASP	Open Web Application Security Project
RFC	Request for Comments
PC	Personal Computer
PKI	Public-Key Infrastructure
SDL	Security Development Lifecycle
SQL	Structured Query Language
SSL	Secure Socket Layer
TLS	Transport Layer Security
URI	Uniform Resource Identifier
UTF-8	Universal Coded Character Set + Transformation Format – 8-bit
W3C	World Wide Web Consortium
WS	WebSocket
WSAPI	WebSocket Application Programming Interface
WSS	WebSocket over Transport Layer Security
XSS	Cross-Site Scripting
ZAP	Zed Attack Proxy

# 1 Introduction

## 1.1 Research Background

In the beginning of the Web the pages were static. They only used to show content with minimal interactivity and functionality. From that time the Web has evolved a great deal bringing so called web applications available for the users of the Web with the same kind of functionality previously only used in desktop applications. There are now bidirectional communication web applications, for example for watching videos, reading emails and writing documents, and they are available for users of the Web as long as they have a device equipped with a modern browser.

In order to deliver these kind of applications as web applications the usage of various technologies is required. They have to be able to handle a huge amount of network traffic and at the same time serve countless concurrent web application users. Also, these technologies need to be as efficient as possible from the network traffic amount and performance point of view.

When bringing new technologies to the world of the Web, the security of web applications is one of the most important matters to consider along with functionality and efficiency. Without proper security decisions already made during the planning phase of a new technology data confidentiality, integrity or availability, any of the three tenets of traditional information security could potentially be compromised.

WebSocket is one of the technologies used to bring bidirectional communication and the needed security, functionality, and efficiency to web applications. However, some of the security decisions are made during the development phase of a service, and some of them are configurable later on by a WebSocket service provider. Still all of the decisions can affect the overall security of any WebSocket implementation. This is one of the reasons why the testing of WebSocket and its security is necessary in the same way as with all the other web technology.

This thesis analyzes how WebSocket implementations are implemented on the Web, what the common security issues in them are, how those security issues can be

tested, and whether it is possible to develop a tool capable of carrying out the major part of the WebSocket implementation testing.

## 1.2 Commissioner

The Security Testing of WebSockets research is made for a company called Silverskin Information Security. Silverskin Information Security is a Finnish security consultancy company with roughly 20 employees.

## 1.3 Problem Definition

Commonly available web security testing tools such as Burp Suite or OWASP ZAP are unsuitable for testing the security of a WebSocket implementation automatically.

This leaves it as a manual task requiring extensive knowledge of how the implementations work, what the common security issues in them are, and how they can be security tested. The objective is to develop a tool that is able to test the security issues in WebSocket implementations easily with intermediate level knowledge of security issues in WebSocket implementations.

In order to reach the objective it needs to be understood what the common security issues in WebSocket implementations are. Issues can reside on multiple levels; therefore, clear scoping must be set to only limit the issues to WebSocket alone.

With knowledge of common security issues available, the reader must understand how these security issues can be tested in theory and in practice. This requires deep technical knowledge of the WebSocket protocol security and web security testing tools.

In order to develop the tool commonly available programming languages must be analyzed to find a language with production-quality WebSocket libraries available. After choosing the appropriate programming language the tool must be designed, implemented, and tested. It might be that some of the security issues are on a level that cannot simply be tested with this kind of tool but instead requires deeper understanding of the specific implementation. This might then lead to a decision to exclude some security issues from the scope of the tool.

## 1.4 Research Methodology

In order to reach the set objectives, a proper research method must be chosen. A design research based approach with qualitative methodologies was chosen for this thesis where the research aims to develop a completely new tool.

Design research is used to solve a problem or to improve a previously known matter (e.g. a product). This research obeys both facts as the problem is well known and the steps for improving the security testing process are known.

There are two separate processes in design research:

- **Development work**, with a target. E.g. a process, a product, a service or an action.
- **Research**, which is the process where the actual thesis is created.

Development work follows a known process for the phenomenon whereas research follows the research process and methods that are dependent on the actual phenomenon and its nature. The researcher must understand the target of development and the development process. The actual research work is targeted at the beginning and the ending stages of the tool development process. (Kananen 2012, 45.)

Qualitative methodologies in this research are split into four sections. (Kananen 2012, 93.):

- **Planning** includes scoping, the research problem, a literature overview for theory parts, and validity.
- **Information gathering** includes observation, theme interview, and literature sources.
- **Analysis** includes programming, categorization, and content analysis.
- **Interpretation** includes conclusions and validity.

All of these methods are used during this research process in order to reach the set objectives.

## 1.5 Literary Overview

WebSocket has been analyzed from a security point of view in the past by other researchers. Tomi Fagerlund published a paper titled *Websocket-protokollan*

*tietoturva selainsovelluksissa* [WebSocket protocol's security in web applications] in 2014. Fagerlund's research was based on WebSocket as a technology in comparison to other similar technologies and theoretical review on WebSocket security issues. (Fagerlund 2014.)

A short paper titled *WebSocket Security Analysis* was published by Jussi-Pekka Erkkilä in 2012. Erkkilä's analysis details the security issues in WebSockets and what kind of steps could be taken to mitigate them. (Erkkilä 2012.)

The focus of the publications mentioned above has been more theoretical than that of this research.

## 2 WebSocket Protocol

### 2.1 History of WebSockets

Historically, developing web applications that need bidirectional communication between a client and a server has required abusing HTTP (Hypertext Transfer Protocol) to achieve the required functionality. Abuse of HTTP for bidirectional communication leads to suboptimal use of HTTP connections, causing unnecessary problems for communication parties. A simpler solution would be to use a single TCP connection for traffic in both directions instead of abusing the HTTP protocol.

In June 2008 a step to solve this problem was taken in the working draft 10 of HTML5 specification, where the project name for this new functionality was TCPConnection, a TCP-based socket API (W3C 2008). Later on in July 2008 in the HTML5 specification the functionality was renamed to WebSocket. WebSocket was introduced with the following objectives. (Comet Daily 2008.):

- Seamlessly traverse firewalls and routers.
- Allow duly authorized cross-domain communication.
- Integrate well with cookie-based authentication.
- Integrate with existing HTTP load balancers.
- Be compatible with binary data.

The WebSocket protocol was originally developed by the W3C and the WHATWG group, however, it was moved under development of IETF in February 2010. After multiple years of draft versioning, IETF published a finished version of WebSocket protocol (RFC 6455) in December 2011 (Hickson and others 2010). The WebSocket protocol was now ready to combine a single TCP connection with the WebSocket API (WSAPI) providing an alternative to HTTP polling for bidirectional communication between the client and the server. (Fette and others 2011, 1-4.)

### 2.2 Bidirectional Communication in Web Applications

Developing web applications that need bidirectional communication between a client and a server has required various HTTP communication methods with varying usage requirements. Nowadays, the most widely used method, Long-polling, involves

abusing HTTP to poll the server for updates while sending upstream notifications as distinct HTTP requests. (Fette and others 2011, 4.)

Commonly used methods (Chopra 2015, 11-12) are listed below:

- **Request/Response:** A mechanism in which the client sends a request to the server and receives a response. This process is driven by some interaction such as the click of a button on the webpage to refresh the whole page. When Asynchronous JavaScript and XML (AJAX) entered the picture, it made the webpages dynamic through the usage of JavaScript automation and helped in loading some part of the webpage without loading the whole page again.
- **Polling:** A mechanism for scenarios where data needs to be refreshed without user interaction, such as the score of a football match. In polling, the data is fetched after a set period of time and it keeps hitting the server, regardless of whether the data has changed or not. This causes unnecessary requests to the server, opening a connection and then closing it every time.
- **Long-polling:** A mechanism abusing Request/Response where the connection kept open for a particular time period. When the client uses long polling the server responds to the client only after the data is ready to be sent, which differs from the traditional Request/Response method where the response is sent to the client right after the request. This is one of the ways to achieve real-time communication, but it works only with known time intervals.

The fact that all of these methods utilize HTTP for communication introduces a variety of problems. (Fette and others 2011, 4.):

- The server is forced to use a number of different underlying TCP (Transmission Control Protocol) connections for each client: one for sending information to the client and a new one for each incoming message.
- The wire protocol has a high overhead, with each client-to-server message having a HTTP header.
- The client-side script is forced to maintain a mapping from outgoing connections to incoming connection to track replies.

The WebSocket protocol was designed to supersede the existing bidirectional communication technologies that utilize HTTP as a transport layer to benefit from the existing infrastructure (proxies, filtering, authentication). The WebSocket protocol addresses the problems of existing bidirectional HTTP technologies in the context of existing HTTP infrastructure. The WebSocket protocol is designed to work over HTTP ports 80 and 443 as well as support HTTP proxies and intermediaries even

if this implies some complexity specific to the current environment. However, the design does not limit the WebSocket protocol to HTTP, and future implementations could use a simpler handshake than the currently used one, over a dedicated port without reinventing the entire protocol. (Fette and others 2011, 4-5.)

### 2.3 WebSocket Benefits

WebSocket is not the only technology to provide bidirectional communication between the client and the server; however, currently it is the most efficient one. The list of the benefits of the WebSocket protocol according to Chopra (2015, 3-4) as follows includes:

- Full-duplex communication.
- Low bandwidth consumption.
- Security.
- Low latency.
- Works over Transmission Control Protocol (TCP) (although it needs HTTP for initial handshake).
- Supported by almost all the web browsers and web servers including mobile browsers.

The typical usage case for the WebSocket protocol is to use it in a web application running on a server that simultaneously handles WebSocket and traditional HTTP requests with a browser as the client. This does not limit the use of the WebSocket protocol to browser applications – it can also be used in any client or server application. This makes the efficiency of WebSocket protocol utilizable e.g. with desktop applications.

### 2.4 WebSocket Browser Support

Browser vendors quickly adapted by providing the support for the WebSocket protocol. After the release of the RFC 6455 standard in December 2011, Chrome released the support for the WebSocket protocol during the same month within their software version 16 (Webkit 2011), compared to Firefox where support for the WebSocket protocol was released within their version 11 in March 2012. (Mozilla Developer Network 2016.)



The WebSocket protocol support in different browsers is listed in Table 1. (Wikipedia 2016.):

Table 1. The WebSocket protocol support on modern browsers (supported since version).

<b>WebSocket protocol</b>	<b>Internet Explorer</b>	<b>Firefox (PC)</b>	<b>Firefox (Android)</b>	<b>Chrome (PC, Mobile)</b>	<b>Safari (Mac, iOS)</b>	<b>Opera (PC, Mobile)</b>	<b>Android Browser</b>
13 RFC 6455	10	11	11	16	6	12.10	4.4

As can be seen in Table 1, all the major browser brands currently support usage of the WebSocket protocol.

Separate non-browser related libraries for creating client side WebSocket implementations are available for different programming languages – e.g. ClientWebSocket for .NET and websocket-client for Python. These libraries are used in applications that are not used as browser applications.

## 2.5 Relationship to TCP and HTTP

The WebSocket protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP server as an Upgrade request to upgrade a connection from HTTP to a WebSocket connection. By default, the WebSocket protocol uses port 80 for unencrypted WebSocket connections and port 443 for encrypted WebSocket connections tunneled over Transport Layer Security (TLS) in the same way as in HTTPS. (Fette and others Ltd 2011, 11.)

## 3 WebSocket Communication

### 3.1 Overview

The WebSocket protocol used with web applications begins its connection to the server with a simple HTTP request. A browser or another client that supports WebSocket send the server a request with specific HTTP headers that asks for a connection upgrade to use WebSocket. This process is called an opening handshake. (Lombardi 2015, 112.)

After the upgrade to use WebSocket has been agreed between the parties, a WebSocket connection is established. The client or the server can then send data both ways (full-duplex) in binary format. Depending on the opcode used in WebSocket frame, the WebSocket protocol converts the provided UTF-8 encoded text to binary format for transit and then converts it back to UTF-8 encoded text or just transmits the data in binary format without data format conversions. The connection is closed after one of the parties has sent a closing handshake.

The possibility to upgrade a connection with a header was introduced in HTTP/1.1 standard (RFC 2616) to allow the client to notify the server of alternate means of communication. It is primarily used at this point as a means of upgrading HTTP to use WebSocket or HTTP/2. (Lombardi 2015, 112-113.)

Table 2 lists the headers used during the WebSocket opening handshake with the explanation of every HTTP header and its value. Additional headers can be used if the application's developer wants to use them e.g. for additional security or functionality.

Table 2. List of possible WebSocket headers during the opening handshake. (Lombardi 2015, 115.)

HTTP header	Required	Explanation
Host	Yes	Header field containing a host name of server from where a WebSocket service is requested.
Upgrade	Yes	<i>websocket</i> as a static value. Used to notify the WebSocket server to which protocol connection is upgraded.
Connection	Yes	<i>Upgrade</i> as a static value. Used to notify the WebSocket server of alternate means of communication.
Sec-WebSocket-Key	Yes	Nonce, randomly generated base64-encoded value by the WebSocket client. When decoded, is 16 bytes in length.
Sec-WebSocket-Version	Yes	A version of the WebSocket protocol with which the client is attempting to communicate. 13 is the latest version of the WebSocket protocol. If the WebSocket client requests a version that is not supported by the WebSocket server, the WebSocket server responds with a list of supported WebSocket versions.
Origin	No	Optional header field used to identify from where a request was originated.
Sec-WebSocket-Accept	Yes (server-side)	Nonce, randomly generated value by a WebSocket server that is used to identify the connecting party in the future communication.
Sec-WebSocket-Protocol	No	Optional header field with a list of values indicating which protocols the WebSocket client would like to speak, ordered by preference.
Sec-WebSocket-Extensions	No	Optional header field with a list of values indicating which extensions the client would like to speak.

### 3.2 WebSocket Uniform Resource Identifiers

The WebSocket specification (RFC 6455) specifies two Uniform Resource Identifiers (URIs) that can be used with the WebSocket protocol:

**ws-URI format:** "ws:" "://" host [ ":" port ] path [ "?" query ]

This format is used with non-encrypted connections of the WebSocket protocol.

**wss-URI format:** "wss:" "://" host [ ":" port ] path [ "?" query ]

This format is used with TLS-encrypted connections of the WebSocket protocol. This requires the WebSocket server to have certificates configured.

The WebSocket protocol utilizes the PKI infrastructure with TLS-encrypted connections. The WebSocket client connecting to the WebSocket server must be able to trust the authority that signed the certificate for the WebSocket server in order to enable a secure connection between the parties.

### 3.3 Opening Handshake

Before being able to send WebSocket messages between the WebSocket client and the WebSocket server, it requires that the opening handshake to be completed between the parties. Typically, in a web application, the opening handshake process starts by a user accessing a website where WebSocket communication is used. The browser then downloads a JavaScript code from the server and runs it with browser's interpreter and instructs the browser to open a WebSocket connection to the WebSocket server prescribed in the code.

The client-side JavaScript code and used WebSocket JavaScript library must contain details for the WebSocket connection opening. To open a new WebSocket connection from a browser, at minimum, the JavaScript code must contain a WebSocket URI (Uniform Resource Identifier). WebSocket connections in browser applications can occur with browser integrated JavaScript API (integrated to a browser by the browser vendor), which handles the WebSocket communication in a browser context. An example URI could be `ws://ws-example-site.com` for a non-encrypted connection or `wss://ws-example-site.com` for an encrypted connection. Additional optional and standardized HTTP headers like `Sec-WebSocket-Protocol` can be used to select a specific WebSocket protocol. Other non-standardized HTTP headers can be used e.g. for authenticating the user to the service. (Lombardi 2015, 113.)

An example opening handshake request sent by a browser based WebSocket client is illustrated as follows:

```
GET /websocket-example/ HTTP/1.1
Host: ws-example-site.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMBDL1EzLkh9GBhXDw==
Sec-WebSocket-Version: 13
Origin: http://ws-example-site.com
```

All the opening handshake request properties are explained in Table 3.

Table 3. Explanation of every property in the opening handshake request. (Fielding and others 2014, 6.7; Lombardi 2015, 115.)

Request property	Explanation
GET /websocket-example/ HTTP/1.1	<ul style="list-style-type: none"> <li>• GET - Indicates the used HTTP method used during the communication.</li> <li>• /websocket-example/ - Indicates the resource name from where the WebSocket service is requested.</li> <li>• HTTP/1.1 - Indicates the HTTP version used.</li> </ul>
Host: ws-example-site.com	Domain name for the WebSocket service.
Upgrade: websocket	Used to notify the WebSocket server to which protocol the connection is upgraded.
Connection: Upgrade	Used to notify the WebSocket server of alternate means of communication.
Sec-WebSocket-Key: x3JJHMBDL1EzLkh9GBhXDw==	Nonce, randomly generated base64-encoded value by the WebSocket client. When decoded, is 16 bytes in length.
Sec-WebSocket-Version: 13	The version of the WebSocket protocol.
Origin: http://ws-example-site.com	Origin from where the WebSocket service was called.

When the handshake request is sent and received by the WebSocket server, it must respond on it. An example response by the WebSocket server is illustrated as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

All the opening handshake response properties are explained in Table 4.

Table 4. Explanation of every property in the opening handshake response. (Fielding and others 2014, 6.7; Lombardi 2015, 115.)

Response property	Explanation
HTTP/1.1 101 Switching Protocols	<ul style="list-style-type: none"> <li>• HTTP/1.1 - Indicates the HTTP version used.</li> <li>• 101 Switching Protocols - HTTP response status code indicating that the WebSocket server has accepted to switch the protocol to the WebSocket protocol.</li> </ul>
Upgrade: websocket	Used to confirm the WebSocket client to which protocol the connection is upgraded.
Connection: Upgrade	Used to confirm the WebSocket client of alternate means of communication.
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=	Nonce, randomly generated value by the WebSocket server that is used to identify the connecting party in the future communication.

### 3.4 WebSocket Frame

When the communication channel between the WebSocket client and the WebSocket server has been established, the parties can start to communicate between each other with WebSocket messages. WebSocket messages use WebSocket frames to communicate. Because the communication between the WebSocket client and the WebSocket server is bidirectional, either party can send data back and forth at any time, as long as no close frame (closing handshake) was previously sent. (Lombardi 2015, 115.)

### 3.4.1 WebSocket Frame Format

A high-level overview of the framing is illustrated in Figure 1.

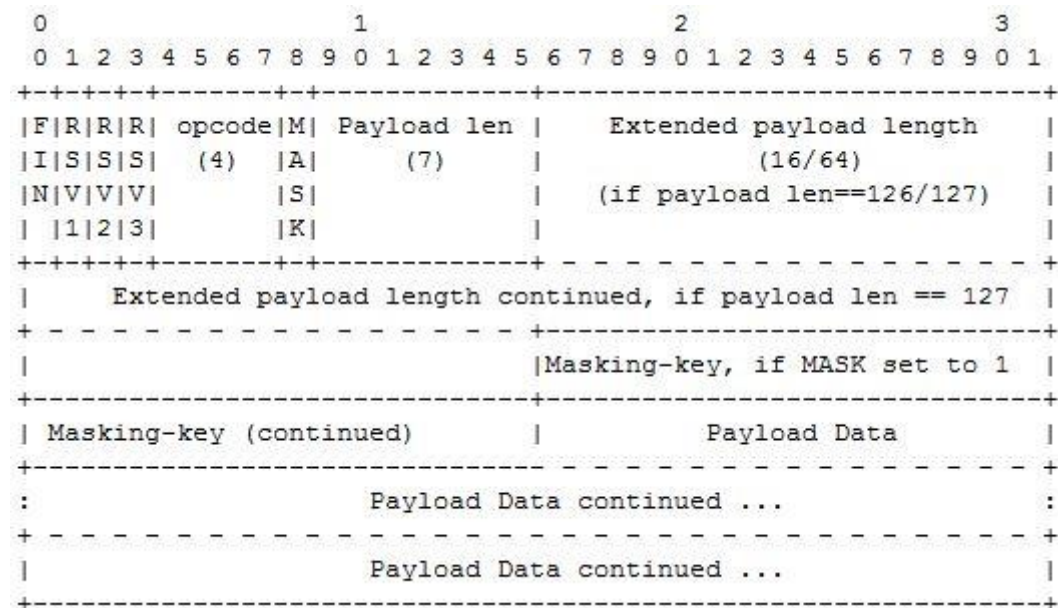


Figure 1. WebSocket frame format.

According to Fette and others (2011, 28-30), segments of the WebSocket frame are described as the following:

**FIN:** 1 bit

Indicates that this is the final fragment in a message. The first fragment may also be the final fragment.

**RSV1, RSV2, RSV3:** 1 bit each

Must be 0 unless an extension is negotiated that defines meanings for non-zero values. If a nonzero value is received and none of the negotiated extensions defines the meaning of such a non-zero value, the receiving endpoint must fail the WebSocket connection.

**Opcode:** 4 bits

Defines the interpretation of the Payload data. If an unknown opcode is received, the receiving endpoint must fail the WebSocket connection.

The following values are defined:

- %x0 denotes a continuation frame
- %x1 denotes a text frame
- %x2 denotes a binary frame
- %x3-7 are reserved for further non-control frames
- %x8 denotes a connection close
- %x9 denotes a ping
- %xA denotes a pong
- %xB-F are reserved for further control frames.

**Mask:** 1 bit

Defines whether the Payload data is masked. If set to 1, a masking key is present in masking-key, and this is used to unmask the Payload data. All frames sent from the WebSocket client to the WebSocket server have this bit set to 1.

**Payload length:** 7 bits, 7+16 bits, or 7+64 bits

The length of the Payload data, in bytes: If 0-125, that is the payload length.

- If 126, the following 2 bytes interpreted as a 16-bit unsigned integer are the payload length.
- If 127, the following 8 bytes interpreted as a 64-bit unsigned integer (the most significant bit must be 0) are the payload length.
- Multibyte length quantities are expressed in network byte order.

Note that in all cases, the minimal number of bytes must be used to encode the length, for example, the length of a 124-byte-long string cannot be encoded as the sequence 126, 0, 124. The payload length is the length of the Extension data + the length of the Application data. The length of the Extension data may be zero, in which case the payload length is the length of the Application data.



**Masking-key:** 0 or 4 bytes

All frames sent from the WebSocket client to the WebSocket server are masked by a 32-bit value that is contained within the frame. This field is present if the mask bit is set to 1 and is absent if the mask bit is set to 0. See Section 5.3 for further information on client-to-server masking.

**Payload data:** (x+y) bytes

The "Payload data" is defined as "Extension data" concatenated with "Application data".

**Extension data:** x bytes

The Extension data is 0 bytes unless an extension has been negotiated. Any extension must specify the length of the Extension data, or how that length may be calculated, and how the extension use must be negotiated during the opening handshake. If present, the Extension data is included in the total payload length.

**Application data:** y bytes

Arbitrary Application data, taking up the remainder of the frame after any Extension data. The length of the Application data is equal to the payload length minus the length of the Extension data.

It is important to note that the representation of data in frames is binary, not ASCII characters. As such, a field with a length of 1 bit that takes values %x0 / %x1 is represented as a single bit whose value is 0 or 1.

Masking is used to mask the Payload data uniquely to prevent cache poisoning attacks against proxy servers. Masking is mandatory for messages sent from the WebSocket client to the WebSocket server, but is not used at all with messages sent from the WebSocket client to the WebSocket server.

### 3.4.2 Fragmentation

One WebSocket message can include one or more frames depending on how the WebSocket server and the WebSocket client decide to send data back and forth. Splitting message into multiple frames is called fragmentation.

With the ability to fragment WebSocket messages, a party (a WebSocket client or a WebSocket server) would have to buffer the entire message before sending so it could send back an accurate count of frames. With the ability to fragment, the endpoint can choose a reasonably sized buffer, and when it is full, send another frame as a continuation until sending of the entire message is complete. (Lombardi 2015, 119.)

Fragmentation is controlled with the FIN-bit within the WebSocket frame like it was described under chapter 3.4.1 *WebSocket frame format*.

### 3.5 Closing Handshake

Closing of an established WebSocket connection requires a frame to be sent with an opcode of 0x08. If the WebSocket client sends the close frame, it must be masked like with every WebSocket message, however, it is not masked when sent by the WebSocket server.

Closing frame may contain information about why the connection was closed. Information contains a status code and a message. The status code is passed in the body of the message as a 2-byte unsigned integer. The reason string (message) is a UTF-8 encoded string. Table 5 lists the pre-defined close frame status codes in the WebSocket protocol specification (RFC 6455).

Table 5. The pre-defined close frame status codes in the WebSocket protocol. (Lombardi 2015, 120.)

Status code	Reason	Description
1000	Normal closure	An application has successfully completed.
1001	Going away	A WebSocket server or a WebSocket client application is shutting down or closing without expectation of continuing.
1002	Protocol error	A connection is closing with a protocol error.
1003	Unsupported data	A WebSocket client application or a WebSocket server application has received a message of an unexpected type that it cannot handle.
1004	Reserved	Reserved range for a future use as per RFC 6455.
1005	No status rcvd	No valid code was received.
1006	Abnormal closure	A connection has closed abnormally.

1007	Invalid frame payload data	A message received was not consistent with the type off the message (e.g., non-UTF-8).
1008	Policy violation	A message received has violated a policy. This is a generic status code that can be returned when there are no more suitable status codes.
1009	Message too big	A message received was too large to process.
1010	Mandatory ext.	An extension that was requested from the WebSocket server was not returned in the WebSocket handshake.
1011	Internal error	A connection is terminated due to an unexpected condition.
1012	Service restart	A WebSocket service is restarted, and a WebSocket client that reconnects should do so with a randomized delay off 5-30 seconds.
1013	Try again later	A WebSocket server is overloaded and a WebSocket client should either connect to a different IP (given multiple targets), or reconnect to the same IP when user has performed an action.
1014	Unassigned	Unassigned and not in use.
1015	TLS handshake	A TLS handshake has failed.

The close frame status codes listed in table 5 are only listing the status codes that are pre-defined in the WebSocket protocol specification (RFC 6455). Other status code ranges are also available for custom usage. Table 6 lists the status code ranges that can be used in other situations.

Table 6. Close frame status code ranges in the WebSocket protocol. (Lombardi 2015, 121.)

Status code range	Description
0-999	Not used for status codes.
1000-2999	Status codes are either defined by RFC 6455 or will be in future revisions.
3000-3999	Reserved for libraries, frameworks, and applications.
4000-4999	Reserved for private use, and is not registered with the IANA. Feel free to use these values in your code between the WebSocket client and the WebSocket server with prior agreement.

### 3.6 WebSocket Subprotocols

The WebSocket protocol provides a support for subprotocols that can be used to support multiple applications under the same WebSocket server. The WebSocket

client can request that the WebSocket server use a specific subprotocol by including the Sec-WebSocket-Protocol header field in its WebSocket opening handshake. If it is specified, the WebSocket server needs to include the same field and one of the selected subprotocol values in its response for the connection to be established. (Fette and others 2011, 12.)

The WebSocket protocol specification defines the recommended naming of subprotocols. The recommendation is to use names that contain the ASCII version of the domain name of the subprotocol's originator. For example, if a Company were to create a Chat subprotocol to be implemented by many servers around the Web, they should name it "chat.company.com". If they would like to implement a newer version of the Chat subprotocol for simultaneous use, they could use subprotocol "chat2.company.com", then the two subprotocols could be implemented by WebSocket servers simultaneously, with the WebSocket server dynamically selecting which subprotocol to use based on the value sent by the client. (Fette and others 2011, 12.)

### 3.7 WebSocket Extensions

Extensions provide a mechanism for implementations to opt-in to additional protocol features. Extensions can be developed freely. To use extensions, the WebSocket client may request extensions to be used with the WebSocket connection by including the Sec-WebSocket-Extensions header field in its WebSocket opening handshake. The WebSocket server may then accept some or all extensions requested by the WebSocket client. (Fette and others 2011, 48-50.)

## 4 WebSocket Security Issues

### 4.1 State of the WebSocket Security

WebSocket is not an exception when it comes to security issues in web technologies. Many of the security controls that are available for general web technologies are also available for WebSocket. But like always, the responsibility of knowing and implementing security controls into a WebSocket application is left for the developer and/or the admin of the service.

When a human is involved in securing applications' implementations, sometimes, the security controls that are available are totally left unimplemented or are implemented wrongly. In most cases, this is due to the developer not knowing how the security controls should be implemented. It could also be due to lack of time or any other reason.

If security is not part of the application since the requirements phase, it usually results with big problems later on. Security is something that can be hard and costly to implement, but even more costly if it needs to be implemented to a ready product afterwards. This is because it usually requires major changes to the whole application. It is important to understand that the best way to secure any application is to have security relevant support processes used with the development. This set of processes is called as Security Development Lifecycle (SDL). It helps the whole development process to reach the necessary security state of an application. (Microsoft.)

Without having the knowledge of the current security state of an application, one of the best ways is to make a penetration test for the application by a trusted party. In the penetration testing, the implementation (e.g. a working web application) is tested for possible vulnerabilities by utilizing different kind of tools and techniques to find out vulnerabilities that might exist in the application. The penetration testing can also include a code review, where the application's source code is reviewed for finding out and confirming vulnerabilities. The penetration testing is part of standard SDL processes.

In addition to penetration testing, when in production, the application or the surrounding infrastructure should have an ability to detect attacks occurring against the application. Allowing attackers to repeatedly attack the application to discover or exploit vulnerabilities that have not been protected against can lead to major issues. Capability to detect attackers trying to discover or exploit vulnerabilities as a security control should not be underestimated.

## 4.2 Authentication

Authentication is a way to verify a legitimate user to a service. According to Fette and others (2011, 53), the WebSocket protocol does not prescribe any particular way that WebSocket servers can authenticate clients during the WebSocket handshake.

On web applications one of the most commonly seen methods of implementing authentication is a form-based authentication. Using the Set-Cookie header field, an HTTP server can pass name/value pairs and associated metadata (called cookies) to a client (e.g. browser). In the form-based authentication, the client is assigned with a name/value pair after the user is successfully authenticated to the service. When the client makes subsequent requests to the server, the client returns the name/value pair as a part of the Cookie header. (Barth and others 2011, 3.)

When combining the form-based authentication with a WebSocket service, the actual authentication must happen before the WebSocket handshake. The assigned the name/value pair received after the authentication is used with the WebSocket handshake as a part of the Cookie header field. Figure 2 illustrates the process.

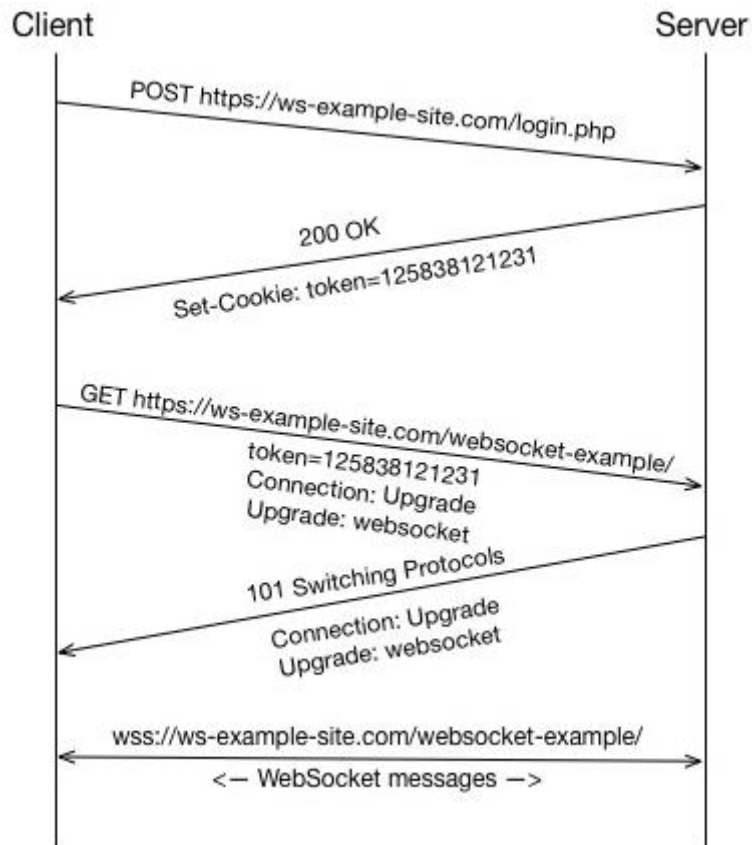


Figure 2. The form-based authentication used in a WebSocket service.

In WebSocket the authentication does not necessary need to occur by using the form-based authentication. The WebSocket server can use any client authentication mechanism available to a generic HTTP server, such as any cookie field value, basic authentication, digest authentication, or certificate authentication. As long there is a possibility to authenticate the user in a secure manner and the WebSocket server verifies it, the authentication mechanism in question is suitable for use.

When there is no defined authentication solution prescribed in the WebSocket protocol, it leaves the implementation of authentication to developer's shoulders. The developer might then build a custom authentication scheme, but building it correctly from a security point of view, can be challenging and could lead to serious security issues. For example, an attacker could bypass the whole authentication process and access the data served by the application.

### 4.3 Authorization

Authorization ensures that the authenticated user has appropriate privileges to access resources. The resources the user have access usually depends on the user's role in the service.

With authorization, the term Principle of Least Privilege is usually mentioned. It means that the user should only have access to resources, that the user is able to perform an assigned job task, nothing else. In case of WebSocket, authorization is heavily application context dependent. There are three possible scenarios that could occur:

- An attacker is able to access a function that requires higher-level authorization to the WebSocket service than originally assigned for the user.
- An attacker is able to access a function that shows other user's restricted content.
- An attacker is able to access the WebSocket service that requires a user authorization, without any authorization.

Figure 3 illustrates the listed attack scenarios.

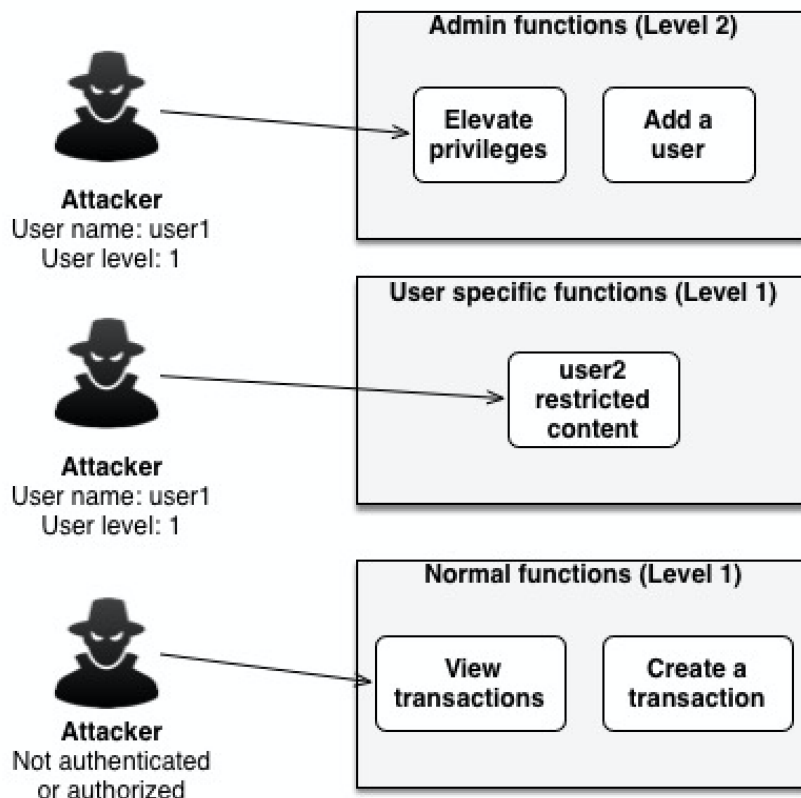


Figure 3. Attacks against authorization levels.



In the first case, the attacker is able to access functions above the authorization level assigned. The attacker has level 1 authorization, but is able to access level 2 functions.

In the second case, the attacker is able to access functions on the same authorization level of another user.

In the third case, the attacker is able to access functions of any authorization level without any assigned authorization level.

In vulnerability categorization, cases one and three are called vertical privilege escalation and case two is called horizontal privilege escalation. From security point of view, all three scenarios are problematic and can lead to serious issues, since the attacker is able to access resources that should be out of reach.

#### 4.4 Cross-Domain Requesting

WebSocket uses the origin-based security model, which is commonly used by web browsers. This means that when a user performs a WebSocket opening handshake request, the web browser adds an HTTP header named Origin to the request. The Origin header field is used to protect against unauthorized cross-origin use of a WebSocket server by scripts using the WebSocket API in a web browser. (Fette and others Ltd 2011, 7.)

An example WebSocket opening handshake request illustrated below:

```
GET /websocket-example/ HTTP/1.1  
Host: ws-example-site.com  
Origin: ws-example-site.com  
Upgrade: websocket  
Connection: Upgrade  
Sec-WebSocket-Key: x3JJHbDL1EzLkh9GBhXDw==  
Sec-WebSocket-Version: 13  
Origin: http://ws-example-site.com
```

As it can be seen from the example, the server is informed of the script origin generating the WebSocket opening handshake request (Origin: ws-example-

site.com). If the WebSocket server does not wish to accept connections from this origin, it can choose to reject the request by sending an appropriate HTTP error code. Browser clients send this header field by default; for non-browser clients, this header field may be sent if it makes sense in the context of those clients. (Fette and others Ltd 2011, 7.)

The problem with cross-domain requesting is that if the WebSocket service should only be available from predefined locations, without the proper validation of origin, an attacker could use the service from any other, potentially malicious origin. Figure 4 illustrates how an example Cross-Site WebSocket Hijacking (CSWSH) attack could occur for a victim using a legitimate service on <https://www.example-site.com> with a web browser.

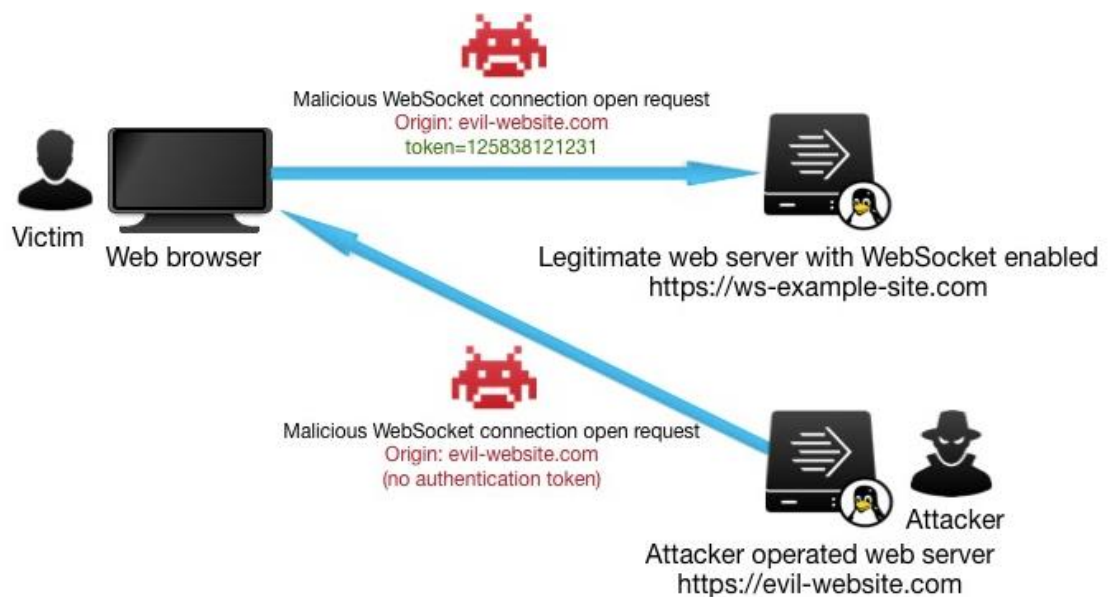


Figure 4. Cross-Site WebSocket Hijacking attack.

At first, the victim is tricked into accessing a malicious web site called <https://evil-website.com> with the victim's web browser. The attacker's web site contains a JavaScript code that instructs the victim's browser to open a WebSocket connection to a legitimate <https://ws-example-site.com> service. When the browser makes the WebSocket opening handshake request, it adds the authentication header (e.g. token=125838121231) to the request that was assigned for the victim when accessed the legitimate service before the scenario happened. Now when the legitimate server inspects the incoming WebSocket opening handshake request, it checks that the token is valid and proceeds with the opening handshake normally and opens the

WebSocket connection. The legitimate server did not validate the request origin and the code loaded from attacker's web site can use the WebSocket service maliciously.

#### 4.5 Traffic Encryption

Encryption is used to convert data into another format, called ciphertext, which can only be read by authorized parties. Without the usage of encryption, appropriately positioned attacker is able listen the communication between the two communication parties. This is called as Man-in-the-Middle (MitM) attack, which is illustrated in Figure 5.

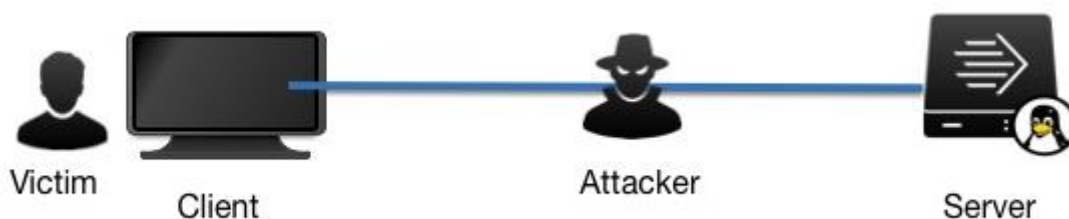


Figure 5. Man-in-the-Middle attack.

If an attacker is able to position like illustrated in Figure 5, in theory the attacker could read and modify all the communication data. This will result in problems with data confidentiality and integrity.

Masking used with WebSocket frames does not protect the data from MitM attacks, since the key used to mask the data is part of the WebSocket frame. According to Fette and others (2011, 51-52), masking was implemented to prevent proxy cache poisoning attacks, not MitM attacks. Cache poisoning attacks are attacks against a user of a service. Attacks occur when an attacker is able to insert malicious code to the cache of a proxy server, that is used by the client with the service.

According to Fette and others (2011, 11), by default, the WebSocket protocol uses port 80 for regular WebSocket connections and port 443 for WebSocket connections tunneled over Transport Layer Security (TLS). TLS is not automatically enabled between the parties and must be configured for the WebSocket service separately.

It must be noted that not all the available TLS cipher suites can be trusted to be secure. Attacks like BEAST, can be used compromise the security of TLS protocol, if

the implementation of the service is done against the best practices. (Qyalys SSL LABS 2014, 8.)

## 4.6 Input Validation

### 4.6.1 Description

In context of vulnerabilities in web applications, input validation refers to failure to properly handle input arriving from the client or the server. Handling input improperly can lead to different kinds of vulnerabilities in web applications, such as interpreter injection (e.g. SQL, LDAP, or HTML), file system attacks and buffer overflows. The server should never trust data arriving from the client, since it has every possibility to tamper with the data. (OWASP 2013.)

Attacks that are targeted against input validation can be split into two main categories:

- Attacks that are directed against the user of the application.
- Attacks that are directed against the server of the application.

Figure 6 illustrates some of the attacks that are targeted against both of the categories.

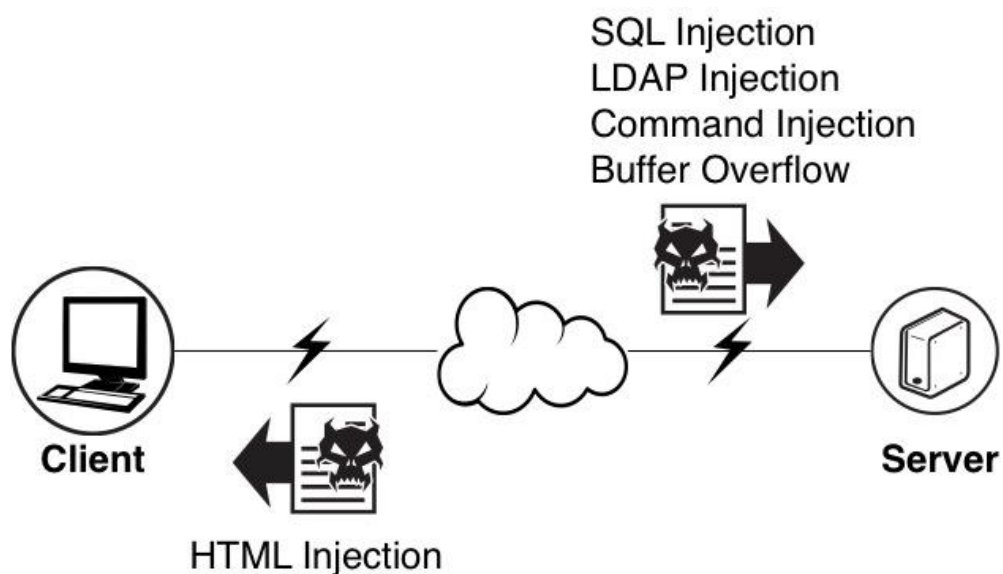


Figure 6. Example attacks against the client and the server.

From security point of view, both of the scenarios are problematic and can lead to serious security issues, since the attacker is able to affect the code or commands run

at either end's interpreter, maliciously. Normally this kind of ability to affect the code ran on either end is out of reach for the attacker.

#### 4.6.2 Server-Side Injection Attacks

SQL injection is one of the most serious forms of injections in web applications. It is caused because the server-side code does not handle the user input in a safe way. Example server-side code is shown below for an application login function:

```
uName = getRequestString("UserName");
uPass = getRequestString("UserPass");
sql = "SELECT * FROM Users WHERE Name ='" + uName + "' AND Pass ='" + uPass + "'";
```

Like it can be seen from the code content of *sql* variable, the values from *uName* and *uPass* variables are simply added as part of the SQL query that will be run by a SQL interpreter later on. Using this functionality legitimately by submitting "admin" as the *uName* and "password123" as the *uPass* the SQL query would be the following:

```
SELECT * FROM Users WHERE Name = 'admin' AND Pass = 'password123'
```

If one or more username with the submitted password is found from the database's *Users* table (SQL server returns one or more records), the login is successful. This kind functionality can be easily exploited, by submitting a malicious code as the input. The attacker could submit "admin" as the *uName* and "' OR '1'='1'" as the *uPass*. The SQL query would be the following:

```
SELECT * FROM Users WHERE Name = 'admin' AND Pass = "' OR '1'='1'"
```

This will transform the SQL query in a way that the SQL server always returns one record, the admin record and the login will occur as the admin user.

Injection attacks against the server-side can be easily avoided by utilizing parameterized queries. With parameterized queries the user input cannot affect the structure of the query, it only accept the user input as the parameter value. With SQL servers, Stored Procedures or Prepared Statements are used to provide parameterized queries to avoid SQL injections.

### 4.6.3 Client-Side Injection Attacks

In many cases, a proper encoding has the potential to defuse attacks that rely on lack of input validation. In case of HTML injection, if HTML entity encoding is used on the input before it is sent to a browser, it will prevent most of HTML injection attacks (e.g. cross-site scripting [XSS]). HTML entity encoding means replacing ASCII characters with their “HTML Entity” equivalents (e.g. replacing the “<” character with “&lt;”). This would then result the browser handling the input as content of an element, instead of handling it as a start of a HTML tag. (OWASP, 2013.)

An example of the HTML injection attack could occur through the following server-side code:

```
(String) page += "<input name='creditcard' type='TEXT' value='" +
    request.getParameter("CC") + "'>";
```

The form gets a value for its input called *creditcard*, from a parameter called *CC*, which is provided by a user as a URL parameter. Legitimate example of a *CC* parameter in a request could be the following:

```
https://ws-example-site.com/function?CC=1234567890
```

In this case, form’s input value would be “1234567890”. An attacker can leverage this functionality and modify the request into a malicious format:

```
https://ws-example-
site.com/function?='><script>document.location='https://evil-
website.com/cookiecatcher?foo='+document.cookie</script>'
```

With the malicious request, the value “><script>document.location='https://evil-website.com/cookiecatcher?foo='+document.cookie</script>” is added to the form.

This will result the browser’s interpreter running the following code:

```
<input name='creditcard' type='TEXT'
value="><script>document.location='https://evil-
website.com/cookiecatcher?foo='+document.cookie</script>'>";
```

The code instructs the browser to run a script to redirect the browser to a malicious http://evil-website.com/ web site and include the Cookie’s name/value pairs used

with the ws-example-site.com web site as an URL parameter. The attacker can record all the requests sent to the malicious web site and use the sent Cookie's name/value pairs to pose as the victim on the legitimate ws-example-site.com web site.

#### 4.6.4 Protecting Against Input Validation Related Attacks

In order to protect against input validation related attacks, the user and the server input must be sanitized (input sanitization). This means that the input must be stripped out or converted out of malicious input, that the system can then process the data in a safe way without causing problems to the user or the server of the application.

The best way to understand the protection against input validation related attacks is that the data originating from the client-side to the server-side as well as the data originating from the server-side to the client-side must be validated for a malicious input. The input should never be able to inject the client-side or the server-side code maliciously (e.g. SQL, LDAP or HTML).

The protection against client-side injections can be accomplished with proper encoding and the protection against many of the server-side injections can be accomplished by utilizing parameterized queries.

It must be noted that creating a system that can defend against all kind of input validation related attacks could be hard to implement. Many of the frameworks or other ready input validation libraries provide functionalities that can handle the input in a safe way, without the developer needing to create own security controls against these kinds of attacks. Utilizing ready functionalities are usually the safest way on approaching this problem. The most important thing is that when implemented, it must be centralized and used with all input.

### 4.7 Resource Exhaustion

Resource exhaustion is a simple denial of service condition, which occurs when the resources necessary to perform an action are entirely consumed, therefore preventing that action from taking place (OWASP 2009).

WebSocket is a connection-oriented protocol, which means that an established connection is kept open until one of the parties decides to close the connection. This can lead to a problem, if there is no proper defense on how many WebSocket connections can be opened and kept open by a single attacker. The attacker could use this weakness to attack against the service and exhaust resources of the service causing denial of service. When the service runs out of resources to open new WebSocket connections, it restricts legitimate users from accessing the service.

This attack is similar to an attack published in 2009 known as Slowloris, which still today causes harm to HTTP servers (Wikipedia 2016).



## 5 Manual Testing of WebSocket Security

### 5.1 WebSocket Security Testing Tools

There is few security testing tools available with various features to WebSocket security testing. Most popular ones of these tools are the proxy tools Burp Suite and OWASP Zed Attack Proxy (ZAP).

#### 5.1.1 Burp Suite

Burp Suite is a commercial product developed by PortSwigger for web applications' security testing. According to PortSwigger (2016), Burp Suite is an integrated platform for performing security testing of web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application's attack surface, through to finding and exploiting security vulnerabilities.

Burp Suite has provided support for viewing, intercepting and modifying WebSocket messages on the fly since January 2014 in its version 1.4.21. However, these functionalities are only part of the professional edition. Figure 7 illustrates Burp Suite's WebSocket history user interface. (PortSwigger 2014.)

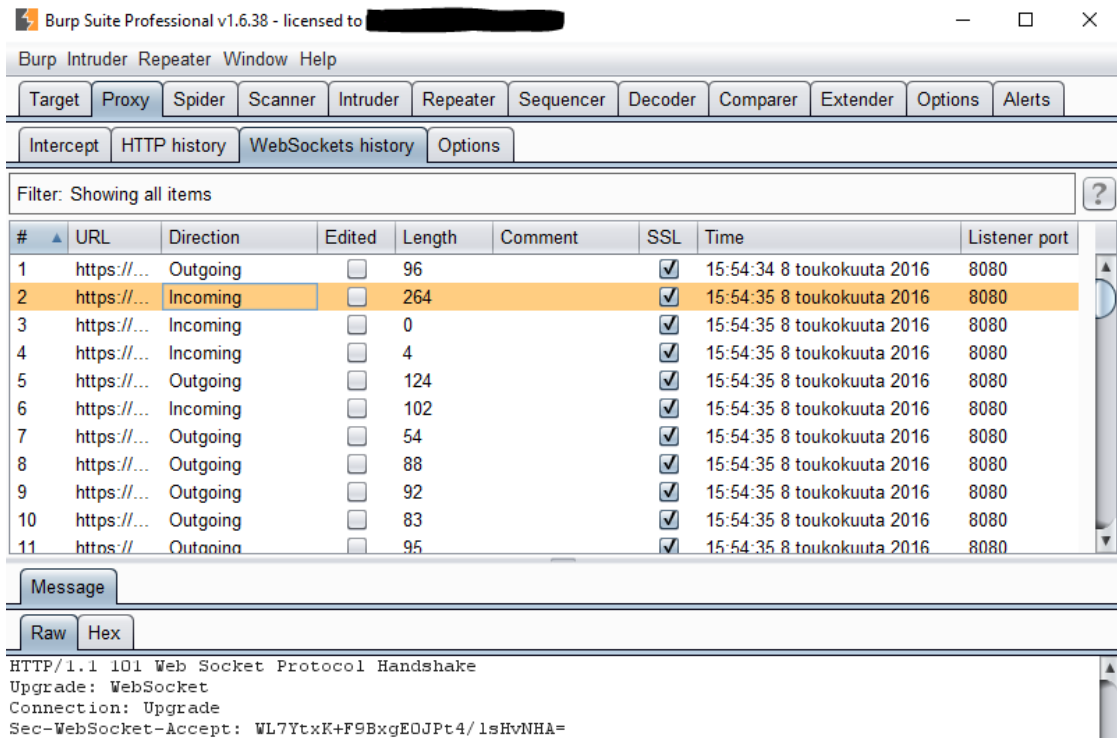


Figure 7. Burp Suite's WebSocket history user interface.

Burp Suite is written in Java and runs on any platform that has Java Runtime Environment (JRE) installed.

### 5.1.2 OWASP Zed Attack Proxy

According to OWASP (2016), the OWASP Zed Attack Proxy (ZAP) is one of the world's most popular free security tools and is actively maintained by hundreds of international volunteers. It can help you automatically find security vulnerabilities in web applications while developing and testing applications.

OWASP ZAP provides support for viewing, intercepting and modifying WebSocket messages on the fly and afterwards. Figure 8 illustrates OWASP Zed Attack Proxy's WebSocket user interface.

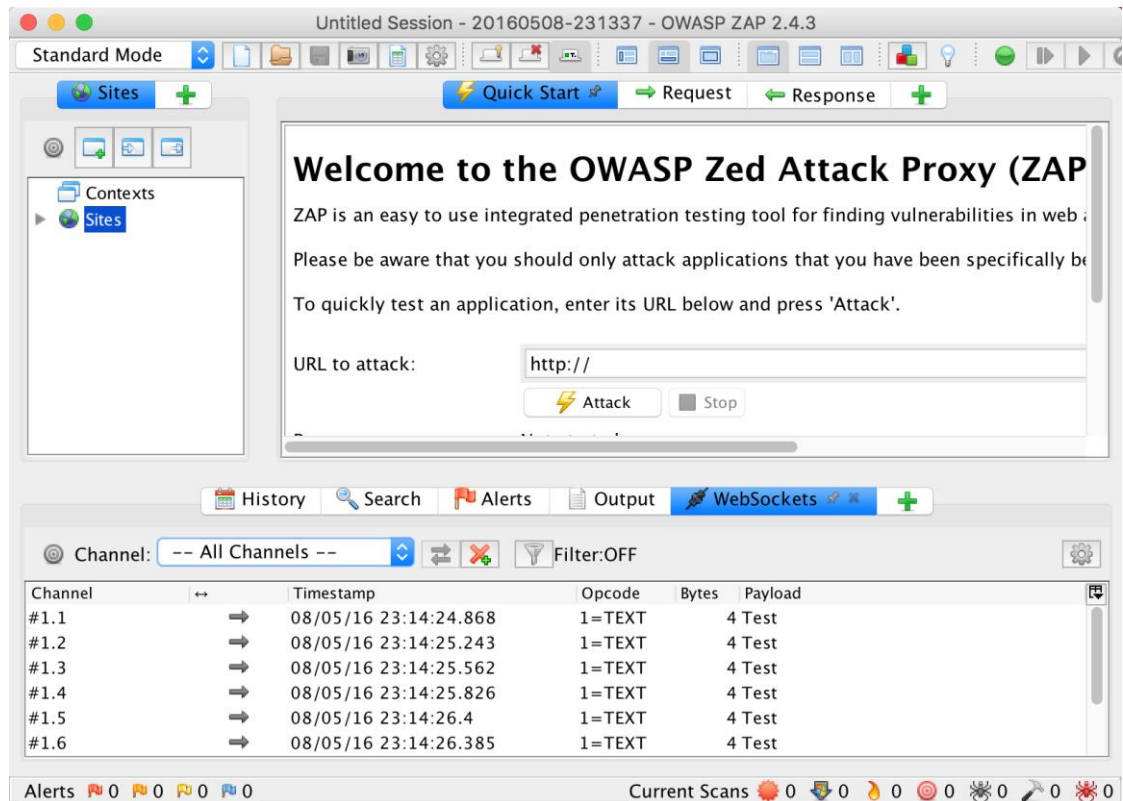


Figure 8. OWASP Zed Attack Proxy's WebSocket user interface.

OWASP ZAP is written in Java. OWASP provides installation packages for OS X, Windows and Linux with a preliminary requirement that Java Runtime Environment (JRE) is installed.

### 5.1.3 Other Tools

There are also other tools available, but they are more for WebSocket debugging purposes, not really about the security testing.

**Google Chrome's developer tools** can be used for viewing WebSocket traffic. This is not a tool for security testing, but is a good tool for inspecting what kind of messages are sent in an established WebSocket connection. Figure 9 illustrates the WebSocket view under Chrome's developer tools. (Google 2016.)

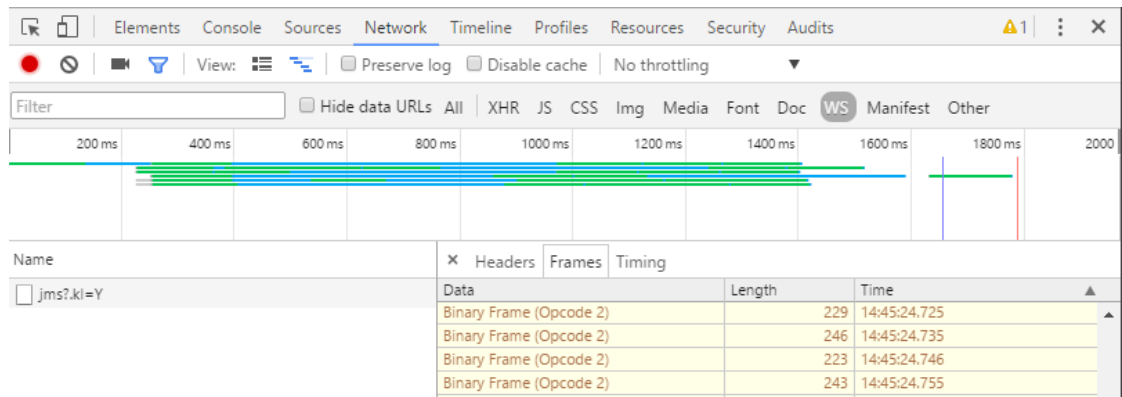


Figure 9. WebSocket traffic view in Google Chrome's developer tools.

**WebSocket.org Echo Test** is a web browser based tool that can be used to test the basic functionality of a WebSocket server. This can be used to send and receive messages with any WebSocket server that does not restrict the origin. Figure 10 illustrates the tool. (Kaazing 2016.)

## Echo Test

The first section of this page will let you do an HTML5 WebSocket test against the echo server. The second section walks you through creating a WebSocket application yourself.

You can also [inspect WebSocket messages](#) using your browser.

### Try it out!

✓ This browser supports WebSocket.

**Location:**

Use secure WebSocket (TLS)

**Message:**

**Log:**

Figure 10. WebSocket.org Echo Test tool.

Echo Test tool can be accessed from <http://www.websocket.org/echo.html>. It can also be downloaded and run from your own server.

## 5.2 Manual Testing

### 5.2.1 Proxy Tool

WebSocket security tools like Burp Suite and OWASP ZAP both utilize proxy functionality to achieve the needed functionality for security testing (PortSwigger 2016; OWASP 2016).

In scope of web security testing, proxy is a tool that can be used to reroute the traffic through a proxy server, acting as a middleman that can be used to view history, intercept, modify, or block the traffic that is occurring between the two parties. The proxy server is integrated into both Burp Suite and OWASP ZAP. When activated, the proxy server's address must be configured to a browser in order to see the traffic between the two parties in the proxy tool. Figure 11 illustrates the principle of the proxy server functionality.

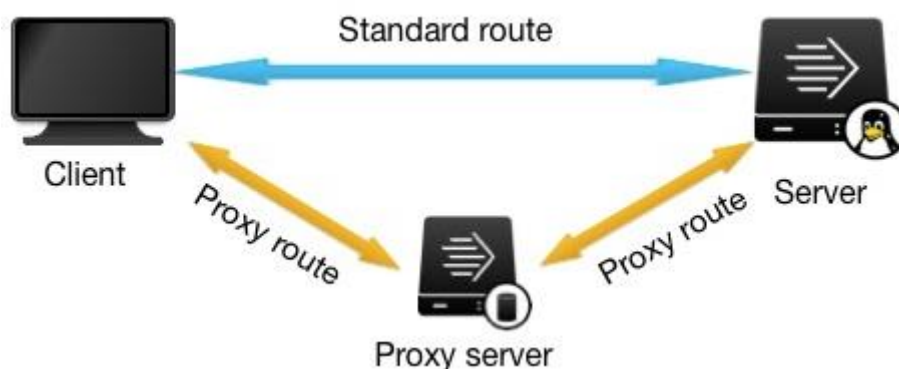


Figure 11. Principle of the proxy server functionality.

When the proxy server is configured for example to the browser client, the incoming and outgoing browser traffic is directed through the proxy server. Proxy server can freely view, intercept, modify, and block the traffic.

It must be noted that if the proxy server is used with encrypted connections (e.g. HTTPS or WSS), the client must trust the certificate authority that has signed the proxy server's certificate. In case of a security testing, root certificate used with the proxy server can be exported and installed to the client to avoid any problems with certificate trusts.

With all the following testing scenarios, it is assumed that the proxy server is configured in between the client and the server.

### 5.2.2 Authentication

To be able to test authentication issues in a WebSocket service the following knowledge about the service needs to be available:

- Address for the WebSocket service.
- Authentication mechanism(s) used (if such exist).
- One set of working credentials (if such exist).

Testing starts by opening the connection to the WebSocket service. In case of a browser based WebSocket service, open the web site where the WebSocket service is located, login to the service by using the credentials (if login is required) and the WebSocket connection should be established.

Once the connection is established, the detail on how the WebSocket opening handshake occurred is available on the proxy server's WebSocket history. From there, find the initial WebSocket client generated WebSocket opening handshake request and send it to a tool, which can resend the request.

In case of Burp Suite, resending requests can be accomplished by right clicking the request in question and selecting "Send to Repeater". Once the request is in Repeater, request can be verified for authentication related details (e.g. Cookie header). Figure 12 illustrates the send to repeater functionality.

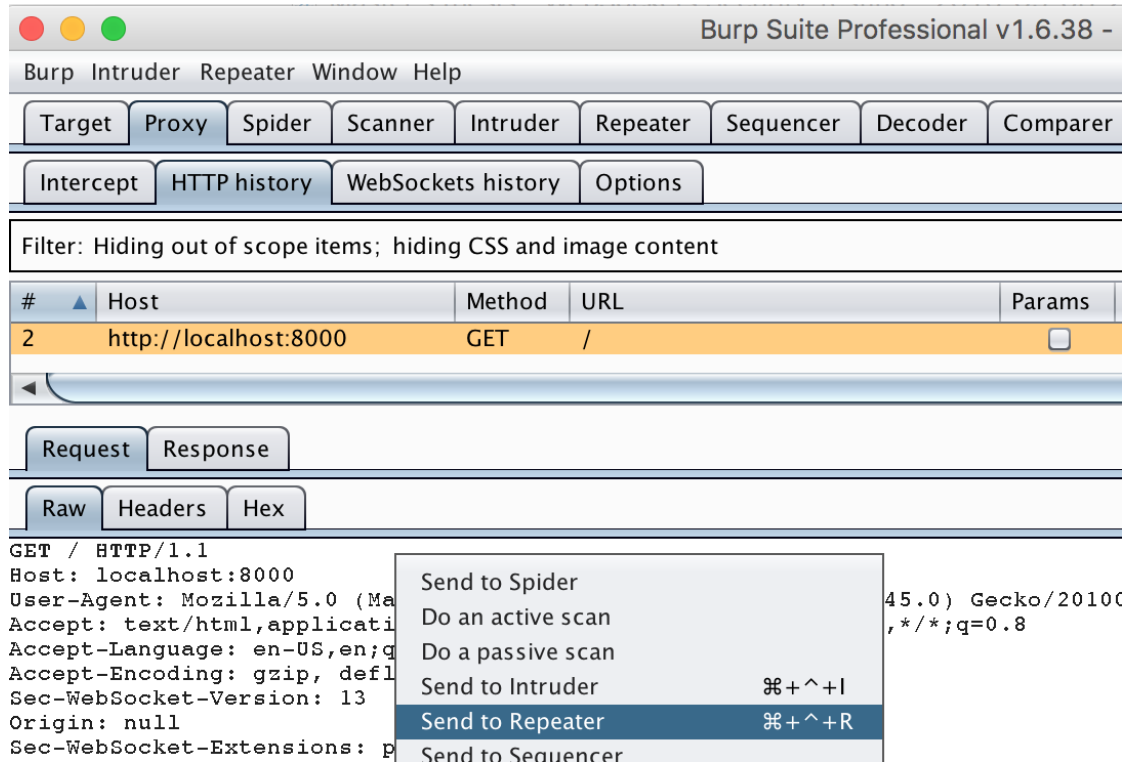


Figure 12. Sent to Repeater functionality in Burp Suite.

If the service uses form-based authentication, the client generated WebSocket opening handshake request should contain a Cookie name/value pair, or other HTTP header (starting with “X-”) that indicates the authenticated user to the server. An example header could be “Cookie: token=12583812131”. The following scenarios needs to be tested at minimum in order to validate the function of the authentication security control (others scenarios may also exist):

- Remove the whole header from the request:
  - ~~Cookie: token=12583812131~~
- Modify the header by removing the name/value pair:
  - Cookie: ~~token=12583812131~~
- Modify the header by removing the value:
  - Cookie: token=~~12583812131~~
- Modify the header by trying other values as the value:
  - Cookie: token=1
  - Cookie: token=0
  - Cookie: token=-1
  - Cookie: token=-0
  - Cookie: token=true
  - Cookie: token=false
  - Cookie: token=null
  - Cookie: token=%00

Figure 13 illustrates the modifying the cookie value to be %00 (nullbyte) character.

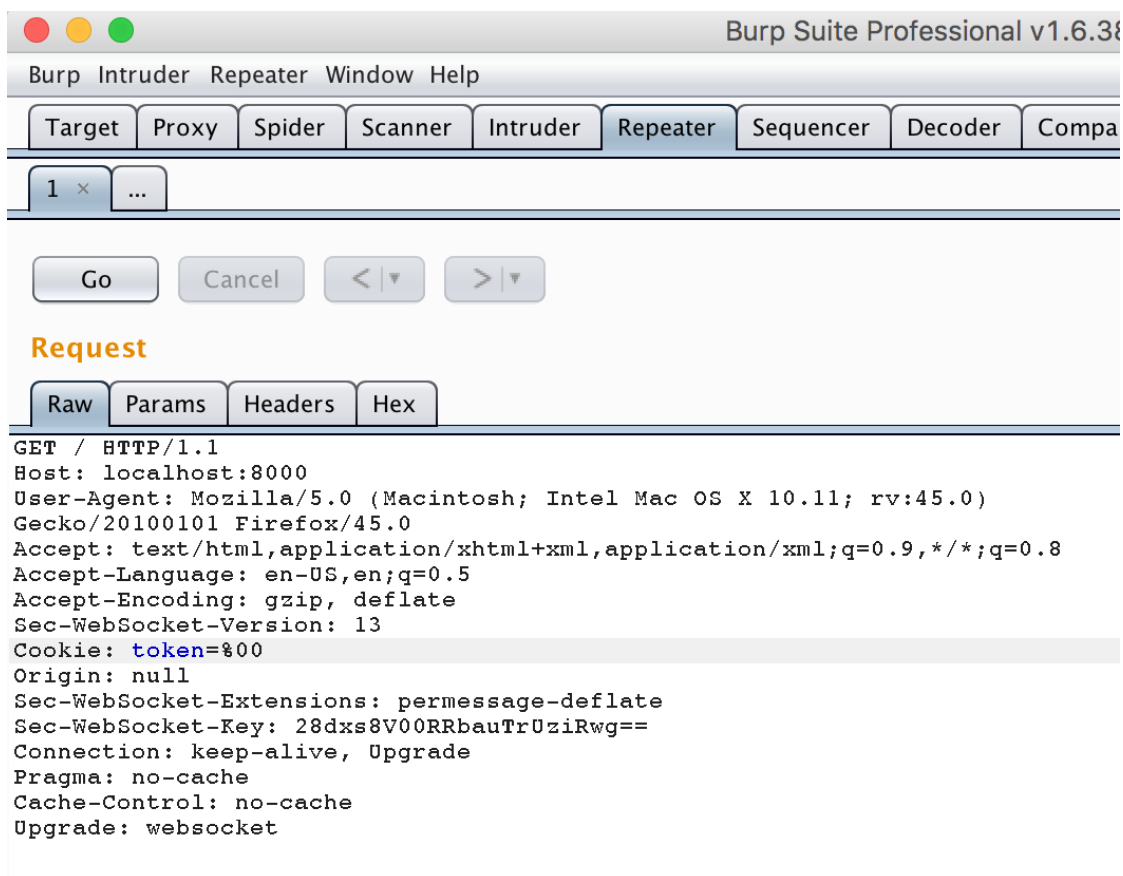


Figure 13. Nullbyte as the cookie's name/value pair's value in Burp Suite Repeater.

If the response to the modified request is similar to the following example, the service has an issue with the security control, since the authentication check can be bypassed:

```
HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: GF3mHw49oEigrTamsouTYopfKxg=
```

In case of the service's security control working correctly, you should see HTTP status code "401 Unauthorized" or similar, which restricts the access to the service.

If the service uses basic authentication, digest authentication, client certificates, or other similar authentication mechanisms, the testing in those scenarios is different. On those cases it must be verified, if the WebSocket opening handshake request can be done successfully without the user authentication. Once the request is known (found from history), Burp Suite's Repeater can be used to make the WebSocket



opening handshake request similar to case of form-based authentication to verify the functionality of service's authentication security control.

Sometimes, the implementation of WebSockets service's authentication could be a combination of using client certificates and form-based authentication. On those cases both of the authentication mechanisms must be tested for security issues.

### 5.2.3 Authorization

To be able to test authorization issues in a WebSocket service the following knowledge about the service needs to be available:

- Address for the WebSocket service.
- Authentication mechanism(s) used.
- Two or more sets of working credentials with different level of authorization (e.g. normal user and admin user).  
*OR*
- One set of working credentials with a normal user level of authorization and the knowledge of admin functions and how they are called through the WebSocket service.
- If the service has user specific functions on same authorization level, two sets of working credentials from the same authorization level.

Testing starts by opening the connection to the WebSocket service. In case of a browser based WebSocket service, open the web site where the WebSocket service is located, login to the service by using the credentials (if login is required) and the WebSocket connection should be established.

The first thing to do is to analyze the service for what kinds of WebSocket messages are sent from the WebSocket client to the WebSocket server when the service is used normally. This includes the usage of the normal user level's functions and resources as well as the functions and resources on the user level(s) above that. This requires that every user account from separate user levels is logged into the service one by one and tested separately. It is not exceptional to have three or even more user levels to be tested. The WebSocket message history in Burp is called WebSockets history and in OWASP ZAP it is called WebSockets.

Once the collection of messages sent by all users on all user levels is available, the collection must be reviewed manually and compared between each user and user level. Usually the case is that the normal user level has access to minimum amount of

functions and resources and the admin has access to all functions and resources. This can however differ between applications. As the end result from the comparison, there should be a list of messages that can be sent with each user level and what are the messages that require the higher user level than the user level in question. E.g. the admin user can create a new user to the service and a user on the normal user level cannot.

Once the list of available higher user level messages available, the proxy tool is used to send the messages to the WebSocket server and see how does it respond. Burp Suite does not have a feature for resending WebSocket messages. However, OWASP ZAP has it. Resend functionality from OWASP ZAP can be found by right clicking any WebSocket message and selecting Resend like Figure 14 illustrates.

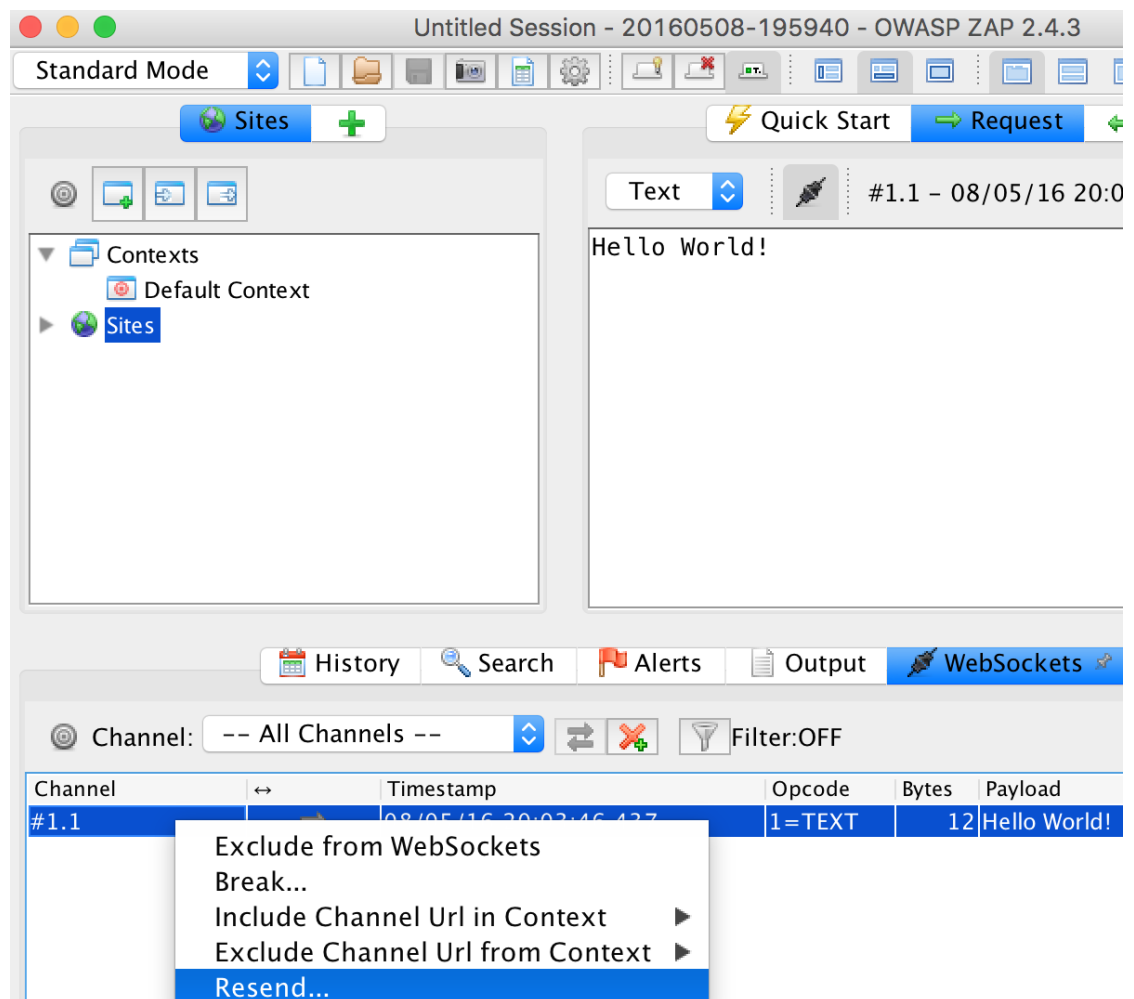


Figure 14. Access to Resend functionality in OWASP Zed Attack Proxy.

Once the Resend functionality is opened, it can be used to manually send any message. Figure 15 illustrates the Resend functionality in OWASP ZAP.

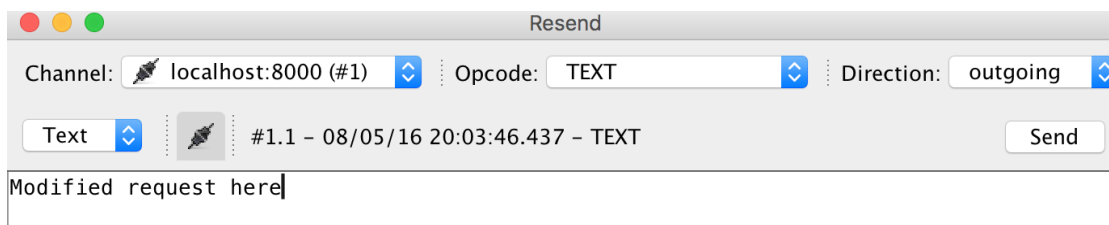


Figure 15. Resend functionality in OWASP Zed Attack Proxy.

Response to the sent message can be seen from the WebSockets history. If there is a response that includes any data that should only be accessible only by higher authorization level user or by another user only, the authorization security control is broken.

It must be noted that if there is a functionality where name/value pair is used in the message (e.g. `userId=123`) and the response should only be accessible by a specific user within the same user level, it must be tested with another user at the same user level. If another user is able to receive the data that was meant for the other user, the security control is broken.

OWASP ZAP also has a feature called Fuzz that can be used to run set of tests with a predefined list of payloads with a selected WebSocket request as separate requests with one payload at a time. Payload list can be loaded from a file or simply pasted to the tool, which can for example be used for authorization testing. Authorization testing manually might take a lot of time to test, so Fuzz feature is a way to ease the testing by automation.

#### 5.2.4 Cross-Domain Requesting

To be able to test origin validation issues in a WebSocket service the following knowledge about the service needs to be available:

- Address for the WebSocket service.
- Authentication mechanism(s) used (if such exist).
- One set of working credentials (if such exist).

Issues related to cross-domain requesting are easy to test. By default in WebSockets, the only thing that controls the cross-domain requesting is the Origin header. Origin header is sent within the WebSocket opening handshake request by the WebSocket client and validated by the WebSocket server. Proxy tools can manipulate the

request headers freely. However, in some cases, the source from where the service can be called is not limited for a reason. In those cases the testing of origin validation is not necessary.

Testing starts by opening the connection to the WebSocket service. In case of a browser based WebSocket service, open the web site where the WebSocket service is located, login to the service by using the credentials (if login is required) and the WebSocket connection should be established.

Once you have the connection established, review the proxy history and find out the WebSocket opening handshake request and send it to Burp's Repeater or ZAP's Resend tool. From the request change the Origin header's value to something else than it originally was. Figure 16 illustrates Origin modified WebSocket opening handshake request in the Burp Repeater tool.

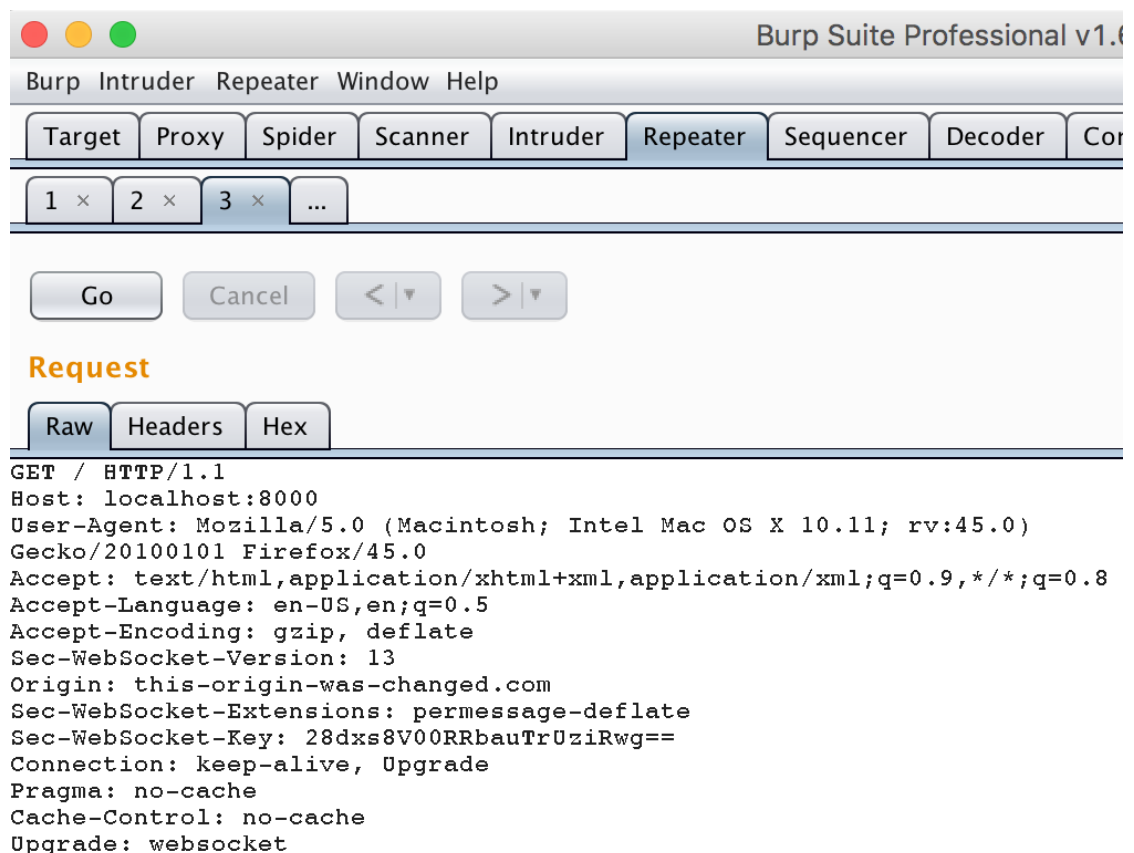


Figure 16. Origin modified WebSocket opening handshake request in Burp Repeater.

If the response to the modified request is similar to the following example, the service has an origin validation security issue, since the origin is not validated:

*HTTP/1.1 101 Switching Protocols*

*Upgrade: WebSocket*

*Connection: Upgrade*

*Sec-WebSocket-Accept: GF3mHw49oEigrTamsoUTYopfKxg=*

In case of the service's security control working correctly, response's HTTP status code is "401 Unauthorized" or similar, which restricts the access to the service from unauthorized origins.

### 5.2.5 Traffic Encryption

If a WebSocket service is not using TLS encrypted connection (URI: wss://) to the WebSocket service by default, it should be tested if the service supports TLS encrypted connections at all.

To be able to test traffic encryption issues in the WebSocket service the following knowledge about the service needs to be available:

- Address for the WebSocket service.
- Authentication mechanism(s) used (if such exist).
- One set of working credentials (if such exist).

Once you have the connection established, look at the proxy history and find out the WebSocket opening handshake request and send it to Burp's Repeater. Once the request is in repeater, select Configure target details and change a proper port and select Use HTTPS. Figure 17 illustrates the Configure target details view in Burp Repeater tool.

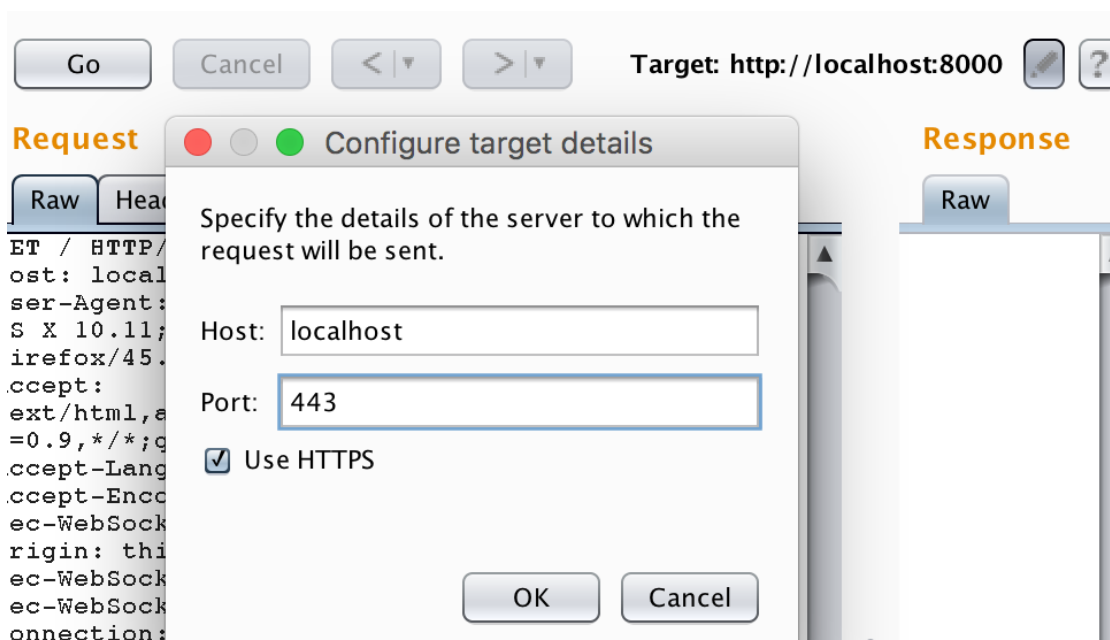


Figure 17. Configure target details view in Burp Repeater.

If the response to the modified request is similar to the following example, the service is working with TLS encryption also:

*HTTP/1.1 101 Switching Protocols*

*Upgrade: WebSocket*

*Connection: Upgrade*

*Sec-WebSocket-Accept: GF3mHw49oEigrTamsouTYopfKxg=*

In all services where TLS encryption is not used by default or TLS is not supported at all, it is a security issue. Encryption should always be used.

### 5.2.6 Input Validation

To be able to test input validation issues in a WebSocket service the following knowledge about the service needs to be available:

- Address for the WebSocket service.
- Authentication mechanism(s) used (if such exist).
- Set of working credentials for every user authorization level (if such exist).

Testing starts by opening the connection to the WebSocket service. In case of a browser based WebSocket service, open the web site where the WebSocket service is located, login to the service by using the credentials (if login is required) and the WebSocket connection should be established.

The first thing to do is to analyze the service for what kinds of WebSocket messages are sent from the WebSocket client to the WebSocket server when the service is used normally (this includes all the messages that are sent on all user authorization levels).

Once you have a list of messages used by the service, it must be reviewed for potentially problematic messages from input validation perspective. In theory every message could be injectable, but due to time or scope limitations they might need to be filtered. In those cases, a prioritization must be done for most potential problematic ones. For example, if there is a message with a content `userId=123`, it should be a good starting point for input validation testing.

To be able to test the message, a list of malicious payloads must be available that are used fuzz for potential issues in input validation. Fuzz testing should contain tests against different kind of vulnerabilities (e.g. SQL injection, XSS, Path traversal, and similar). For traditional web applications, these kinds of fuzz tests are integrated to Burp Suite's and OWASP ZAP's features (active scan), but using them with WebSocket have limitations. Since the Burp Suite does not support resending of WebSocket messages, an effective testing cannot be done with it. This is why OWASP ZAP is a better choice for this.

When there is a list of payloads available, automated testing can be accomplished by using OWASP ZAP's Fuzz tool with a customized payload list. In most cases the better way to approach this testing is to do it manually. Figure 18 illustrates Resend feature in OWASP ZAP with manually added example SQL injection payload.

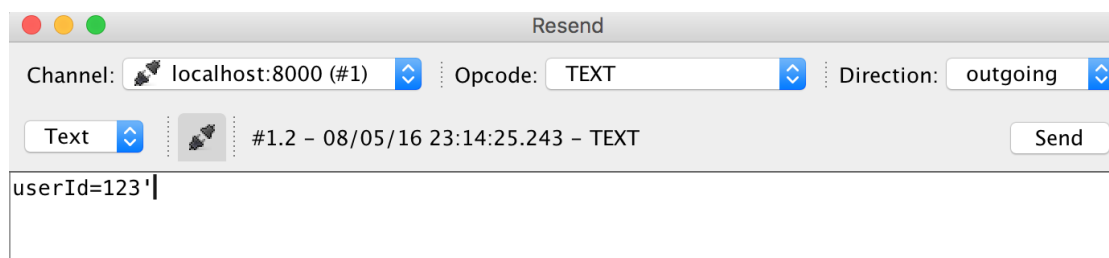


Figure 18. WebSocket message resend feature in OWASP Zed Attack Proxy.

The payload used in the Figure 18 tries to inject ' -character into the interpreted SQL code. The result for a sent message must be validated from WebSockets history for potentially problematic responses. If the response for the sent message returns a

SQL error message, it might be SQL injectable and would require some more testing to validate if that really is the case.

Input validation test scenarios with different payloads require deeper knowledge of different kind of vulnerabilities, how they can be tested and how they might impact the system. It is not efficient to describe all the possible payloads for every single type of vulnerabilities in this research, since the amount of them is huge and they are also highly context dependent.

### 5.2.7 Resource Exhaustion

To be able to test resource exhaustion issues in a WebSocket service the following knowledge about the service needs to be available:

- Address for the WebSocket service.
- Authentication mechanism(s) used (if such exist).
- Set of working credentials for every user authorization level (if such exist).

Starting point for testing is that multiple WebSocket connections must be established in order to perform the testing. At first, it must be tested if active WebSocket connections are limited to one connection per session or not.

In case the service is usable with a web browser and the service's own JavaScript code handles the WebSocket opening handshake request, it can be easily tested by opening the same service to multiple tabs and validating if it is usable in every tab. If the service is not a browser application, automation scripts must be written. Such scripts can utilize the JavaScript WebSocket API integrated into browsers or other 3<sup>rd</sup> party libraries available for different programming languages. For Python there is a library called *websocket* (part of *websocket-client*) that can be utilized for this kind of testing.

A JavaScript based testing tool to be used with the WebSocket API can be created with a few lines of JavaScript code. One example of this kind of testing tool can be found from the *simple-websocket-server* project. The tool is an html page containing JavaScript code for opening any WebSocket connection, reading messages, sending messages and closing the connection. The JavaScript based testing tool's source code can be found from appendix 1. (Pallot 2015.)



If the service is checking the Origin before establishing the connection, the Origin must be manipulated. The Origin manipulation can be accomplished for example with Burp Suite proxy while doing the testing. Figure 19 illustrates the manipulation of Origin in Burp Suite. Same tool can be used for adding an authentication header if is needed for the testing.

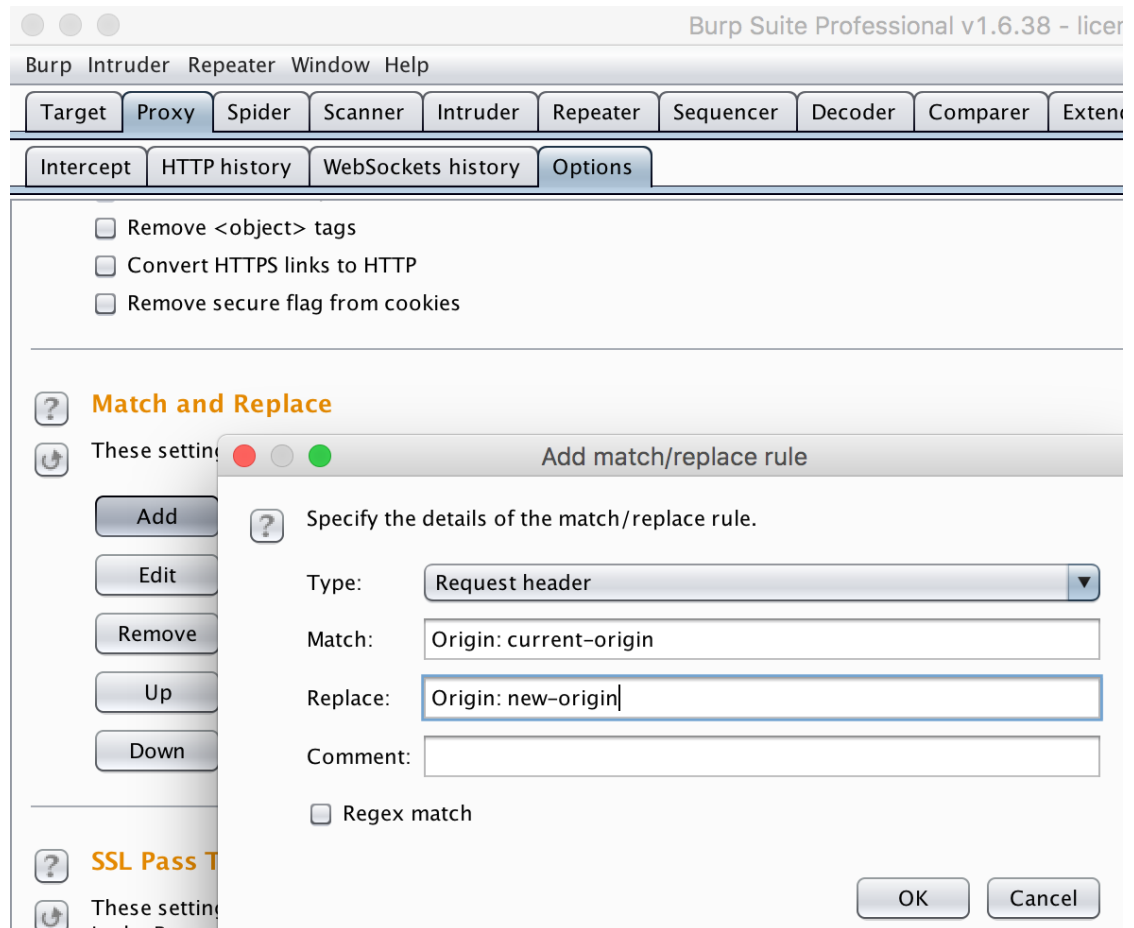


Figure 19. The Origin manipulation with Burp Suite.

A Python based testing tool can also be written with few lines of code. Code below illustrates a WebSocket client that can be used to open the WebSocket connection and send WebSocket messages (Github 2014):

```
from __future__ import print_function
import websocket

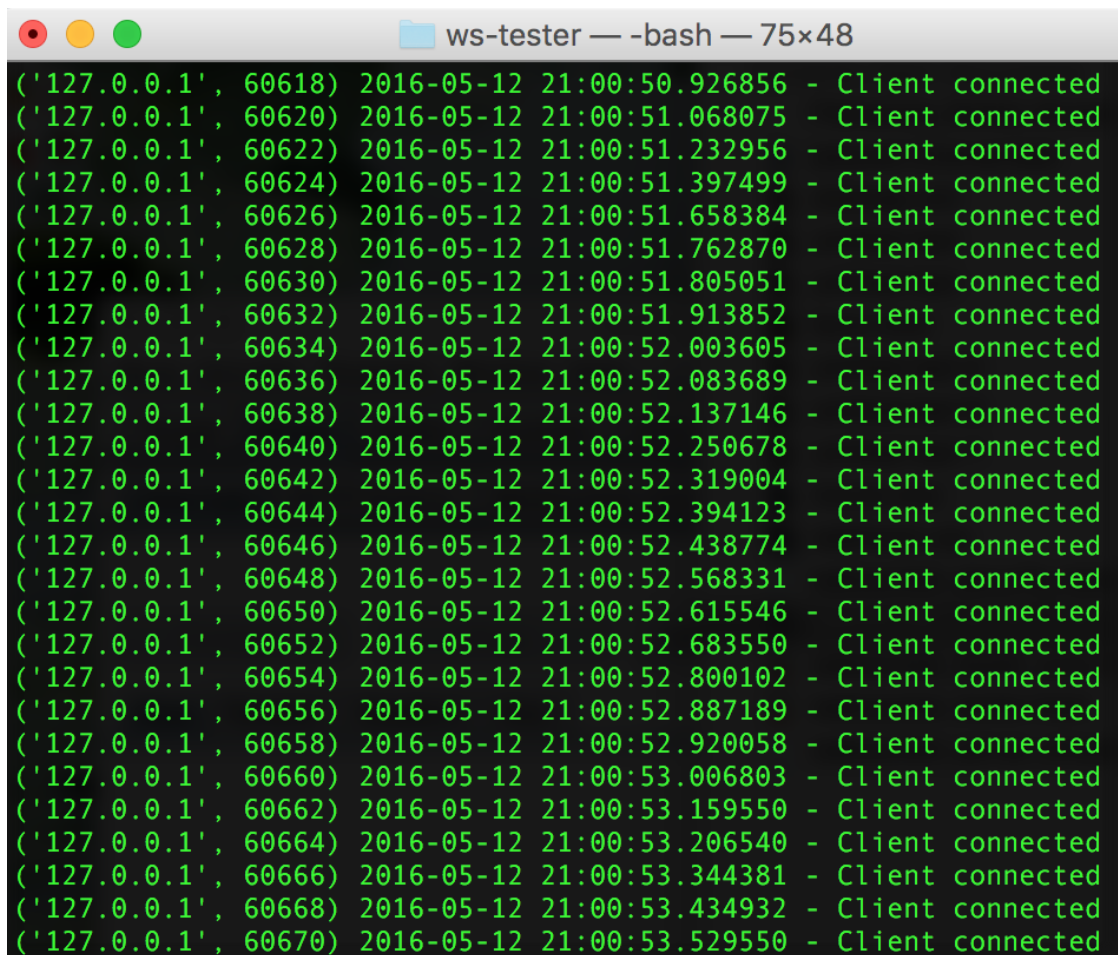
if __name__ == "__main__":
    websocket.enableTrace(True)
    ws = websocket.create_connection("ws://ws-example-site.com/")
```

```
print("Sending 'Hello, World'...")  
ws.send("Hello, World")  
print("Sent")  
print("Receiving...")  
result = ws.recv()  
print("Received '%s'" % result)  
ws.close()
```

The Origin manipulation can be accomplished by giving the Origin as an argument to the `websocket.create_connection` function. Adding a custom header (e.g. authentication header) can be added the same way. Example for this below:

```
options["origin"] = "example-domain.com"  
options["header"] = ["Cookie: token=125838121231"]  
websocket.create_connection("ws://ws-example-site.com/",  
**options)
```

Both the scripts are meant for opening one connection per running script to the service. To use them to open multiple connections at the same time, the scripts must be modified based on the specific use case. In some cases, it is simply enough to run the same script simultaneously. For example, the JavaScript based script was tested in the Firefox browser by opening it to the browser for 200 times to separate tabs without any problems, which is enough for most of the testing cases. Figure 20 illustrates this situation from the WebSocket server point of view.



```
ws-tester — -bash — 75x48
('127.0.0.1', 60618) 2016-05-12 21:00:50.926856 - Client connected
('127.0.0.1', 60620) 2016-05-12 21:00:51.068075 - Client connected
('127.0.0.1', 60622) 2016-05-12 21:00:51.232956 - Client connected
('127.0.0.1', 60624) 2016-05-12 21:00:51.397499 - Client connected
('127.0.0.1', 60626) 2016-05-12 21:00:51.658384 - Client connected
('127.0.0.1', 60628) 2016-05-12 21:00:51.762870 - Client connected
('127.0.0.1', 60630) 2016-05-12 21:00:51.805051 - Client connected
('127.0.0.1', 60632) 2016-05-12 21:00:51.913852 - Client connected
('127.0.0.1', 60634) 2016-05-12 21:00:52.003605 - Client connected
('127.0.0.1', 60636) 2016-05-12 21:00:52.083689 - Client connected
('127.0.0.1', 60638) 2016-05-12 21:00:52.137146 - Client connected
('127.0.0.1', 60640) 2016-05-12 21:00:52.250678 - Client connected
('127.0.0.1', 60642) 2016-05-12 21:00:52.319004 - Client connected
('127.0.0.1', 60644) 2016-05-12 21:00:52.394123 - Client connected
('127.0.0.1', 60646) 2016-05-12 21:00:52.438774 - Client connected
('127.0.0.1', 60648) 2016-05-12 21:00:52.568331 - Client connected
('127.0.0.1', 60650) 2016-05-12 21:00:52.615546 - Client connected
('127.0.0.1', 60652) 2016-05-12 21:00:52.683550 - Client connected
('127.0.0.1', 60654) 2016-05-12 21:00:52.800102 - Client connected
('127.0.0.1', 60656) 2016-05-12 21:00:52.887189 - Client connected
('127.0.0.1', 60658) 2016-05-12 21:00:52.920058 - Client connected
('127.0.0.1', 60660) 2016-05-12 21:00:53.006803 - Client connected
('127.0.0.1', 60662) 2016-05-12 21:00:53.159550 - Client connected
('127.0.0.1', 60664) 2016-05-12 21:00:53.206540 - Client connected
('127.0.0.1', 60666) 2016-05-12 21:00:53.344381 - Client connected
('127.0.0.1', 60668) 2016-05-12 21:00:53.434932 - Client connected
('127.0.0.1', 60670) 2016-05-12 21:00:53.529550 - Client connected
```

Figure 20. WebSocket server's log when multiple connections are opened by one WebSocket client from the same source.

## 6 Customized Tool for WebSockets Security Testing

### 6.1 Preliminary Analysis

The first step in developing the new tool is to set the requirements for what the tool should be able to perform. The following are the requirements that were set for the tool.

**Easy to implement.** From the development point of view the tool must be easy to implement. This includes a programming language that must be easy to program and if possible, have previous knowledge about the selected programming language. Chosen programming language for this tool was Python, because it is a really easy language to understand and the previous experience I have programming with it.

**Cross-platform support.** It must be possible to run the tool on different platforms like Windows, OS X and Linux, without unnecessary dependencies to 3<sup>rd</sup> parties.

**WebSocket client-side libraries available.** The selected programming language must have WebSocket client-side libraries available, because it is not reasonable to implement them from a scratch during this project. Python has a library called websocket, which is part of the websocket-client package that can handle all the client-side traffic needed for this tool.

**Extensibility and support for the future.** The tool must be programmed in a way that it can be extended for new features in future. If for example the WebSocket specification is updated or if there is need for a new feature.

**The tool must be easy to use.** The tool that is going to be developed must be easy to use. This does not mean that there should be a graphical user interface, but instead should be a clear way to use it from terminal/console. As a user of the tool there is no need to do any changes to the actual script's code when using it in various scenarios.

**The tool has basic features for WebSocket security testing.** The tool must be able to test different kind of problematic security related scenarios, either automatically or manually. These features include:

- Request header manipulation testing.
- Traffic encryption testing.
- Cross-domain requesting testing (Origin validation).
- Manual input validation testing.
- Logging of all requests.

## 6.2 Design

In order to implement the tool in an effective way, the decisions how the tool is going to be implemented were decided during the design phase. Design phase includes what kind of user interface will be created, what kind of logic flows are there, the different functions implemented and so on.

### 6.2.1 User Interface

User interface for the tool will be a non-graphical terminal/console –text based tool.

User must be able provide the needed details for basic testing as attributes.

Launching of the tool must occur like following:

```
python ws_security_tester.py -attribute1 value -attribute1 value
```

If the user does not know the parameter, available parameters can be printed with *-h* parameter. Help printing must occur like following:

```
python ws_security_tester.py -h
```

*Result:*

*positional arguments*

*argument\_name                    argument description*

*optional arguments*

*-argument                        argument description*

Once the script is launched, a small description about the tool is printed to user's terminal/console. This must include the name of the tool and what where the arguments provided by the user during the launch.

If there is a need to ask some specific questions (interactive user input), those are asked after printing the description. Questions can be asked as yes/no questions where user can write the input. If yes/no is not suitable, other input can also be asked and supplied by the user.

Once all the needed details are in place, the testing starts and the status about what is occurring is informed interactively on user interface.

There must be a way for the user to stop the execution of the test and user must be informed about this possibility.

### 6.2.2 Functions

In order for the tool to work, it must have different kind of security testing functions. The following functions must be implemented.

**Open WebSocket connection.** The function must be able to open a connection to any WebSocket server. Server can be used either with `ws://` (non-encrypted) or `wss://` (encrypted) connection. It must also support manipulation of the Origin header and adding of any custom header. User provided subprotocols must be supported as well.

**Close WebSocket connection.** If requested by the user or by the WebSocket server, it must be possible to close the WebSocket connection.

**Send and receive messages and keep the connection alive.** The tool must be able to send user provided messages, receive server provided messages and keep the connection alive (ping/pong frames). Messages sent as UTF-8 encoded text (opcode 0x1) as well as binary (opcode 0x2) formatted must be supported.

**Log data.** The tool must be able to log any data that is sent to a log function and at the same time log all the WebSocket messages while the WebSocket connection is established. Log must be written to a file.

**Print data.** The tool must be able to print data to the user interface for informational purposes.

**User input.** The tool must be able to ask for user input and handle it.

**Proxy support.** The tool must be able to support the usage of HTTP proxies.

### 6.2.3 Logic Flow of the Tool

The tool must follow a specific logic flow during the whole execution. Figure 21 illustrates the logic flow in the tool.

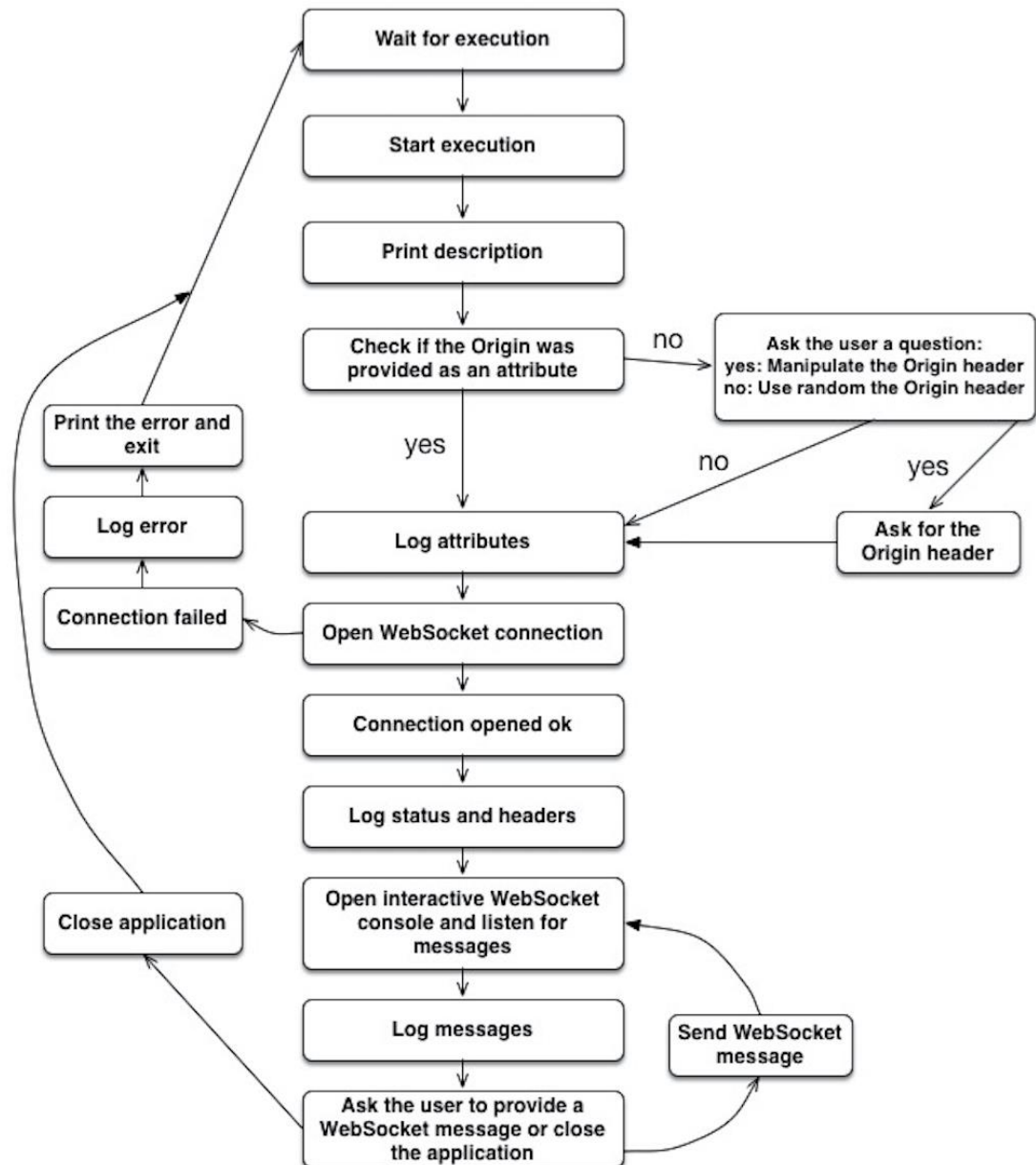


Figure 21. Logic flow chart of the tool.

## 6.3 Implementation

### 6.3.1 Wsdump Tool

Originally the idea for the implementation was to write the whole tool from a scratch, but during the initial steps of implementation I found a tool, which was almost the same as what the whole implementation was all about. The tool is called `wsdump.py` and is a part of the open-source `websocket-client` package, which contain the `websocket` library. (Liris 2016.)

`Wsdump.py` is a Python script, which was developed for testing purposes to be used with `websocket` library. Initially it is not meant for the security testing.

### 6.3.2 Programming the Tool

From implementation point of view this tool was to be modified for security testing purposes. By default, `Wsdump` provided support for many of the needed features, but not logging at all.

Like described in design phase the tool must be able to log all `WebSocket` messages that are being sent during the testing. This also includes the `WebSocket` opening handshake request and response.

The first thing is to add the logging possibility to the tool. `Logging` is a Python library that can be used for this case. In order to provide the functionality, the following code was needed:

**Import logging library:**

```
import logging
```

**Define a new argument for logging:**

```
def parse_args():
```

```
    ...
```

```
    parser.add_argument("log_file", metavar="log_file_path",  
                        help="log file path. e.g. filename.txt")
```

```
    ...
```



**Define a write\_log function:**

```
def write_log(data):
    logging.info(data)
```

**Set logging configuration:**

```
def main():
    ...
    logging.basicConfig(
        filename=args.log_file,
        level=logging.DEBUG,
        format = "%(message)s",
        datefmt='%d-%b %H:%M:%S',
    )
    ...
```

When the basic configuration for logging was in place, the next step is to log all the needed messages. It can be accomplished with the written *write\_log* function.

Logging was added to all necessary parts of the tool. An example of logging usage is following:

```
write_log('Script start time: %s' % time.ctime())
```

As a second additional feature, support for custom headers was to be added. With it any custom header can be added. For example in cases where authentication is needed, there needs to be a Cookie header with key/value pair or similar. In order to provide the functionality, the following code was written:

**Define a new argument for the custom header:**

```
def parse_args():
    ...
    parser.add_argument("--header", action="store_true",
        help="Custom header e.g. Cookie:
        token=1234567890")
    ...
```

**Add check for the argument and add it to connection options if exist:**

```

if (args.header):
    options["header"] = args.header

```

**Later on options are used in the connection opening:**

```

ws = websocket.create_connection(args.url, sslopt=opts, **options)

```

As a third feature a check if the Origin was not provided as an argument was added. This feature asks a question from the user to confirm on how the Origin is to be used. By default the Origin is set to the WebSocket services address (e.g. http://127.0.0.1).

```

if (args.origin):
    options["origin"] = args.origin
else:
    print("\nOrigin not provided, please provide it here or set
        it with -o parameter. If not provided, example-
        domain.com will be used.")
    write_log("\nOrigin not provided, please provide it here
        or set it with -o parameter. If not provided,
        example-domain.com will be used.")
    origin_question = ""
    while origin_question not in ("yes", "no"):
        print('Do you want to provide it here
            (yes/no?)')
        origin_question = raw_input()
    if origin_question == "yes":
        print("\nOrigin (e.g. example-
            domain.com):")
        options["origin"] = raw_input()
        write_log ("User provided origin: %s" %
            options["origin"])
    else:
        options["origin"] = "example-domain.com"

```

```
print("Origin not provided, example-  
domain.com will be used.")  
write_log("Origin not provided, example-  
domain.com will be used.")
```

As the fourth feature, interface messages about what is happening in the tool execution was added. Example message:

```
print("\nConnection opened successfully to " + args.url + "\n")
```

With all the mentioned changes the tool is in a state where it can be used for the security testing.

## 6.4 Testing

The testing of the tool mainly occurred during the implementation phase of every new feature. The functions that were already implemented to Wsdump tool previously were tested for the necessary part also. This included opening a WebSocket connection, sending WebSocket messages and receiving WebSocket messages in both UTF-8 encoded text and binary format, as well as routing the traffic through a proxy server.

When testing any of the new features added to the tool, testing was done with a local WebSocket server to confirm the functionality. From testing point of view all of the new functionalities were simple to test with this setup. Burp Suite was used as the proxy tool during the testing.

Logging functionality was tested by adding new items to be logged one by one. Operability of a new log item was easy to confirm by simply reviewing the log file, which was generated during the tool execution. All problems that existed with this feature were easy to fix.

Functionality of adding a custom header was tested by adding a new custom header as an argument and confirming the operability. The custom header was confirmed from WebSocket opening handshake that was logged Burp Suite history or by reviewing the log file.



```

-c, --console          ws console with cleartext output
-s [SUBPROTOCOLS [SUBPROTOCOLS ...]], --subprotocols
[SUBPROTOCOLS [SUBPROTOCOLS ...]]
                        Set subprotocols
-o ORIGIN, --origin ORIGIN      Set origin e.g. test.domain.com
--eof-wait EOF_WAIT      wait time(second) after 'EOF' received.
-t TEXT, --text TEXT      Send initial text
--timings               Print timings in seconds
--header                Custom header e.g. Cookie:
                        token=1234567890

```

**Starting the tool.** Tool starts by launching it from terminal/console with the following command:

```
Python ws_security_tester.py [arguments] ws_url log_file
```

**Startup banner.** The tool prints a banner giving all the information about the arguments provided during the startup. The banner is illustrated in Figure 22.

```

*****
*****
*****
                        WebSocket Security Tester

                        Provided arguments:
console=False, eof_wait=0, header=False, log_file='ws_log.txt', nocert
=False, origin='ws-example-site.com', proxy=None, raw=False, subprotoco
ls=None, text=None, timings=False, url='ws://127.0.0.1:8000', verbose=2
*****
*****
*****

```

Figure 22. Banner shown during the tool launch.

**Origin verification.** If the user does not provide the origin within the launch arguments, the user is asked a verification about how to handle origin. Figure 23 illustrates the functionality.

```

Origin not provided, please provide it here or set it with -o parameter. If not provided, ex
ample-domain.com will be used.
Do you want to provide it here (yes/no?)
yes

Origin (e.g. example-domain.com):
own-domain-url.com

```

Figure 23. Origin verification.

**WebSocket opening handshake request and response.** When the WebSocket handshake is happening, the handshake request and response are printed to the user interface for inspection. Figure 24 illustrates the functionality.

```

Starting security testing:

Trying to open a connection to ws://127.0.0.1:8000
--- request header ---
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: 127.0.0.1:8000
Origin: ws-example-site.com
Sec-WebSocket-Key: 2qUyvEwpRWbDxNWi2GCTvw==
Sec-WebSocket-Version: 13

-----
--- response header ---
HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: wuxtmgFTqf2SXM56eNd+DunJTB8=
-----

Connection opened successfully to ws://127.0.0.1:8000

```

Figure 24. WebSocket opening handshake on the user interface.

**Sending messages.** Once the WebSocket connection is established, messages can be sent to the WebSocket connection by simply typing the message and sending it by pressing enter. Figure 25 illustrates communication between a Python client and another client who is connected to the same WebSocket session from another source.

```

Connection opened successfully to ws://127.0.0.1:8000

[> Test message from the tool
send: '\x81\x9a\xe2`p}\xb6\x05\x03\t\xc2\r\x15\x0e\x91\x01\x17\x18\xc2\x06\x02\x12\x8f@\x04\x15\x87@\x04\x12\x8d\x0c'
< text: 127.0.0.1 - Test message from a separate WebSocket client
> █

```

Figure 25. WebSocket messaging with another client.

**Log data.** The tool provides a log file, which can be used later on to review what happened during the WebSocket connection. Figure 26 illustrates the functionality.

Script start time: Sun May 22 23:37:36 2016

```
*****
*****
*****
```

WebSocket Security Tester

Provided arguments:

```
console=False, eof_wait=0, header=False, log_file='ws_log.txt',
nocert=False, origin='ws-example-site.com', proxy=None, raw=False,
subprotocols=None, text=None, timings=False, url='ws://127.0.0.1:8000',
verbose=2
```

```
*****
*****
*****
```

Starting security testing:

Trying to open a connection to ws://127.0.0.1:8000

--- request header ---

GET / HTTP/1.1

Upgrade: websocket

Connection: Upgrade

Host: 127.0.0.1:8000

Origin: ws-example-site.com

Sec-WebSocket-Key: 2qUyvEwpRwBdXNwi2GCTvw==

Sec-WebSocket-Version: 13

|

-----  
--- response header ---

HTTP/1.1 101 Switching Protocols

Upgrade: WebSocket

Connection: Upgrade

Sec-WebSocket-Accept: wuxtmqFTqf2SXM56eNd+DunJTB8=

-----  
Connection opened successfully to ws://127.0.0.1:8000

```
send: '\x81\x9a\xe2`p}\xb6\x05\x03\t\xc2\r\x15\x0e
\x91\x01\x17\x18\xc2\x06\x02\x12\x8f@\x04\x15\x87@\x04\x12\x8d\x0c'
text: 127.0.0.1 - Test message from a separate WebSocket client
```

Figure 26. Log file from the tool.

**Problem situations.** If the tool cannot establish a WebSocket connection, it informs the user about it by testing with different arguments. Figure 27 illustrates the situation.

```
Starting security testing:
Trying to open a connection to ws://127.0.0.1:8001
Error while trying to open a connection to ws://127.0.0.1:8001. Please
provide a working url and try again.
Exiting script
```

Figure 27. Failed WebSocket connection establishment.

## 7 Results

The initial requirements for the tool required the capability to handle different kinds of security testing scenarios. All the features included in the requirements were successfully implemented in the tool. In order to confirm the reliability of the results reported by the tool, all the tests were performed to record all the traffic handled by the tool and reviewing them for any errors.

Now that the tool is complete it can be used in WebSocket security testing assignments to improve testing efficiency (time- and nerve-wise). The tool can handle all the relevant security testing scenarios involving a WebSocket service during a penetration test. However, the user of the tool must have a sufficient understanding of how the WebSocket protocol works, limiting the usability of the tool. This is not a problem as the tool was intended to be used by experts in the web application security testing field.

Before the existence of this tool, testing the security of WebSocket services required use of tools that were suboptimal for this purpose. This tool will ease and comfort the testing process in the future. In every aspect this tool is beneficial for future testing of WebSocket services. Currently the tool is only available for internal use at Silverskin Information Security.

From every point of view, the developed tool and the research was a success.



## 8 Conclusion

The main objective of this master's thesis was to research if it is possible to develop a tool to ease the security testing of WebSocket services. In order to implement the tool the following questions had to be answered:

- How does the WebSocket protocol work?
- What are the common vulnerabilities in WebSocket services?
- What are the security testing tools available for WebSocket testing?
- How can the vulnerabilities be tested with WebSocket security testing tools?

The WebSocket protocol technical specification and other literature were reviewed in order to understand how the protocol works. The specification was heavy to read but researching the specification thoroughly helped a lot in understanding the protocol and its potential security issues.

The available literature was reviewed to discover the vulnerabilities commonly found in WebSocket services but it did not introduce a comprehensive set of possible vulnerabilities. A better source for information was to review articles found on the Web that were written mostly by independent security researchers. At this point the researcher's previous experience in web application penetration testing was a huge help in understanding what the vulnerabilities are, how can they be exploited, and the impact of someone exploiting them.

Finding tools suitable for WebSocket security testing was the easiest phases of this research since the information was widely available on vendors' web sites. Using these tools in WebSocket security testing was more difficult, because in order to test a WebSocket service, a server needed to be running a WebSocket service, that at the same time was vulnerable to a specific kind of vulnerability. The other fact was that the researcher had to have an understanding of how the tools worked with WebSocket security testing.

Once all the previous phases of the research were completed, the actual development of the tool began. The first step was to perform a preliminary analysis of the features required for the tool. This analysis was relatively easy to perform due to the information previously gathered on the vulnerabilities commonly found in WebSocket services.

From there the next phase was the design phase that was used to plan what the application should look like, how it can be used, and what technologies should the tool use in order to work properly. This phase took more time than the preliminary analysis, since a design had to be drawn, choose a programming language, review the libraries available for the programming language, among others.

The next phase was the actual implementation phase of the tool. During the first steps of implementation, a tool was found that already supported several of the required features. The tool was a non-security WebSocket testing and was chosen as the base for the new tool. This shortened the development time by a large margin.

New features of the tool were tested while implementing them in the tool. The testing phase was easy and there were no severe problems during the testing phase. The development process resulted in a new WebSocket service testing tool able to test for the commonly found security vulnerabilities in WebSocket services. While the amount of programming needed to implement the functionality was not massive, it was still educational.

Some of the objectives set for this thesis were easy to understand and document and others were heavy learning experiences and time-consuming. The researcher's previous knowledge in web application security testing was extremely beneficial, because some of the security vulnerabilities and tools were well-known. Otherwise this research could have been too large a project to handle with all the limitations set.

Using the design research based approach with qualitative methodologies was the correct choice for this research. It supported the overall process of developing the tool as well as documenting the process in the form of this thesis. In retrospective, spending the hours developing the custom tool and searching for the right ways to implement was overall a great experience.

## 9 Future Research

Currently the state of security of WebSocket services is stable, meaning no new types of vulnerabilities have been found lately. However, if a new type of vulnerability is found in a WebSocket service, it must be verified whether the tool is already able to test for the scenario. If not, then it needs to be investigated how the support for testing the new scenario can be implemented.

There are at least two features that could be seen as valuable in the future. One of them is making the tool more automated. Currently the tool is used manually for the most part, meaning that the user must know how to use the tool properly and what kind of arguments to use in certain situations. Improving automation of the tool would require an extensive mapping of what sort of functions the tool should be able to perform automatically. Only after that it can be implemented. However, not all tests should be automated, as the ways a WebSocket service can be implemented in varies a lot. Some services may use WebSocket only for video streaming, and one may use it for direct database access.

Another interesting addition to the tool would be support for fuzzing with preset payloads. Currently the arguments support one initial WebSocket message when launching the tool. It could be expanded to support a set of messages that would then be sent when the WebSocket connection is established. Currently the fuzzing feature only supports manually typed payloads.

Other than these, there are no known features that would provide a clear improvement to the functionality of the tool.

## References

- Barth, A. & Berkeley, U.C. *HTTP State Management Mechanism*. Internet Engineering Task Force (IETF). Accessed on 7 May 2016. Retrieved from <https://tools.ietf.org/html/rfc6265>
- Chopra, V. 2015. *WebSocket Essentials – Building Apps with HTML5 WebSockets*. Birmingham: Packt Publishing.
- Comet Daily. 2008. *Independence Day: HTML5 WebSocket Liberates Comet From Hacks*. Accessed on 30 April 2016 Retrieved from <http://cometdaily.com/2008/07/04/html5-websocket/>
- Erkkilä, J. 2012. *WebSocket Security Analysis*. Accessed on 7 May 2016. Retrieved from <https://secfault.fi/files/writings/Websocket2012.pdf>
- Fagerlund, T. 2014. *WebSocket-protokollan tietoturva selainsovelluksissa* [WebSocket protocol's security in web applications] (Pro Gradu). Tampere: Tampere University of Technology
- Fielding, R., Adobe, Reschke, J. & Greenbytes. 2014. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing – RFC 7230. Internet Engineering Task Force (IETF). Accessed on 1 May 2016. Retrieved from <https://tools.ietf.org/html/rfc7230>
- Google. 2016. *Chrome DevTools*. Accessed on 8 May 2016. Retrieved from <https://developers.google.com/web/tools/chrome-devtools/>
- Fette, I., Google Inc, Melnikov, A. & Isode Ltd. 2011. *The WebSocket protocol - RFC 6455*. Internet Engineering Task Force (IETF). Accessed on 19 March 2016. Retrieved from <https://tools.ietf.org/html/rfc6455>
- Hickson, I. & Google Inc. 2010. *The Web Socket protocol - draft-hixie-thewebsocketprotocol-75*. Network Working Group. Accessed on 30 April 2016. Retrieved from <https://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-75>
- Kananen, J. 2012. *Kehittämistutkimus opinnäytetyönä [Design research as thesis]*. Jyväskylä: Tampereen Yliopistopaino Oy – Juvenes Print.
- Kaazing. 2016. *Echo Test Tool*. Accessed on 8 May 2016. Retrieved from <http://websocket.org/echo.html>
- Liris. 2014. *websocket-client – echo\_client.py*. Github. Accessed on 12 May 2016. Retrieved from <https://github.com/liris/websocket-client>
- Liris. 2016. *websocket-client – wsdump.py*. Github. Accessed on 13 May 2016. Retrieved from <https://github.com/liris/websocket-client/blob/master/bin/wsdump.py>
- Lombardi, A. 2015. *WebSocket: Lightweight client-server communications*. Sebastopol: O'Reilly.

- Loreto, S., Ericsson, P. Saint-Andre, Cisco, Salsano, S., University of Rome “Tor Vergata”, Wilkins, G. & Webtide. 2011. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. Internet Engineering Task Force (IETF). Accessed on 19 March 2016. Retrieved from <https://tools.ietf.org/html/rfc6202>
- Microsoft. *What is the Security Development Lifecycle*. Accessed on 24 May 2016. Retrieved from <https://www.microsoft.com/en-us/sdl/>
- Mozilla Developer Network. 2016. *WebSockets*. Accessed on 7 May 2016. Retrieved from [https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API)
- OWASP. 2013. *Data validation*. Accessed on 4 May 2016. Retrieved from [https://www.owasp.org/index.php/Data\\_Validation](https://www.owasp.org/index.php/Data_Validation)
- OWASP. 2009. *Resource exhaustion*. Accessed on 7 May 2016. Retrieved from [https://www.owasp.org/index.php/Resource\\_exhaustion](https://www.owasp.org/index.php/Resource_exhaustion)
- OWASP. 2016. *OWASP Zed Attack Proxy (ZAP)*. Accessed on 8 May 2016. Retrieved from [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)
- OWASP. 2016. *OWASP Zed Attack Proxy (ZAP) - Getting Started Guide*. Accessed on 8 May 2016. Retrieved from <https://github.com/zaproxy/zaproxy/releases/download/2.4.0/ZAPGettingStartedGuide-2.4.pdf>
- Pallot, D. 2015. *simple-websocket-server – websocket-html*. Github. Accessed on 12 May 2016. Retrieved from <https://github.com/dpallot/simple-websocket-server/blob/master/SimpleWebSocketServer/websocket.html>
- PortSwigger. 2016. *Burp Proxy*. Accessed on 8 May 2016. Retrieved from <https://portswigger.net/burp/proxy.html>
- PortSwigger. 2016. *Burp Suite*. Accessed on 8 May 2016. Retrieved from <https://portswigger.net/burp/>
- PortSwigger. 2014. *Burp Suite v1.5.21 Release Notes*. Accessed on 8 May 2016. Retrieved from <http://releases.portswigger.net/2014/01/v1521.html>
- Qyalys SSL LABS. 2014. *SSL/TLS Deployment Best Practices*. Accessed on 7 May 2016. Retrieved from [https://www.ssllabs.com/downloads/SSL\\_TLS\\_Deployment\\_Best\\_Practices.pdf](https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices.pdf)
- W3C. 2008. *HTML5 - A vocabulary and associated APIs for HTML and XHTML*. Accessed on 30 April 2016. Retrieved from <https://www.w3.org/TR/2008/WD-html5-20080610/single-page/>
- WebKit. 2011. *Changeset 97249*. Accessed on 7 May 2016. Retrieved from <https://trac.webkit.org/changeset/97249>
- Wikipedia. *WebSocket – Browser implementation*. Accessed on 3 April 2016. Retrieved from <https://en.wikipedia.org/wiki/WebSocket>
- Wikipedia. *Slowloris (computer security)*. Accessed on 7 April 2016. Retrieved from [https://en.wikipedia.org/wiki/Slowloris\\_%28computer\\_security%29](https://en.wikipedia.org/wiki/Slowloris_%28computer_security%29)

## Appendices

Appendice 1.                    simple-web-socket-client – websocket.html

```
<!DOCTYPE html>
<meta charset="utf-8" />
<title>WebSocket Test</title>
<script language="javascript" type="text/javascript">
  function init()
  {
      document.myform.url.value = "ws://localhost:8000/"
      document.myform.inputtext.value = "Hello World!"
      document.myform.disconnectButton.disabled = true;
  }
  function doConnect()
  {
      websocket = new WebSocket(document.myform.url.value);
      websocket.onopen = function(evt) { onOpen(evt) };
      websocket.onclose = function(evt) { onClose(evt) };
      websocket.onmessage = function(evt) { onMessage(evt) };
      websocket.onerror = function(evt) { onError(evt) };
  }
  function onOpen(evt)
  {
      writeToScreen("connected\n");
      document.myform.connectButton.disabled = true;
      document.myform.disconnectButton.disabled = false;
  }
  function onClose(evt)
  {
      writeToScreen("disconnected\n");
      document.myform.connectButton.disabled = false;
      document.myform.disconnectButton.disabled = true;
  }
  function onMessage(evt)
```

```

{
  writeToScreen("response: " + evt.data + '\n');
}
function onError(evt)
{
  writeToScreen('error: ' + evt.data + '\n');
  websocket.close();
  document.myform.connectButton.disabled = false;
  document.myform.disconnectButton.disabled = true;
}
function doSend(message)
{
  writeToScreen("sent: " + message + '\n');
  websocket.send(message);
}
function writeToScreen(message)
{
  document.myform.outputtext.value += message
  document.myform.outputtext.scrollTop =
document.myform.outputtext.scrollHeight;
}
window.addEventListener("load", init, false);
function sendText() {
  doSend( document.myform.inputtext.value );
}
function clearText() {
  document.myform.outputtext.value = "";
}
function doDisconnect() {
  websocket.close();
}
</script>

<div id="output"></div>

```

```
<form name="myform">
<p>
<textarea name="outputtext" rows="20" cols="50"></textarea>
</p>
<p>
<textarea name="inputtext" cols="50"></textarea>
</p>
<p>
<textarea name="url" cols="50"></textarea>
</p>
<p>
<input type="button" name=sendButton value="Send" onClick="sendText();">
<input type="button" name=clearButton value="Clear" onClick="clearText();">
<input type="button" name=disconnectButton value="Disconnect"
onClick="doDisconnect();">
<input type="button" name=connectButton value="Connect"
onClick="doConnect();">
</p>
</form>
</html>
```