On WebSockets in Penetration Testing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Robert Koch

Matrikelnummer 0726330

an der

Fakultät für Informatik der Technischen Universität Wien

Mitwirkung: M.Sc. Martin Mulazzani

Wien, 11.03.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)
On WebSockets in Penetration Testing

MASTER’S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Robert Koch
Registration Number 0726330

to the Faculty of Informatics
at the Vienna University of Technology

Assistance: M.Sc. Martin Mulazzani

Vienna, 11.03.2013
(Signature of Author) (Signature of Advisor)
Erklärung zur Verfassung der Arbeit

Robert Koch
2011 Obermallebarn 25

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen - die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)
Acknowledgements

First I want to thank Simon Bennetts from Mozilla, who is the project lead of ZAP. His motivation, enthusiasm and guidance on extending the intercepting proxy helped me a lot.

I also want to acknowledge Martin Mulazzani, my internal supervisor at Vienna University of Technology, for the support and guidance he has provided throughout the work.

Not to be underestimated, the moral support provided by persons close to me. It has greatly contributed to this work. Special thanks to my family which supported me during my studies.
Abstract

In recent years the Web has grown to a platform, where standalone application types known from desktop computers run direct in a browser. The finalization of the HTML5 standard aids this development. WebSockets emerged also with HTML5, although their specification was outsourced to its own standardization process, which is nearly finalized. This new way to communicate enables two endpoints to send & receive messages at the same time. A new WebSocket connection starts with a HTTP-based handshake. Afterwards a lightweight protocol takes over, enabling real-time communication. WebSocket connections are based on TCP and feature asynchronous & bi-directional communication. With more powerful standards the importance of browsers grow, while browser-plugins such as Flash or Java become obsolete. Users may profit from this development, because they can remove potential insecure third-party plugins. While modern browsers care about security, third-party vendors often do not. So the web as a platform becomes safer, but we need not to forget about securing our web applications.

Developers and security testers need to keep up with new standards and be aware of new threats. Only if they are educated well enough, they are able to think out-of-the-box, to think like an attacker. This is required especially in the field of penetration testing, where you try to find vulnerabilities by acting like an attacker. Intercepting proxies, a sub-type of web application scanners can aid the search for weaknesses. These easy-to-use graphical tools are put as Man-in-the-Middle between your browser and the Internet. While you are browsing your web application under test, the communication is recorded and displayed to the penetration tester. Thorough support for WebSockets does not exist for any web application scanner, which may result in untested parts.

Within this work, attack vectors for WebSockets are identified and support for the open-source intercepting proxy Zed Attack Proxy (ZAP) is developed. Consequently, web applications using WebSockets can be analysed in-depth. While the usage of WebSockets on the Internet was already evaluated by other researchers, I did an evaluation on WebSocket usage in smartphone apps. Therefore I fetched over 15,000 freely available Android apps and examined if they make use of WebSockets. 14 apps proofed to use the soon-standardized WebSockets.

In the future, usage of WebSockets is expected to rise. With tool support, developers and security testers are able to deliver more secure applications.


In Zukunft wird die Verbreitung von WebSockets zunehmen. Tool-Unterstützung erleichtert es Entwicklern und Sicherheits-Testern die Sicherheit ihrer Anwendungen zu gewährleisten.
## Contents

1 Introduction .......... 1
   1.1 Problem Description .... 1
   1.2 Motivation ........ 2
   1.3 Expected Result ....... 2
   1.4 Methodological Approach ... 3
   1.5 State of the Art .......... 3
   1.6 Structure of the Work ...... 3

2 Background .......... 5
   2.1 WebSockets ........ 5
   2.2 Vulnerabilities ........ 27
   2.3 Penetration Testing ...... 28
   2.4 Web Application Scanners ... 30

3 Penetration Testing on WebSockets .......... 36
   3.1 Security Implications of WebSockets .......... 36
   3.2 Attack Vectors for WebSockets ........ 38
   3.3 Vulnerability Exploration Techniques ...... 42

4 Implementation of WebSockets Support for an existing Web Application Scanner .......... 44
   4.1 Scope of the WebSockets Extension .......... 44
   4.2 Features of the WebSockets Extension ........ 45
   4.3 Approach of Implementing the WebSockets Extension ...... 45

5 Evaluation of WebSocket Usage in Android Apps .......... 63
   5.1 Approach & Analysis .......... 63
   5.2 Limitations on Analysis Approach .......... 66
   5.3 Manual Verification .......... 67
   5.4 Evaluation Result .......... 70

6 Related Work .......... 77

7 Future Work .......... 79

8 Conclusion .......... 81

Bibliography .......... 82
1.1 Problem Description

Testing web applications is crucial for ensuring their security. Existing vulnerabilities can be exploited by attackers for malicious purposes. The Open Web Application Security Project (OWASP) focuses on improving the security of software. Its OWASP Top Ten project aims at identifying the most critical web application security flaws and wants to raise awareness of common vulnerabilities. In 2010 they released their latest awareness document, describing 10 widely spread vulnerabilities. Penetration tests are often used to find such security holes, by simulating attacks against the application. Testers can also use tools that support their search for these critical security flaws.

The upcoming HTML5 standard will include a new method for communication: WebSockets. It is based on a lightweight protocol which is layered on TCP. WebSockets enable asynchronous bidirectional communication and are intended to replace techniques based on Polling or Streaming. Arbitrary data can be sent over the wire. As developers start using WebSockets in their web applications, the need for testing arises. Network protocol analysers, such as Wireshark, can only be used to view WebSocket communication. Moreover it requires some effort by the developer to read the payload sent back & forth. With the release of version 1.8 in the end of June 2012, it includes a dissector for the WebSockets protocol, which provides you a more human-friendly representation. However, its usage is not really suitable for web application testers, nor for their developers.

1.2 Motivation

In the area of web based applications, intercepting proxies are very popular. While you are surfing through your web application in the browser, all HTTP requests and responses get tracked by such an intermediary. You can either inspect the communication in detail by yourself, or you can use various scanner tools that are often included in such proxies, to find security holes. Such scanners examine applications in an automated fashion \cite{18}. One of the biggest advantages of these applications is their ease-of-use. Every developer can start and run such tools and gain insight into possible security problems.

1.3 Expected Result

The OWASP Zed Attack Proxy (ZAP\cite{3}) is an intercepting proxy that is easy to use for developers and functional testers, which are new to penetration testing. As part of this master thesis this open source tool will be extended to include support for WebSockets. The extension should allow a tester to not only inspect communication, but also to change message payloads. Moreover it should ease the discovery of various vulnerabilities, especially those from the OWASP Top Ten.

The other part of this master thesis will deal with web application scanners. Such vulnerability scanners crawl the web application as black box. Often this is done in an automated fashion. The detection rate is fairly good for historical Cross-Site-Scripting (XSS) or SQL injection vulnerabilities, while there is room for improvement for detecting more advanced attacks, such as Cross-Site-Request-Forgery or second order XSS \cite{8}.

Finally I want to analyse how popular WebSockets are on mobile applications. I plan to do an evaluation on the most popular freely available Android apps.

To conclude, I want to deal with the following research questions in my thesis:

- What are possible attack vectors for WebSockets?
- How do WebSockets affect web developers? Which impact do WebSockets have on the security of web sites/applications?
- Find out if there are automated ways to test WebSockets for vulnerabilities. While the protocol itself seems fairly stable, this question aims at the payload that is transmitted back & forth from servers to web applications. Are there any other web application scanners that take WebSockets into account?
- How widespread is the usage of WebSockets in mobile applications?

\footnote{\textsuperscript{3} “zaproxy - OWASP ZAP: An easy to use integrated penetration testing tool for finding vulnerabilities in web applications.”. Google Code. Accessed May 28, 2012.}

\url{http://code.google.com/p/zaproxy}
1.4 Methodological Approach

First, I have to get into contact with the community of the open source intercepting proxy ZAP. Then I plan to implement WebSockets support for ZAP with the Java programming language using an object-oriented design method. For the implementation I will have to follow the WebSockets specification, which is split into two parts:


2. RFC6455: *The WebSocket Protocol* [16] - describing the wire protocol

Besides the practical part I will conduct a literature search about current research in web application scanners and their approach. Additionally I plan an evaluation where I want to fetch top freely available Android apps and do some pattern matching on their binary. This aims at finding apps that set-up a WebSocket connection. Positives will be verified by running them manually on the Android emulator, where ZAP is used to track all HTTP/WebSocket traffic.

1.5 State of the Art

In the beginning the WebSockets protocol was very minimalistic, but it had been extended to support more advanced features. Moreover there has been a critical vulnerability in the protocol, which was discovered & reported by [26]. So the protocol has evolved over time and is now a proposed standard. There is nearly no support by security tools. The browser Chrome (version 22 or higher) allows inspecting WebSocket frames in its Development-Tools [4]. Fiddler2 is another Intercepting Proxy like ZAP, but Windows only. It has also basic support for inspecting WebSocket frame content [5].

With new technologies come new security problems. The TOR browser bundle for Firefox that provides anonymity on the Internet, failed with the introduction of WebSockets. The DNS resolution for WebSocket connections bypassed the TOR network, which could be used to find out which sites were visited by the ”anonymous” user [6].

1.6 Structure of the Work

Besides the problem description and staking of the work in Chapter 1, the rest of this paper is organized as follows. In Chapter 2 I want to introduce the reader to the relevant technologies and

---


concepts of penetration testing web applications. A lot of space is dedicated to the introduction of WebSockets, because this technology is the focus of my work and it is a new upcoming standard for communication. Furthermore a comparison is made to SPDY, another upcoming protocol proposed by Google.

Chapter 3 takes a look at the weaknesses of the security mechanisms of WebSockets. Additionally attack vectors on WebSockets are examined.

Chapter 4 describes how I extended the web application scanner ZAP with support for WebSockets. It explains the features of my extension and how they support a web application tester. Moreover the architecture of the extension and the code behind is described.

Chapter 5 is dedicated to the conducted evaluation to find out how many Android apps make use of WebSockets. Furthermore the limitations of my evaluation are highlighted. Besides that, the usages found are examined with the support of my ZAP extension.

Chapter 6 places my work in the context of related work, Chapter 7 states possible future work and Chapter 8 finally concludes.
First, we will have a look at WebSockets: how they are standardized, their properties and intended usage. There is also a section about Server-Sent Events and SPDY, highlighting the differences of these technologies. Then I want to give some basic idea about penetration testing. Subsequently we will have a look at web application scanners and their goal.

### 2.1 WebSockets

WebSockets enable browsers to establish a persistent, full-duplex – bi-directional – and asynchronous connection to a server. The communication channel can be protected against eavesdropping with TLS, much like HTTPS. The default ports are 80 or 443, such that enterprises are not required to open additional ports in their firewalls. The initial handshake is done via HTTP(s). Afterwards the connection is "upgraded" and the TCP connection remains open as WebSocket channel, where the lightweight WebSocket protocol takes over. WebSockets are part of the upcoming HTML5 standard. The early stages of the WebSocket specification were incorporated in the HTML5 standard up to W3C Working Draft 12 February 2009. Afterwards it was outsourced to provide more flexibility. See the history section in Chapter 2.1. Today the relevant specification parts are:

- **The WebSocket API** [24]: Defines the interface for working with WebSockets in web browsers. The access to basic functionality is provided via JavaScript.

- **The WebSocket Protocol** [16] (RFC6455): Defines the lightweight wire protocol on top of TCP and the initial upgrade process from a HTTP-based handshake. This specification includes references to various registries at the Internet Assigned Numbers Authority (IANA), combined under the WebSocket Protocol Registries [52].

---

WebSockets can be used for real-time applications, due to its low protocol overhead. Additionally no more polling is required to retrieve updates, resulting in better scalability. The new standard offers a completely new communication paradigm. No more workarounds are necessary. The names for the techniques in use are various and sometimes mean the same. Just to name a few: Reverse Ajax, Comet, HTTP Streaming, HTTP Long Polling, BOSH and Reverse HTTP. See Chapter [6] for references to related work dealing with those workarounds.

The protocol was defined with extensibility in mind, such that you can use a WebSocket connection to tunnel higher level protocols such as Java Message Service (JMS), Extensible Messaging and Presence Protocol (XMPP or Jabber), Advanced Message Queuing Protocol (AMQP), Streaming Text Oriented Messaging Protocol (Stomp) or Remote Frame Buffer (RFB or VNC) over the web through firewalls and proxies [33].

Various sites exist that list browser support for new HTML5 features. One of them is http://caniuse.com/#feat=websockets. Another HTML5 compatibility table that is focused on mobile device support, can be found under http://mobilehtml5.org/.

See the following section for more background on the specification process. Afterwards I will introduce you to the WebSocket API and to the wire protocol. Finally I will deal with security considerations and provide a short comparison to other competing technologies.

History of WebSocket specification

The standardization body World Wide Web Consortium (W3C) decided to outsource various parts from the HTML5 standard, in favour of more modularity and faster progress. This included the WebSocket specification.

Table 2.1 shows an overview of the evolution of The WebSocket Protocol. In January 2001 it started with a draft from Ian "Hixie" Hickson, who is the author of the HTML5 standard. In May 2012, the Internet Engineering Task Force HyBi Working Group (IETF HyBi WG) took on responsibility on this wire protocol. After several drafts draft-ietf-hybi-thewebsocketprotocol-17 finally became RFC6455 [16]. There was one severe flaw in the protocol prior to version draft-ietf-hybi-thewebsocketprotocol-04: Huang et al. [26] discovered a problem where it was possible to conduct a cache poisoning attack, allowing to attack all clients of a proxy that is unaware of WebSocket connections. See the protocol security section in Chapter 2.1 for more information. Since this vulnerability was identified, outgoing message payloads are masked with a random key on a per-frame basis.

---

https://www.w3.org/  

http://tools.ietf.org/wg/hybi/  

https://datatracker.ietf.org/doc/draft-hixie-thewebsocketprotocol/history/  

https://datatracker.ietf.org/doc/rfc6455/history/
<table>
<thead>
<tr>
<th>date</th>
<th>document name</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 January 2009</td>
<td>draft-hixie-thewebsocketprotocol-00</td>
<td>First draft from Ian Hickson.</td>
</tr>
<tr>
<td>6 May 2010</td>
<td>draft-hixie-thewebsocketprotocol-76</td>
<td>Latest Hixie-draft.</td>
</tr>
<tr>
<td>23 May 2010</td>
<td>draft-ietf-hybi-thewebsocketprotocol-00</td>
<td>First HyBi-draft.</td>
</tr>
<tr>
<td>30 September 2011</td>
<td>draft-ietf-hybi-thewebsocketprotocol-17</td>
<td>Latest HyBi-draft.</td>
</tr>
<tr>
<td>12 December 2011</td>
<td>RFC6455</td>
<td>Proposed Standard</td>
</tr>
</tbody>
</table>

Table 2.1: History of The WebSocket Protocol

The other part of the standard, The WebSocket API, is maintained by the Web Applications Working Group (WebApps WG). Their first W3C Working Draft was released on 29 October 2009. After changing its status to a W3C Candidate Recommendation on 8 December 2011 the specification was forced to take a step back to W3C Working Draft on 24 May 2012. Finally it became a W3C Candidate Recommendation again on 20 September 2012.

The WebSocket API

The API specifies how you can create and access a WebSocket in your browser. Via JavaScript you can set-up & tear-down a connection, send & receive messages and do simple error handling. In Listing 2.1 you can see a simple HTML5 page. Its JavaScript code establishes a WebSocket connection to echo.websocket.org, sends a message and closes the connection when the echo-server responded the sent message. This is all done via event handlers that are called asynchronously. Every action in the example is logged on the HTML page.

The example uses the ws:// protocol instead of the encrypted wss://. After calling the constructor, various event handlers are set.

Listing 2.1: WebSocket API usage demonstration

```html
<!DOCTYPE html>
<html>
<head>
<title>WebSocket demo</title>
<meta charset="UTF-8" />
</head>
<body>
<h1>WebSocket API usage</h1>
<div id="log"></div>
<script type="text/javascript">
var logCtr = document.getElementById("log");
function log(msg) {
    logCtr.innerHTML += msg + "\n<br/>";
}
</script>
</body>
</html>
```

---

```javascript
var socket = new WebSocket("ws://echo.websocket.org");
socket.onopen = function(evt) {
  log("connection_open");
  log("send message");
  socket.send("WebSockets rock!");
};

socket.onmessage = function(evt) {
  log("received message: "+ evt.data);
  socket.close();
};

socket.onclose = function(evt) {
  log("socket is closed");
};

socket.onerror = function(evt) {
  log("error happened");
};
```

The WebSocket Protocol

Each WebSocket message consists of one or more frames. In Figure 2.1 you can see how such a frame looks like. There are some fields that are either not used (mask) or where its size is dynamically determined (extended length) resulting in different overhead. There was a little mistake of the size label under the extended length field. The maximum size is not 4, but 8 bytes. In Figure 2.2 you can see the overhead of a WebSocket frame depending on the direction and payload size. The overhead is:

- **minimal**: When a message is sent from the server to the client. For the other direction the overhead increases from 2 to 6 bytes. These 4 extra bytes are necessary due to masking (see protocol security section in Chapter 2.1).

- **maximal**: A frame is allowed to carry 0 to $2^{63} - 1 = 9,223,372,037 \times 10^{18} - 1$ bytes of data. This limit is set since some platforms do not support unsigned 64-bit integer. At most there are 8 bytes for the length field, resulting in 10 bytes overhead for incoming frames, respectively 14 bytes when the frame is sent from the client to the server.

The initial handshake starts on the level of HTTP. After completion, WebSocket frames are sent over the still-open TCP connection. The handshake is shown in Figure 2.3. The most important headers Upgrade: websocket and Connection: upgrade appear in both, request and response. The response for a successful handshake has to contain the HTTP status code 101, which stands for Switching Protocols. It is defined in the RFC2616 [17] and indicates the willingness of the server to change the application protocol immediately after the empty line of the
HTTP response headers. Hereby the target protocol is specified in the Upgrade-header of the request.

In addition there is another mandatory header called Sec-WebSocket-Key containing a random 16-byte nonce, which is base64 encoded. When the server receives the handshake request, it concatenates the original base64 encoded nonce with the fixed GUID “258EAFA5-E914-47DA-95CA-C5AB0DC85B11”. After concatenation, a SHA-1 hash is calculated, the hash value is base64 encoded, and the result is set in the Sec-WebSocket-Accept-header. Only if the right value is retrieved by the client (browser) the connection should be allowed. A possible scenario looks like this:

1. Client chooses random 16-byte nonce, e.g. abcdefghijklmnop
2. Client encodes nonce with base64: YWJjZGVmZ2hpamtsbW5vcA==
3. Server receives Sec-WebSocket-Key-header and concatenates its value with the fixed GUID: YWJjZGVmZ2hpamtsbW5vcA==258EAFA5-E914-47DA-95CA-C5AB0DC85B11
4. Server calculates a SHA-1 hash from this concatenated string: 7a7e4463d7d822b3c71ae7062c3e8393f11738de
5. Server encodes this 160-bit (20 byte) hash value with base64: eN5EY9fY1rPHGucGLD6Dk/EXON4=

Note: Not the string value is encoded, but the bytes.
6. Client receives `Sec-WebSocket-Accept`-header and checks if expected value is provided. If not, the connection is terminated.

This procedure shows the client the server’s willingness to accept this WebSocket connection. If there was no challenge/response handshake, malicious form submissions or XMLHttpRequests may be able to trick a server into establishing a WebSocket connection right away. So it proves the client that the server speaks WebSocket \[16\]. More information is provided in the next section of Chapter 2.1 that covers security considerations in the protocol.

If the server does not support the specified version from the client, it must terminate and send a `Sec-WebSocket-Version`-header indicating supported versions. As of October 2012, there is only version 13 available, but it might happen that servers & clients still support some earlier draft versions. While there was a link between WebSocket versions and draft numbers in `draft-ietf-hybi-thewebsocketprotocol-0X` from version 0 to 8, versions 9 to 12 remained unassigned, until draft 17 became version 13.

After a successful handshake, WebSocket frames have to be used for communication. While Figure 2.1 gives a rough overview, a more detailed view on the structure of WebSocket frames is provided in Figure 2.4. The size of the WebSocket packet headers depend on the direction and the size of the payload. First, no masking key is present when packet is sent from server to client. Second, if the payload gets longer, more bytes (ranging for an additional 1 to 8 bytes) are used to indicate the payload length. The first two bytes are always present. They are called `frame header` and consist of various flags & `opcode`, beside the payload length & the mask-flag.

---

Figure 2.4: Detailed WebSockets frame structure according to RFC6455 [16].

As shown in the **Opcode**-note in Figure 2.4, there are two categories of WebSockets frames: non-control & control frames. A non-control WebSockets message is allowed to consist of an arbitrary number of WebSockets frames. The first frame of a non-control message indicates the message’s type, which is either of opcode **text** (1) or **binary** (2). The following fragments of the current message are marked with opcode **continuation** (0). Finally, the last fragment will
have the FIN-flag set. If there is only one frame, the FIN-flag is allowed to be set on the first frame. The text-opcode demands the payload data to be encoded as valid UTF-8 characters. If it contained illegal values, either endpoint has to terminate immediately. While there might be a partial UTF-8 sequence in one frame, the whole (reassembled) message must contain valid UTF-8. For non-text, there is the binary-opcode, where arbitrary bytes can be sent. While a message is being sent, i.e. from its first to its last frame, the protocol forbids a new message. To send a new non-control message, of same or different opcode, the last message must be finished first.

Besides non-control frames there are control frames. In contrast to non-control frames, they are allowed to be sent at any time, e.g. while a huge text message is sent in several frames. This is possible, as control frames fit into one frame - splitting them into several frames is not allowed. Though, they are allowed to contain some textual (UTF-8) payload.

Endpoints that receive a frame of opcode ping (9), must respond to it as soon as possible with a pong (10) frame containing the same payload. This mechanism can be used to derive Quality-of-Service (QoS) parameters (e.g. Round-Trip-Time). Pong-frames are also allowed to be sent unsolicited in order to serve as unidirectional heartbeat. This may be used to prevent connection timeouts.

Another control frame is defined as opcode close (11). While the WebSocket connection could be closed theoretically by closing the underlying TCP connection, the specification demands for an in-band closing handshake. See Figure 2.5 for the closing procedure, where there is no requirement which endpoint may send the first close frame. The goal or reason for this mechanism is to avoid problems with intermediaries. Close frames are allowed to contain a status code consisting of 2 bytes. More detailed descriptions about status codes can be found in RFC6455 [16].

For the payload length the specification demands that “in all cases the minimal number of bytes must be used to encode the length” [16]. If the payload size is below 126, the Extended Payload Length field is not present. When the payload size increases up to \(2^{16} - 1 = 65535\) (64KB), the Extended Payload Length field is 2 bytes long. If the payload length is above that value, all 8 bytes are in use. How many bytes are used to determine the length of the payload is indicated by the value of the field Payload Length. As shown in Figure 2.1 if the value is 126, the next 2 bytes, if 127, the next 8 bytes are used as payload length. As a result there is minimal overhead for small messages with payload sizes of up to 125 bytes, because they fit into one WebSocket frame that has got only 2 bytes overhead, respectively 6 bytes when sent from the client to the server due to masking.

The reason for masking and its approach are described in the protocol security section of Chapter 2.1. Beforehand, I want to point out the extensibility of the WebSocket protocol. Besides reserved flags (RSV1-3) & unassigned opcode values that allow future expansion there are two mechanisms you can build upon to extend the protocol:

(a) Sub-Protocols: Application-level extension that defines how the payload must look like. You can use WebSockets as transport layer for any protocol. Some of them are formally defined for usage over WebSocket channels. See Table 2.2 for a list of current sub-protocols.
Figure 2.5: In-band WebSocket closing handshake before TCP connection terminates with status codes according to RFC6455 [16] and IANA WebSocket Close Code Number Registry [52].

(b) Extensions: Protocol-level extensions are allowed to make use of reserved flags and change how the payload is represented. Some drafts for extensions are shown in Table 2.3.

Everybody can define sub-protocols & extensions. Some of them have registered their names at the IANA WebSocket Registry [52].

---


http://go.microsoft.com/fwlink/?LinkId=231897&clcid=0x409

http://wamp.ws/spec
identifier | name
--- | ---
MBWS.huawei.com | The MessageBroker WebSocket Subprotocol
MBLWS.huawei.com | SOAP Over WebSocket Protocol Binding Specification
soap | The WebSocket Application Messaging Protocol
wamp | Simple Text Oriented Messaging Protocol
v12.stomp | An XMPP Sub-protocol for WebSocket
xmpp | The WebSocket Protocol as a Transport for the Message Session Relay Protocol
msrp | The WebSocket Protocol as a Transport for the Session Initiation Protocol (SIP)
sip | NETCONF over WebSocket
netconf

Table 2.2: Existing specifications for WebSocket sub-protocols.

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Multiplexing Extension for WebSockets</td>
<td>Allow several logical WebSocket connections through one underlying TCP connection.</td>
</tr>
<tr>
<td>WebSocket Per-message Compression</td>
<td>Use of compression algorithms to reduce payload size for non-control frames.</td>
</tr>
</tbody>
</table>

Table 2.3: Existing specifications for WebSocket extensions.

WebSocket Protocol Security

In this section I want to deal with security features that come with WebSockets. I want to explain why there was the need to mask the payload sent from the client to the server.

In the beginning I want to say that many existing HTTP security mechanisms apply to Web-

Sockets too. Being able to leverage existing mechanisms was a reason to stay HTTP compatible. Let us start with encryption:

The WebSockets protocol supports encryption. If the scheme used to connect is wss:// a TLS-handshake must be performed before the WebSocket-handshake. As the WebSocket-handshake is HTTP-based, the TLS-handshake for the secure connection does not differ from a request to an encrypted website. Using encryption results in the CIA-properties for the communication channel, which are:

- **Confidentiality**: Payload sent over the channel cannot be read by third parties.
- **Integrity**: Payload sent over the channel cannot be changed by third parties in a way that the receiving party will not notice that.
- **Authenticity**: Payload received is guaranteed to be from the other party.

However, encryption does not prevent SSL Stripping attacks [35]. When the initial URL to a secure site is entered as http://, but not https:// a man-in-the-middle is able to exploit the redirect to HTTPS to deliver all content to the victim unencrypted and therefore read/change all messages. Regarding WebSockets, this attack vector does not apply. When the demanded scheme is wss:// a non-secure connection is not accepted by the browser. Moreover if a plain connection is requested, but the server wants the client to use an encrypted session instead, a redirect will fail - at least in Firefox. See the browser configuration section in Chapter 2.1 for more details.

*Note:* Using encryption provides secure transport, but it does not mean that you have got a secure application. You have to take care of the payload content in your application logic [45].

Another feature from HTTP that is available to WebSockets is HTTP-authentication as defined in RFC2617 [19]. Before the WebSocket protocol takes responsibility a challenge-response authentication – either basic or digest – can be conducted on HTTP-level.

Besides HTTP-authentication a WebSocket server can make use of an existing, possibly authenticated session, as Cookie values are sent with the handshake request. This enables servers to skip recurring authentication, when given Cookie value already indicates an authenticated session.

Security for WebSockets rests upon the browser’s origin model. Browsers submit an Origin-header containing the base URL of the site requesting a WebSocket connection. The concept of an origin is defined in RFC6454 [7] as a scheme, a host and a port. Due to this address value, which is used for modelling trust relationships on the web, the server can decide if it permits the connection request. This protects against “unauthorized cross-origin requests by scripts using the WebSocket API in a web browser” [16]. Of course, the Origin-header can be set to an arbitrary value by non-browsers or ignored by the server, but this is not the intention. The idea behind is to allow a WebSocket server to decide to whom it wants to talk to. Shema says [44]: “Hackers can spoof this header for their own traffic (to limited effect), but cannot exploit HTML, JavaScript, or plugins to spoof this header in another browser”. This means that trusted clients (browsers) experience protection.
Another protection against misuse is realized using the \textit{Sec-WebSocket-Key}-header. Only if the server is able to respond with an appropriate \textit{Sec-WebSocket-Accept}-header, the connection is allowed by the browser. This "prevents an attacker from tricking a WebSocket server by sending it carefully crafted packets using XMLHttpRequest or a form submission" \cite{16}.

\textbf{Note:} Header names starting with \textit{Sec-} are not allowed to be set when using the browser’s XMLHttpRequest object, where you have got the possibility to use the API method \texttt{setRequestHeader(header, value)} for setting custom headers\cite{18}. You are also not allowed to set e.g. the \textit{Origin}-header.

The challenge/response mechanism with the \textit{-Key} and \textit{-Accept} header also prevents cross-protocol attacks. Consider some XSS-weakness in Twitter, where a WebSocket connection is set up to some SMTP server in order to send some spam \cite{44}. Everybody visiting the exploited site would aid the spam mail campaign by simply reading a Tweet. By ensuring both parties speak WebSockets such attack is not possible. No browser would allow establishing a WebSocket connection to foreign – \textit{non-WebSocket} – servers.

WebSockets were intended to be a lightweight layer on top of TCP that allows sending and receiving payloads in clear-text. But some problems emerged with network intermediaries. Caching proxies could be attacked when using the unencrypted \textit{ws://} scheme. Consider the following scenario, also shown in Figure \ref{fig:network-intermediaries}: An attacker sets-up a WebSocket connection through a targeted proxy to his own server. Then he sends some payload, which mimics a HTTP request: e.g. requesting \url{http://www.google-analytics.com/ga.js}. Next, the WebSocket server responds with some crafted HTTP response headers and a malicious JavaScript file. The headers indicate to cache this file for several days, which is done by the proxy. We have achieved cache poisoning, as the caching proxy is not aware that this is a WebSocket channel. Imagine that this proxy belongs to some bigger company - all of the users inside the company will get the malicious JavaScript file injected, whenever they visit a website that includes the Google Analytics file. This attack and some countermeasures were described by Huang et al. in \cite{26}.

A variant to avoid this issue was adopted by the WebSockets standard: The outgoing bytes for the payload are masked using a XOR-operation with a random key consisting of 4 bytes. For every WebSocket frame another random key is generated (by the browser). The key is sent within the frame, allowing the receiver to unmask again, using the XOR-operation a second time. As a result some JavaScript code injected via e.g. XSS, cannot send bytes that look like a HTTP-request, as the payload bytes are masked with a random key. If the attacker would know the key in advance, he could calculate which bytes he has to send to achieve the attack again. Unfortunately for the attacker, the bytes used are not known beforehand. Consequently the black-hat cannot control the bytes on the wire (from a browser). The algorithm applied is shown in Figure \ref{fig:xor-algorithm} and can be described as \cite{16}: Byte $i$ of the transformed data (\textit{transformed-}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{xor-algorithm.png}
\caption{XOR-algorithm for payload masking.}
\end{figure}


**Figure 2.6:** Attacker could poison the cache of some proxy (picture from\textsuperscript{[19]} \textsuperscript{[26]})

**Figure 2.7:** The payload is masked one byte after another with the repeated mask\textsuperscript{[19]}

\[ j = i \mod 4 \]

\[ \text{transformed-byte-}i = \text{original-byte-}i \oplus \text{masking-key-byte-j} \]

With masking it does not suffice that the victim within the target network visits some attacker-controlled site. Moreover exploiting a XSS-vulnerability does also not aid this type of attack. But, if the attacker is able to use a non-browser client, he is able to control the masking keys, choosing arbitrary ones, allowing to conduct the attack described above. Think of Java Applets or executable files. As a result it is recommended to use WebSocket-aware proxies & firewalls. For these reasons masking does not eliminate this type of attack, but harden it. Another attack vector emerges when the random number generator used to create the masking keys is weak and the keys used in future are predictable. Consequently the attacker is able to control how the payload will appear on the wire, bypassing masking. Therefore it is important to use a cryptographically strong random number generator as also described in [16].

To be save from cache poisoning attacks, one could configure their firewall to enforce TLS-handshakes for WebSocket-upgrade-requests. This means no HTTP-request containing an Upgrade: websocket-header should be allowed to proceed with the insecure ws:// scheme. As a
result of forcing the wss:// scheme, no clear-text messages can be misinterpreted by network intermediaries.

There are probably two reasons why this masking is only done in one direction. First, a possible HTTP-response sent by the server has no effect without the existence of a HTTP-request. Second, masking causes 4 byte overhead as seen in Figure 2.2 Why masking is also done on the secure transport channel is not stated by the standard - maybe for consistency reasons such that it is not unnecessarily complex. Another possible reason could be to ensure that no network intermediary is tricked by this attack, when only the transport channel between networks is secured, but the frames are forwarded without TLS within a company’s network.

The RFC6455 [16] stresses also the importance that an application should not be possible to change the payload while transmission is already in progress. Imagine a large frame, where the masking key and the first bytes are already on the wire. If the application developer is then able to set the bytes in the back area of the frame, he/she already knows the mask and could control the bytes on the wire.

On the one hand masking prevents cache-poisoning attacks, but on the other it impedes work for data loss prevention, intrusion detection- & intrusion prevention software and firewalls. When they are not aware of WebSockets, no data is analysed on this communication channel (or even worse misinterpreted). As a result no Malware or malicious JavaScript will be identified nor will data leakages be detected [50, 45].

Within the WebSocket protocol handshake, a sub-protocol for the payload can be defined via the Sec-WebSocket-Protocol-header. This is also a security feature as this allows protocol-aware intermediaries to validate the payload. As soon as firewalls and gateways understand the WebSocket protocol and the sub-protocol in use, security will improve a lot.

Use Cases

Usage of WebSockets in web applications is recommended when you have got one of the following requirements according to [45]:

- time critical data delivery
- requirement for true bidirectional flow
- interactivity
- higher throughput

Shema et al. conclude that WebSockets are well suited for browser games, update-intensive widgets (consider stock quotes or live ticker) and other interactive applications (consider maps or chat applications).

Later on Vaagn Toukharian, researcher at Qualys, suggests in a blog post to adopt WebSockets when there is need for higher throughput, full duplex communication or lower latency.
But you should always remember security basics (authentication/authorization, session management, state handling), because the WebSocket protocol isn’t aware of these [50].

Different use cases for WebSockets, theoretical and practical ones are considered in various papers. In some work, performance is measured & compared to related technologies:

Klimek et al. [30] found an interesting alternative to speed up HTTP utilizing WebSockets. Like SPDY [40] their approach tries to reduce the number of TCP connections but in a transparent manner, requiring no additional protocol support. Not a restricted bandwidth slows down a connection, but the Round-Trip-Time. Increasing the bandwidth from 5 to 10 Mbps has less effect on the page load time, than the reduction of the time a packet needs to travel from your browser to the server and back (Round-Trip-Time). They introduce a network-based optimization tool, called Instant Page Load. It aggregates a requested web-page and their associated resources together and delivers them in only 2 TCP connections. One for the initial web page and another for a WebSocket connection that is set-up after loading the landing page. The WebSocket connection is then used to loading all other resources (scripts, style-sheets, images, etc.).

A data-binding framework for the web, based on WebSockets and HTML5 Microdata[21], is proposed by Heinrich & Gaedke [23]. It allows linking client-side HTML elements to server-side models in a declarative manner. While the binding is specified in the HTML via three attributes, namely itemscope, itemtype and itemprop, the model or UI changes are communicated through WebSockets. An advantage of such approach has been seen with Desktop Binding Frameworks that “speed up the reoccurring development task of coupling UI elements and data objects”. The authors argue that their declarative approach with WebSockets doesn’t require the web developers to learn complex JavaScript binding libraries, but only HTML, JavaScript and a server-side programming language. Moreover no expensive polling technique is leveraged, but the lightweight two-way WebSocket protocol.

Kazi & Deters [29] attest WebSockets a “very efficient and reliable form of communication”, as they allow “for significantly faster data transmission rates” compared to polling. Their work focuses on mobile devices as they become more important every day. They fundamentally change the way we use IT infrastructure. The authors cite an ABI research report that says that by 2016 there will be 2.1 billion HTML5 browsers on mobile devices. The paper looks at current publish-subscribe approaches for mobile devices and summarizes existing work based upon WebSockets.

Another reason for fast adoption of WebSockets in mobile devices comes up with the introduction of the next generation mobile network LTE (Long Term Evolution). The low overhead, real-time capable WebSocket protocol will be able to unfold its whole power when used in low-latency network environments. Various papers report Round-Trip-Times of less than 10ms in LTE networks [9, 43]. This is comparable low to a measurement from Jurvansuu et al. on

a live operational network utilizing HSDPA technology, where the Round-Trip-Time has been 80ms [27]. Arjona et al. [2] state the expected Round-Trip-Time for the last technology in use before LTE, HSUPA as roughly 65ms. As a result LTE will be able to significantly boost the mobile web.

Mandyam & Ehsan [34] examined WebSockets with regard to power consumption on mobile devices. As the keep-alive of WebSocket connections is not clearly specified, developers can use different methods. They can send ping-, unsolicited pong-frames, or application level keep-alive messages. Such messages keep the TCP connection alive, but at the same time mobile devices are kept from switching to the energy-saving Fast Dormancy mode. As a result, the authors suggest falling back to Ajax with longer intervals between messages, when the battery of the mobile device is low or reports fast consumption. This is possible due to the W3C battery API[22].

**Browser configuration**

Firefox version 16.0.1 has got several WebSocket related options that can be found if you type in about:config in the URL-bar and then search for "websocket". Additional information was taken from the developer’s page for WebSockets[23]. We will look at the directives and figure out what they are supposed to do:

- `network.websocket.allowInsecureFromHTTPS`: boolean [false]
  Normally the WebSocket API does not allow connecting to WebSocket servers using the ws://-scheme if the connecting page was loaded through HTTPS. This does not apply e.g. to Google Chrome 23, which is considered bad practice.

- `network.websocket.auto-follow-http-redirects`: boolean [false]
  Per default, no redirection is allowed when connecting to a ws(s):// URL. As a redirection is denied per default, SSL-stripping attacks are not possible.

- `network.websocket.delay-failed-reconnects`: boolean [true]
  Might prevent immediate re-connections after a previous request has failed for the sake of preventing Denial-Of-Service-attacks.

- `network.websocket.enabled`: boolean [true]
  Switch to turn WebSockets completely off or on.

- `network.websocket.extensions.stream-deflate`: boolean [false]
  Would enable the stream-deflate extension. It was disabled per default in Firefox 8 as the specification for this extension got deprecated.

---


• **network.websocket.max-connections**: integer [200]
  Maximum number of WebSocket connections across all tab instances.

• **network.websocket.max-message-size**: integer [2147483647]
  The theoretical size of messages is limited to a default of $2^{31} - 1 = 2147483647$ bytes, which is 2GB. It has been 16MB per default before Firefox 11. From the specification side [16], the maximum payload size is stated as $2^{63} - 1$ bytes, but using implementation-specific limits is perfectly valid. This should prevent Denial-Of-Service attacks.

• **network.websocket.timeout.close**: integer [20]
  When a close-frame is sent, and there is no answer from either side, the connection will be terminated after 20 seconds

• **network.websocket.timeout.open**: integer [20]
  If the connection is not set-up, i.e. the handshake is not completed within this timeout, the try is aborted.

• **network.websocket.timeout.ping.request**: integer [0]
  Intention not clear.

• **network.websocket.timeout.ping.response**: integer [10]
  Intention not clear.

According to Shekyan & Toukharian [45] browsers have got different policies regarding the maximum number of concurrent WebSocket connections. See Table 2.4 for current values.

<table>
<thead>
<tr>
<th>browser</th>
<th>max-connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chrome</td>
<td>3237</td>
</tr>
<tr>
<td>Chromium</td>
<td>924</td>
</tr>
<tr>
<td>Firefox</td>
<td>200</td>
</tr>
<tr>
<td>Opera</td>
<td>900</td>
</tr>
<tr>
<td>Safari</td>
<td>2970</td>
</tr>
</tbody>
</table>

**Table 2.4**: Maximum number of concurrent WebSocket connections allowed by different browsers according to [45] in August 2012.

**Browser support & polyfills**

You can look up specific browser support at various websites; two of them were mentioned at the beginning of the WebSockets background section in Chapter 2.1. When developers make use of new technologies they often care about backward compatibility regarding the browser version in use. If a user’s browser does not support feature X, polyfills can be used. Polyfills provide browser-fallbacks, written in JavaScript or Flash, which allow developers to use new browser features also in older browsers that do not support the required functionality. The missing feature is then emulated. In case of WebSockets, there are several ways to emulate it.
A concrete library that provides you with the best real-time communication channel available is *Socket.IO*[^24]. If your browser does not support WebSockets it will automatically fall-back to:

- Adobe Flash Socket
- Ajax Long Polling
- Ajax Multipart Streaming
- *Forever iframe*
- JSONP Polling

Besides *Socket.IO* there are many other JavaScript libraries providing polyfills. It was just mentioned due to its popularity. Another possible fall-back requires an active Java plug-in in your browser, where you can use raw TCP Sockets in Java Applets.

**WebSockets versus SPDY?**

SPDY, pronounced as “SPeeDY”, is a proposal from Google mainly to speed up page loading time. Its idea is to wrap HTTP and send all resources required by websites through only one TCP connection. SPDY’s main features are[^49]:

- **Request Multiplexing**: Is inspired by HTTP’s pipelining. However, it avoids performance penalties, as resources in the response are not expected to occur in a particular order. Moreover it supports prioritization.

- **Encryption**: There are various reasons to use SSL by default: First, the performance overhead of SSL is comparable low, as it had to be done only once for the single TCP connection. Second, with SPDY over SSL, transparent proxies do not cause problems. Last but not least, according to Google, SPDY outperformed unencrypted HTTP[^40].

- **Header Compression**: The same HTTP headers reoccur often. Compression techniques can be used to lower the overhead.

- **Server Push**: Round-Trip-Times can be saved, when resources are pushed from the server to the client. Imagine you are loading the landing page of some website that is delivered via SPDY. After loading the *index.html* file, the browser may load style-sheets, script files and images. If the server pushes these resources before the main html file is processed without an explicit request from the browser, it can save requests for each resource already received. The downside of this feature may be wasted bandwidth when resources are already cached.

• **Server Hint**[^25] Avoids loading resources when they are already in cache, as done with **Server Push**. Only the URL of the resource the client will need is sent to him, but not its data. Subsequently the client looks for a cache hit. If the resource is found, no request must be made. In the case of a cache miss, the resource has to be requested as usual.

SPDY might look like an alternative to WebSockets, but it is not. Some people on the internet try to set up a competition. Especially the features **Server Push & Server Hint** may be a reason for this. Both can be used to deliver resources to browsers more quickly, but you cannot listen to a **Server Push** from JavaScript like you can do with WebSockets.

SPDY aims to improve the performance of web pages, but stays backward compatible. It incorporates the HTTP protocol. Consequently existing infrastructure, such as caching proxies, do not have to be replaced. They can continue to work outside the SPDY connection.

With WebSockets there is no HTTP, except the initial handshake which is done utilizing HTTP’s upgrade mechanism. In contrast to SPDY, WebSockets target the real-time interactive web. In this web, stock quotes & live sports data are communicated, multi-player games are run and collaborative applications bring people together (these ideas were taken from a blog post[^26]).

So SPDY and WebSockets do not compete, but complement each other. There is even a proposal[^27] to set-up WebSocket channels within SPDY connections, beside other HTTP requests or responses. Otherwise an additional TCP connection would be required for WebSockets, when SPDY is used to deliver a website. This would result in wasted resources.

**HTML5 Connectivity Features**

WebSockets belong to the connectivity feature set of the upcoming HTML5 standard. A badge with the HTML5 connectivity logo is shown in Figure 2.8. You can put that badge onto your website in order to show off that you are using specific HTML5 features.

---


[^27]: “WebSocket Layering over SPDY [public draft]”, Google Docs. Last modified August 27, 2012. Accessed October 25, 2012. [https://docs.google.com/document/d/1zUEFzz7NCl5s3Yms8hXx1Y4wGXX3EEvoZc3G1hrqPQcR0/edit](https://docs.google.com/document/d/1zUEFzz7NCl5s3Yms8hXx1Y4wGXX3EEvoZc3G1hrqPQcR0/edit)
Figure 2.9 shows an overview of all connectivity methods. While the initial Website is loaded via HTTP or SPDY, different technologies are available to keep the site up-to-date. With Server-Sent Events one TCP connection can be used to provide one-way communication from your server to clients. With the XMLHttpRequest object, you can issue requests in the background, leveraging the existing HTTP request/response infrastructure. In HTML5 its extended specification allows for cross-origin requests. WebSockets provide a two-way communication channel within one TCP connection. Finally cross document messaging can be done via the postMessage API, allowing you to send messages between frames of different origins.

In the following sub-section I want to deal with Server-Sent Events, as they are somehow related to WebSockets.

http://www.w3.org/html/logo/  
https://speakerdeck.com/marakana/the-html5-connectivity-revolution
Server-Sent Events

With Server-Sent Events (SSE) you are able to send a continuous stream of data from servers to browsers. You can access this functionality via a JavaScript API called `EventSource`. SSE provides a one-way stream of data, which is in contrast to the bi-directional nature of WebSockets. In use cases where the client (browser) does not need to communicate something back to the server, SSE are a viable alternative to WebSockets. Moreover SSE do not require a special protocol or server implementation to be useful.

The specification of SSE specifically addresses mobile devices with its *connectionless push* feature that “reduces the total data usage, and can therefore result in considerable power savings” [24].

See Listing 2.2 for a SSE client implementation. The client connects via the `EventSource(url)` constructor to a server side script called `sse_server.php` shown in Listing 2.3. The `EventSource.onMessage` handler is called for every event that has got no specific event handler. For the *server-time* event, a specific handler is defined. After 10 seconds, the server script terminates.

**Listing 2.2:** Server-Side Events API usage demonstration. See Listing 2.3 for simple server script.

```html
c<!DOCTYPE html>
<
<html
<head
<
title>Server-Sent Events demo</title>
<meta charset="UTF-8" />
</head
<body
<h1>Server-Sent Events API usage</h1>
<div id="log"></div>
<script type="text/javascript">
var logCtr = document.getElementById("log");
function log(msg) {
  logCtr.innerHTML += msg + 
<br/>
}
var url = "http://localhost/sse_server.php";
var stream = new EventSource (url);
stream.onopen = function () {
  log("connection_established");
};
stream.onerror = function () {
  stream.close();
  log("connection_closed");
};
stream.onmessage = function (evt) {
  // receives messages, not processed by custom handlers
  log("received_event_message: " + evt.data);
};
```
Listing 2.3: Simple PHP script emitting some Server-Side Events.

```php
<?php
    /*
     * Server-Sent Events Demo
     * Server Side PHP Script creating some events
     */
    header("content-type: text/event-stream");

    // start output buffering
    if (ob_get_level () == 0) {
        ob_start();
    }
    echo "data: Hello! \n";
    echo "\n";
    ob_flush();
    flush();
    $startTime = time();
    while (true) {
        // This is an example, no connections should exceed 10 seconds
        $time = time();
        if (($time - $startTime) > 10) {
            break;
        }
        echo "event: server-time\n";
        echo "data: \n" . ($time) . "\n";
        echo "\n";
        ob_flush();
        flush();
        sleep (2.4);
    }
    ob_end_flush();
```

Automatic reconnection is another feature specific to SSE. In Firefox 18 this `dom.server-events.default-reconnection-time` is 5000 milliseconds per default. Combined with the Last-Event-ID header, this is very useful for broadcasting: Every time the browser receives an event containing a line like this `id:<some-value>`, it sets an internal attribute
called \textit{lastEventId}. If the connection is closed, the client attempts to re-connect automatically. This time it includes the HTTP header \textit{Last-Event-ID} with the value of the internal \textit{lastEventId}.

### 2.2 Vulnerabilities

Vulnerabilities are errors that attackers can exploit. They arise from defects and are either implementation-level bugs or design-level flaws \cite{36}. Humans make mistakes/errors that result in flaws/defects/faults in software. If code containing such flaws is executed, it causes a failure. Flaws may or may not result in failures. Some flaws might stay dormant for the whole lifecycle of a software product.

<table>
<thead>
<tr>
<th>Notation</th>
<th>alert raised</th>
<th>really vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positive</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>False Positive</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>False Negative</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>True Negative</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

\textbf{Table 2.5:} Notation for (un-)detected (non-)existing vulnerabilities.

In Table 2.5 you can see the common notation when talking about security tests that either discovered vulnerabilities or missed existing ones - the two notations that are important to security testers are:

- \textit{false positive}: An alert is raised for a possible vulnerability that does not exist. If tools report a false positive, it costs the tester time to manually determine that this alert is a "false alarm".

- \textit{false negative}: There is an uncovered vulnerability – i.e. a flaw that awaits detection – that can be exploited. A tool should not report false negatives, because this way it misses detection of existing vulnerabilities.

In the two remaining cases we observe the correct behaviour. According to Li and Xue \cite{32}, attacks can be classified into two cases:

- \textit{input validation attacks}: User-provided input is not treated correctly, allowing malicious code to be executed. HTML form fields or cookie values are examples for areas where an attack may be initiated. These entry points are called \textit{input vectors}.

- \textit{state violation attacks}: Logic flaws of the application are exploited e.g. to get access to sensitive information, bypass authentication, or run restricted functionality. This may be possible due to forgotten authentication checks on specific admin pages.
2.3 Penetration Testing

Penetration tests can be used to find vulnerabilities in web applications. The tester, also referred as *white hat*, uses the same methods as a *black hat* (attacker) would use to discover weaknesses. White hats literally perform attacks on applications. The only distinction to black hats is the end result. While the motivations of black hats are numerous, the goal of white hats is to increase the security of the application under test, by finding vulnerabilities and provide information about possible weaknesses to developers. Penetration tests are sometimes abbreviated as *pentests* and the approach is also called *ethical hacking*.

Penetration testing of web applications may include, amongst other steps, the following actions [20]:

- errors are generated where possible
- unexpected input is supplied
- interfaces are assaulted
- protocols are examined and altered
- cookie contents are abused
- tools are employed

A success of penetration testing cannot be expected. There is no guarantee that your system is secure when no additional problems are found during testing. Geer & Harthorne understand penetration testing as art of proving the existence of the unexpected [20]. Moreover the performers of the tests are described as artists that have got the vital characteristic of creativity. Penetration testing doesn’t follow a standard process [3]. As a result as penetration tester you have to think out-of-the-box [10]. It does not suffice to test the expected behaviour according to an application’s design document, but to test and look for vulnerabilities and misconfigurations also beside the core functionality.

The *NIST Special Publication 800-115* describes penetration testing as a four-stage methodology [41]. In Figure 2.10 you can see the following phases:

- **Planning**: Rules and scope are identified, management approval is awaited and the objectives are defined.
- **Discovery**: Information about the system under test is gathered and compared against vulnerability databases.
- **Attack**: Potential vulnerabilities are used to exploit the system; gained knowledge during attack is used in an additional discovery phase.
- **Reporting**: Continuous phase that accompanies other phases, documenting an assessment plan, identified vulnerabilities, a risk rating and mitigation strategies.
There are several ways to conduct security tests: While e.g. static code analysis is based on a white-box approach, a penetration test is basically a black-box test. In the latter category you have got no insights into the application. You know nothing about the application’s design, its code or deployment. You try to find out as much as possible about an application from the outside like an attacker. Then you leverage your gained knowledge to break into the system. At this point you can aid the testing process, by using some internal information about the application. You still conduct your test as black-box test, but you use additional information to identify more possible entry points or attack vectors. As it is a combination of white-box and black-box, it is referred to as grey-box testing [53].

McGray said that penetration testing is best suited to find configuration problems and other environmental factors that affect an application’s security [35]. In [4] Austin & Williams compared some vulnerability discovery techniques against each other:

- **Static Analysis**: Found most implementation bugs in their experiment.
- **Systematic Manual Penetration Testing**: Found most design flaws.
- **Automated Penetration Testing**: Most effective way to find implementation bugs (i.e. the most bugs in short time).

Arkin et al. stated that “penetration testing is the most commonly misapplied mechanism as well” [3]. When performing a penetration test it does not suffice to apply tools in an unstructured manner or focus manually only on one type of vulnerability. Instead you have to protect against all potential vulnerabilities, because as developer or security tester you strive for a thorough application security. This is in contrast to black hats that have to discover only one weakness in your application. Matt Bishop said: “All types of testing can show only the presence of flaws and never the absence of them!” [10]. McGraw finds similar words in [36]: “Passing a low-octane penetration test indicates little about your actual security posture, but failing a canned penetration test indicates that you’re in very deep trouble”. Additionally penetration testing is not only about hacking into an application, but it “requires a detailed analysis of the threats and

**Figure 2.10**: Four-Stage Penetration Testing Methodology taken from NIST SP800-115 [41].
potential attackers to be most valuable” [10]. Moreover it should be used as "last check" before your application goes into production and not as "first check" if your application is secure [36]. In order to gain the most advantage of penetration testing, it should be integrated into a secure software development life cycle (SDLC). There are proposals in [3] and [53]. Penetration testing should be integrated into the development lifecycle for various reasons [3]:

- Problems discovered late in the development are expensive to fix. “Time and budget severely constrain the options for remedy.”
- Mitigation strategies are developed from root-cause analysis of found vulnerabilities. These strategies often include awareness trainings for developers. The identified type of vulnerability is avoided pro-actively with best practices & coding guidelines.
- Findings from penetration testing improve on-going design, implementation and deployment practice.

2.4 Web Application Scanners

Tools can aid the search for vulnerabilities. As penetration testing does not pretend specific actions, the variety of tools in use is large. Each penetration tester has got its own collection of tools. It has to be noted that tools are often only able to find the lowest hanging fruit [20]. Without thorough skills of the penetration tester the fruit higher up are hard to reach. There are several advantages when tools are part of the testing process:

- **Grunt work**: Repetitive work can be done by tools, when used effectively, leading to less workload left to be carried out by security experts [3].
- **Metrics**: Tools produce output that can be used to get an overall picture about an application and its progress. While you cannot ensure the absence of bugs you can gain advantage when your metrics improve compared to earlier values, i.e. if you lower the number of potential vulnerabilities you gained something positive [3].
- **Automation**: While manual penetration testing is labour-intensive, automated penetration testing can be a very effective way to discover implementation bugs in terms of the number of vulnerabilities found in a given amount of time [4].

**Definition**

To define the term web application scanner we need to state what is a web application first. I want to give you two complementary definitions:

- “A web application is a software application, executed by a web server, which responds to dynamic web page requests over HTTP”, given by the Web Application Security Consortium (WASC) in [51].
• Another definition by Kappel in [28]: “Web applications are software systems based on technologies and standards of the World Wide Web Consortium (W3C). They provide Web-specific resources such as content and services through a user interface, the Web browser.”

Now we can continue to define: “A web application scanner is an automated program that examines web applications for security vulnerabilities.” [18]. Doup’e et al. describes a web application scanner as composition of 3 modules [12]:

1. **Crawler**: Identifies as much entry points as possible, i.e. URLs with possible GET parameters and HTML form input values (POST parameter).

2. **Attacker**: Tries to exploit vulnerabilities, e.g. inject JavaScript or SQL operators.

3. **Analysis**: Looks at the application’s response in order to detect if vulnerabilities were found. Of course, this is not possible e.g. for blind SQL injection vulnerabilities. In this case response times can be measured and compared: If the response does not yield an error message, a longer operation can be injected to slow down the processing, e.g. issuing 100 pings on command line or `waitfor delay '0:0:20'` for SQL injection vulnerabilities. If the response time is higher than usual, a potential vulnerability was found.

The most important tool for web application testers is, of course, the web browser. Today, browsers include developer toolbars and support custom extensions [31] that can be used to inspect a web application from the outside. Web application scanners are focused on the security of applications, like some browser extensions, but browser-independent. These security tools incorporate several helpers that are able to aid the search for vulnerabilities. The most common parts are known as:

• **Spider**: It crawls the web application on a given URL and identifies further hyperlinks on the fetched page. The pages behind these hyperlinks are then scanned recursively. The outcome is a list of existing URL’s and represents possible entry points for attacks.

• **Fuzzer**: Input values such as form values, URL parameter values or HTTP header values can be varied to get a different output. Through this value enumeration, potential vulnerabilities can be detected.

Automation is key with fuzzing. Manual enumeration of several parameters at one time would be mind-numbing and time consuming. Moreover you will not be able to try a large number of potential identifiers or a syntactic range of identifiers that are believed to be used by the targeted application [48].


[https://www.owasp.org/index.php/Phoenix/Tools#Browser](https://www.owasp.org/index.php/Phoenix/Tools#Browser)
Vulnerability Scanner: Gathers information about software in use. This includes operating system, web server, firewalls, scripting language interpreters and further more. Afterwards the found versions are looked up in vulnerability databases to find out about potential weaknesses and exploits.

There is a wide variety of tools that focus on different aspects. A list of existing commercial and free/open source tools can be found here: [32]

Evaluation, Limitations & Challenge

Doupé et al. examined 11 web application scanners and concluded that there is a high rate of false negatives, i.e. a lot of uncovered vulnerabilities. Additionally the authors list several other publications reporting a similar result. As conclusion they “warn against the naive use of web application scanners (and the false sense of security that derives from it)” [12]. As not every scanner operates like the others, different tools should be used, when the security of a web application is assessed. Some common problems for web application scanners are [12, 13, 21, 48]:

- **client-side code:** Often tools do not execute client-side code, e.g. JavaScript. Modern applications do a lot of dynamic stuff, such as loading content via Ajax. If the used web application scanner does not run such client-side code, it may miss various execution paths and the detection of potential vulnerabilities.

- **“deep” vulnerabilities:** Exploration of hidden vulnerabilities on pages, which can be reached only e.g. after submitting complex forms, is hard to achieve with a tool. Before you can detect vulnerabilities on a page you have to find that page first. Doupé et al. call this issue "discoverability problem". This incompleteness is also stated in [21]: Halfond et al. stress the importance of complete identification of an application’s input vectors. As web crawlers are unable to do so, they suggest using their static analysis technique for identifying potential input vectors before executing the black-box penetration attack phase.

- **application state:** Tracking of an application’s state, e.g. login status or multi-step forms. If you do not take an application’s state into account, you won’t be able to explore the whole site. You have to e.g. fuzz parameters on each page in its different states to reach as many execution paths as possible.

This problem is addressed in [13], where a state-aware web application scanner is developed. A symbolic Mealy machine (an automaton) is used behind the scenes utilizing a state-change algorithm. The same page may behave differently when accessed in another state. Consider the homepage of some web mail service. If you’re logged in, the page shows your in-box. Otherwise a login form is presented to you. Another advantage of

state-awareness is the ability to recover from irreversible state changes due to successful fuzzing/probing for attacks [13]. Li and Xue [32] also proposed a black-box testing technique that focuses on state violation attacks.

- **vulnerability detection** [21]: Not every input that triggers a vulnerability causes a response where this faulty behaviour can be detected. Consider blind-injection vulnerabilities (SQL, command-line), although they may be detected via time-measurements as described above. Besides manual verification that is error-prone and time-consuming automation in response analysis is complex.

- **context awareness**: Scanners are often not aware of value ranges for parameters. When you use a brute-force approach to enumerate them, there is a practical limit on the variety of checks. Not every value can be tried for a given set of parameters, even when done automatically [48].

  The combination of specific values to different parameters could also be important to detect vulnerabilities. Consider a bad parameter value triggering an error, and another parameter value containing a XSS payload [48].

- **false positives**: According to Baral [6] this is a major problem for web application scanners. You need a lot of time and effort to figure out that reported vulnerabilities do not exist. It is needless to say that you cannot afford to miss a real vulnerability, but you can also not afford investing too much time in false positives.

In general, automation is hard to achieve. For a scanner to be effective, it has to understand error messages. If it is not able to interpret error messages about invalid data in submitted forms, it will not be able to detect "deep" web pages or reach authentication status [48].

Various upcoming standards may worsen the situation. More and more data is also held on the client-side. With the Web Storage [33] specification, the Indexed Database API [34] and the increasing performance of client-side JavaScript, more & more functionality is moved into the browser. As a result not only server-side code has to be probed for vulnerabilities, but also business logic on the client-side.

Every tool behaves different and might be good at detecting vulnerabilities of category X, while another one will be able to find other vulnerabilities. As a result you should never rely on just one tool, but use several. Moreover output of tools has to be interpreted. A machine cannot replace security testers with years of experience. No matter how good web application scanners will be in the future, they will never replace a security professional, because such tools do not integrate artificial intelligence [6].

In my work I focus on a specific part of web application scanners, called **intercepting proxies**.

---


Intercepting Proxy

This type of security tools often also include spiders, fuzzers and other scanners and fulfil our stated definition of web application scanners that demands automation. But intercepting proxies do more.

First, how do you use an intercepting proxy? - Your browser is set-up to tunnel all TCP/IP connections through the proxy. The proxy itself records all communication between your browser and a web application. It provides you the contents of HTTP and HTTPS conversations. The output is useful for inspection and the tool often aids further (manual) investigations. Like a network packet sniffer an intercepting proxy collects network packets and represents them to the tester. As a web application security tester isn’t primarily interested in low-level protocols, intercepting proxies focus on the application layer (OSI-layer 5-7), i.e. mainly HTTP and HTTPS. While you’re browsing a web application all requests and responses are recorded and presented in an easy-to-read manner. Intercepting proxies are not always used for security-related reasons, but for aiding the developer’s debugging actions. As a result these tools are sometimes referred as web debugging proxy or simply HTTP debugger.

In Figure 2.11 you can see how intercepting proxies are used. First you start with the reconnaissance & analysis phase, where you browse through the web application under test. The proxy collects all requests, responses and other items discovered while doing passive spidering. With active Spiders more items and sites can be discovered. The more entry points to an application, the more likely it is to find vulnerabilities. In the next phase, various tools are unleashed on the attack surface, trying to find vulnerabilities. The most important step has to be done at the end: Confirm that reported vulnerabilities really exist in the web application and fix the problem [48].

<table>
<thead>
<tr>
<th>Name</th>
<th>License</th>
<th>Platform</th>
<th>Version</th>
<th>Last Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acunetix WVS</td>
<td>commercial</td>
<td>Windows-only</td>
<td>8</td>
<td>05.02.2013</td>
</tr>
<tr>
<td>Burp Suite</td>
<td>commercial plus free-edition</td>
<td>cross-platform</td>
<td>1.5.04</td>
<td>09.01.2013</td>
</tr>
<tr>
<td>Fiddler</td>
<td>Freeware</td>
<td>Windows-only</td>
<td>2.4.2.6</td>
<td>21.01.2013</td>
</tr>
<tr>
<td>IBM Security AppScan</td>
<td>commercial</td>
<td>Windows-only</td>
<td>8.6</td>
<td>11.06.2012</td>
</tr>
<tr>
<td>OWASP Zed Attack Proxy</td>
<td>open source (Apache license)</td>
<td>cross-platform</td>
<td>2.0.0</td>
<td>30.01.2013</td>
</tr>
<tr>
<td>ParosPro Desktop Edition</td>
<td>commercial</td>
<td>Windows-only</td>
<td>1.9.12</td>
<td>28.03.2011</td>
</tr>
<tr>
<td>WebScarab NG</td>
<td>open source (GPLv2)</td>
<td>cross-platform</td>
<td>0.2.1</td>
<td>22.01.2011</td>
</tr>
</tbody>
</table>

Table 2.6: List of some intercepting proxies, created in February 2013.

---

There are many intercepting proxies in the wild. See Table 2.6 for a few examples. These intercepting proxies have in common an easy-to-use graphical user interface that specifically supports developers & functional testers new to penetration testing. This is in contrast to most of the scanners that are command line tools. Of course, intercepting proxies can also be used by experienced pentesters.

In my work I contributed to the open source project **OWASP Zed Attack Proxy**.

---

**Figure 2.11:** Typical work flow of an integrated testing suite according to Stuttard & Pinto [48].
Penetration Testing on WebSockets

In this chapter I want to answer various research questions: Chapter 3.1 takes a look at the security implications when WebSockets are used. Developers & security testers should know about new standards and their impact on existing and future applications.

Attack Vectors in the context of WebSockets are examined in Chapter 3.2. Moreover advices for minimizing or avoiding threats are given.

Finally Chapter 3.3 tries to state some methods how WebSocket-specific vulnerabilities can be found. While my extension for ZAP will support a tester’s effort on finding vulnerabilities, it is always the tester that needs to be aware of what is going on. A tool can only ease this process.

For an introduction to WebSockets and its security considerations see Chapter 2.1.

3.1 Security Implications of WebSockets

Every system is just as strong as its weakest link. This applies to security too. If a black hat is able to exploit just one vulnerability he/she may gain access to the whole system. With WebSockets there are some underlying assumptions on its security that every developer & security tester should know about:

- For the wss:// scheme only: The encryption of SSL is secure and nobody is able to read or change communication. Attacks may only be possible due to SSL misconfiguration or bad libraries. Popular attacks are known as BEAST or CRIME attack\(^1\).

Note: Due to the persistent nature of WebSocket connections, the initial overhead becomes negligible. As a result it is recommended to avoid the plain ws:// scheme and use the encrypted wss:// scheme.

• When authentication is done via session cookies or HTTP authentication the developer for that part is responsible to do this in a reasonable way, such that no WebSocket connection is allowed for illegitimate users.

• The Origin-header cannot be trusted. Although browsers set this header and do not allow altering it, a client does not have to be necessarily a browser. Consequently no data in WebSocket messages can be trusted.

• WebSocket-agnostic web application firewalls are able to scan the messages’ payload for malicious content. If you use a firewall that does not speak WebSockets, it could easily be bypassed using this new communication channel. As a result it should be aware of WebSockets and the sub-protocol in use. If both is the case for the firewall, the use of WebSockets as transport channel actually aids security, as the payload can be inspected by the firewall without special configuration.

• While browser vendors kept their early support for WebSockets up-to-date, there are a lot of server implementations that still support older draft versions that may contain vulnerabilities. Using the latest, standardized version of WebSockets is highly recommended. Moreover servers should not accept connection requests for out-dated draft versions.

Safeguard WebSocket Connectivity with the Content Security Policy

Even if your website or web application does not make use of WebSockets you are at risk. Black hats could exploit another vulnerability to inject JavaScript code that sets-up a WebSocket connection to its evil servers. This is possible because the Same-Origin-Policy does not apply for WebSocket connections. The Content Security Policy (CSP) may help out here [47].

A web application can inform modern browsers where it wants to load its resources from. This is done via the HTTP-header Content-Security-Policy. Several directives can be set to control the origin of scripts, style-sheets, images, etc. When the web application demands resources from an excluded domain, the browser refuses to load them according to the CSP header of the current web page. The CSP does not only restrict origins of resources, but also JavaScript connections to external servers. To tell the browser that it should only allow WebSocket connections to one specific domain, the following directive can be used:

```
Content-Security-Policy: default-src 'self';
connect-src echo.websocket.org
```

If the WebSocket constructor is used to establish a connection to some excluded URI, “the user agent must act as if it had received an empty HTTP 400 response” [47].

The connect-src directive does not only affect WebSockets, but also the EventSource constructor (Server-Sent Events) and the XMLHttpRequest.open() method (cross-domain Ajax calls).

Besides the fact that the proposed standard is not yet fully standardized (Candidate Recommendation for now), version 1.1 is under way to fix various design issues. Moreover browser
support is still incomplete: First, browser vendors use custom header names. Second, not every directive is supported. The following example allows your website to connect to Kaazing’s Echo Server within Chrome 24:

```
X-WebKit-CSP: default-src 'self';
   connect-src ws://echo.websocket.org
```

Firefox 18 still complains about an unknown `connect-src` directive. Mozilla has chosen a different name: `xhr-src`. For this reason, the following header would work:

```
x-content-security-policy: default-src 'self';
   xhr-src ws://echo.websocket.org
```

Besides all problems, the `Content-Security-Policy` will evolve and in future it will be supported by all browsers. Developers can start using it right now and protect users that already use modern browsers supporting CSP.

### 3.2 Attack Vectors for WebSockets

Here I want to deal with several possible attack vectors for WebSockets, look for the reasons why an attack is possible and state some countermeasures.

Vaagn Toukharian said: “From a security perspective, WebSockets don’t make applications more secure – but they do provide a new attack vector for hackers. Traditional web attacks like Cross Site Scripting (XSS) and Man-in-the-Middle (MitM) attacks can find a new home in WebSocket traffic”

#### Eavesdropping

If the communication is not encrypted, an intermediary or Man-in-the-Middle is able to read and/or change the communication. Using the `wss://` scheme in the URL-parameter of the `WebSocket(url, protocols)` constructor provides protection against eavesdropping and modification.

As already discussed in Chapter 2.1 an SSL-Stripping attack is not possible, as a standards-compliant browser won’t accept a non-secure connection when the URL used starts with the `wss://` scheme.

#### Denial-Of-Service (DOS)

When all available resources of a server are in use, no more connections can be accepted and no more messages can be processed fast. In general, no more service can be offered. In the area of WebSocket servers, exhausting resources may be easy when no server-side checks are imposed

---

regarding the number of open connections. On the browser side there is no severe restriction in this direction, easing an attack to a specific server. See Table 2.4 for the number of maximum WebSocket connections allowed in different browsers. Consequently there is potential to use up all available resources of a server by opening several connections simultaneously. One browser probably will not be able to bring down a server. In combination with a XSS-vulnerability on a popular site, a huge amount of connections may be created [45][50]. Another idea to screw up a server is to send endless amounts of small frames or one huge frame.

Another idea to bring down a server is spread by Shema [44]: If a server reserves as much space as needed immediately after reading the length value of a WebSocket frame, one could exhaust all available memory. As mentioned earlier, the maximum payload size per frame is 2GB. If an attacker pretends a huge frame size but sends only a small payload, a vulnerable server would have set aside the requested amount of memory before realizing that it is indeed not required. As you can see, this way it is easy to consume a lot of space.

Generally, the WebSocket client shall never be trusted and the server should perform several checks:

- Allow only a specific amount of open connections in total and per client.
- Restrict the number of frames a WebSocket message consists of.
- Check for the size of received frames first and dismiss their payload if too big to handle.

**Port Scanning**

Kuppan released a network scanning tool called *JS-Recon* in December 2011 [31]. It uses either WebSockets or Cross-Origin-Requests to determine if targeted ports are open. Based upon response times given in Figure 3.1 port status can be determined. *JS-Recon* is valuable especially when it comes to information gathering in a company’s Intranet, as a firewall would block such scans from the outside. When the requests originate from the inside, they are not recognized by the firewall as it listens only to passing connections.

Fortunately browsers restrict access to specific ports. If access to well-known ports is allowed, cross protocol attacks would be possible. Consider a form submission or an *XmlHttpRequest* to port 21, where an FTP-server runs. The server could be tricked into accepting a connection with a spoofed client. In Firefox the restriction is called *Port Banning* [3] while in Chrome there is a list of blocked ports called *Restricted Ports* [4]. According to Zalewski [54] Internet Explorer blocks only few & different ports than other common browsers. As a result,

---

https://developer.mozilla.org/en-US/docs/Mozilla_Port_Blocking

4 “Chrome SVN Repository - static variable kRestrictedPorts containing blocked ports”, The Chromium Projects.  
http://src.chromium.org/viewvc/chrome/trunk/src/net/base/net_util.cc
Figure 3.1: The time-to-failure is used to determine port status [31].

Port Scanning is much more effective in Microsoft’s browser. In other browsers it is not as effective as desired by black hats. JS-Recon can be used for other scans too, e.g. to scan for valid IP-addresses of hosts in the current client-side network. If run on a computer inside a company, the internal network structure can be revealed.

The reason why it was not possible beforehand is the relaxation of the Same-Origin-Policy for WebSocket connections and the introduction of Cross-Origin-Requests. With XmlHttpRequest's, only requests to resources covered by the Same-Origin-Policy are allowed. On one hand you gain more flexibility when connections are accepted across origins, on the other hand the attack surface increases and vulnerabilities are more likely.

WebSocket Botnet

Schmidt [42] describes a high level topology of a Botnet based on WebSockets. As long as enough users stay on a popular website that is hacked - there are a lot of zombies that establish a WebSocket connection to the Command & Control server. As long as the site is open in the user agent, the zombie will not disappear. The attacker controlling the server may display any content in the affected website. Moreover he may issue cross-origin requests, harming your online identity in authenticated websites.

In combination with Web Workers [25], computational tasks can be performed in the user agent’s background, without the user noticing it.

WebSocket Remote Shell

WebSockets can be used to create a remote shell during that time a malicious site is kept open in the user agent. An attacker might be able to control the behaviour of the user agent within the functionality of JavaScript [42].

As with the Botnet before, the victim has to either visit and stay on the site of an attacker or the attacker has to find some XSS-exploit on a popular website that allows him to control visitors. Heiderich et al. [22] provide an excellent overview of XSS. Although their paper deals with script-less attacks, valuable references are provided and mitigation strategies are mentioned. Moreover they deal with an emerging standard called Content Security Policy [47] that aims to reduce the attack surface for content injection vulnerabilities.

A showcase for a remote shell was created by Shekyan and Toukharian for the conference Black Hat USA 2012. Once a victim.js file is injected into a website, a persistent WebSocket
connection is set up, while the original site remains in an iframe, allowing the attacker to fetch information seamlessly or inject other scripts [45].

Masking Attacks a.k.a. Cache Poisoning Attack

The reason for masking is explained in the security section of Chapter 2.1. If the implementation of the client is non-conformant or the used masking keys are predictable, a cache-poising attack can be achieved, leading the masking mechanism ad absurdum. If you are able to control how the bytes appear on the wire, caching proxies that are not WebSocket-aware may misinterpret given bytes.

According to the WebSocket protocol specification [16] a WebSocket client must not send the first bytes of some payload when the later payload is not fixed. This means if you are able to set the latter bytes of the payload after the first bytes already got on the wire, you could read the masking key. As a result you would be able to control the bytes on the wire.

For the unpredictability of the masking key, the specification says: “the masking key MUST be derived from a strong source of entropy, and the masking key for a given frame MUST NOT make it simple for a server/proxy to predict the masking key for a subsequent frame”. Further it refers to another Request For Comments, namely RFC4086 [14] that deals with Randomness Requirements for Security.

Payload Injection Attacks

WebSocket payloads consist of user-provided input. Therefore you have to treat received messages on the server properly. Input validation prevents a set of attacks ranging from SQL injection, code injection, stored XSS attacks and further more. A proper approach is to whitelist allowed input values. Further, special characters should always be escaped, e.g. ticks in SQL parameter values.

On the server not only incoming messages should be treated carefully, but also outgoing messages. With proper output validation client-side code execution is not possible.

Sub-Protocol Attacks

Various sub-protocols may be tunnelled over a WebSocket connection. See Chapter 2.1 for some examples. Weaknesses in sub-protocols do not disappear when used in a WebSocket channel.

As mitigation strategy to sub-protocol-level attacks, developers should not surrender testing a protocols validity. WebSocket agnostic web application firewalls could also do this job for a developer.

Protocol Level Attacks

WebSocket endpoints could also be tested on the protocol level. Shema suggests several things to try [44]:

- set invalid length values
• set unused flags
• mismatch masking flags and masking keys
• replying messages
• sending out of order frames or overlapping fragments
• setting invalid UTF-8 sequences in text messages

A sub-part of protocol level attacks are attacks onto WebSocket protocol extensions. Such WebSocket extensions increase the attack surface. Consequently their specifications and implementations should also be checked thoroughly.

So far, there isn’t yet a finalized specification for an extension. Though, one might try to set reserved bits (RSV1-3) or reserved opcode values and see how the WebSocket endpoint reacts.

### 3.3 Vulnerability Exploration Techniques

With WebSockets vulnerability detection is not as easy as with HTTP. When you send a HTTP request there is always a response, where you can easily detect changed output, or slow response times. With WebSockets, communication is bi-directional and asynchronous. There is no request/response pattern, although a sub-protocol in use may define such. As a result automated detection is fairly hard and has to be done manually. However, tools can aid the search for vulnerabilities. The approach to take is similar to stored XSS attacks, where there is often no indication of a vulnerability. While the attack payload is stored in a database, it may appear on any page of the website.

If there exists a request-response pattern in the WebSocket communication, you can measure the response time after sending crafted requests. Like with Blind SQL-injection attacks, one could send values that delay the execution of database or file system queries. If the execution slowed down, a hidden vulnerability may be detected.

While the content may vary, there will always be field values that are user-provided. In order to find vulnerabilities you can enumerate such values. The payloads in use are sometimes called fuzz strings. In advance you will not know how to identify found vulnerabilities. The goal when sending crafted messages is to cause anomalous behaviour [48].

With automation you are able to test huge amounts of strings. Stuttard & Pinto [48] describe different attack payloads ranging from random values and syntactic enumerations to character blocks that probe the server for buffer overflow vulnerabilities.

Besides submitting huge amounts of values, Edge Case Testing aims at sending only few payloads that are carefully chosen. Take the Sec-WebSocket-Version-header as example. Normally it contains the value "13". Interesting values to try are non-existing values such as 14, 0 or even negative ones. The same accounts to values in the payload. The goal is to cause the business logic to behave anomalous.
WebSocket-Aware Security/Debugging Tools

I already mentioned tool support in Chapter 1.5. Besides Wireshark for low-level network inspection, there is only one web application scanner that allows to inspect WebSockets, namely Fiddler2. I took a look at several products, shown in Table 2.6.
CHAPTER 4

Implementation of WebSockets Support for an existing Web Application Scanner

“The Zed Attack Proxy (ZAP) is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications.” It is written in Java, fully internationalized and cross-platform. Most importantly it is open source under the Apache License, version 2. The core part is based on the free, but out-dated Paros Proxy. For the time this thesis is being written, ZAP is an OWASP flagship project. Consequently it receives special attention and advertising. WebSocket support for ZAP is realized as extension, although several changes to the core were required. In this chapter I want to describe the design decisions and the code structure, but first I want to define the scope of my extension.

If you would like to grasp the basic idea behind an intercepting proxy like ZAP, please take a look at the section about web application scanners in Chapter 2.4.

4.1 Scope of the WebSockets Extension

A tool for WebSockets could operate on different levels. At the time of designing my extension for ZAP there was just Fiddler2, another web debugger, with rudimentary logging support for WebSocket frame content. For development I have used Wireshark to inspect and learn about basic frame structure. With version 1.8 released on June 21, 2012 it got a WebSocket dissector to ease readability. See Chapter 1.5 for references.

In contrast to the tools above, I wanted to create a tool on a higher level. While frames are the basic building blocks, the interesting parts in the context of web applications take place on the message level. While one could build a tool to test WebSocket servers & clients on the network

---


http://www.parosproxy.org
level, I wanted to examine web applications that make use of WebSocket communication. Any user input to an application can change the way it behaves or the output it produces. Without proper input handling the application might contain vulnerabilities. A tool that allows viewing and modifying WebSocket payloads may be able to aid the search for security holes.

### 4.2 Features of the WebSockets Extension

Here I want to list the implemented features and the rationale behind them. The implementation of them is explained in detail in Chapter 4.3.

- **Message Capturing**: Messages are collected from various frames. Besides support for the `ws://`-scheme, it also works with the secure `wss://`-scheme, when you have imported the SSL certificate generated in ZAP into your browser.

- **Message Display**: Every message is shown in the WebSockets-tab. You are able to filter the messages view by channel, direction (either incoming or outgoing messages) and by message type (opcode). This allows developers and security testers to inspect communication and explore possible **Information Disclosures**.

- **Breakpoints**: Allow pausing the communication and stepping through the messages on some WebSocket channel. You can even change the payload when doing so. Breakpoints can be set in a generic way for all WebSocket messages or very specific for custom message types with specific payloads. Changing the bytes on the wire enables black box testing. Unexpected output may be observed due to modified input.

- **Filtering**: Filters are applied automatically on messages, when passed through ZAP. There is a filter allowing you to replace message payloads using a defined pattern. In contrast to setting breakpoints and changing payloads manually the communication is not paused preserving the communication flow.

- **Craft Messages**: Custom messages can be created and sent to any open WebSocket channel. This feature also allows re-sending existing messages.

- **Value Enumeration**: Sometimes also referred to as **fuzzing**. Allows sending huge amounts of messages where a specific part of the payload is enumerated. There are various fuzz collections in ZAP that can be also utilized for WebSockets.

### 4.3 Approach of Implementing the WebSockets Extension

This chapter will deal with various parts and aspects of my extension. I will start to describe some necessary changes to the core. Then I will have a look at the main part of my extension,
before explaining further details about the database tables behind. Finally I will have a look at the user interface integration & the interaction with other ZAP extensions.

I started by studying the WebSockets protocol specification (RFC6455 [16]). Additionally I read into the ZAP documentation and code. In order to ensure that my WebSockets implementation does not alter any frames, I have used a test suite called AutobahnTestsuite. This test suite contains about 300 test cases to “verify client and server implementations of The WebSocket Protocol for specification conformance and implementation robustness” [5]. See Chapter 4.3 for more details about ensuring implementation conformance.

Core Integration

Originally I wanted to build my extension upon Java’s NIO features, although ZAP is based on blocking I/O, because Java’s NIO allows for non-blocking I/O. As a result more performance could have been achieved. It worked fine for non-SSL connections, where I had to switch from a java.net.Socket instance to a java.nio.channels.SocketChannel. The problem emerged when trying to transform an instance of javax.net.ssl.SSLSocket to a java.nio.channels.SocketChannel with some instance of javax.net.ssl.SSLEngine, as this would have been the way to go with NIO. It was also not possible to address the root cause, because rewriting the core of ZAP would have consumed too much time. Consequently I went with several threads doing blocking reads on java.net.Socket instances. Every WebSocket channel consists of two threads:

- one listener on the outgoing connection from your browser to ZAP
- another listener on the incoming connection from the web server to ZAP

Another problem with the core was the ignorance of the HTTP status code 101 that appears in a successful WebSocket handshake. The solution was an upgrade of the Apache library Commons HttpClient from version 3.0 to 3.1. This small version upgrade allowed me to extend this library in a way that allowed me to access the java.net.Socket & its java.io.InputStream instance. Moreover I was able to prevent the closing of the socket in case a WebSocket handshake is successful.

Basic Extension Structure

The main entry point is the class ExtensionWebSocket, which is the starting point of my contribution. It is located in the org.zaproxy.zap.extension.websocket package. Most of the WebSocket related stuff is located there. The extension class initializes all components and hooks them into ZAP. To be independent from the core as much as possible, I make use of the Observer pattern: I introduced the interface PersistentConnectionListener with one method called onHandshakeResponse(...). The observed class that informs all observers is called ProxyThread. It notifies all

---


registered observers on each TCP connection. The first observer that returns *true* takes responsibility on the connection and its closing is prevented. If no observer takes responsibility, the connection is closed. To register an observer on the class `ProxyThread` you can use the method `ExtensionHook.addPersistentConnectionListener(PersistentConnectionListener)` in the `hook(...)` method of your extension. Possible use cases for the persistent connection listeners, where the underlying TCP connection is kept open, are *HTTP-Protocol-Upgrades* as with WebSockets, *Server-Sent Event* streams or other HTTP streaming techniques.

When a new connection is detected in `ExtensionWebSocket.onHandshakeResponse(...)` the method `ExtensionWebSocket.addWebSocketsChannel(...)` is called. It expects the `HTTPMessage` instance of the handshake, the socket for communication with the browser, the socket for communication with the server and the current `InputStream` from the server. The latest argument is of importance, as first WebSocket messages are allowed to appear in the same TCP packet after the HTTP response. As the `InputStream` instance may buffer bytes, first messages would be lost if another `InputStream` is opened on the *outgoing socket*. The WebSocket protocol does not care how its frames are split across TCP packets. This characteristic is called TCP-clean. As a result you cannot make any assumptions about the underlying packetizing mechanism of a TCP stream. You have to be able to cope with any possible.

The `ExtensionWebSocket` class creates a new instance of `WebSocketProxy` via the factory method `WebSocketProxy.create(...)` that returns a version specific `WebSocketProxy` instance. For now `WebSocketProxyV13` is the only implementation of the abstract class `WebSocketProxy`. It contains an inner class `WebSocketMessageV13` extending the abstract class `WebSocketMessage`. This should ease support for future versions. There is already some conceptual framework in the `WebSocketProxy` class. This way, external developers will be able to provide their own version-specific implementations.

Each `WebSocketProxy` instance creates two instances of `WebSocketListener`. These instances are threads listening to one of the given sockets, either for incoming or outgoing messages. If the first byte arrives, it calls `WebSocketProxy.processRead(...)` that handles the received WebSocket frame. Until processing is finished, no further WebSocket frame within the responsibility of the `WebSocketListener` is handled.

The `WebSocketProxy` class implements the *Observer*-pattern, allowing instances of `WebSocketObserver` to get notified about new frames or a change of the state of a WebSocket connection. Figure 4.1 shows the current observers and the interface of the observer. The observer with the lowest observing order is called first. Whenever a message arrives or a part of it (i.e., a frame) the `WebSocketObserver.onMessageFrame(...)` method is called. When further processing and notification should be stopped, *false* can be returned. When the internal state of the `WebSocketProxy` changes, the `WebSocketObserver.onStateChange(...)` method is called. The following states are possible:

- **Ready states defined as in the WebSockets API [24]** (numeric values are stated in parenthesis):
  - *CONNECTING* (0): The connection has not yet been established.
  - *OPEN* (1): The WebSocket connection is established and communication is possible.
• Proxy specific states indicating that channel got either black- or whitelisted. While a WebSocket is in a ready state all the time, the following two states are orthogonal:
  - EXCLUDED: Blacklisted channels are forwarded through the proxy, but not further processed nor shown in the user interface.
  - INCLUDED: Default state that is propagated only when channel was previously excluded and now got included.

Here is also a description of the observers, containing the current observing order in parenthesis. The lower the value, the earlier it is called.

• WebSocketFilterListener (0): Calls all enabled WebSocketFilter instances, allowing them to change e.g. the payload. There is a WebSocket-specific filter implementation called FilterWebSocketPayload which is added to the Filter-extension in the ExtensionWebSocket.hook(...) method. This filter allows changing the payload via a regular expression. See the section about filter integration in Chapter 4.3 for more details.

• WebSocketProxyListenerBreak (95): Halts if a breakpoint applies. When halting, no further input is read from the socket, until the current message is forwarded or dropped. This mechanism allows you to change the payload or to drop messages. When dropped, the message is not forwarded. See the section about the integration in the existing break feature of ZAP in Chapter 4.3 for more information.
• **WebSocketStorage** (100): This listener keeps track of new channels and stores finished messages in the database. The saved data is used for display in the user interface. See the section about database structure in Chapter 4.3 for more insights.

• **WebSocketPanel** (105): Triggers view updates for the channels and messages in the user interface under the WebSockets-tab. See the section about user interface integration in Chapter 4.3 to gain knowledge about the graphical user interface.

• **WebSocketPanelSender** (106): Maintains a list of connected WebSocket channels, where custom crafted messages can be sent to. See the section about the manual request extension integration in Chapter 4.3 for more details.

• **WebSocketFuzzerHandler** (110): Shows fuzzed messages in the user interface under the fuzz-tab. See the section about fuzz integration in Chapter 4.3 to learn more.

As you can see, this mechanism is a very powerful way to get informed about what is going on. In the class diagram in Figure 4.1, you can see that each instance of WebSocketProxy has got its own observerList. If you want to observe all instances you can add your WebSocketObserver instance to the ExtensionWebSocket.allChannelObservers list, either directly as in Listing 4.1 or via the hook(...) method in your Extension* class as in Listing 4.2. Each time a new WebSocketProxy instance is created, every observer from this list is added to the proxy-specific WebSocketProxy.observerList.

**Listing 4.1:** Add observer for all WebSocket channels that are initialized afterwards.

```java
ExtensionWebSocket extWs = (ExtensionWebSocket) ControlgetSingleton()
    .getExtensionLoader().getExtension(ExtensionWebSocket.NAME);
extWs.addAllChannelObserver(myWebSocketObserver);
```

**Listing 4.2:** Add observer for all WebSocket channels when starting up ZAP in your Extension-Adaptor instance.

```java
@Override
public void hook(ExtensionHook extensionHook) {
    // 'this' implements WebSocketObserver
    extensionHook.addWebSocketObserver(this);
}
```

WebSocket messages are processed in WebSocketProxy.processRead(...) as mentioned before. There are several types of messages, which are specified by a 4-bits opcode header. The assigned values are:

- **non-control frames**
  - `continuation` (0)
  - `text` (1)
  - `binary` (2)

- **control frames**
- close (8)
- ping (9)
- pong (10)

A non-control message may be split up across several frames. For this purpose a continuation-frame (0) is sent, resuming the last binary- or text-frame that was received before. In between arbitrary control frames are allowed to occur.

In order to achieve loose coupling across different components, I have introduced two Data Transfer Objects (DTO), namely WebSocketChannelDTO & WebSocketMessageDTO, shown in Figure 4.3. They contain public attributes and small helper methods. From an architectural point of view DTO’s should not contain any business logic that needs to be tested. The fuzzing part in this class diagram is explained in the section about fuzz integration of Chapter 4.3. The DTO’s can be retrieved via:

- public WebSocketMessageDTO WebSocketMessage.getDTO();
- public WebSocketChannelDTO WebSocketProxy.getDTO();
- various methods from the TableWebSocket-class, which is the interface to the database, return or demand them.

Database Structure

ZAP uses HSQLDB to store processed data. For the WebSockets extension I created three database tables (see also Figure 4.4):

- websocket_channel: Stores information about each WebSocket connection.
- websocket_message: Contains information about all messages from each channel.
- websocket_message_fuzz: If WebSocket messages are issued with the fuzz-extension, additional information is stored here.

When ZAP starts it initializes the class TableWebSocket and creates these tables if they did not exist.

Primary key values are not created via an autoincrement-feature as known from MySQL, but within the application. Instances of java.util.concurrent.atomic.AtomicInteger are used as type for attributes WebSocketProxy.channelIdGenerator, WebSocketProxy.messageIdGenerator and WebSocketFuzzableTextMessage.fuzzIdGenerator.

Fields such as websocket_channel.end_timestamp or websocket_channel.history_id are not required. The field websocket_channel.history_id is a foreign key and may link to the HTTP message of the WebSocket handshake. Fields websocket_channel.host and websocket_channel.url

---


http://hsqldb.org

50
are not the same. The first contains the result of the Java call `Socket.getInetAddress().getHostName()`, while the latter contains the requested URL of the WebSocket handshake. When connecting to the Kaazing Echo Server, the host is `echo.websocket.org`, but the url field contains `https://echo.websocket.org/?encoding=text` as this value appeared in the connection process.
HTTP-handshake-request as shown in Listing 4.3

Listing 4.3: WebSocket handshake request to Kaazing’s Echo Server

```java
GET https://echo.websocket.org/?encoding=text HTTP/1.1
Host: echo.websocket.org
Connection: keep-alive, upgrade
Sec-WebSocket-Version: 13
Origin: https://www.websocket.org
Sec-WebSocket-Key: Q9KqDUb99fp0iM8+DadDw==
Upgrade: websocket
Content-Length: 0
```


https://www.websocket.org/echo.html
Figure 4.4: Entity Relationship diagram of WebSocket specific database parts.

The table websocket_message contains two columns for payloads, namely payload_utf8 and payload_bytes. For binary-opcode messages the column payload_bytes is filled. For all other types of messages, the column payload_utf8 is set with the readable representation. This way, integration into the search-extension should be easier, as searching could be done on database level. The constraint websocket_message_payload ensures that at least one of these two columns is set. The field payload_length contains the number of bytes and may differ from the number of characters of the payload_utf8. With an upgrade of the underlying HSQLDB version 1.8.0 to 2.2.9 it was possible to take advantage of the new column types CLOB/BLOB: Only a reference to the large object’s content is returned, allowing you to retrieve only a substring, respectively only some bytes. This is used in the payload preview of the WebSockets-tab, which speeds up the user interface for bigger payloads.

User Interface

There are several areas where the WebSocket extension shows up in the user interface. This section will deal with these areas and explain the code structure behind.

The most important part appears as WebSocket-tab in the lower region of ZAP. The class behind this panel is called WebSocketPanel. It contains all the user interface elements visible there. The most important parts are (from left to right & from top to bottom):
• **scope button**: See section about integration with *Contexts & Scopes* in Chapter 4.3 for explanation.

• **channel selector**: This dropdown box stored in the `channelSelect` attribute allows filtering the messages shown by channel. It is backed by a `ComboBoxModel` instance that is based upon another `SortedListModel`. This relationship among these models is shown in Figure 4.5. There you have got the `channelsModel` attribute of the `WebSocketsPanel` that is exposed via `WebSocketPanel.getChannelsModel()`. New items are added to this instance of `ChannelSortedListModel`, excluded channels are removed from this list. This model instances can be used directly for `JList`’s or for `JComboBox`es when wrapped by an instance of `ComboBoxChannelModel`. To clear up things, look at Listing 4.4, which explains the complex relationships.

Consequently this allows placing several channel selectors, either dropdowns or lists, in the user interface that have got the same items behind. Modifications on the content must only be done to `WebSocketPanel.channelsModel`. Every change is propagated to other user interface elements automatically.

In Figure 4.5 you can also see a relationship from this main model to `WebSocketUiHelper`. This helper class is used to ease usage & aid consistency of various WebSocket-related user interface elements across pop-up dialogues. There are several independent channel
Figure 4.6: UML class diagram excerpt for messages view in the WebSockets tab.

selectors, e.g. in the ’Add custom breakpoint’-dialogue or in the filter ’Tools > Filter > Replace WebSocket payload using defined pattern’. They all take advantage from one central model. They are retrieved through a helper instance, avoiding not only duplicate code, but also providing a consistent view on user interface elements across dialogues.

Listing 4.4: Variable initializations collected from WebSocketPanel to clarify how channel selector models belong together.
1 // dropdown box in WebSockets tab
2 channelSelect = new JComboBox<>(channelSelectModel);
3
4 // model for above dropdown
5 channelSelectModel = new ComboBoxChannelModel(channelsModel);
6
7 // base model, where items are added and removed
8 channelsModel = new ChannelSortedListModel();
9
10 // method to retrieve central model is defined as:
public ChannelSortedListModel getChannelsModel() {
    return channelsModel;
}

- **handshake button**: When a channel is selected in the dropdown, this button is enabled. When clicked, it shows the `HttpMessage` from the handshake in the Request/Response-tab.

- **break button**: Allows adding some custom breakpoint, specific to WebSocket messages. See the Brk-extension integration section in Chapter [4.3](#) for more information.

- **filter button**: Opens up the `WebSocketMessagesViewFilterDialog`, which allows changing the types of messages shown in the WebSockets-tab.

- **options button**: Brings up the options dialogue defined by `OptionsWebSocketPanel`. It is backed by an instance of `OptionsParamWebSocket`, which is the interface to the saved settings. These options are available as of November 6, 2012:
  
  - **forward all**: Do only forward messages for all channels, but do not process them further. This means that WebSocket communication will not be shown in the user interface nor will it be stored to the database. You can enable it, when you are not interested in WebSockets communication but the site under test makes heavy use of it, resulting in a slow-running ZAP. **Default**: disabled, i.e. process all channels
  
  - **break on all**: Catch also WebSocket messages when breakpoints via all-request/all-response breakpoint-buttons are enabled in Break-Toolbar. This does not affect custom breakpoints set in WebSockets tab & shown in Breakpoints tab. **Default**: disabled, i.e. catch only HTTP messages on global controls
  
  - **break on ping/pong**: Catch also Ping & Pong messages when breakpoint is set for all-requests/all-response or when stepping through several messages. This does also not affect custom breakpoints where Ping & Pong are explicitly chosen. Ping & Pong messages are often used by servers to test connection health. **Default**: disabled, i.e. do not catch them, unless explicitly set to do so

- **messages view**: A `JTable` is used to display communication details. The class making use of it is called `WebSocketMessagesView`. It uses a special model instance from type `WebSocketMessagesViewModel`. This is also shown in Figure [4.6](#). Both are initialized in the `WebSocketsPanel` and held in the private attributes `messagesView` & `messagesModel`. The class of the `messagesModel` attribute is special, as it inherits from the abstract class `PagingTableModel`. This abstract model class is able to handle thousands of entries as it holds only `PagingTableModel.MAX_PAGE_SIZE` entries in memory at any given time, but the scrollbar of the `JTable` appears as it would contain all entries. When scrolling down or up to other messages, or when new messages arrive, a new page is (re-)loaded. While in load, place-holder values are shown in the rows. The loading strategy is determined by abstract methods, which have to be implemented in subclasses.

In case of `WebSocketMessagesViewModel` the database is used to query the total row count & new pages consisting of several entries. It is aware of the filters applied to the messages.
view and adjusts the query to the database accordingly. In Figure 4.6, you can see that messages can be filtered by opcode, direction & scope. The current channel selection – either all or one specific – is handled outside WebSocketMessagesViewFilter. The WebSocketPanel sets the active channel id in the model class when the channel selector changes. To lower the number of queries, the number of rows is cached for the method WebSocketMessagesViewModel.getRowCount().

As I mentioned before, the WebSocketUiHelper class is instantiated in various pop-up dialogues. Its goal is to bring in more consistency across user interface dialogues. It is used in:

- WebSocketBreakDialog: Specify custom conditions for breakpoints.
- FilterWebSocketReplaceDialog: Allows entering some payload pattern for specific WebSocket messages. When there is some matching message, the payload pattern match is replaced with another string defined by the tester.
- WebSocketMessagesViewFilterDialog: Restrict types of messages shown in the WebSockets tab.

Integration with Filter-Extension

The FilterWebSocketPayload class allows for modification of WebSocket-payloads on specific messages. It is set up in the ExtensionWebSocket.hook(...) method. It implements the WebSocketFilter interface with its only method WebSocketFilter.onWebSocketPayload(WebSocketMessage) and modifies a message’s payload if criteria are met. There is a WebSocketObserver instance called WebSocketFilterListener that calls this method for all WebSocket-specific filters.

Other WebSocket related filters can be implemented easily by just implementing the onWebSocketPayload-method that is called, whenever a message arrives. The code for adding your own filter instance is shown in Listing 4.5. The filter’s ordering can be influenced by overriding its getId() method.

Listing 4.5: Add custom WebSocket-related filter instance. The addWebSocketFilter(WebSocketFilter)-method takes care of adding the filter also to the Filter-extension.

```java
1 ExtensionWebSocket extWs = ( ExtensionWebSocket ) Control.getSingleton();
2 .getExtensionLoader().getExtension(ExtensionWebSocket.NAME);
3 if (extWs != null) {
4     WebSocketFilter f = new MyOwnWebSocketFilter();
5     extWs.addWebSocketFilter(f);
6 }
```

Integration with Brk-Extension

There are several options for the break-behaviour of WebSocket messages. See the user interface section of Chapter 4.3 for a description. These options are enforced in the WebSocketBreakpointMessageHandler class. The decision if ZAP should hold on the arrival of a specific message, i.e. if a breakpoint applies, is determined in WebSocketBreakpointMessage.match(Message,
boolean). Beforehand WebSocketProxyListenerBreak.onMessageFrame(int, WebSocketMessage) does some initial checks before passing on the power of decision. This includes the check if this operation is allowed regarding the current mode. See the integration of Contexts & Scopes section in Chapter 3.3

Integration with Fuzz-Extension

When you right click on some selected payload of a WebSockets message in the Request- or Response-tab, a fuzz-menu entry becomes available. After selecting the desired fuzz-file, the sent messages are shown in the Fuzzer-tab.

When WebSocket channels are fuzzed, the messages view shown in the Fuzzer-tab inherits from the view in the WebSockets-tab. The WebSocketFuzzMessagesViewModel also loads its entries from the database. For every fuzzed message, there is an entry in the database table websocket_message_fuzz providing more information on the fuzzed messages. Unsuccessful fuzzed messages do not pass the WebSocketStorage class, which is responsible for saving messages into database. As a result there is an extra list for failed messages in WebSocketFuzzMessagesViewModel.erroneousMessages. A reason for unsuccessful fuzzing attempts may be closed WebSocket-channels, due to endpoint errors.

WebSocketFuzzMessageDTO extends WebSocketMessageDTO and holds additional information on the fuzzing process. When an instance of WebSocketFuzzMessageDTO arrives at the WebSocketStorage class, additional information is saved to the websocket_message_fuzz-table.

You can not only retrieve a Data Transfer Object from a WebSocketMessage, but also create a WebSocketMessage from a given WebSocketMessageDTO. The given Data Transfer Object is saved as base Data Transfer Object in the WebSocketMessage.dto. When you request the Data Transfer Object from a WebSocketMessage no new instance is created, but the base Data Transfer Object is returned with updated values. This way the origin of a WebSocket message is preserved, weather it originated from a fuzzing-process or was received from either endpoint.

Integration with Manual-Request-Extension

This add-on allows sending of either custom-crafted or of existing messages to an active WebSocket channel. For the first, there is a Tools-menu entry that brings up the Manual Send WebSocket Message dialogue. The latter option appears when you right click on a message in the WebSockets-tab. It uses another instance of the dialogue class, such that the behaviour is equal to the Manual Request Editor- & Resend-feature for HTTP.

The core class of this extension is the dialogue class ManualWebSocketSendEditorDialog that extends the abstract ManualRequestEditorDialog with WebSocket-related code. It delegates the sending to an instance of WebSocketPanelSender, which implements the WebSocketObserver-interface. This allows the sender class to keep track of connected channels. When a message should be sent, it checks first if the desired channel is valid (i.e. connected) and some opcode is set. If checks are passed, the message is passed to WebSocketProxy.sendAndNotify(WebSocketMessage).
Integration with Contexts and Scope

ZAP has got a dropdown menu in the top toolbar that allows its users to select the current mode. There are three of them, called safe mode, standard mode & protected mode.

My contribution of WebSockets supports these modes. The idea behind is to support different kinds of users. In safe mode potentially dangerous operations are not possible. ZAP does not perform active scanning & fuzzing, nor does it apply filters or break on messages. It disables the trigger buttons and menu items. This is useful for developers who want to have a look at the communication of their web applications. They can inspect messages and even get alerts from passive scanning rules. But they can be sure that no request, response or WebSocket message is issued that could cause your application to crash due to XSS-detection, SQL-injection tries, etc. Not only developers benefit from this safe mode, but also pentesters that initially do not want to destroy the application under review.

In standard mode, ZAP allows the user to do anything he/she wants. For the protected mode we need to introduce the concept of Contexts & Scope. A Context is basically a set of URL patterns that is included in the current context & another set of URL patterns that is explicitly excluded. When patterns from both sets apply to a given URL, it is considered as excluded. You can define different contexts and name them for easier identification. Afterwards you can choose for every context if it is in the scope or not. For those resources out-of-scene, ZAP behaves like in safe mode. Other resources in-scene – due to matching of at least one included URL pattern of a Context in-scene – are treated by ZAP as in standard mode.

With WebSockets, you can right-click on any message and add or remove the message’s channel to some context. If the channel is in-scene, the icon of the channel gets a target symbol.

The following operations are not allowed/executed on WebSockets channels that are out-of-scene when in protected mode:

- **Filter**: Payload is not changed if pattern matches. Each WebSocketFilter instance is responsible on its own for avoiding unsafe operations.

- **Fuzzing**: Sending a modified message due to fuzzing is only allowed on channels in-scene. The fuzz-menu entry is disabled for all messages from channels out-of-scene.

- **Breaking**: No breakpoint will apply for messages from channels out-of-scene.

- **Manual Send**: Sending manually crafted messages is only allowed on channels in-scene. This applies also for the Re-Sending feature.

In safe mode none of the operations above is allowed on any WebSocket channel.

Like in other tabs, you can use the scope button – left-most button in WebSockets tab – to filter the items of the channel selector and the messages shown below. If enabled, only those channels and messages are shown that are in-scene.

The URL taken into account when checking if a channel is in-scene, is the requested URL from the handshake request. Note: In the channel selector the URL from the host is shown.
Exclude WebSockets from Proxy

My extension enables users to leave specific channels out of ZAP. By right-clicking on a message, someone can choose to "Exclude from WebSockets". Then the message’s channel does not show up in the WebSockets tab. Moreover its communication will not be stored in the database. But all WebSocket frames are forwarded, ensuring that the site under test works properly.

This can be advantageous when you are not interested in specific WebSocke channels, but their communication slows down ZAP too much, due to its huge number or big size of messages. You can reintegrate existing channels in ZAP by deleting the excluded WebSocket URL from the session properties dialogue.

If you want to exclude all WebSocket channels you can enable the option for "forwarding only" in the WebSocket-specific options dialogue. See the user interface section in Chapter 4.3 for more information about this option.

Ensuring Implementation Conformance

I have used the AutobahnTestsuite [5] to check if my WebSocket extension behaves well in any circumstance. Here I will describe the set-up of the test environment and its findings:

1. Download and installation:

   $ git clone git://github.com/tavendo/AutobahnTestSuite.git
   $ cd AutobahnTestSuite/autobahntestsuite
   $ sudo python setup.py install

2. Start testing server:

   $ wstest -m fuzzingserver

   Note: Per default, Autobahn starts a web server on port 8080. This is the same port, ZAP uses per default for intercepting traffic. You can change Autobahn’s web port by editing its configuration file shown in Listing 4.6. The configuration file is created after the first call of the wstest command, as described below. We assume that we changed the webport directive to 8081.

3. Run tests: Start your browser and visit the URL:

   http://localhost:8081/test_browser.html

   Then start the test runs once with proxy settings enabled and another time without them. There is an extension called FoxyProxy[7] that allows switching between specific proxy settings with 2 clicks. Between test runs change the User Agent Identifier form field, such that two reports are generated and none is overwritten.

---

4. **Analysis**: When finished, test results from both runs can be compared at:

http://localhost:8081/cwd/reports/clients/index.html

Results from both runs should be equal. Test cases that pass in the browser should also pass when routed through ZAP. See Table 4.1 for the test protocol. It lists test cases that are still not passed, as the browser underneath does not pass it too.

A configuration file named `fuzzingserver.json` is created in the working directory (where the `wstest` command was issued), when called the first time. The content of mine is shown in Listing 4.6. As you can see the performance tests (9.*) are kept out, as some of them are really slow. However, I included them in the test run on November 13, 2012. Four of them, namely 9.3.1, 9.3.2, 9.3.3, & 9.4.1 failed, as these tests timed out after 100 seconds. Another browser – Chrome 23 – passes all tests without any conformance problem. Unfortunately I was not able to use this browser to check ZAP’s compliance on Ubuntu Linux, because Chrome does not tunnel its WebSocket communication through the system wide proxy settings nor through the FoxyProxy extension for Chrome.

Listing 4.6: Content of configuration file `fuzzingserver.json` created on first `wstest` call.

```json
{
  "url": "ws://localhost:9001",
  "options": {"failByDrop": false},
  "outdir": ".\reports\clients",
  "webport": 8081,
  "cases": ["*"],
  "exclude-cases": ["9.*"],
  "exclude-agent-cases": {} 
}
```

<table>
<thead>
<tr>
<th>Date</th>
<th>7.3.1</th>
<th>7.3.2</th>
<th>7.9.*</th>
<th>Behaves like</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 1, 2012</td>
<td>PASSED</td>
<td>FAILED</td>
<td>FAILED</td>
<td>Firefox 12 without ZAP</td>
</tr>
<tr>
<td>July 7, 2012</td>
<td>FAILED</td>
<td>FAILED</td>
<td>FAILED</td>
<td>Firefox 13 without ZAP</td>
</tr>
<tr>
<td>July 20, 2012</td>
<td>PASSED</td>
<td>PASSED</td>
<td>partly passes, but fails 2-3, 6-8 &amp; 10-13</td>
<td>Firefox 14 without ZAP</td>
</tr>
<tr>
<td>November 13, 2012</td>
<td>PASSED</td>
<td>PASSED</td>
<td>partly passes, but fails 2-3, 6-8 &amp; 10-13</td>
<td>Firefox 16 without ZAP</td>
</tr>
</tbody>
</table>

Table 4.1: Test protocol listing problematic test cases in terms of conformance problems by my WebSocket implementation for ZAP (performance tests were skipped).

When I ran these tests the first time on May 23, 2012 I got a lot of failed tests. With the test output and Wireshark I was able to identify various problems:

- The first WebSocket frame is allowed to occur immediately after the newline of the successful HTTP handshake response. In my implementation I managed to get the `java.net.Socket` instance out of the core, when realizing that the TCP connection is for a WebSocket
channel. At the core the Apache HttpClient library is used, which reads the HTTP hand-
shake response with some instance of java.io.InputStream. In my extension I opened
another java.io.InputStream on it. Unfortunately it did not contain the first frame, as this
was already read (buffered) by the first reader. Consequently I had to retrieve also the
reader from the core library, such that I had not missed the first WebSocket frame. This
occurred when the first WebSocket frame appeared in the same TCP packet as the HTTP
handshake response.

- WebSocket frames are allowed to be split across TCP packets. An InputStream.read(byte[]) does not block until the whole buffer is filled, but returns the number of bytes retrieved. If it differs from the length of the given buffer, you have to call it again via:

  InputStream.read(buffer, bytesAlreadyRetrieved, buffer.length - bytesAlreadyRetrieved)

- WebSocket frames may contain invalid UTF-8 payload values. I had to catch those UTF-8 errors and forward these frames, although they were invalid. This behaviour is in contrast to RFC6455 [16], where either endpoint has to close immediately when invalid values are read. Since I want to test these endpoints, sticking to the specification would not make much sense in this case.

- WebSocket frames may consist of thousands of frames resulting in a message of several Megabytes. To improve performance I could forward each frame as soon as it is read. Filters and breakpoints can only be applied on full messages. As a result, immediate forwarding of frames is only possible for excluded channels, where nothing has to be further processed. Due to this performance problem, I added a “forward all” option & the exclude-feature for WebSocket channels. When enabled for a channel, waiting until the whole message is collected is avoided, improving performance a lot.

- WebSocket close-frames may contain a close code consisting only of one byte, instead of two required bytes. When I encounter such invalid close code, I ignore that and forward the frame regardless of validity.

- After an unfinished text-frame, another non-control frame might be sent that is not the awaited continuation-frame. As a result I rewrote the WebSocketProxy.processRead(...) method to avoid failing, but forwarding the invalid frame. The receiving endpoint has to close the connection immediately according to the WebSocket protocol specification [16].

- Furthermore I was able to improved logging and the shutdown procedure of a WebSocket-
Proxy instance in ZAP as soon as one side fails.

In conclusion I can say that you should never make any assumptions about the underlying TCP packet structure. A WebSocket frame may be split up into several TCP packets or several WebSocket frames may appear in one TCP packet. Finally I was able to fix all of the problems listed above until the first documented run on June 1, 2012 as shown in the test protocol in Table 4.1.
 Evaluation of WebSocket Usage in Android Apps

Shema, Shekyan & Toukharian [45] examined WebSocket usage in the Alexa Top 600K websites. In their study they have looked only at the landing page. I would not expect heavy use of WebSockets on the main page of a website, with the exception of one-page-websites. Instead the characteristics of WebSockets make it an ideal candidate for mobile applications. I have already discussed use cases of WebSockets on mobile devices in Chapter 2.1. Consequently I examined WebSocket usages in the top free apps of the Google Play Store ¹ for devices running the Android operating system. I downloaded over 15,000 apps and ran various commands on them in order to detect WebSocket usages in non-obfuscated apps in an automated manner. Be aware of the limitations of my analysis approach described in Chapter 5.2. The approach itself is described in Chapter 5.1.

5.1 Approach & Analysis

I wrote a Java program based on the Google Play Crawler JAVA API ². It allowed me to gain a list of 34 category names of the Google Play Store, each belonging to one of two divisions. The Games-division has got 8, while the Application-division contains 26 categories. For each category a list of the top 500 freely available apps was created. This amounts for a total of 17,000 apps. After creating a list on January 15, 2013, the API was also used to fetch the APK-files. 1204 apps out of total 17,000 appeared two or even three times in different categories. As a result there were only 15,710 apps left for download. From these 15,710 apps, the download

failed for 646. In most cases, the API in use reported that this app is not available in my country. Consequently I based my evaluation on the top 15,064 freely available apps from the Google Play Store.

An Android app is basically a ZIP file with some basic folder/file structure. It uses the file ending *.apk, which stands for application package file. Java is the predominant language used to program Android apps. However, the resulting code archive (*.jar file) is converted into the DEX file format that is run on the Dalvik virtual machine. With their own byte-code and virtual machine, Google was able to achieve better performance and lower resource consumption on mobile devices. Besides Java developers are also able to use native languages and compile that code into portable libraries (*.so files).

In order to detect usage of WebSockets, one could start an application and try out every feature. Meanwhile network traffic is recorded e.g. via tcpdump, Wireshark or ZAP. Afterwards you can analyse the traffic and search for packet occurrences of the WebSockets protocol. The difficulty of this approach is obvious: it is time-consuming. Various procedures make testing cumbersome: Apps...

- ...may require registration
- ...may require specific hardware features on the phone
- ...are built with different guidelines in mind and behave differently
- ...may hide their features behind edge use cases that are not easy to discover
- ...may use encryption (HTTPS), which hinders analysis on a network dump

As a result I employed another method. I searched the contents of the APK-file for occurrences of specific strings, indicating WebSocket usage. The search was applied case insensitive with the keywords: websocket, ws:// & wss://. The keywords may occur in asset files, such as HTML & JavaScript, or in the classes.dex (byte-code compiled from Java). This approach, as easy and time-effective as it might be, has also got some caveats - see Chapter 5.2 for limitations. See Listing 5.1 to get an impression about the script, which was applied to all apps in an automated manner. The excerpt shown, searches the contents of the APK-file. If possible it does not grep on the APK (ZIP) file, but on its decoded resources. Decoding is done using the android apktool resulting in more accurate matches.

Listing 5.1: Centrepiece of automated analysis script checking the APK-file for WebSocket usages.

```bash
#!/bin/bash
# script expects APK-filename as parameter
apk=$1
workspace="/home/robert/test_$1/
```

---


grep_words="websocket\|ws://\|wss://"
# grep -i => case insensitive
# grep --binary-files=text => print matches of binary files as text
# grep --color=never => coloring hinders subsequent matching
# grep -R => recursive search, when applied on directory
grep_options="-i--binary-files=text--color=never-R"
grep_lines=""
grep_count=0

# 1. decode resources to $workspace
java -jar ~/Software/apktool-install-linux-r05-ibot/apktool.jar decode \
   --force --keep-broken-res $apk $workspace > /dev/null 2>&1
apkToolSucceeded=$?

# 2. search in contents
if [ $apkToolSucceeded = 0 ]; then
   # grep on $workspace
   grep_line=$(grep $grep_options "$grep_words" $workspace)
fi

if [ ! $apkToolSucceeded = 0 ]; then
   # grep on APK file
   # unzip -p => output to stdout
   # unzip -aa => force extraction as text file
   grep_line=`unzip -p -aa $apk | grep $grep_options "$grep_words"`
fi

# 3. exclude certain results (known false positives):
# grep -v => returns only lines that do not match
# a) lines with "window.Modernizr"
grep_line2=`echo "$grep_line" | grep -v $grep_options \
   "window.Modernizr\|Build:http://www.modernizr.com/download/"`
if [ ! ${#grep_line} = ${#grep_line2} ]; then
   echo "contains Modernizr";
   grep_line=$grep_line2
fi

# b) lines with "views://" or "news://"
grep_line2=`echo "$grep_line" | grep -v $grep_options "views://\|news://"`
if [ ! ${#grep_line} = ${#grep_line2} ]; then
   echo "contains views:// or news://";
   grep_line=$grep_line2
fi

# c) lines with "jboss/netty/"-library
grep_line2=`echo "$grep_line" | grep -v $grep_options \
   "jboss/netty/handler/codec/http/websocket/"`
Due to obfuscation described in the limitations section, the evaluation is only able to state a clear result for non-obfuscated apps. There might be more usages of WebSockets so far than reported. However, the reported usages are valid as they were verified manually.

5.2 Limitations on Analysis Approach

My evaluation is limited in several ways:

- First, the application selection is biased towards popularity.
- Second, only freely available apps are investigated.
- Third, the Play store from Google is not the only app store. Consequently the apps under investigation might have been not the most popular ones at the evaluation time. Moreover, download rates in Figure 5.2 refer only to Google’s app store. In reality the installation count is expected to be higher.
- Fourth, if no usage is reported it does not mean that there is not any. As a result false negatives may occur. Reasons for false negatives are discussed below.

Obfuscation is a technique to hide implementation details. It makes it harder to decompile and interpret application code. There are various tools that support such techniques, two of them are: ProGuard is a freely available tool that is already built into the Android toolkit. DexGuard is another commercial tool from the same developer that can be used to hide implementation details with more advanced features such as string encryption. While the first tool does not hinder search for the WebSocket URL with a ws(s):// schema, the second tool with string encryption makes it impossible to gain insight with this simple technique. To get an impression about the number of obfuscated apps, I conducted another evaluation on the overall top

---

480 apps (across categories). From these 480 apps, 41% definitely use obfuscation techniques. I used the same heuristic taken by Enck et al. [15] to check for obfuscation. If an app contains a file called `a.class` in the main package (e.g.: `com/foo/appname/a.class`), it is considered as obfuscated. Additionally I also consider an app as obfuscated when `a.class` appears in the root directory. Obfuscation does not mean that the keywords are not found. Often only class & variable names are renamed, but strings are not obfuscated. When string encryption is employed, then you would have to invest more time to find out what is going on. How many of the obfuscated apps use string encryption was not evaluated and is left for future work.

If one could determine automatically if an app uses string obfuscation techniques, the script `d2j-decrypt-string` may help as described by lohan+ in a blog post.\(^6\)

Another obstacle for static analysis can be shared libraries (`*.so` files). Native languages, such as C or C++ may be used as implementation language for critical parts and its code is then provided as library file. Decompiling such libraries may require more effort and was not done.

To avoid static analysis completely, an app could also hide its business logic by downloading important assets at the time of the first application start up. Such behaviour can only be detected on dynamic analysis where the app is executed.

### 5.3 Manual Verification

My analysis script reported over 150 apps that may make use of WebSockets. The majority turned out to be false positives. Over 30 apps included JavaScript library such as ExtJS, Backbone.js, Modernizr, etc. that provide polyfills as described in Chapter 2.1 or support extensions based upon WebSockets. Another 25 apps included the Java Netty library, which is an asynchronous event-driven network application framework. This library includes support for WebSockets, but the relevant package was not used by the apps under investigation. 12 apps included strings that referred to another protocol such as `news://` or `views://`. The keywords in use did not exclude such matches. 10 apps were mobile browsers that do not establish a WebSocket connection on their own and the remaining false positives account to other libraries, where WebSocket-related code was detected but not accessed.

After an initial identification of false positives, 32 apps remained to be looked at. Their analysis report revealed interesting matches. Consequently they were installed on an Android Emulator and their traffic was captured & analysed with ZAP. For older WebSocket versions, inspection with ZAP is not possible. Support for older draft-versions was left for future work. Instead, I used Wireshark to inspect payloads of non-encrypted WebSocket connections. Additionally the app’s DEX code archive was transformed back into a `*.jar` file with a script

---
called `dex2jar`. The command $ d2j-dex2jar.sh classes.dex results in a file called classes-dex2jar.jar. `JD-GUI` can be fed with that. It decompiles all *.class files and presents the sources of the *.jar file immediately. This way more detailed analysis was possible.

WebSocket usage was verified in 14 apps. See the result section in Chapter 5.4 for more details. Another 7 apps contained WebSocket-related code that may be used in the future.

**Setup for Android**

In order to allow ZAP to capture the network traffic from the Android Emulator, the following procedures have to be done:

*Note: The dollar sign ($) indicates input for the command line and must not be entered.*

1. Get ZAP’s certificate file. It needs to be installed onto the emulator in order to allow HTTPS traffic flowing through ZAP and appear as valid communication on the emulator.
   - a) open ZAP
   - b) go to **Tools > Options > Dynamic SSL Certificates**
   - c) **optional:** generate a SSL certificate if you have not created one before
   - d) click onto **Save** and temporarily keep the saved `zap.cer` file

2. Create a virtual SD-card containing the certificate of ZAP:
   - a) open a terminal
   - b) go to your Android SDK installation
   - c) **enter** $ tools/mksdcard 32M ws_sdcard
   - d) **enter** $ sudo mount -o loop,uid=1000 ws_sdcard /media/sdcard
   - e) copy certificate retrieved in the previous procedure onto sdcard via: $ cp zap.cer /media/sdcard/
   - f) **Alternatively** you can also use `adb` to transfer the certificate file onto your emulator, when it is already running. In this case create the emulator and start it first. Then transfer the certificate file onto it as described below:
     i. open a terminal and go to Android SDK directory
     ii. run $ platform-tools/adb remount
     iii. run $ platform-tools/adb push zap.cer /sdcard/

3. Create a new emulator:

---


a) use the terminal and run $ tools/android from your Android SDK directory
b) go to Tools > Manage-AVD
c) click onto New... to create a new Android Virtual Device (AVD) – choose whatever version and device you like – I have made good experience with version 4.2
d) enter "ws_avd" as AVD name
e) as SD-Card choose the ws_sdcard file created in the previous procedure
f) click okay and close the tool

4. Start the emulator and install certificate:

a) use the terminal to start the emulator via $ tools/emulator -avd ws_avd -http-proxy http://localhost:8080
b) after booting, go to Menu > Settings > Security > Install Certificate from SD card (this is for Android 4.x devices) – the previous copied *.cer file is detected and installed – before you may have to setup a pattern, pin or password for your emulator device
c) while browsing HTTP sites might work immediately through the proxy, there might be an error for HTTPS sites - follow this procedure to fix:
   • go to Menu > Settings > (Mobile Networks) More... > Mobile Networks > Access Point Names
   • choose the only item that appears
   • enter 10.0.2.2 for the Proxy setting
   • enter 8080 for the Port setting
d) test if HTTPS is now working by opening ZAP in the background and visit e.g. https://google.com in the Android browser

For installing an APK-file called myapp.apk on the virtual Android device, do the following steps:

1. open a terminal and go to Android SDK directory
2. run $ platform-tools/adb remount
3. run $ platform-tools/adb install <path-to-app>/myapp.apk
4. app will appear in the phone’s menu if installation succeeded

For some apps installation may fail due to missing shared libraries or they may crash when starting them on the emulator. In this case, a real phone may help. In the following I describe one way how communication can be intercepted by ZAP when running apps on a real phone. It requires your device to be rooted, as it depends upon installation of an app called SSHTunnel⁹

The computer where ZAP is running needs to have an SSH server installed that must be available from the internet. For the configuration part, I will refer to this computer as http://my-zap-computer.org. The idea is to set-up a SSH tunnel, where all traffic flows through. At the same time, the traffic is redirected through the port where ZAP is running on. This way you can carry your phone with you, while ZAP records all traffic.

1. on the phone (running Android 4.x) go to Menu > Settings > More Settings -> Mobile networks -> Access Point Names

2. select the active entry and change the following settings:
   - Proxy: 127.0.0.1
   - Port: 8080

3. install SSHTunnel from the Google Play Store

4. run it and configure it as follows:
   - Host: http://my-zap-computer.org
   - Username & Password that is able to connect via SSH
   - Port Forwarding: off
   - Local Port: 8080
   - Remote Address: 127.0.0.1
   - Remote Port: 8080
   - Global Proxy: on

5. turn the tunnel on via the first configuration option called Tunnel Switch

6. if working, the traffic of your browser should appear in ZAP

5.4 Evaluation Result

From the top 15,064 apps, 14 make use of WebSockets. This amounts for a total of about 0.093%. I want to mention again that there might be some false negatives due to the caveats mentioned in the limitations section of Chapter 5.2.

The true positives occur across several categories. See Figure 5.1 to see the distribution. The Social category outshines all other categories, as WebSockets are very popular for chat applications. However, due to the low number of true positives, no clear statement can be derived.

The Google Play Store is divided into two major app divisions: Application & Game. Interestingly all true positives with one exception belong to the application group. I would have expected more usage on apps in the games division, because the characteristics of WebSockets make them an ideal candidate for interactive games.

I have also evaluated how popular these apps are. See Figure 5.2 to get an impression about download numbers of found apps. There is one app, namely Airdroid that excels the rest in
download numbers. The majority of usages appear for apps within the download interval of 100,000 and 1,000,000. No conclusion can be derived from that distribution, as there were also two usages below an installation rate of 5000 downloads. The new technology is used across different apps, regardless of popularity.

Besides category assignment and popularity I investigated which version of the WebSocket protocol was in use. In Figure 5.3 you can see that less than 50% of true positives use the latest protocol version that has become the standard in RFC6455 [16]. While version 8 is similar to version 13, the use of old versions entitled as draft-3/draft-0 or earlier is really alarming. About one third of the apps make use of these old specifications. Here, the cache poisoning attack described by Huang et al. [26] may work and affect the systems security. Moreover, further security flaws, fixed until the finalization of the protocol, may be exploitable.

Last but not least I took a look at encryption. In Figure 5.4 you can see that only 2 out of 14 apps, use the encrypted wss://-schema. All others may suffer from Man-in-the-Middle attacks.

With the soon ending standardization process of the WebSockets API for the browsers and the adoption of the low-latency LTE technology by mobile network operators, I would expect more usages. Companies may not use WebSockets in web applications as its users need to have a modern browser supporting WebSockets, whereas in mobile applications you are able to deliver the browser engine or library within your application. Thus usage of WebSockets is easier in mobile applications aside from old browser versions on desktop computers or laptops.

Details about Apps using WebSockets

In this section I want to describe the apps that definitely make use of WebSockets. I start with the most popular:

- Airdroid
Figure 5.2: Download numbers of apps using WebSockets in the Google Play Store.

Figure 5.3: Apps make use of different WebSocket versions.

Figure 5.4: Distribution of the plain ws://- & the encrypted wss://-schema.
When you are in the same WLAN as your phone, then you can access the phone’s functions & storage via your browser. Once you have logged onto your phone, a WebSocket connection is set up from your browser to your phone. The communication channel is used to keep track of the phone’s battery status, its number of unread SMS and missed calls.

- **GroupMe**
  - package name: com.groupme.android
  - category: social
  - downloads: 1,000,000+
  - WebSocket version: 13
  - WebSocket endpoint: https://push.groupme.com/faye

With GroupMe you can chat with other people or groups of people using the same app. The WebSocket connection is used to handle the group subscriptions and the chat messages. The data format in use is JSON.

- **Grooveshark Remote**
  - package name: uk.co.awesomedigital.gsremote
  - category: music and audio
  - downloads: 500,000+
  - WebSocket version: 8
  - WebSocket endpoint: http://io.gsremote.com:3000/socket.io/1/websocket/<some-id>

This app lets you control your Grooveshark music player on your laptop or desktop computer. The data format in use is determined by Socket.IO.

- **Jokes To Offend Everyone**
  - package name: com.jokestooffend
  - category: social
  - downloads: 500,000+
  - WebSocket version: draft-3 or earlier
This app lets you share jokes with others. It uses the WebSockets connection to load jokes. The data format is determined by Socket.IO.

- **Schoener Fernsehen**
  - package name: com.onlinetvrecorder.SchoenerFernsehen2
  - category: media and video
  - downloads: 500,000+
  - WebSocket version: 8
  - WebSocket endpoint: http://static.peer-stream.com:8080/socket.io/1/websocket/<some-id>

You can watch TV with this app. Moreover it includes a chat functionality that uses WebSockets to discuss the current broadcast with other users.

- **Twoo - Meet new people**
  - package name: com.twoo
  - category: social
  - downloads: 500,000+
  - WebSocket version: 13
  - WebSocket endpoint: http://77.73.177.243:80/poll

This is a location-aware app that makes it easy to get to know new people. It includes a chat functionality that uses WebSockets with a JSON data format.

- **Couple, formerly Pair**
  - package name: com.tenthbit.juliet
  - category: social
  - downloads: 100,000+
  - WebSocket version: draft-3 or earlier
  - WebSocket endpoint: https://api-wss.tenthbit.com/socket.io/1/websocket/<some-id>

Couple makes it easy to stay in contact with your partner. There is also a chat functionality inside the app. The data format in use is JSON.

- **Crazy Eights Online**
  - package name: de.same.krautmaumau.android
Features the classic card game also known as "Mau Mau" for 2 to 5 players. It uses a WebSockets connection to feed the game engine. The data format in use is JSON.

- **droid VNC server**
  - package name: org.onaips.vnc
  - category: communication
  - downloads: 100,000+
  - WebSocket version: 13
  - WebSocket endpoint: http://<your-phone-ip>:5901/websockify

  This app is a VNC server that allows using a browser as VNC-client. The data sent is base64-encoded and sent within text-frames.

- **Livestream for Producers**
  - package name: com.livestream.livestream
  - category: media and video
  - downloads: 50,000+
  - WebSocket version: draft-3 or earlier
  - WebSocket endpoint: http://sio-4.sio.new.livestream.com/socket.io/1/websocket/<some-id>

  Livestream is a live blogging service that uses WebSockets to retrieve comments, post assets and more. The data format is determined by Socket.IO.

- **Sport.pl LIVE**
  - package name: pl.sport.live
  - category: sports
  - downloads: 10,000+
  - WebSocket version: 13
  - WebSocket endpoint: http://sportpllive.app.gazeta.pl/socket.io/1/websocket/<some-id>

  Keeps you informed about sport events, such as soccer games, basketball, boxing and many more. It uses a WebSocket connection to push the information to your phone. Socket.IO determines how the data looks like.
• **Loxone**
  - **package name**: com.loxone.app
  - **category**: lifestyle
  - **downloads**: 5000+
  - **WebSocket version**: draft-0 or earlier
  - **WebSocket endpoint**: http://ukdemominiserver.loxone.co.uk:7778/

With this app you can control your own loxone server for your smart home. It lets you control lighting, heating, burglar alarm, etc. The payload of the connection is not human-readable. Of course, to control your own smart home, you must not use the address of the demo server, but your own.

• **Blockchain**
  - **package name**: piuk.blockchain
  - **category**: business
  - **downloads**: 1000+
  - **WebSocket version**: 13
  - **WebSocket endpoint**: http://api.blockchain.info/inv

Blockchain is a Bitcoin wallet, where you can buy, send & control your Bitcoins. The payload is mixed. Some messages are readable JSON; others are cryptic text-frames.

• **4XPhone**
  - **package name**: air.com.pandats.A4xp
  - **category**: finance
  - **downloads**: 1000+
  - **WebSocket version**: 8
  - **WebSocket endpoint**: http://tradingtools.dealserv.com:81/Quotes/cometd

This app shows stock quotes. It is able to retrieve them via WebSockets. At start-up it requests a config.xml file, which contains a switch for WebSockets `<WebSockets-Protocol>`. Per default it is off, but when its value is changed within ZAP, a WebSocket connection is set up, transferring stock quotes.
Shema, Shekyan & Toukharian [45] examined security of WebSockets and scanned the Alexa Top 600K websites for WebSocket usages. 900 websites were using WebSockets on their landing page, which amounts for 0.15%. Leaving out a single vendor’s customer support chat system, 45 usages remain. Among these, only 9 use encryption for their connection. Within their presentation at Black Hat USA, they presented a tool called Waldo\(^1\) which is a simple tool to demonstrate how easy it is to use WebSockets. In my opinion the expressiveness of this scan is very limited. Use cases for WebSockets would expect us to see them mainly in tools & games off the landing page.

Ruottu and Markus created a proof-of-concept called BrowserSocket\(^2\) that utilized WebSockets for browser-to-browser communication. They have developed a Firefox plug-in that created a socket server. It was available for Firefox version 3.6 to 4. Another browser capable of WebSockets was able to connect to the previously created BrowserSocket. This is a very interesting project that featured peer-to-peer communication within browsers. Unfortunately it is no longer deployable, but its code-base is available on GitHub.

Several guys at LearnBoost did an evaluation on “Socket.IO and firewall software” [46]. They have investigated 16 personal firewalls (a.k.a. internet security tools) to find out how they treat WebSocket connections. 3 of them blocked WebSocket connections. For the remaining firewalls there was always a fall-back port available. Interestingly, some browsers were blocked even on port 80. They concluded that a move of the WebSocket traffic to port 443 fixed all connection problems.


WebSockets emerged as low-overhead alternative to complex workarounds. In former days Ajax appeared that subsequently boosted the web as application platform. Further techniques for satisfying the demand for real-time communication popped up to extend these use cases such as Polling, Long Polling & Streaming. There are some papers dealing with a performance comparison to WebSockets:

Pimentel & Nickerson [39] examined usage of WebSockets to retrieve wind sensor data in a continuous manner. They measured delay times to retrieve data from different areas in the world and compared them to other underlying technologies, namely Polling and Long Polling. Due to the periodic delivery of measurements, Long Polling performed as well as WebSockets, except the case where the Round-Trip-Time was greater than the periodic report interval. In my opinion WebSockets unfold their whole power and outshine competing technologies when used for infrequent two-way communication. Both are not the case for this weather station.

Argawal [1] investigated net-throughputs of HTML sockets, namely WebSockets and XHR-polling, and compared these to raw TCP sockets in use by native applications: When transmitting 256-byte data chunks from the client to server there was a 1.16x overhead for WebSockets and 5x overhead for XHR-polling compared to raw TCP sockets. Tests were conducted on a traffic-shaped network and a 3G cellular data network by sending chunks of up to 50 kilobytes. For the mobile phone network, he concluded that 90% of all data chunks arrive at the client within an inter-chunk delay of 760ms for WebSockets, 1382ms for XHR-polling, but only 360ms for raw TCP. I think that analysis of smaller payloads, which are less than 1 kilobyte would be more interesting, because for bigger payloads the overhead is comparable low. The overhead of the WebSockets protocol version 13 (RFC6455) for 256-byte payloads is 8 bytes from client to server (due to the masking requirement) and 4 byte from server to client.

Back in 2003, the XMPP community started to work on the BOSH-standard, which stands for Bidirectional-streams Over Synchronous HTTP [38]. BOSH is a transport protocol leveraging a “long-lived, bidirectional TCP connection between two entities (such as a client and a server) by efficiently using multiple synchronous HTTP request/response pairs without requiring the use of frequent polling or chunked responses”. If WebSockets were not supported, BOSH may be a considerable option for bidirectional communication. However, its complexity and comparable high overhead discourage its use.

If WebSockets are not available, techniques like Ajax, Comet or HTTP Streaming proofed to work. Bozdag et al. [11] did a performance evaluation upon these techniques and compared push & pull approaches for real time data delivery. Their result indicated better network performance with push approaches.
Future Work

For the implementation specific part, there are various possible enhancements:

- **Support for upcoming WebSocket protocol extensions:**
  Existing drafts for extensions are discussed in the WebSocket protocol section of Chapter 2.1. As of November 2012, none of those drafts has reached a final state. Moreover there is no demo known to me that makes use of such extension.

- **Integration with ZAP’s API**:  
  WebSocket communication could be exposed via the REST API that is accessible via http://zap/ in your browser.

- **Integration with the Script Console of ZAP**:  
  It allows accessing internal data structures via various scripting languages, including JavaScript, Python, Ruby and many more.

- **Integration with the search extension**:  
  Via searching you are able to find usages of specific patterns in HTTP communication. Including WebSocket messages in the search scope would be another useful enhancement.

- **Support of older draft versions**:  
  My evaluation has shown that there are many old WebSocket versions in use. Supporting them in ZAP would make testing & inspection easier.

---


The number of WebSocket endpoints, i.e. servers and clients, will rise in the near future due to the standardization progress. While my extension for the penetration testing tool focuses on the payload content, I left an evaluation on the robustness of WebSocket endpoints for future work. The focus could be on the protocol itself and how these endpoints deal with errors, how vulnerable they are to Denial-of-Service attacks and if there is support for older draft-versions with possible vulnerabilities.

In the field of mobile networks, an evaluation of WebSockets in Long-Term Evolution networks can be conducted. On the one hand, performance regarding latency (Round-Trip-Times) and throughput is interesting. Especially the real-time capabilities via the web should be evaluated. On the other hand, stability and battery consumption are worth to be looked at.
Conclusion

With WebSockets a new communication paradigm for the web has arrived. It has got no request-response pattern known from HTTP, but offers an asynchronous & bi-directional channel. WebSockets bring real-time capabilities to the web. The upcoming standard enables developers to realize more powerful use cases, but they also pose risks. As API’s grow, the attack surface also increases. In this thesis, I identified possible attack vectors for WebSockets. Proper handling of WebSockets can hinder some attacks or even eliminate them. Developers and security testers need to know shortcomings and security implications of this standard.

Even network administrators should be aware of WebSockets. Current firewall products, also specialized web application firewalls, are not WebSocket-agnostic. As a result, data within this new communication channel may by-pass security-controls. Moreover Cross-Site-Scripting (XSS) and other injection attacks find a new home in WebSocket messages. Web developers must not trust the client and are required to do proper input handling. Programmers writing WebSocket servers need to create robust software too that is resilient against Denial-of-Service (DOS) attacks and that does not crash on invalid protocol usage.

Tools can support developers to create secure applications and security testers to verify correct behaviour. As part of this work I have extended the open source intercepting proxy OWASP ZAP with support for WebSockets. Such proxy is put as Man-in-the-Middle between your browser and the web application under test. It allows inspecting communication and changing of payloads. My extension can even fuzz parts of WebSocket messages. For now it is the only known penetration testing tool that has got support for WebSockets.

Finally I conducted an evaluation on over 15,000 Android apps. The goal was to find out how widespread WebSockets are on mobile devices. 14 freely available top apps establish a WebSocket connection. An older protocol version is used in 8 apps. Encryption is employed only in 2 cases. Besides low usage, these facts are alarming – developers are strongly recommended to upgrade their WebSocket protocol version and make use of encryption.
Bibliography


