The WebSocket Protocol and Security: Best Practices and Worst Weaknesses

University of Oulu
Department of Information Processing Science
Master's Thesis
Juuso Karlström
14.12.2015

Abstract

Modern web applications need reliable communication between the servers and the clients in order to access information from databases or to insert user defined input into the applications. Even today, when the web sites are something completely different from what they were originally designed to be, they still rely on the original protocols. These protocols, e.g. HTML, have been updated a few times. The transition from HTML 4.1. to HTML5 introduced many new features and techniques, such as the WebSocket protocol.

Auditing different protocols from the security perspective is one of the key methods for enhancing the reliability of the protocols under testing. The results provided by the testing often reveal vulnerabilities or at the very least suggestions for future development. These results are then assigned to the developers or the community and hopefully these issues are then addressed.

In this thesis *Design Science Research Methodology* was used to research the WebSocket protocol and also a few commonly used server implementations for this protocol. Moreover, statistics on how widely WebSockets are used in web applications was also looked into.

The research showed that the protocol in itself has dealt with the security aspect and that the protocol specification states clearly on how the protocol should work when applied according to the documentation. However, as there is a delicate balance between usability and security, the scale has favoured usability over security on a number of occasions by reducing the safety of the protocol to some degree.

Acknowledgements

First of all, I would like to thank Professor Juha Röning for his thorough commentary and feedback on the reviews during our meetings.

I would also like to express my gratitude to Jouni Lappalainen, my supervisor in the Department of Information Processing Science, for his support guidance throughout the work. In addition, our discussions motivated me greatly and kept me going.

Christian Wieser and all the others at the Oulu University Secure Programming Group (OUSPG) also deserve a big thank you for their guidance, support and expertise during the writing of this thesis, thus making it possible.

Last but not least, a special acknowledgement needs to be made to the people close to me: Sanna, thank you for motivating me in times of doubt and giving me all the moral support that I needed; Joona and Samuel, my two sons, thank you for your delightful distractions.

Abbreviations

API	*Application Programming Interface.* API expresses a component in software which defines functionalities, operations, inputs, and underlying types for building software applications.
CPU	*Central Processing Unit*, processor in computers, which performs calculations, basic Input/Output operations, and control operations specified by instructions.
DCI	*Deep Content Inspection,* form of network filtering where packets are inspected for malware, viruses and such when they pass an inspection point.
(D)DOS	*Denial-of-Service* or *Distributed-Denial-of-Service,* an attempt to make an internet resource unreachable for its intended users, normally with so large an amount of traffic that the services provided by the host are suspended. DDOS uses several locations for attacking the host.
HTML	*Hyper Text Markup Language,* a set of markups and code used for creating displaying a World Wide Web browser page.
HTTP	*Hypertext Transfer Protocol*, used as a data communication method in World Wide Web.
TCP	*Transmission Control Protocol*, one of the core protocols of Internet protocol suite. It provides a reliable and ordered stream of octets between applications.
TLS	*Transport Layer Security,* a protocol for encrypting and securing Internet communication.
XSS	*Cross-site scripting,* a vulnerability found in web applications that allows an attacker to inject harmful code to web applications.
URL	*Uniform Resource Locator,* an address that points to a resource on the Internet.

Contents

1. Introduction

Bruce Schneier, a well-known computer security and cryptography expert, describes security consisting of two different aspects. This is how he defines security in the context of an infant abduction from a hospital, and the countermeasure that hospitals use, viz. the use of RFID tags on a bracelet placed on a baby's ankle.

> Security is both a reality and a feeling. The reality of security is mathematical, based on the probability of different risks and the effectiveness of different countermeasures. We know the infant abduction rates and how well the bracelets reduce those rates. We also know the cost of the bracelets, and can thus calculate whether they're a cost-effective security measure or not. But security is also a feeling, based on individual psychological reactions to both the risks and the countermeasures. And the two things are different: You can be secure even though you don't feel secure, and you can feel secure even though you're not really secure (Schneier, 2007).

Information security and cyber security are notions that are challenging these days. On the other hand, we are investing more and more money to protect ourselves against threats that are nearly non-existent, and there are many things that need much more attention and funding than they are receiving now (Schneier, 2008).

As the world continues to enjoy the services of different web applications in an ever-increasing manner, there arise new security challenges. HTML5, which is the newest version of the HTML-protocol (*HyperText Markup Language)*, is a standard for creating and presenting web sites. The updated HTML provides a platform for fast, responsive, and correspondent applications that are expected in the modern web application development. HTML5 has introduced new features for providing a platform for the new generation of web applications that serve clients with highly usable web applications. In addition for providing these newest features, the new technologies also provide new security faults and vulnerabilities that might go unnoticed for a certain period of time making them exploitable by malicious actors. Information security is as important as ever, when these new technologies are being adopted for serving new types of services for customers and clients (Zhang, 2012).

One of these new technologies that was developed and published alongside HTML5 is the WebSocket protocol. The WebSocket protocol was developed to provide real-time, full-duplex, and asynchronous communication between the web application and the server, thus server making the web applications more usable and lightweight and to enable the creation of more functional applications without a need to abuse the already existing protocols not intended for that kind of usage. Creating a new protocol for such a need increases the usability of those applications and makes the development of those applications easier and better (Wang, Salim, & Moskovits, 2013).

From the security perspective there is a lot of interest towards new protocols that are trying to solve a problem which exists widely. It is assumed that a new protocol that tries to rectify the problem is assumed on being adapted quickly, since developers have been waiting for solutions to certain problems for a while. Quick adaptation means competitive advantage against the competition in application development between

different developing parties, but the fastness might come with a price. It could be deduced that newer protocols lack the exhaustive security auditing that can only be conducted in the time span of years by security researchers and other motivated instances, including criminals and other wrong-doers. The lack of security auditing also give malicious actors an advantage since they are also learning the protocols alongside with the developers. Malicious actors do not report their findings in a responsible manner – instead they use the detected vulnerabilities to exploit the target system (Dowd, McDonald, & Schuh, 2006).

Securing web applications is a crucial task for the developers, but it is also relatively challenging. Fred Brooks's important quote from 1986 about software engineering can be applied to security aspects as-well – *there are no silver bullets (Brooks, 1986).* Security testing is crucial during the development of web applications, and it can be argued that a secure application can not be developed without proper testing procedures. Also the protocols and underlying technologies that the applications use must be as secure as possible. This can also be achieved by vigorous and proper testing (Meucci & Muller, 2014).

1.1. Research Problem

The goal of this work is to answer these four questions:

1. What are the vulnerabilities in the WebSocket protocol/implementations?

2. What kind of preventive and defense mechanisms are there for defending against different threats?

3. How widely is the WebSocket protocol used in the web?

4. How can WebSocket protocol be improved from the security point of view?

The first question will be answered in this work in Chapters 3 and 4. The vulnerabilities and weaknesses in the WebSocket protocol can be mostly covered with the documentation and existing research. Chapter 4 will reveal some problems that were found in some libraries that implement WebSockets. A thorough review of the WebSocket protocol and the WebSocket API reveals known or unknown implementation faults, vulnerabilities, and bad security practices that might lead to serious security issues. Learning how those vulnerabilities could be used for malicious purposes is the key on understanding how the countermeasures against vulnerabilities and weaknesses work.

The second question and its results will depend on the results of the *Background*-chapter. In the empirical part of this work an attempt will then be made to resolve these issues and to provide fixes and – at least – some mitigation techniques to make the vulnerabilities less severe and the probable breaches less damaging. Countermeasures and the behavior of the preventive mechanisms will be reviewed and explained. The risks and threats that do not have a working security mechanisms available will be presented as well.

The usage percentage of the WebSocket protocol in the web will be presented in Chapter 4.2. *WebSocket Usage Statistics*. This rate will be determined by performing an analysis on existing web sites. With this information we will be able to determine how many of the existing web sites that use WebSockets apply encryption in the connection from a client to the server.

The last question will be answered in the Chapter 5 after the results have been found. After that, certain assumptions and suggestions for the future development of the WebSocket protocol can be applied. These answers will deal with both the WebSocket protocol and the implementations of the WebSocket as well as the good practices concerning these applications.

1.2. Expected Results

WebSockets are a relatively new standard in the field of software development where standards, techniques and trends continuously evolve. The first draft of the WebSocket protocol was submitted in 2010 according to *IETF Datatracker* and it received its RFC status in the following year (IETF, 2012). The young age of the protocol could indicate that there are certain aspects that have gone unnoticed during the development process of the WebSocket protocol.

However, it is clear that there are no foolproof protocols from the security perspective, and one would always assume that there are vulnerabilities that are yet to be detected. Even if the protocol in itself seems to be secure there are certainly applications that use WebSockets that lack the proper safety precautions. This could be explained by the fact that there are developers who are not so security oriented and therefore do not pay an equal amount of attention to security. Granted, there are also applications that do not need extra attention towards security – e.g. applications that are not designed to handle sensitive information or to store personification data in the first place.

It is expected that some of these applications can be found, and that most of these applications can be more or less easily adapted to follow the most common security principles. The results could then be presented as follows:

- There are vulnerable applications and software that use WebSockets as a technique for providing real-time communications
- There are solutions available for fixing the vulnerable applications
- Secure practices can be applied to applications implementing the WebSocket protocol

Fixing these vulnerabilities is the key to providing safe applications for the customers. Depending on the type of the vulnerability, the attacker can take advantage of it in very surprising ways to attack the targeted application.

1.3. The Scope of this Thesis

This research and thesis is limited to the WebSocket and its API in itself. Although the WebSocket also uses other well-known protocols for its functionalities, e.g. *TLS (Transport Layer Security)* for data encryption, no security audit of that protocol will be conducted. Furthermore, although some references will be made to TLS and although it will propably be necessary to highlight some of the most common vulnerabilities, a thorough security audit of the TLS protocol will nevertheless be outside the scope of this thesis.

The focus will be on the WebSocket protocol technical implementation and therefore some other common networking components will be disregarded in this thesis. It is to be admitted that minor references to these components might be needed but a thorough research focusing on e.g. *Transmission Control Protocol (TCP)* will not be embarked on.

Applied security measurements will not be evaluated from an economics point-of-view so it is therefore up to the organization and the developers themselves to decide whether the modifications provided in this thesis are suitable solutions to their problem from the economic perspective.

2. Research Methodology

In this chapter the theory of the research method used and general information about this thesis is presented. Later, testing methods that have been used in this thesis will also be explained.

2.1. The Design Science Methodology

This thesis uses Design Science as its research method for providing results from the security aspect of the WebSocket protocol. Design science method consists of two different operations: build and evaluate. During the build phase, a suitable test environment for the intended tests is created, along with selecting the tools that can be used in this work. The results gathered by using the selected tools are then evaluated in the evaluation phase, and their significance from the security perspective is then estimated. Figure 1 presents the general structure of the design science (Hevner, March, Park, & Sudha, 2004).

After the creation of the test environment we need to select WebSocket libraries for creating applications and WebSocket server implementations for testing purposes, because different need different applications and different tools. There is no dedicated method for the selection of the WebSocket libraries. The building of the suitable test applications – or using existing ones that have already been developed - is also part of the build-process of the design science-method. The applications and tools created are created so that they carry tests for certain test cases (Järvinen, 2008).

The evaluation criteria is the security aspect. We test the built testbed and applications for security evaluation, focusing on the WebSocket protocol and some of the key concerns that the protocol specification contains. Since some of the vulnerabilities of WebSocket have already been found (and mostly fixed) we are focusing on finding new methods for exploiting WebSockets. The evaluation process aims for improvement of WebSockets, since known vulnerabilities can be fixed and therefore the security can be enhanced.

Auditing application protocols can be a tedious task. Communication protocols like WebSocket usually have a documentation readily available for anyone who wishes to know more about the protocol and its in-depths in detail. If the documentation is not 30 available – and even if it is – Dowd et. Al (2006) suggest on performing a security related audit for the purpose of seeing the protocol in-action, performing the tasks it is supposedly developed for. The most common way of learning how a protocol works is to capture the packets that are send using the protocol under research (Dowd et al., 2006).

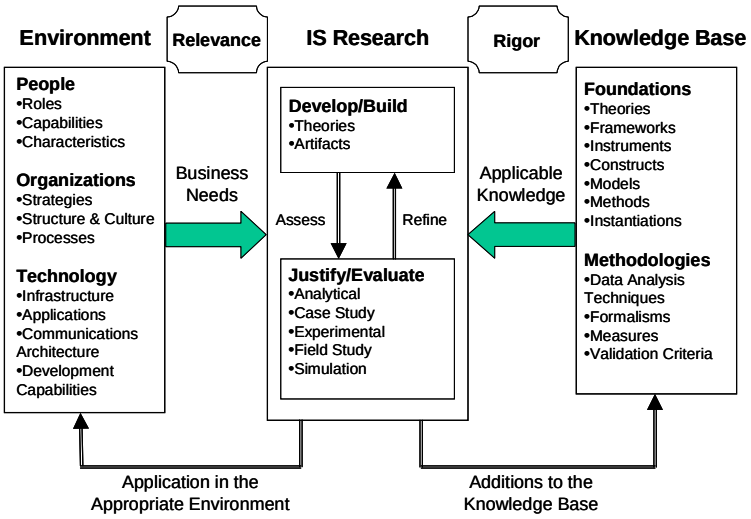

Figure 1. A graph picturing the different stages, parties, and artefacts involved in the *Design Science* method (Hevner et al., 2004),

The evaluation is the part where the results are to be found. It is a key component in the design science method. The artifacts are weighed upon the general principles of information security, and how we have defined security in networks and applications in previous chapters (Hevner et al., 2004).

What this thesis is aiming to prove is that the WebSocket protocol has its own weaknesses and vulnerabilities which are needed to be addressed in order to implement the protocol correctly on an application. Correct in this context means that the known and most common vulnerabilities are noted and identified when developing the application, without significantly harming the usability of the application. It is the key for a successful application to keep the usability and the security aspect in a delicate balance, where the security aspects are taken care of – in the best case – invisible to the user. Depending on the use-case – e.g. banking application where client sensitive data is handled en route from client to the provider – there might be need to address the security of the application deliberately. The results of this thesis can be used as general tips for developers or organizations that are implementing WebSocket protocol to their services and helping them to use WebSocket in a more secure way.

3. Background of WebSockets and Information Security

Modern web applications are growing in size, features and user base. Newer features require newer protocols to comply with the growing need to replace older protocols that are not necessarily suited for certain purposes. WebSockets are developed to meet with the demand for high speed, light overhead, and real-time communication (Wang et al., 2013).

The following chapter will present the WebSockets protocol and the WebSocket *Application Programming Interface (API)*, their dominant features, use-cases and security issues. After that the basic information security concepts and the importance of information security in general will be explained. Finally, in the light of these first sub-chapters concrete issues and threats concerning web applications that use WebSockets will be presented.

3.1. WebSockets

In computer science, a socket is a software component, which is used to connect computers to local or wide area networks. The socket is opened for a program which then uses this socket to transfer data to another location (Techterms.com, 2014). WebSockets provide persistent connection between client and server, which means that there is a connection open for bidirectional communication. The WebSocket protocol uses ports 80 and 443, depending on whether TLS is enabled or not. This way enterprises do not have the need to open additional ports from firewall in order to use WebSockets (IETF, 2011b).

WebSocket is a technology, that provides asynchronous full-duplex communication between the WebSocket server and the client. Earlier, this type of communication was implemented with HTTP, but since HTTPs original purpose was to provide a way to display static content on web applications there was a specific need for a new protocol. Displaying static content and polling data from the server using HTTP requires much more bandwidth than WebSockets.

WebSocket can be used as a transfer medium with other higher level protocols – in a somewhat similar manner as WebSockets use TCP as a lower-level implementation. Protocols that can be used with WebSockets are for example the *Extensible Messaging and Presence Protocol (XMPP* or *Jabber), Remote Frame Buffer (RFB* or *VNC),* or *Streaming Text Oriented Messaging Protocol (Stomp).* Using these protocols on top of WebSockets makes sense, since using sub-protocols on top of WebSockets does not require additional firewall rules or proxies because WebSockets have already established a working connection that the firewall has already allowed (Koch, 2013).

Web sites have been static and stale in nature in providing information and content for users, but nowadays we can observe that the web applications – formally more generally known as web sites – are becoming more and more dynamic, interactive and constantly changing, which is why there has arisen a need for more flexible protocols. WebSocket provide more flexibility for developers than protocols that have been known for displaying only static content on web applications (Kaazing.com, 2014). Earlier

solutions for real-time communication between browser and server operate on top of HTTP, which is difficult to develop and configure to work as its supposed to. HTTP as a foundation for creating simple real-time communication applications increases complexity unnecessarily (Wang et al., 2013).

3.1.1. History of WebSockets

The early Internet-draft of *draft-ietf-hybi-thewebsocketprotocol-17* was submitted in May 2010 by Ian Hickson, the author and maintainer of the HTML5 specification. The draft was appointed to *Internet Engineering Task Force (IETF)*, and the goal was to provide advanced capabilities for providing cross-platform communication in HTML5. After several drafts the *draft-ietf-hybi-thewebsocketprotocol-17* switched to RFC version. The WebSocket protocol was then known as RFC 6455, and it is the latest version of the protocol still (IETF, 2012; Wang et al., 2013).

The WebSocket protocol is maintained by IETF, but its API is developed by *World Wide Web Consortium (W3C)*. The WebSocket API is mainly supported by most widely used modern desktop browsers (Chrome, Firefox, Internet Explorer, Opera, Safari), and also the mobile platform is mostly supported by the same vendors' mobile versions of browsers (Mozilla Developer Network, 2014). API provides the methods and functions that are needed to provide WebSocket communication for web applications (Alexis Deveria, 2015; W3C, 2011).

In an interview with Bruce Lawson in 2013, Hickson stated that in hindsight giving the WebSockets specification process to IETF “was a huge mistake”. He explained that the process delayed the development of the WebSockets specification by a year, and the IETF made changes that reduce the safety of WebSockets. He also stated that features such as multiplexing and compressing could be available to WebSockets now if the IETF was not involved (Lawson, 2013).

3.1.2. State of The Art

The WebSocket protocol specification document RFC 6455 was last updated in 2012. In February 2014, Yataka Hirano submitted a new internet draft for standardizing WebSocket protocol over HTTP version 2.0, which is also known as HTTP/2. Hirano notes in the Introduction-chapter that the RFC 6455 specification has scalability issues. That specification is designed so that one WebSocket connection uses one TCP connection. This solution works for smaller applications, but there are applications that use multiple WebSocket connections for one user to provide flexibility. However, this increases the load on the network intermediaries and the end hosts that provide the application. The goal of the draft is to layer WebSocket protocol to use HTTP/2s framing multiplexing functionality rather than TCP connection. This would fix the scalability issues with TCP, since WebSocket protocol could use HTTP/2 stream, which is being developed to handle such issues as HTTP/1.1s lack of support for TCP multiplexing (Hirano, 2014).

Shema, Shekyan and Toukharian researched the number of the different WebSocket implementations in the Alexa[1] Top 600,000 websites in 2012. The results showed a relatively low adaptation rate: of the 600 000 most downloaded web sites only around 0.15% use WebSockets on landing page.

[1] Alexa Internet is a subsidiary company of Amazon.com which provides analytics data of web page and network traffic (Alexa.com, 2015).

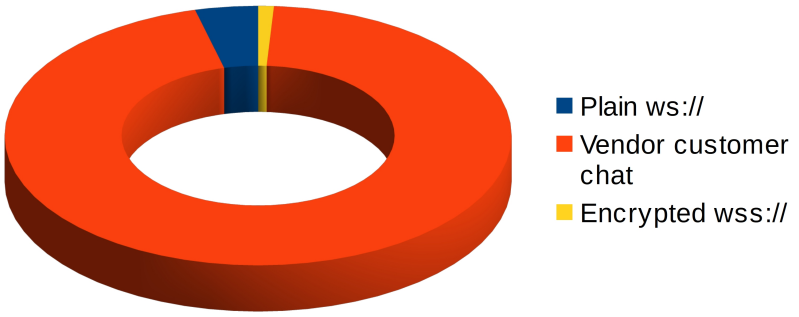

Figure 2. Pie chart displaying the usage of WebSockets on 600,000 most used web sites in 2012.

The pie chart in Figure 2 shows quite clearly where WebSockets are mostly used. 95% of the detected WebSocket instances were made up of vendor customer support chat that was widely applied to different sites. The remaining five percent was split in two: *plain ws://* took the slice of four percent and the last percent was the encrypted *wss://*.

Jussi-Pekka Erkkilä performed a security analysis in 2012 on the WebSocket's latest specification RFC 6455 analyzing the WebSocket protocol and its security features. He concluded that as WebSocket "provides many advantages it also opens up potential security issues". Many of those security issues are not necessarily bound to WebSocket or its API in itself, but they are nonetheless risks concerning WebSocket and its implementation. WebSocket relies on several different networking components (TCP, TSL/SSL, proxies) and web browsers comply differently to specifications of WebSocket protocol, therefore many different scenarios for potential attacks are easy to find.

Robert Koch studied WebSockets in 2013 from the perspective of penetration testing. He applied the WebSocket protocol to the open-source intercepting proxy *Zed Attack Proxy (ZAP)*. Intercepting proxies are used for helping developers in finding weaknesses in web applications. He also did an evaluation on the WebSocket usage in smartphone applications, where he concluded that in 15000 Android applications only 14 were found to use WebSockets. During the research, the WebSocket API was still in a standardization process, which could explain at some level these low numbers in the use of WebSocket applications in smartphones.

Considering the relatively young age of the WebSocket protocol, there are not that many scientific studies being made at the moment, although the internet and many different bloggers is packed with different types of WebSocket-related findings and issues. Although they are not scientific sources, some of them can be used as a resource for ideas and examples when testing schemes are planned. The sources and the information they provide will be tested and the results can be reported accordingly to validate them.

3.1.3. Features and Advantages

Traditionally, real time communication in a web application has needed some kind of a workaround. When a client loads a web page and sends an HTTP request for the server, the server acknowledges the client and responses accordingly. At that moment the server is capable of providing valid, real time data. But after a some time the information might not be valid anymore, for example in the case of stock values. In this way the data are updated only when the page refreshes, which creates a lot of traffic and overhead for the communication if a refresh is needed only for a few bytes of updated information (Wang et al., 2013).

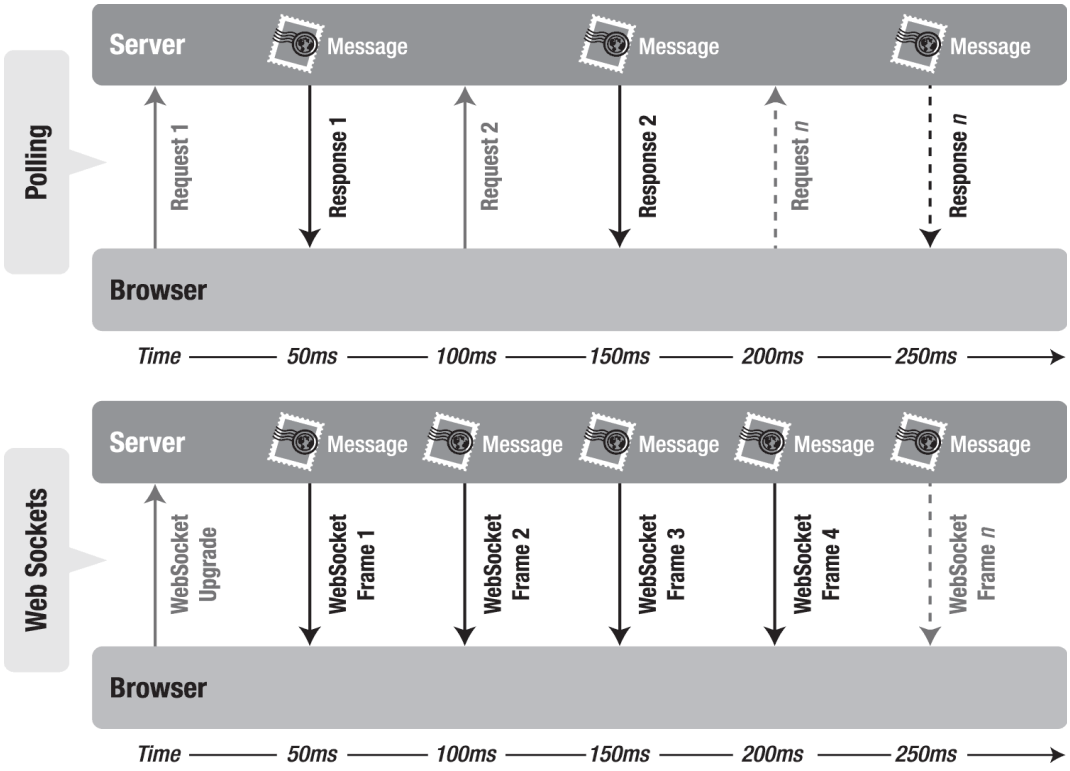

Figure 3. A graph displaying the difference in latency between regular HTTP polling and WebSockets frame exchange. The WebSocket protocol does not need additional requests for receiving frames or sending frames.

Formerly real time communication was applied to web applications by abusing the HTTP protocol by delaying the responses from the server. *Polling*, or otherwise known as Comet, is a technique where a client sends request to a server in a regular interval whether or not the server has any new information to provide for the browser. If there is new information, the server then responds to the request. Figure 3 displays the difference how regular polling and WebSockets work in the context of frame exchange (Lubbers, Albers, & Salim, 2010). This solution works when you know the exact interval of updates on information, but it is still not real time and it creates unnecessary overhead for the communication when the client sends new requests when there is no new data available (Loreto, Saint-Andre, Salsano, & Wilkins, 2011; Wang et al., 2013)

Another workaround is *Long polling*, where the server delays its response arbitrarily, and when the server notices that there is new information sent to the browser it then responses to the original request. If the request performs timeout, the browser automatically sends a new request to the server that hangs until the server has something new to send to the browser (Loreto et al., 2011).

The WebSocket protocol is meant to replace the techniques such as HTTP polling and long polling by providing more real time communication and by reducing overhead on networks. Using HTTP polling for displaying dynamic content on a website is a kind of a workaround since the original idea of HTTP was to display only static content on a web site. The latest HTTP protocol specification was released in 1999 when the Internet was a whole different concept than what it is today (Fielding et al., 1999). Since then, the Internet as a concept has developed to a whole new meaning, and web applications are leaning away from their original purpose. They are still very much used in the “traditional way”, but the spectrum of the use-cases has widened (Kaazing.com, 2014).

WebSockets also enable the establishment of client side connection, which was not possible with previous solutions (Wang et al., 2013). The WebSocket protocol uses TCP connection for providing tunneled messaging in a bidirectional manner. This way the HTTP headers are stripped out – resulting in much lower overhead and bandwidth usage since there is no need for refreshing and polling to update information displayed on the web page. (IETF, 2011b).

3.1.4. WebSocket API

The WebSocket API is an interface that enables developers to use the WebSocket protocol in web applications. The following chapter presents the most common examples and features that are needed for creating useful web applications.

The WebSocket protocol consists of two separate actions: a handshake, that informs the server that client wants to upgrade the HTTP connection to a WebSocket connection, and the data transfer. Once the underlying TCP connection is upgraded to WebSocket, the client can send WebSocket compliant data and vice versa. After that, the asynchronous event listeners handle the different situations in WebSocket connection life cycle.

Table 1. A WebSocket object has these four different events:

Event handler	**Event handler event type**
onOpen	open
onMessage	message
onError	error
onClose	close

Table 1 displays the event handlers that must be supported by all objects implementing the WebSocket interface. Here is an example where the event handlers are attached to functions to correlate with WebSocket events:

```
if ("WebSocket" in window){
   websocket = new WebSocket("ws://127.0.0.1:8080/", "echo-protocol");

   //attach event handlers
   websocket.onopen = onOpen;
   websocket.onclose = onClose;

   websocket.onmessage = receiveOutput;
   websocket.onerror = onError;
   }
 else {
   alert("WebSockets not supported on your browser.");
}
```

After this we can assign functions to those previously attached variables, and define what the application does when a message arrives from the server:

```
function onOpen(evt){
  console.log("Websocket open, waiting for traffic....")
  sendCommand()
}
```

```
function onClose(evt){
  websocket.close()
  console.log(“Disconnected”)
}

function onError(evt){
   window.alert("Error establishing WebSocket-connection to server" +
evt.data)
}
```

These are the main events that are needed to handle WebSocket traffic on the clients side.

3.1.5. Establish WebSocket Connection

Figure 4 presents the handshake operation between client and server when establishing a WebSocket connection. The client sends an upgrade request to the server for upgrading the HTTP connection to a WebSocket based connection. In order to create a successful connection the client must send all the necessary information to the server to parse. If the server fails to parse all the necessary header fields the server stops processing the handshake request and responses with an appropriate error code, for example *400 Bad Request* (IETF, 2011b; Ronacher, 2012) .

Figure 4 demonstrates the handshake operation of a WebSocket connection without TLS enabled. If the handshake operation is happening on via port 443 the TLS handshake must be made over that connection. If the handshake operation for TLS is successful, then all the communication must be made through the encrypted tunnel, including the server handshake response. (IETF, 2011b).

One of the most interesting fields from the security standpoint is the *Origin*-header. It consists of a scheme, host, and port. For example, in Figure 4 the request for an upgrade comes from the following origin:

```
localhost:8080/
```

This handshake request was performed on a local client and a local server instance. The communication was directed to port 8080 on the server side, therefore the *Origin*-header displays the server port. If the port number is the default HTTP assigned port (80) then the port number is omitted. This applies to every situation where the port number is the same as the default port for the scheme (Wang et al., 2013).

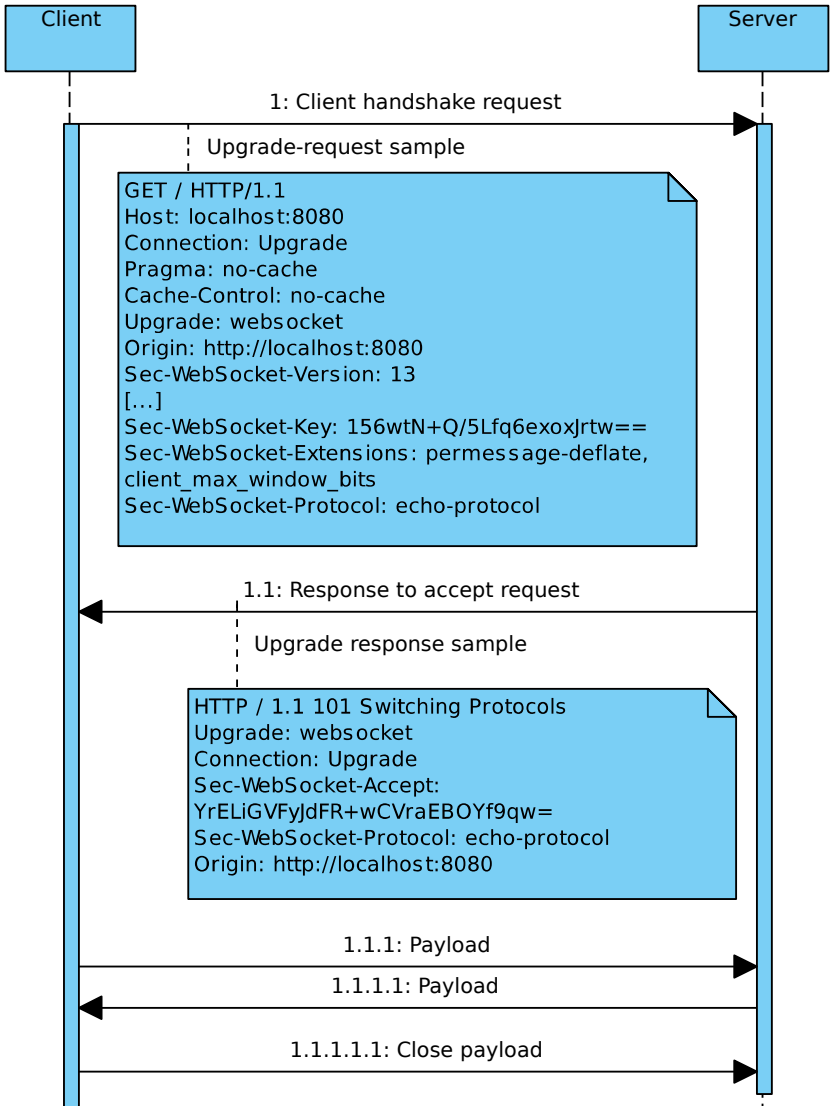

Figure 4. Example of WebSocket opening handshake without TLS. After a successful handshake the frame exchange begins.

The server can naturally be configured so that it rejects connections from unknown sources or from sources which it does not want to handle (Wang et al., 2013). This mitigates the risk of a DOS-attack. However, the risk still exists since the header parsing is done by the browser instance, which means that the parsing uses the server's CPU to compute the parsing. If the number of malicious origins exceeds the servers capacity to handle handshake operations the server could be made unreachable for clients that are authorized to connect to the server (Jain & Singh, 2012).

3.1.6. Frames in WebSocket Protocol

Payload is the section of the data packet which contains data that is needed to be transferred from one place to another. Generally, packets in networks consist of two different sections:

- Headers, which contain information about the origin and the destination

- Payload, which specifies the actual data being sent.

In the WebSocket protocol, data is transmitted using a sequence of frames. In Figure 5. the base framing protocol scheme is presented, obtained from the WebSocket protocol documentation RFC 6455:

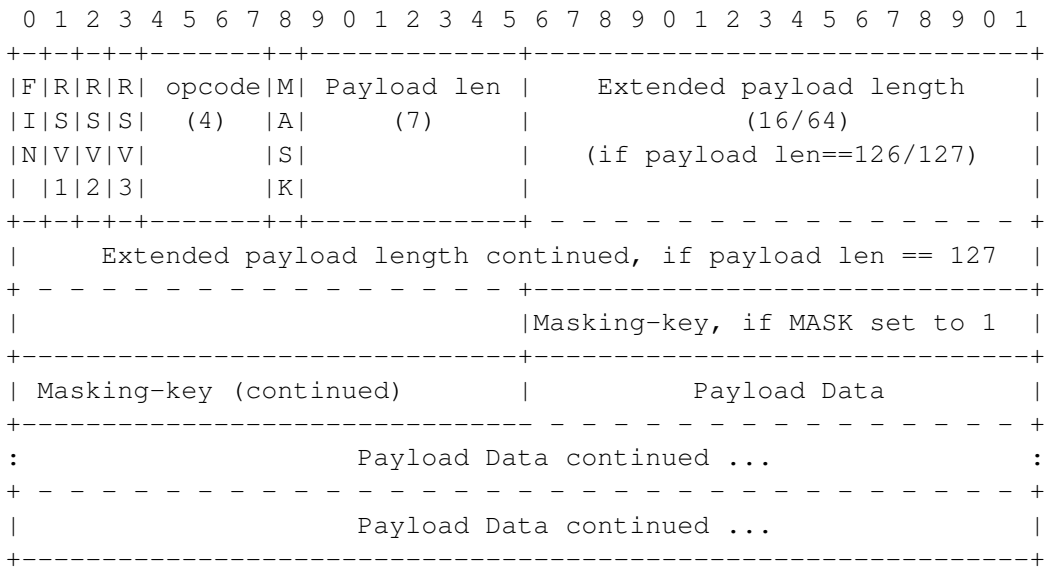

Figure 5. The framing scheme of the WebSocket protocol.

Frames consist of fields with different uses. Some of them are used on any WebSocket based communication, some of them are reserved for future development. Here are the fields explained (IETF, 2011b; Ronacher, 2012):

- *FIN* (1 bit): When the bit is set the frame is the last fragment in a message.
- *RSV1, RSV2, RSV3* (1 bit each): Bits that are reserved for future development. The receiver must fail the WebSocket connection if a nonzero value is received and none of the negotiated extensions defines that very value.
- *opcode* (4 bits): Contains information that informs the recipient of the content of the frame. The following are the currently supported values:
  - *%x0*: Continues the payload from the last.
  - *%x1*: Contains the text data in UTF-8.
  - *%x2*: Contains binary data.
  - *%x3-7*: Reserved for future usage.
  - *%x8*: Terminates the connection.
  - *%x9*: Contains a ping.
  - *%xA*: Contains a pong.
  - *%xB-F*: Reserved for future usage.

- *Mask* (1 bit): Indicates the masking of the “Payload data”. If set to 1 – as it must be when sent from a client to the server – a masking key is presented in the masking-key field which is used to unmask the payload.

- *Payload length* (7 bits, 7+16 bits, or 7+64 bits): The length of the payload field in bytes. The values between 0-125 indicate that the payload is that long. In the case of 126 the next two following bytes are the payload length. If the payload length is 127 then the next 8 bytes indicate the length.

- *Masking-key* (0 or 4 bytes): All frames coming from the client to the server are masked. If the frame is masked – as it should be – this field has the value 1, but if the value is set to 0 this field is not present.

- *Payload data* (x+y bytes): The actual data that is transferred – consists of *Extension* and *Application data.*

- *Extension data* (x bytes): Contains x amount of bytes if an extension has been established, otherwise it is set to 0. Any extension and its behavior has to be negotiated in the opening handshake.

- *Application data* (y bytes): Takes up the remainder of the data that does not fit into *Extension data*.

For security reasons, the client must mask all data that is being sent to the server in the manner specified above. If the client fails to mask the data accordingly, the server must close the connection upon receiving the non-masked frame. This type of data frame forms the data transmission scheme between the WebSocket client and the host, and can be transmitted from either counterpart anywhere between the opening handshake completion and the sending of a *Closing frame* (IETF, 2011b).

Masking is a function, where the payload data is transformed unreadable using bitwise operations. The frames that are being sent from the client to the server need to be masked using a 32-bit key *Masking-key*, which is transferred in the frame to the server. If the *Mask* is set to 1, the *Masking-key* needs to be present. The key is then used to unmask the payload when the server receives it. All frames that are masked have to have a new *Masking-key* in order to make them more unpredictable for the attacker. Masking-operation is required for additional security of the WebSocket protocol, but there is no requirement for the server to perform masking. Masking was added as a security feature in order to mitigate the possibility of a *cache poisoning attack*, which will be explained in more detail in Chapter 4.3.5. Since the *cache poisoning attack* affects mostly the HTTP GET-requests (where the client requests data from the server), the server masking the frames that are sent to the client was determined unnecessary (IETF, 2011b).

3.2. Information Security

The following chapters present information security as a general science and practice, and underline the importance of information security in networks and web applications that are the key areas of this thesis. Understanding the importance of security in general is the key to understanding more specific concepts, such as protocols, and the security on that level.

3.2.1. The Importance of Information Security

Security has always been a difficult concept to measure. Lord William Thomson Kelvin (1824-1907), a mathematical physicist and engineer who mostly worked in the nineteenth century (when terms like software were unheard of) is quoted saying "*If you can not measure it, you can not improve it*." He might have been referring to his own field of expertise, but this also rings true in the context of security.

Measurements are evaluating concepts from different aspects using different techniques. For example, the stock market is one concept that can be measured. In people's minds the performance in stock market can be considered as an abstract notion. Conceptually the concepts that are being measured are a specific set of companies, which are measured from the value and growth perspective. One of the methods that is used to measure the stock market is the Dow Jones average which in itself is error-riddled that does not provide the full picture of the situation of the organizations under measurement. Luckily, the Dow Jones average is only one way to measure the concept of stock market performance (Viswanathan, 2005).

As with the situation with the stock market, there is no one bona fide solution for measuring how effective or ineffective information security or additional security measurements are. One of the ways to evaluate the need for additional security measurements and their effectiveness on the risk that is in need for added security is risk management. Depending on the organization and the risk at hand, different approaches and techniques can be used. Manuel Santander from SANS Institute suggests *Information Security Management System (ISMS)* as the security management protocol for the organization (Santander, 2010). ISMS gives guidelines on how appropriate measures are assigned to given stages of risks and how they are managed.

From the customer's point of view, some might see added security features as unnecessary, complexity increasing, and difficult to use. For others, how the security is handled by the company might be a key deciding factor whether to use the company's products or not. Integrating security and usability is a key factor for the success of using security mechanisms. Security practices are to be implemented so that the effect on user experience is minimal, and that there is only one way to do something – the secure way. Too many options tend to lure the user to use use something that is easy, not something that is secure (Whitman & Mattord, 2011).

Security can also be seen as an expensive investment that does not add to the company's value directly. Security consists of highly technical and specific issues in computer systems, but it also has an impact on the economics. Stock market and its behaviour has always been a strong indicator on a company's value when something goes awry. When a company is struck by an attack from the outside, it is not that common to see the company's stock value go down. Granted, some studies also show that information security breaches have no significant relation to the company's stock value. However, this does not deflate the meaning of security when it comes to the organizations valuable assets. Losing those assets altogether could lead to devastating results when it comes to customer satisfaction and reliability. Losing the public image of reliability could lead to losing potential new customers and therefore decrease the number of customers. In addition, the company could be sued for losing customer data, and these potential legal liabilities could also prove to be very harmful and costly for the company (Gordon, Loeb, & Zhou, 2011).

There are many recent examples that display the importance of good security measurements. In 2014, an American home improvement retailer Home Depot was hacked, and as a result 56 million credit cards were affected. The hackers that are

believed to be behind this breach presumably installed malware on the retailers self-checkout systems. During a five month period from April 2014 to September 2014 they were able to go unnoticed, successfully capturing 56 million debit and credit card numbers, which is the largest retail card breach in history. Early estimates from Home Depot state that the gross costs for damage repair are around 62 million US dollars. Those costs include investigation, providing credit monitoring services to the customers, increasing call centre staffing, and paying legal and professional services needed for the investigation. Currently, it is rather difficult to estimate how much this is eventually going to cost Home Depot, but this breach will leave a dent on the company's public image on how they handle security. The repercussions could have been much more severe, if besides losing card numbers additional payment card data had been leaked, too. The company has as of January 2014 “rolled out new enhanced payment protection, which scrambles the raw payment card information unreadable and virtually useless to hackers”. The roll-out was completed in September 13, 2014, eleven days after the breach was first detected. The breach happened just before the new encryption mechanisms were fully implemented in all Home Depot US-retail stores (Krebs, 20AD; The Home Depot, 2014)

Revelations in the 2013 about National Security Agency's (NSA) surveillance program has also had a great impact on general mindset about security and privacy. New America's Open Technology Institute in 2014 states that the revelations have increased the value of the organizations that claim they can defend against NSAs surveillance. It is a competitive advantage, that only benefits companies outside the United States because NSA seems to be connected to every organization in the country in some form or another. Customers' increased interest in privacy and security has increased the demand for services that provide privacy protection and value security visibly (Dehl, Greene, Bankston, & Morgus, 2014).

3.2.2. The C.I.A. - triangle

The C.I.A. triangle – displayed on Figure 6 - consists of three different aspects of information that are valuable to the corporation when they are valuing the data handled by themselves: confidentiality, integrity, and availability. Although these aspects are still valid and important, they are lacking certain aspects of information security (Whitman & Mattord, 2011). Janine L. Spears criticized in 2006 that the definition of the C.I.A. triangle is not up to date with the modern concept of information security. She noted that the C.I.A. “does not acknowledge the employee or strategic related aspects of information security.” Since the original definition of information security the networks, the computers and the users have changed drastically which makes the “traditional” definition inadequate. She also points out that ethics, trust and responsibility are visibly lacking because these concepts are increasingly associated with information security (Spears, 2006).

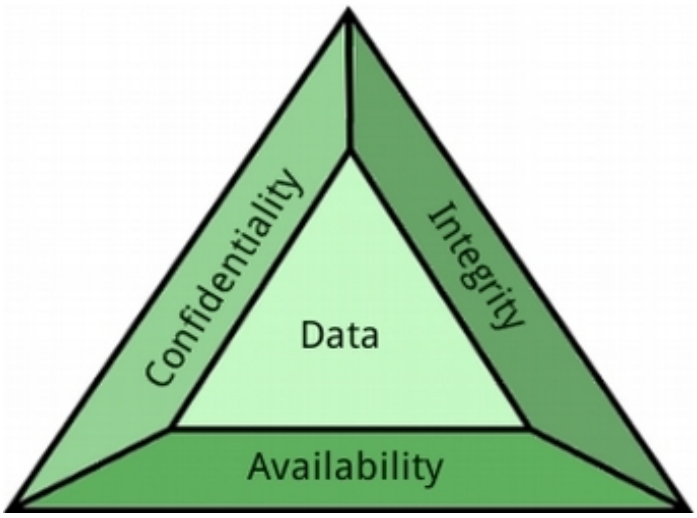

Figure 6. The C.I.A.-triangle

In today's world the term cyber security seems to be used more often than the term information security. Although they seem to synonyms to each other, there are minor details that make up the lack of human aspect pointed out by Janine L. Spears. Information security and cyber security as a concept differentiate from each other in one particular aspect: cyber security takes humans into account as well, in accordance with the familiar C.I.A. concept (von Solms & van Niekerk, 2013).

In relation to WebSockets, *integrity*, *availability*, and *confidentiality* can be achieved by using a TLS connection (*wss://).* However, TLS is still prone to *SSL Strip*-attacks where the attacker reroutes the traffic via an unencrypted channel, thus performing a *man-in-the-middle* attack – explained in detail in Chapter 4.3.3 - and allowing the attacker to read and/or write the traffic. This is possible if the secure site still uses *http://* instead of *https://* in its URL (Marlinspike, 2009). However, the WebSocket protocol is designed so that if the TLS is enabled in the connection, then all the frames sent or received must be using TLS – otherwise the connection is dropped (IETF, 2011b).

In this thesis the traditional definition of information security and the C.I.A. triangle are still valid since the aspect of the so-called human factor is disregarded in this thesis. The only security related factor that could specifically be tied to humans is the implementation. However, are the developers, system administrators, and other people who are responsible for developing and maintaining modern web applications skilled and educated enough to take the security aspect of their applications seriously. In any case, the factors that are being weighed as to the overall security in this thesis are *confidentiality, integrity, and availability*.

3.2.3. Security in the Networks

At this moment the Internet is by far the most widely used network in the world. People are accessing different services that use different devices for accessing them. Connecting to a network makes devices visible to others as well, which increases their risk for an attack from the outside. Adding protocols, devices, ports and services available by accessing them through network by authorized entities makes them vulnerable for unauthorized access, too (Kizza, 2013).

Computer networks generally consist of entities, that have the desire to share data with each other. These entities need a way for communicating: hardware for acting as a

transmission medium and software and protocols for successful exchange of information. At the beginning of the development of computer network the device count for the network was a handful of computers, but the rapid success and adaptation rate of the internet has increased the need for more complex standardizations. To standardize the equipment and the software the *Open System Interconnection (OSI)* model was created by the *International Standards Organization (ISO)*. It is an open and layered architecture, which works as network communication protocol standard. Due to its layered structure, every layer is dependant on the layer below. If the layer below does not provide the needed protocols for the layer above to function then the functionalities from the working layer are not going to work. Every layer has its own task, but also distinct vulnerabilities and weaknesses (Pant, 2014). Some of these issues can be fixed with proper implementation or adapting extra features, but some are quite fundamental and require whole overhaul of the protocol *(Kizza, 2013).*

TCP/IP reference model is a competitor with the OSI reference model. It was designed for the Internet, and although the name of OSI would suggest the same, it was originally developed before the Internet. TCP/IP model consists of four layers against OSIs seven layer hierarchy: link, Internet, transport and application. TCP/IP model is used as a model for the modern Internet (Wang et al., 2013).

In the future there will be newer networks infrastructures approaching, such as *Software Defined Network (SDN).* SDN is answering to the need to replace older infrastructures that were not originally designed to work with modern load. SDN is considered to be adjusted to high-bandwidth, dynamic traffic that is needed in modern applications. The architecture relies on openness and vendor-neutral approach for making the network more versatile and more configurable for the controllers. The network resources in SDN are all configurable by software, which makes it more dynamic for different needs. One of the key elements in SDN is the goal to reduce costs in running networks with high capacity (ONF, 2015b). SDN can be deployed with *Network Functions Virtualization (NFV),* which is designed for addressing operational challenges and high costs which are the current challenges in today's' networks. With NFV network functions are virtualized with cloud technology and made available for network operators to achieve greater agility and modifiability, while at the same time reducing costs for running networks (ONF, 2015a).

3.2.4. Impact of Vulnerabilities and Bad Practices

Using networks and the internet requires the implementation of several different protocols. All of these protocols have several different uses, some of which provide fewer functionalities for the network to operate properly. TLS and its predecessor *SSL (Secure Socket Layer)* provide encryption for packets that move in the network, but they also provide integrity and origin authentication (Dierks, 2008). They enhance network security, but those protocols that services rely heavily on are always under development and research. Recently, a team of engineers at Codenomicon and Neel Mehta from Google Security found a bug at the OpenSSL library version 1.0.1. which allowed remote attackers to obtain sensitive information from the process memory. This is a good example of a case where one of the most widely used protocols for encrypting network traffic can be vulnerable and exploited by a malicious actor. The bug was disclosed responsibly, which allowed the developers to work on a patch for the vulnerability and then publish it for the system administrators to patch their systems (Codenomicon, 2014).

In 2014, Ponemon Institute conducted a research where they found that the organizations systems and application administrators do not take proper measurements in securing their Secure Socket Shell (SSH) implementations, which exposes critical

vulnerabilities within organizations. 2,136 respondents from Global 2000[2] enterprises were reviewed, and out of those organizations a very large percentage mismanage their SSH-keys, allowing ex-staff and previous attackers to gain access to the systems. 51% out of the organizations questioned responded that they had been affected by a SSH-key-related compromise in the previous 24 months. SSH-keys were set to never expire in 82% out of those companies, and 53% responded that they did not have centralized key management system in place (Ponemon Institute, 2014). This is not a vulnerability as a software fault, but it is a prime example of bad implementation and bad security practices. Keys should be set to expire in a reasonable amount of time to get rid of the SSH-keys that are not authorized, for example when the work contract of an employee ends.

These examples show, that even the most de-facto standard of the industry can be vulnerable due to e.g. bad practices and software bugs. It is safe to assume that using even the most secure protocols as if they were vulnerable is much safer than assuming that systems are secure and impenetrable. Preparing for a possible breach and lessening possible damages is a useful practice and an advisable act even for an organization with a lower level of security.

3.2.5. Web Application Security

Software development faces new challenges as the number of web applications has increased rapidly. Modern interaction primarily uses *Hyper Text Transfer Protocol (HTTP)* as a communications protocol. HTTP is a protocol that can be useful for different scenarios, although the original purpose of it was quite different. The increase in the number of different web applications available has also made new security threats and vulnerabilities arise from new practices and protocols that are being used and developed. Additionally, many of these web applications use external frameworks to provide functionalities and foundation (Dowd et al., 2006).

Mathias Karlsson found a method for bypassing AngularJSs “sandbox”, which is an isolated environment that restricts the users' access to its own AngularJS implementation. AngularJS is a JavaScript framework, maintained by Google and the open source community (Google, 2014). Karlsson found out that AngularJS allows developers to execute function constructors as callback, because function executing callbacks are allowed. Bypassing sandbox can be initialized when a payload that generates a valid function is constructed, using both of the required arguments and the function body. This is a prime example of external framework vulnerability which affects also the web application that implements this framework. As it uses external frameworks for better functionality where some features are already implemented and which thereby reduce the amount of work for the development of native functionalities, it is a risk that needs to be acknowledged. Luckily, there are many responsible researchers who report these vulnerabilities responsibly to the framework developers, who then patch these frameworks to make vulnerabilities unusable for entities with malicious purposes (Karlsson, 2014).

In order to enhance web application security, several different steps could be made. There are many different attack scenarios and attack vectors that need to be validated for security purposes, presumably before launching the web application to the public. For detecting risks and lowering the risk of probability, OWASP has published a top ten list of the most important security risks for web application development. The vulnerabilities are listed as follows (Wichers & OWASP, 2013) :

[2] The Global 2000 enterprises are the worlds biggest public companies, listed by Forbes (Forbes, 2014).

1. Injection
2. Broken authentication and session management
3. Cross-site Scripting (XSS)
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-site Request Forgery (CSRF)
9. Using Known Vulnerable Components
10. Invalidated Redirects and Forwards.

Web application security is a concept that has many different attacking techniques and security measurements. *The Web Application Firewall (WAF)*, presented later in the Chapter 3.2.7) is one way of protecting the application and the data, and firewalls that are on the layers lower than the application layer need to be applied. These are needed when the application is up and running, but during the development of the application *Secure Software Development Life Cycle (SSDLC)* needs to be applied for developing more secure applications. The SSDLC and SDLC with added security testing are quite similar when it comes to their goals: both have the purpose of reducing security related issues and of making the software more secure. In SSDLC, certain threats for specific targets are identified and tackled with during the development. Creating specific controls and countermeasures enables good Internet hygiene, thus minimizing the probability of an attack. The OWASP top ten vulnerability list is a prime source for threats that need to be treated accordingly during the development process (Akamai, 2014; Whitman & Mattord, 2011).

3.2.6. Threat Type – Cross Site Scripting

The dynamic nature of web applications requires user inputs and server responses, depending on the user's request. In a *cross-site scripting attack (XSS)* a malicious script is injected into an input field on a web application if the user input is not properly sanitized or encoded. If the application fails to validate the input – or if it is lacking validation altogether – the attacker is able to steal cookies and hijack the user's account, manipulate the web content and steal private information. If an attacker can steal a cookie from a certain instance that has already established a session with the HTTP server and therefore acquired a valid session cookie, the attacker can act as the original valid user (Hydara, Sultan, Zulzalil, & Admodisastro, 2013; IETF, 2011a; OWASP, 2014a).

There are two different types of XSS attacks: stored and reflected. A stored attack - also known as a Persistent XSS attack or Type I – is stored on the server side e.g. in a message forum or in a comment field. Persistent XSS includes basically every type of stored data, that is not validated for being safe to render in a browser. Reflected XSS, which is also known as Non-Persistent or Type II, is stored on the server side in a similar manner as the stored XSS injection. A reflected attack occurs instantly when the

user request is returned by a web application in an error message or in a search result, for example (Hydara et al., 2013; OWASP, 2014a).

Amit Klein identified in 2005 an XSS attack, that does not fit in with the original understanding on how XSS attacks work. Normally, XSS attacks are considered to require a transaction with the server in order to send malicious data using a certain vulnerability, and then the server echoes back the message in "an HTML context of some sort, and the Javascript code gets executed". Klein states that these attacks that are called DOM-based (*Document Object Model*) attacks do not fit in with the aforementioned types of XSS: stored and reflected. This type of an attack moves the attackers input from the safer attacker-controlled sources, such as *document.location*, into more sensitive APIs from the security perspective, e.g. *document.write*. The attacker is able to achieve this using unsafe data flows, that are caused by an incorrect client-side code rather than by server-side code (Lekies, Stock, & Johns, 2013). This type of attack does not require any code to be sent back to the server for it to be exploitable (Klein, 2005).

OWASP provides several different ways and guidelines for protecting web applications against XSS attacks. XSS vulnerabilities can be difficult to detect, which is why a thorough code review is one of the most efficient ways for detecting vulnerabilities.

3.2.7. Web Application Firewall – WAF

Web Application Firewall (WAF) was developed to protect web applications and the data handled in the web applications from attacks. WAFs in general have a variety of different options to choose from: both open source and commercial tools are available. Generally, WAF protects web applications from the common attacks such as the aforementioned cross-site scripting attacks and SQL injections. The rules for the firewall can be applied in such a way that identified attacking attempts are blocked. Normally, the functionalities of WAF are based on HTTP spoofing, where the traffic from the client is validated before it is transferred for processing on the server (OWASP, 2014b).

The WAF can also defend against application layer DOS attacks, but it can be the target for a DOS attack like any other network component. Even though the WAF can detect when it is under the attack, the attack normally is detected by doing packet analysis. Packet analysis – even on a smaller scale – requires computing time and resources. When there are a certain number of packets under testing the resources will run out and the DOS attack is successful. Therefore, since the WAF detects application layer traffic, and the layered architecture suggests this, the lower layers need to have security applications that mitigate the effect of the DOS attack. For example, allowing only legitimate HTTP (port 80) and HTTPS (port 443) traffic into the network drops other non-application traffic. DOS attacks can take many forms, such as UDP packet flooding and TCP SYN flooding attack. This sort of filtering will more or less affect the performance of the server more or less, depending on how aggressive the DOS attack is, but it is still necessary the keep servers operational during an attack (Akamai, 2014).

3.3. Summary

In addition to the needs such as reliability and functionality, some applications have a need for additional security. Banking applications, for example, have generally been developed with security as a critical requirement. Some instances have been successful in creating secure software and keeping the users and their data safe, but news of the breaches on web services are flowing constantly and there are no signs of slowing down in the immediate future. Security in the context of web applications relies heavily on the

protocols that are used for networking, data exchange, the software libraries that are used, the devices intended to be used with application in question, etc. The WebSocket protocol is a recent addition to the family of transmission protocols, and there are certainly advantages in using WebSockets. Even though the developers of the WebSocket protocol are taking security seemingly seriously, there are different WebSocket libraries that do not necessarily comply with the protocol standards. The examples and use-cases presented earlier in the chapter present threats that are related to the security of web applications and therefore might use the WebSocket protocol as a medium for attacks.

4. Testing and Results

To define and detect the WebSockets' vulnerabilities and configuration faults a thorough review of both the theoretical and the practical side must be conducted. The first step is to define the attack vectors for WebSocket protocol and that is done in Chapter 4.3. *Attack Vectors for WebSockets.*

The present chapter will find the answers to the research questions mentioned in Chapter 1.1. Firstly, to determine how safe the WebSocket protocol is we need to define safety. Safety, in this context is basically synonymous with the term security, how well the WebSocket protocol addresses the C.I.A triangle and what the means to obstruct these concepts are.

Subsequently, after we have identified the attack vectors related to WebSockets we will solve these issues from the developers' perspective. The goal is to make WebSocket implementation as secure as possible, mitigating the risk for a possible breach or some kind of attack.

4.1. Defining the Testing Process

Fuzz-testing is a testing method for finding out how well a targeted system handles anomalous data in order to find bugs and to increase the reliability. In essence, it is a highly automated testing method where preparing certain, purposefully crafted test-cases takes time but which could prove to be a highly effective method for detecting bugs that might otherwise go unnoticed. Security and quality assurance are usually known to take advantage of the fuzz-testing, but it is not limited to this kind of development. For instance, if security experts are interested in this specific method, most likely the wrong-doers are also interested; basically they are looking for the same thing but for (arguably) different reasons (Takanen, Demott, & Miller, 2008).

In essence, in fuzz-testing purposefully malformed data is sent to a target entity to see how the target handles the malformed data. In the case of WebSockets, a series of malformed requests for the WebSocket server can be sent from the client and monitor what happens when the server replies to these requests that contain malformed data. Theoretically, if the server implementation comes across with a frame which it does not know how to handle, the server could crash. This can lead to availability issues or overflows (Takanen et al., 2008).

To test how the different libraries implementing the WebSocket protocol function when they are used to handle anomalous data, we need tools for fuzzing the data in question. In this testing, we will use a regular string – a piece of text – which is sent to the server and then echoed back. To fuzz the string we will use *radamsa*, a general purpose fuzzer developed in OUSPG (OUSPG, 2015).

```
output = subprocess.check_output(['echo "Hello World!" | radamsa'],
shell=True)
output = output.decode(encoding)
ws.send(output)
```

In the python script that is used to communicate with the server, the message "*Hello World!"* is mutated with radamsa. This produces outputs like the following:

```
Hello Woqld
Helko World!
```

These results are then encoded by using different encodings to increase the risk that the server might not be able to handle the data in question in a coherent manner. It is to be noted that when encoding the fuzzed data, the client script that is being used to send data to the server will also regularly fail to send fuzzed output since the UTF-8 encoding fails in the first test case. That said, since the server and client handle messages differently and use different encoding engines, unwanted errors are nevertheless gathered.

A common class of security problems arises when text data is sent using the wrong encoding. This protocol specifies that messages with a Text data type (as opposed to Binary or other types) contain UTF-8- encoded data. Although the length is still indicated and the applications implementing this protocol should use this length to determine where the frame actually ends, sending data in an improper encoding could lead to unpredictable results.

This message is then echoed back to the client application that checks whether the original payload sent to the server is the same as the one echoed back. This method can obtain results on how these servers behave when they are receiving payloads that should not be sent at all. Table 2 displays the used encodings and the number of frames sent by using the encoding in question.

Table 2. Encodings used in testing of the WebSocket server implementations on how to handle anomalous data ("Character Sets," 2013).

Encoding	**Description**	**# of Frames**
UTF-8	8-bit Unicode	200
ISO-8859-6	Arabic encoding	200
ISO-8859-9	Turkish encoding	200

Additionally, the selected libraries are tested against a DOS-attack, and how they handle multiple connection attempts from one source. The key in this testing is how the handling of the origin is managed in the server implementations. If the attacker can successfully establish multiple connections from one source, the attacker might be able to overload the server and cause availability issues for legitimate users.

Table 3. Table displaying some examples of the WebSocket libraries implemented to different environments. All of them comply with the RFC 6455 standard.

Environment	**Name**	**Role**	**Source**
C	libwebsockets	Client & Server	https://libwebsockets.org
Python	websocket-client	Client	https://pypi.python.org/pypi/websocket-client/
Java	Java WebSockets	Client & Server	https://github.com/TooTallNate/Java-WebSocket

Python for example has a large variety of different WebSockets implementing libraries. Some of these libraries were tested on how they handle malformed data that is purposefully sent to the server and then requested to echo the payload back to the client.

4.2. WebSocket Usage Statistics

To find out how widely the WebSocket protocol is deployed in the web, we need to perform a statistical analysis of a selection of web pages. A common way to perform a statistical analysis on web pages is by using a crawler, a software that downloads the web page and its source code completely or with some filters applied. In this research the statistical analysis was carried out by downloading the web sites to a local machine, in which the analysis of the source code was then performed. From the source code the protocol headers *ws://* and *wss://* were searched to obtain results to answer the research question number three: how widely is the WebSocket protocol applied on the web? In addition to that, we can see how many of the WebSocket instances found use the encrypted tunneling rather than plaintext communication.

In relation to the Shema et al. (2012), this study also uses the Alexa top 600,000 most downloaded web sites for the analysis of the source for web sites. Out of these 600,000 web sites 416,982 were successfully downloaded to be analyzed locally. Table 4 presents the results of this analysis.

Table 4. WebSocket instances found in the sample set of 416,982 of the most downloaded web sites.

	Numbers	**%**
ws://	1314	0.32
wss://	1081	0.26
TOTAL	2395	0.57

These results show that the number of the WebSocket instances used in web sites has indeed grown over these past few years, showing a change from approximately 0.15 percent to about 0.57 percent. This makes a total increase of 0.42 percentage points,

which means a total increase of 26.3 percent during the course of approximately two years. The results can be explained by saying that as the general knowledge about the protocol has been increased, the web developers have taken advantage of that.

In this analysis the WebSocket instances are found on the landing page, which is the first page that is downloaded when visiting a URL with a browser. Hence the number of WebSocket instances on these web sites could be higher if an extensive crawl over the whole web site were to be made. Also, the numbers do not specify that the WebSocket instances found need to be on separate web sites – there could very well be several connection rules on the same web site.

These results also reveal that the number of the WebSocket connections using encryption is higher than in the original research. This could indicate that the WebSockets are used in a more content sensitive manner, where it is necessary to encrypt the traffic. Also, when the site in question uses *https://*, plaintext communication in the WebSocket connection is not authorized (IETF, 2011b). In the original research, one-fifth of the WebSocket connections used the encrypted version. However, in those results it was not specified whether the vendors' support chat application used encrypted *wss://* or not. The revisited results show that the current portion of the encrypted connection is approximately 45 percent, which is almost half of the total number of connections.

4.3. Attack Vectors for WebSockets

What follows is a brief explanation of the defined attack vectors for the WebSocket protocol explained briefly. Some of these attacks can be applied to many different networking protocols, but they can still be used to attack specifically WebSockets.

4.3.1. Cross Site WebSocket Hijacking (CSWSH)

Cross site request forgery (CSRF) is an attack that is related to web applications. In an CSRF attack the attacker forces an authenticated user to perform unwanted actions on a web application. Depending on the user type the results of the attack differ. If the targeted user is an end user who is logged in to the service, the attack might successfully change the users password, gather information about the account, and basically modify and gather information that is available to the user. If the targeted user is the admin for the application the repercussions might be more severe, depending on the success of the attacker. Then the whole application can be compromised (Meucci & Muller, 2014).

Christian Schneider proposed an additional attack threat concerning WebSockets that is similar to CSRF. He describes that during the handshake operation, where the handshake request is sent using HTTP-protocol, the hijacking of the WebSocket connection is possible. Hence the name for the attack: *Cross-site WebSocket Hijacking (CSWSH)*. In CSWSH, it is possible to establish a WebSocket connection to a legitimate service from outside the original application, whether or not it uses HTTPS. When a WebSocket connection is being established, an HTTP upgrade request is sent from the client to the server and the server then replies accordingly. The server then replies to the request with an HTTP status 101 message, along with the credentials that were used to authenticate a legitimate user in an another session. The credentials are then sent to the attacker to be used in hijacking the authenticated session. What kind of credentials are vital to hijack the session depends on the authentication method used in the web application. If for example the site in question uses authentication cookies when the user is authorized then the attacker will be able to hijack the cookie in question if the attack is successful. After this, the attacker has the authority of a legitimate user to perform actions on the web site (Schneider, 2013).

CSWSH is made possible by the lack of a proper *Origin*-management in the WebSocket protocol. If the service has applied proper handling of the origin, this kind of attack should not be possible when alternative origins are blocked from using the service.

4.3.2. Denial-of-Service (DOS)

Denial-of-Service or DOS is one of the most common types of threat to network applications and hosting services. These attacks make the network unavailable for its intended users and overload the system's hard drives, CPU's, peripherals and other components. The purpose of a DOS attack is to overwhelm the target service by taking advantage of specific protocols and flooding the service with huge quantities of requests. The goal is to take the service off-line and make it unreachable for its intended users. DOS attacks usually come from one source (Whitman & Mattord, 2011).

A Distributed-Denial-of-Service (DdoS) attack is an attack in which a request for the target system comes from multiple destinations. These computers used for attacking are usually malware-infected machines which form a massive botnet that the attackers can use. The number of these infected machines is easily in the thousands or tens of thousands, which makes this type of attack hard to defend since the services are designed to handle potentially thousands of customers' requests in peak hours (Akamai, 2014).

One of the DOS or DdoS attack types is the TCP SYN flood attack. This attack uses the three-way handshake of TCP protocols as its mean of an attack. In the TCP three-way handshake the client sends an SYN to the host, the host replies with an SYN/ACK to synchronize with the client, and finally the client sends an ACK message to the host to acknowledge that the earlier message was received successfully. The TCP connection is now established between the client and the host. The way that the TCP SYN flood attack works is that the attacker sends several SYN requests to the host with a forged source address. Then the host allocates the memory and the resources for establishing the connection and sends the SYN/ACK back to the attacker. However, the attacker never responds to it so the resources are never freed in the host's server. This will eventually lead to memory exhaustion and make the host unreachable for users (Jain & Singh, 2012).

Like many other networking protocols, the WebSocket protocol is not immune to DOS attacks. Although the handshake process includes only one frame exchange between the client and the server, the WebSocket server is still vulnerable to an DOS attack although the *Origin*-header mitigates the risk. Nevertheless, the parsing of the handshake frame from the client is performed by using the server's CPU and memory, which might eventually run out if the scale of the DOS attack is too immense for the server to handle.

4.3.3. Man-in-the-Middle (MITM)

Man-in-the-Middle (MITM) is synonymous with TCP hijacking. An attacker hijacks an established TCP connection between two instances by recording data, possibly modifying this data, and then by inserting the modified packages back into the network. The attacker can act as one of the involved parties or it can act as an intermediary between both parties (Chen, Guo, Zheng, & Li, 2009). Encryption mechanisms such as TLS provide origin authentication which makes this attack scenario more difficult if the encryption mechanism is set up and configured properly and no zero-day vulnerabilities have been used to break the implementation of TLS (Whitman & Mattord, 2011).

There are two types of MITM attacks: active and passive. A passive attack means eavesdropping, staying unnoticed in the network and collecting data between two

instances and then using that information for different purposes. An active attack also uses eavesdropping as an information gathering method, but it also includes the behavior of infecting modified data back to the network. Interfering is the key in the active types of attack, but staying out of sight and unnoticed is a primary target for both of these types which makes hacker intrusion even harder to detect (Kaufman, Perlman, & Speciner, 2002).

Although TLS is considered to be a valid protocol for securing TCP traffic, it still has its weaknesses. One particular attack scenario is related to the handshake operation, which exchanges keys between the client and the server thus establishing a secure connection. When the server is not properly verified, an attacker can act as a middle man in? the key exchange while impersonating a server to? the user. Then the attacker is able to steal the user's credentials and act as a valid user for the legitimate server by using the stolen credentials (Karapanos & Capkun, 2014).

Man-in-the-Middle is relevant to this thesis because the WebSocket protocol uses the TCP protocol for tunnelling the traffic from the WebSocket server to the client. This means that the vulnerabilities that threaten the TCP protocol and its safety are also related to the WebSocket protocol.

4.3.4. Payload Injection Attacks

When the WebSocket connection is established with the server, the proper data exchange begins. The data consists of frames, which have two different fields: header and payload. The payload carries the user input from a client to the server and vice versa. The user-provided input should be validated, since stored cross-site scripting attacks and SQL injections could be performed via WebSockets like any other communication protocol.

Zhang (2012) stated that organizations should implement *deep content inspection (DCI)* mechanisms for preventing attacks on the services that HTML5 offers, including WebSockets. DCI is a mechanism that surveys the traffic by continuously detecting potential harm-causing attacks or schemes while blocking the undesired traffic from the network. It thus increases the security of the used devices and the infrastructure of the organizations as a whole. He also mentions that one of the key benefits of the WebSockets is the lower overhead compared to HTTP generated traffic. That could benefit the whole organization when large files can be sent from one instance to another with a significantly lower overhead. That said, one could assume that adding a DCI mechanism to a WebSocket communication platform that continuously surveys and detects traffic could lead to increased traffic and much higher overhead than the original HTTP traffic and therefore lose the advantages that the WebSockets could offer (Zhang, 2012).

4.3.5. Cache Poisoning Attack

The WebSocket protocol was developed to handle a certain need in the current web development era. There is a need for a protocol which provides lightweight and cleartext communication that is also reliable and which works concurrently with the existing protocols that web applications rely on, such as HTTP. However, there is a problem that is related to the plaintext version of WebSockets, *ws://*. Transparent proxies are network intermediaries or nodes whose purpose is to channel traffic effortlessly from one endpoint to another. Transparent proxies are also called “intercepting” proxies, which, according to Huang et al., is a better description of them (Huang, Chen, & Barth, 2011).

A WebSocket handshake is basically an HTTP request to *Upgrade* the connection to a WebSocket based connection. Huang et al. discovered that many of these transparent proxies didn't handle the *Upgrade*-mechanism correctly, which leads to a potential security vulnerability. The *Upgrade*-mechanism consists of two different fields: a *connection* header which includes only the string “Upgrade”, and an *Upgrade*-header which has the name of the protocol to which the client wants to switch. The problem lies with the *Upgrade*-header in itself, which – according to Huang et al. - is not so widely applied and used. This means that the WebSocket handshake request appears to be similar with a *Post*-request, which also uses the *Upgrade*-method upon postage (Huang et al., 2011; Koch, 2013).

This implementation could be exploited by the attacker by setting up a WebSocket server and then a connection to the desired proxy server using WebSockets. The proxy handles the modified payload from the attacker's WebSocket server as a regular HTTP request, which attempts to acquire a certain JavaScript file from another domain. The headers of the request indicate that the malicious file should be cached for several days in the proxy server. This way anyone who is trying to acquire that certain file routing through that infected proxy will automatically load the malicious JavaScript instead of the original, non-malicious file that they are trying to acquire (Koch, 2013).

A proposition for fixing this vulnerability was issued, and a variant of that proposition was adapted by the WebSocket standard. The masking of the payload is now done by using XOR cipher that prevents the attacker from controlling the bytes sent on the wire. The XOR-operation is done with random key consisting of 4 bytes. For every outgoing frame the browser generates a new key, that is then sent within the frame. The receiver is then able to unmask it again using the XOR the second time.

4.3.6. Plaintext Communication

Plaintext – or “cleartext” - is a message sent from the user to the receiver without any applied cryptography to render the text unreadable without the correct keys to encrypt the data. Plaintext in the WebSocket based communication means that the WebSocket is used without TLS to encrypt the traffic between the client and the server. A normal, unencrypted, plaintext connection can be established by using the standard WebSocket protocol identifier ws://. TLS enabled communication can be established by using wss:// (Wang et al., 2013).

Plaintext communication is related to man-in-the-middle attacks, since plaintext communication is readable and modifiable to third parties. Modified communication might go unnoticed from the original communication between two users. In order to keep the communication more secure and the information that is exchanged private, TLS should be enabled. Depending on the scale the web service, the TLS encrypted traffic might cause some overhead to the service and in some cases it might be advisable not to use encryption. If e.g. authenticating a user is done via WebSocket, the TLS should be enabled to protect the user's credentials for packet sniffing.

4.3.7. WebSocket Botnet

Michael Schmidt displayed a network topology in 2011, which presents a botnet that consists of user agents that act as zombies[3]. As long as the malicious site is open in the user agent's browser the connection remains and the attacker is able to use the connected user agent's connection to perform DOS-attacks to different destinations, for example.

[3] A zombie computer is a computer infected with a virus or Trojan that allows a remote attacker to run malicious code on the infected machine remotely (Phillips, 2005).

Cross-origin requests can be performed as well, which would affect the user's online identity (Schmidt, 2011).

Web Workers, which allows the web applications to run scripts on the user's computer to safe computing resources on the server side, could be used to run tasks on the user's computer without the user even realizing it. Depending on the size, this could be very useful for attackers to safe their resources and do some computation on the hijacked user agents (Hickson, 2012).

4.4. Defense Mechanisms for WebSocket Attack Vectors

Enabling encryption on the connection makes the frame exchange secure but it does not make the application in itself secure. Keeping the focus on the security of the application during the developmental process increases the reliability of the application. OWASP has published a reference guide for a secure web application development that can be adapted to software development life cycle (OWASP, 2010).

Payload filtering is also something that should be done to secure the application that uses WebSockets for its data transfer. For example, if parts of the user-provided input are used as a SQL database query, escaping that input is needed to avoid an SQL injection where the attacker might be able to steal information that should be inaccessible (IETF, 2011b).

OWASP suggests in its wiki page titled *Input Validation Cheat Sheet* that whitelist input validation is much more effective than the character blacklisting. A blacklisting input consists of bad keywords (for example <*script*> tag) and bad characters (such as ' and <), but an attacker can circumvent these quite easily – depending on how thorough the list of the blocked characters is. Nevertheless, whitelisting and forming regular expressions is a more dynamic and effective approach for validating the user-provided input but that method comes with difficulties, too. For instance, if the application is a chat, there is not much point in whitelisting basically every word in the most common dictionaries. The handling of that type of dictionary could cause significant overhead on the server side. That said, developing a chat application that also handles SQL databases is an unlikely need for anyone. In that case, blacklisting certain characters or words might be a more feasible thing to do than whitelisting (Wichers, 2014).

4.4.1. The Problem with the Origin

The risk for SQL injections or other types of attack is considerably higher because of the way in which the WebSocket protocol handles different origins for the client. WebSockets are to be used by different scripts on a web page that need an efficient frame exchange with the server. However, regardless of the intention, the WebSocket protocol can be used outside of a web page, even outside of the browser (Erkkilä, 2012). This is generally a good thing, because it makes the WebSocket protocol more dynamic and it can be used in normal – and mobile – applications too. In the IETF specification of WebSockets, the model defaults to open, which means that the WebSocket server blindly accepts the origin that the client offers. Depending on the client, the *Origin*-header can be manually manipulated to point to a non-existent web page, and the server happily accepts it, if there is no *Origin*-verification implemented (Hickson, 2014).

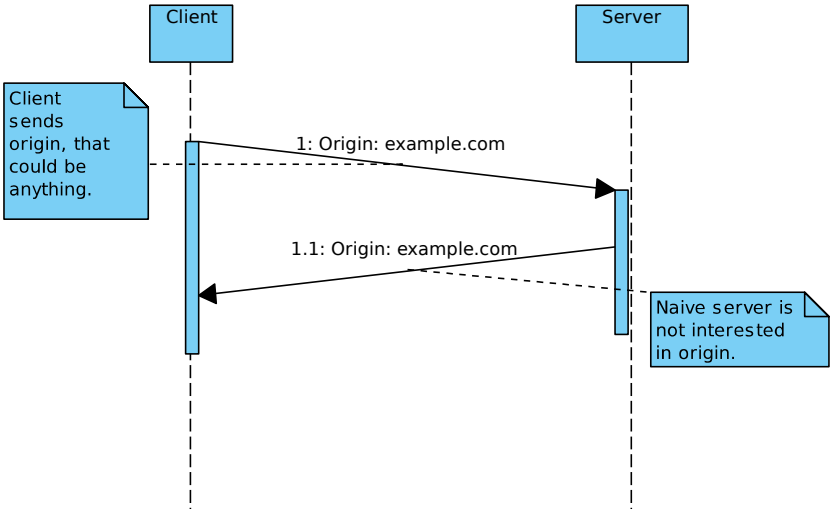

Figure 7. A sequence diagram displaying the current *Origin*-header handling in WebSocket protocol.

Ian Hickson responded to an email query about an interview which he gave in 2013 and in which he stated that the original model of the WebSockets was designed to be more secure. He elaborated that one of the aspects that were changed during the specification process was the handling of the origin. In the original model the server was the one dictating which origins were allowed. In the current model the WebSocket implementation defaults to open, which allows all origins to pass on the handshake process. It is the developers' task to enforce origin policy on the server instead of allowing it to remain as the default option (Hickson, 2014).

In Figure 7 the original scheme of origin handling is presented. As implemented in this fashion, the client is responsible for matching the origin that the server requires. This increases the level of security since the connections with mismatching origins are dropped, thus restricting the access to the intended application. Allowing connections from differentiating origins increases the risk for unintended use or hacking attempts.

Figure 8 displays the more secure origin handling by the server than the original scheme.

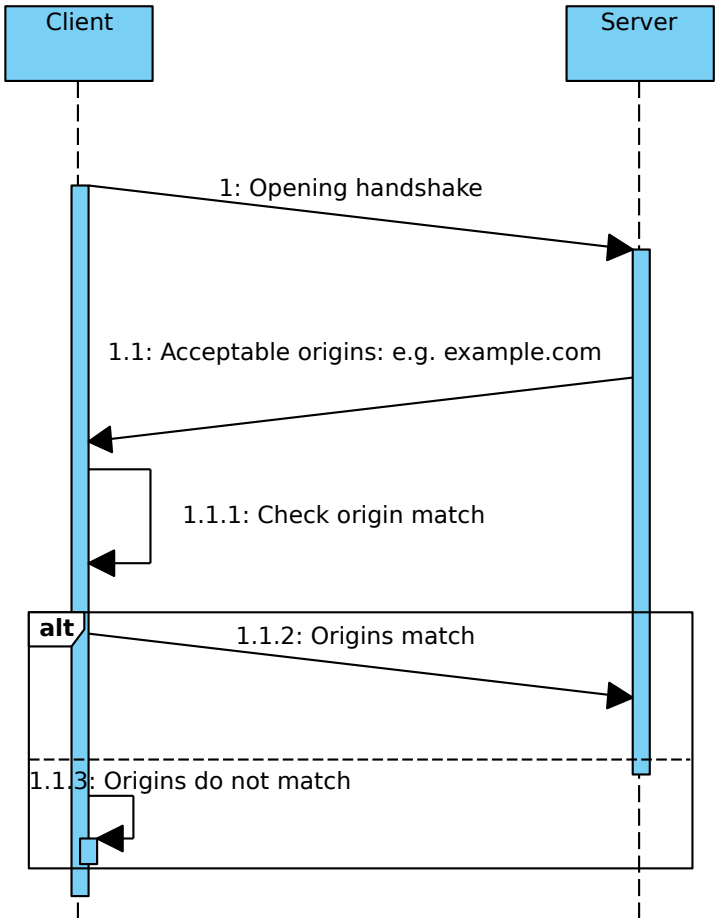

Figure 8. Suggested *Origin*-header management by the server.

Granted, even the suggested origin handling does not make the server instance foolproof. The attacker could do two different handshake attempts to get the origin that the server requires, then doing the same handshake operation with a fixed value of the received origin. This could elaborated as the following sequence:

1. Attacker sends a handshake request for the server without *Origin*-value.
2. The server responds with the required *Origin*-value.
3. The attacker drops the original handshake request, and then...
4. … sends a different handshake request, that has the *Origin*-value the server responded with the first time.
5. The server accepts the connection, since the *Origin* is the one it requires.

This is a trivial obstacle for the motivated attacker to tackle, but it increases the level of difficulty to create a valid connection from an alternative origin.

The other option for blocking unwanted origins of granted access is to blacklist certain addresses. However, this method is not as effective and restrictive as whitelisting and

therefore not advisable. For the most usable option and the greatest performance overall the application should use fixed values for the origin if it is possible.

4.4.2. Defending Against (D)DOS-attacks

Denial-of-Service (DOS) attacks are subject mostly to the availability aspect of the C.I.A triangle. A DOS-attack most commonly aims to render a network instance or service unavailable for the authorized and targeted users, thus harming the quality of the service and quite possibly causing financial damage. These attacks are the most difficult of the most common attack types to trace back and prevent from happening (Jain & Singh, 2012).

A DOS-attack regularly comes from one source. The DOS-attack that comes from several different sources is called the *Distributed-Denial-of-Service (Ddos)*. Its range can be from a few different sources to thousands of different illegally acquired *zombie*-machines. The sheer size of the volume and the spread of the sources makes it much more difficult to defend against (Jain & Singh, 2012).

In the context of WebSockets this means that the attacker makes the WebSocket server unavailable for the intended users by for instance flooding the server with multiple connection attempts from multiple connections (with different origins). The way the server is implemented dictates what the effects of the (D)DOS-attack are towards WebSockets, but the volume of the connection attempts dictates how many resources have to be gathered for the attack to be successful. The higher the load attempted to require, the less is the need for additional *zombie*-machines. Therefore the attacker might attempt to make the server perform multiple compute-intensive tasks (such as rendering PDFs or downloading large video-files) to exhaust the server (Akamai, 2014).

Three selected WebSocket server implementations were put to a test on how they handle arbitrarily large numbers of concurrent WebSocket connections from one origin. Table 5 shows these results.

Table 5. WebSocket implementations that were tested against simple DOS-attack, where numerous connection attempts were made from one source.

Name	**Platform**	**Connection attempts**	**Successful connections**
nodejs-websocket	Node.js	40,000	40,000
libwebsocket	C library	40,000	10,000 - 28,000
AutobahnPython	Python/twisted	40,000	10,000 – 36,000

As argued later in this chapter, servers should limit the WebSocket connections from one source. For example, a *nodejs-websocket* has this properly implemented. During the testing it successfully closed the open connection before opening another from the same source. This is a reasonable and justifiable decision because limiting open connections from one source makes a successful DOS-attack less likely. In some cases, the *nodejs-websocket* dropped the connection earlier than estimated – before the 40,000 threshold – but the server did not crash. Thus it is still usable and available for intended users.

In this test the *libwebsocket* did not perform as well as the *nodejs-websocket.* The results ranged from about 10,000 to approximately 28,000. When the connection attempts failed, the program opening connections were restarted, and the results were always significantly lower than in the previous test. This could indicate that the *libwebsockets* allocate memory to each connection, and when a certain number of connections have been opened the library runs out of memory and then fails to allocate memory to new connections. However, these testing servers were run on a virtual machine that had the default 512 megabytes of memory allocated to the machine. The results did not change when the memory allocation was increased to two gigabytes of memory. The program (*libwebsocket-test-echo.c)* does not give out any error message whatsoever, so debugging the error could require the use of a separate memory debugging tool, e.g. Valgrind.

The third server implementation tested is *AutobahnPython,* an open source subproject of *Autobahn,* which is a real-time framework for web applications. It additionally uses the *twisted*-framework for some functionalities. When opening new connections it works similarly as *nodejs-websocket,* and upon a new connection attempt it closes the already existing one. However, *AutobahnPython* does not reach the selected threshold of 40,000 connection attempts in any of the test attempts. The reason for that might be similar to that of the *libwebsockets,* where the allocated memory runs out. Whatever the ultimate reason might be, the actions are erratic in the *AutobahnPython* and the *libwebsockets* and modifications on how they handle multiple simultaneous connection attempts from one origin are advisable.

The aim of the defense mechanisms for the (D)DOS-attacks is to block the extensive requests before they reach the critical infrastructure. Even in a situation where the origins are checked in the firewall before they enter the server's scope can exhaust the firewall, since the origin checking requires computing power, too. Again, it is still related to the volume of the attack.

The WebSocket protocol in itself does not have any safety mechanisms for the DOS-attacks. There are, however, a few guidelines which a developer might want to follow in order to mitigate the risk of a successful attack:

1. *Limit the number of requests which a user can make during a certain time interval.* This depends highly on the goal of the application and the limit should be considered individually for every application.

2. *Limit the user's ability to perform resource intensive tasks.*

3. *Implement a way for limiting the number of WebSocket connections per user.* There is no limit built in WebSockets that limits the number of connections that a sole user can create. There has been some discussion about this aspect among the developers of WebKit and this functionality may be implemented somewhere in the future (Proskuryakov, 2009).

4. *If possible, implement a separate, physical firewall on the network that filters out extensive traffic.* This way it does not reach the server, and mitigates the risk of data theft and resource hogging of the server.

5. *Use WAF for blocking extensive requests.* Many WAFs have methods for detecting possible (D)DOS-attacks, so use it to mitigate the possibility of a successful attack.

The phrase *"there is no silver bullet"* can be applied to many of the previous attack vectors, but this is especially true when talking about defending against (D)DOS-attacks. A proper defense tactic against them is a sum of many different actions that, when applied, lessen the severity of an attack – hopefully to the point that it requires so many resources from the attacker that it is not meaningful to continue the attack. To summarize, it all comes down to the (1) motivation, (2) skill, and (3) the resources of the attacker.

4.4.3. The Problem with Error Handling

In case something happens that causes an error, there ought to be proper error handling in place for both the client and the server. The errors that have been prepared for can deal with such cases gracefully without interrupting the application or closing the connection unnecessarily. Situations where errors are bound to happen from time to time should be covered with proper planning as to what to do in these situations. This way it is possible to minimize user harm, costs or other factors that might prove costly to the reliability of the application.

The WebSocket API has four different built-in events that have their own methods: *open, close, message* and *error.* Since the WebSocket protocol is an event-driven protocol that follows the asynchronous programming model, the connection is listening to these events and act accordingly when such an event has happened. When the *error*-method is called, the WebSocket connection is bound to close as a result of that (Wang et al., 2013). This is a good place to implement reconnecting rules to the server in certain situations to provide better availability and user experience. The following code sample presents how the error-events can be dealt with:

```
WebSocket.onerror = function(evt) {
   console.log("An error occurred: " + evt.data);
   handleError(evt);
}
```

In the sample, the *handleError(evt)* is a function that takes the *evt*-data as a parameter and works out from there how to continue from there. When the application is under development it is advisable to collect statistics from certain errors and determine how often these errors occur and how they can be mitigated.

This aspect of the WebSocket implementation is related to the *Availabilit* side of the previously described *C.I.A triangle*.

4.4.4. The Problem with Encoding

The text that is readable for humans does not look the same when it is being transferred from one computer to another. In order to transfer a text between computers the text needs to be encoded, which means that it is translated at the sender's end and then translated again at that of the receiver (Techterms.com, 2010). This is done by using *character encoding.* The most popular types of encoding are ASCII and Unicode, and the one which the WebSocket protocol requires is UTF-8, a subset of the Unicode family. All payloads must be UTF-8 encoded when data type is defined as *Text* for the WebSocket protocol to function properly. None of the other encoding standards are supported in WebSockets (IETF, 2011b).

The support for UTF-8 encoding in the WebSocket protocol is specified in the documentation. The encoding in question is a valuable piece on how the frames in the WebSocket protocol work. In Chapter 3.1.6 *Frames in WebSocket Protocol* the

structure of the frame was explained. The section that is closely related to this chapter is the section where the payload data as well as the length of the payload is located (*Payload len*, 7 bits). Implementing the same character encoding on both ends enables the length of the payload to be the same on both the client's and the server's side. The length is used to determine where the frame ends so it is a key factor when the frame is received. When the masking is enabled, the payload is masked using the masking-key. If the payload is longer than the *Payload len*, then the unmasking in the server could lead to a false unmasking. Applications developed that are implementing the WebSocket protocol but do not comply with this and use different encoding may lead to security bugs or loss of data (IETF, 2011b).

The testing revealed that some of the WebSocket server implementations failed to parse the different encodings correctly, some of them crashing and dropping the established connection altogether.

The targeted test server was the echoing server, implemented by using the *nodejs-websocket* library. Figure 10 shows an example of what happens when the character encoding breaks in the application, which then leads to the breaking of the character encoding on the application and on the terminal itself.

Figure 9. An example of a character encoding overflow on a *nodejs-websocket* echo-server. The server fails to encode characters properly so it breaks the encoding of the server and renders the text in console unreadable.

Table 6 shows the numbers acquired from sending 200 frames in three different encodings, using anomalous data fuzzed with radamsa. The test cases were performed three different times so the numbers presented in table 9 represent the average values of those tests. None of the test cases had results that were not along the lines of the other tests so the numbers are quite compliant with each other. The number of the client encoding errors is also presented in the same table but the numbers have naturally nothing to do with the server under the testing.

Table 6. Results for nodejs-websocket echoing server.

	UTF-8	**ISO-8859-6**	**ISO-8859-9**	**TOTAL**
Success	140	102	110	352
Errors	0.33	14	90	104.33
Client Encoding error	*59.67*	*84*	*0*	*143.67*

The testing showed that although the server handled anomalous data fairly well, there were still clear signs that the *nodejs-websocket* library did not handle correctly the data that it was not supposed to receive. The problem may lie in the encoding method that it uses to check whether the payload is UTF-8 compliant or not. Nevertheless, it is an error that should not happen. The potential of this bug for the attacker to exploit is difficult to determine accurately, but it shows that the server could be vulnerable for such attacks as take advantage of the problems that the libraries have with encoding.

4.5. Summary

It is sometimes difficult to see the differences between the WebSocket protocol in itself and the libraries and software which *implement* the WebSocket protocol. The protocol specification dictates certain instances as to how to deal with security issues, such as the handling of the *Origin* header and encoding. The protocol specification takes an approach of giving the developer the opportunity to develop applications that are quite free to extend the security specifically in the applications under development. The specification makes compromises with usability and security when it states that the client provides the origin that the server then approves. This makes it easier to develop applications that do not rely so heavily on security or where the origin really has no effect on how the application is used.

Another aspect that is related to the *Origin* headers is the number of simultaneous connects from one source. The testing revealed that two of the three tested server implementations clearly limit the number of concurrent connections, which is not currently stated in the protocol specification. There might be available some applications where multiple WebSocket connections from one source are justified, but it is relatively difficult to think of a good reason for that.

When the application that uses WebSocket protocol uses *wss://* as the means of connecting to the server and a *Web Application Firewall*, some of the most critical issues are dealt with. The origin is properly authenticated (for communicating with the server, but it is not *authorized)* and the risk for a *man-in-the-middle* attack is mitigated to some extent. Taking care of the error handling aspect in the server is another important factor when developing an application for the clients to use.

5. Discussion

The WebSocket protocol is one of the many networking protocols that have been developed for the different types of networks. WebSockets were developed to answer the need for a lightweight, reliable, real-time and two-way communication in the web applications. It was meant to replace the older solutions, which mostly took advantage of the even older protocols not designed with the modern internet in mind. These older solutions caused unwanted overhead and had generally unpreventable drawbacks since the protocols like HTTP were not meant to be used in that manner. These *polling* methods worked to some extent, but it was clear that there was a need for a standard that could provide the needed functionalities for modern web applications.

It is clear that the WebSocket protocol was not developed with security as the driving factor. The aforementioned aspects are the most motivating needs but the protocol specification in itself shows that being secure and safe is one of the goals of the WebSocket protocol. Indeed, there are some aspects that could be considered to be compromises between the ease of use and the security, for example:

- The handling of the *Origin* header, where the server approves the origin that the client provides, not the other way around.
- Plaintext communication is still supported by the WebSocket protocol.
- The number of concurrent WebSocket connections from one origin is not limited.

These could all be considered to be favors towards development instead of security. One could argue that these particular things were nudges towards the developers to start developing an application that would take advantage of WebSockets. Because it is a new protocol in the market, developer adaptation is the key for the protocol to succeed. IF the WebSocket protocol were to be developed with security being the first-hand objective, the adaptation rate could be even lower than what it was in 2013, when 0.15% of the most visited websites used WebSockets on their landing page. The testing showed that even if this is the case with the protocol, there is nothing to prevent the developers from adapting additional security measurements in their application.

In the testing procedure it became clear that some of the existing WebSocket server implementations had adapted additional security mechanisms that were not provided by the protocol in itself. Limiting the number of connections was adapted in two of the three tested libraries, as it is suggested as a *best practice* in this thesis. Instead of defaulting to an unlimited number of connections, it would be advisable for the protocol to limit the number of concurrent connections as default?. Implementing this on the protocol level could be quite challenging, and it would require the handling of several headers instead of only one that identifies the user and then allows the user one? WebSocket connection. This is challenging because:

- The value of the *Origin* header is the same for all the clients reaching from the same web application.
- *Sec-WebSocket-Key* is calculated separately for every connection.

As a result, neither of these headers can be used as a connection identifier that restricts the number of simultaneous connections to – for example – one. Adding a further header to the handshake process might be the easiest solution, since the framing of the WebSocket protocol has reserved bits for future additions (*RSV1*, *RSV2*, and *RSV3*). For example, one of these bits could change from 0 to 1 if an existing WebSocket connection is already established. The handling of this implementation requires maneuvering on how the applications are developed, as the value of the bit which indicates that the WebSocket connection is already open needs to be stored somewhere, for example in a variable in the script that connects to the server. If the client detects that upon connecting to the server there is already a connection established, it drops the connection attempt or the existing connection.

Since this could be a laborious task for the protocol specification, we will probably not see this kind of change in the near future, at least not in this form. After all, the most popular server implementations seem to comply with the general recommendation to restrict the number of connections coming from one origin. Whether or not this is effective enough can be argued to some extent, but in general it is an advantage that the servers limit the number of concurrent connections from one source.

It is generally up to the developers or the developing organizations to decide what kind of security practices are needed when developing WebSocket applications. If the application under development is e.g. a game which relies more on the fastness and low latency of the protocol and which provides a platform for multiple connection attempts at the same time, added security features could only harm the development process and the playing experience. On the other hand, if the WebSocket connection is used to provide real-time online support for the customers that have authenticated to the service (for example in a banking application), added security features might be needed. As stated earlier in the *Background* chapter, if the connection handshake is done via TLS, the established WebSocket connection needs to use TLS as well to encrypt the WebSocket traffic. In addition to this, an extra frame payload verification might be needed.

It all comes down to the procedure called *risk assessment.* During this procedure, the risks that could affect e.g. the application, end-user data and critical infrastructure are identified and rated accordingly. These ratings are then used to add security features, to encrypt data, and to add other verification and/or authentication methods. If the application under development seems to require additional security measurements, the procedure would have to include a process where the risks are being mitigated by security features or actions that make that the possibility of an attack less probable.

6. Conclusion

A raw connection from the client to the server (or vice versa) can be established with WebSockets. After a successful handshake some kind of tunnel is created where the data of the WebSockets is then exchanged. The frames of the WebSockets carry only the bare minimum that is required for the transfer. This, however, makes the situation interesting from the security point of view. Security often means added overhead, increased costs and very strict implementation rules for securing the data. Since the WebSocket protocol aims to be a highly usable and modifiable protocol, the security solutions should be low overhead, easily implemented and effective against targeted attacks.

As mentioned earlier, the protocol in itself and the libraries implementing the WebSocket protocol are two different things. When this is the case, the protocol specification needs to be as precise as possible in order to address as many aspects and scenarios as possible, whether they are intended attacks or the results of misconfiguration. This is one of the things where a proper error handling is the key. The WebSocket protocol specification does not dictate error management guidelines as strictly as one might assume. Instead of the sections in the documentation where they could say “in this situation the server responds with this error code”, the documentation adopts a more neutral role, leaving it up to the developer to decide how to specifically deal with the situation. The protocol specification is filled with sentences as “...in case of an error, the server drops the connection”. This kind of vaguer approach makes the error management procedures in the various libraries very different.

The same applies to how the different WebSocket implementing libraries handle origins. The testing revealed that some of the libraries developed implement separate guidelines in addition to the official specification documents. These guidelines e.g. suggest that the number of concurrent WebSocket connections from one origin should be limited to one to mitigate the risk of a DOS-attack. It is difficult to think of a situation where multiple WebSocket connections from one origin could be needed instead of only one. These guidelines should be added to the official documentation as requirements for making the WebSocket implementations more unified and more secure. If there is a need for the developer to create an application that uses WebSockets with multiple connections from one origin, it should also be allowed, but the protocol specification should default to one connection per origin.

As time progresses and the trends change it is refreshing to notice that older solutions that are not properly suitable for certain problems are disappearing from the application development. Since WebSocket is a rather new technology, it is constantly implemented to new or existing applications. Its updated adaptation rate, presented in Chapter 4.2., suggests that WebSockets are being adapted at an increasing rate and that the number of encrypted connections exceeds the number of unencrypted connections. This suggests that WebSockets are being used in places where added security and encryption is required. There are a few situations where added security might actually prove to be harmful – for example if an online shop fetches catalog information from the server using WebSockets, encryption might be unnecessary. If the user is logged on to the application in question, the wss:// is definitely needed as the attacker might be able to hijack the authenticated session. Unnecessary encryption causes unnecessary overhead, so it is advisable to avoid encryption or added overhead where it is regarded as unnecessary.

All things considered, the WebSocket protocol can be used as a secure, efficient and low overhead solution for the web applications that need real-time and two-way communication. Security is a very burdensome task for a new communication protocol to be successful since times have changed from the times when the now antiquated TCP and even HTTP were invented. Security was a very different issue in those days – even though TCP and HTTP are approximately 20 years apart from each other – and from modern-day security requirements. HTTP was (originally) used in an unwanted way to provide the same type of services as the WebSocket protocol. The WebSocket protocol can ease the transition to the new era of web applications that provide real-time communication without using the HTTP protocol in a way it was not intended to be used.

From the security perspective it would be interesting to see research into the WebSocket protocol in HTTP/2 in the future. An update to the specification is under way in IETF to standardize a WebSocket connection in HTTP/2 to work on top of HTTP/2 data stream rather than TCP. This should not affect – at least not at this moment – how the WebSocket protocol behaves on the utmost level, but changing the underlying technique could reveal some errors from the security perspective.

References

Akamai. (2014). *A guide to multi-layered web security*. Retrieved from http://www.akamai.com/dl/akamai/akamai-ebook-guide-to-multi-layered-web-security.pdf

Alexa.com. (2015). Alexa Internet - About Us. Retrieved February 25, 2015, from http://www.alexa.com/about

Alexis Deveria. (2015). Can I Use WebSockets? Retrieved October 1, 2014, from http://caniuse.com/#feat=websockets

Brooks, F. P. J. (1986). No Silver Bullet: Essence and Accidents of Software Engineering. Retrieved January 26, 2015, from http://www.cs.nott.ac.uk/~cah/G51ISS/Documents/NoSilverBullet.html

Character Sets. (2013). Retrieved February 27, 2015, from http://www.iana.org/assignments/character-sets/character-sets.xhtml

Chen, Z., Guo, S., Zheng, K., & Li, H. (2009). Research on Man-in-the-Middle Denial of Service Attack in SIP VoIP. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing* (Vol. 2, pp. 263–266). IEEE. http://doi.org/10.1109/NSWCTC.2009.326

Codenomicon. (2014). Heartbleed Bug. Retrieved September 19, 2014, from http://heartbleed.com/

Dehl, K., Greene, R., Bankston, K., & Morgus, R. (2014). *Surveillance Cost: The NSA's Impact on the Economy, Internet Freedom and Cybersecurity*. Retrieved from http://newamerica.org/downloads/Surveilance_Costs_Final.pdf

Dierks, T. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. Retrieved from https://tools.ietf.org/html/rfc5246

Dowd, M., McDonald, J., & Schuh, J. (2006). *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* (Vol. 20). Pearson Education.

Erkkilä, J. (2012). WebSocket Security Analysis.

Fielding, R., Irvine, U., Gettys, J., Compaq/W3C, Mogul, J. C., Compag, … Berners-Lee, T. (1999). Hypertext Transfer Protocol - HTTP/1.1. Retrieved October 2, 2014, from http://www.ietf.org/rfc/rfc2616.pdf

Forbes. (2014). The World's Biggest Public Companies List. Retrieved September 23, 2014, from http://www.forbes.com/global2000/list/

Google. (2014). AngularJS: Frequently Asked Questions. Retrieved September 23, 2014, from https://docs.angularjs.org/misc/faq

Gordon, L. a., Loeb, M. P., & Zhou, L. (2011). The impact of information security breaches: Has there been a downward shift in costs? *Journal of Computer Security*, *19*(1), 33–56. http://doi.org/10.3233/JCS-2009-0398

Hevner, A. R. von, March, S. T., Park, J., & Sudha, R. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105.

Hickson, I. (2012). *Web Workers*. Retrieved from http://www.w3.org/TR/2012/CR-workers-20120501/

Hickson, I. (2014). personal communication.

Hirano, Y. (2014). *WebSocket over HTTP/2*. Retrieved from http://tools.ietf.org/html/draft-hirano-httpbis-websocket-over-http2-01

Huang, L., Chen, E., & Barth, A. (2011). Talking to yourself for fun and profit. *Proceedings of W2SP*, 1–11. Retrieved from https://www.truststc.org/pubs/840/websocket.pdf

Hydara, I., Sultan, A. B. M., Zulzalil, H., & Admodisastro, N. (2013). Current state of research on cross-site scripting (XSS) - A systematic literature review. *Information and Software Technology*. http://doi.org/10.1016/j.infsof.2014.07.010

IETF. (2011a). *RFC 6265 - HTTP State Management Mechanism*. Berkeley. Retrieved from http://tools.ietf.org/pdf/rfc6265.pdf

IETF. (2011b). *The WebSocket Protocol*.

IETF. (2012). The WebSocket protocol RFC6455 History.

Jain, A., & Singh, A. K. (2012). Distributed Denial of Service (DDoS) Attacks - Classification and Implications. *Journal of Informations and Operations Management*, *3*(1), 136–140. Retrieved from http://www.bioinfopublication.org/files/articles/3_1_30_JIOM.pdf

Järvinen, P. (2008). Mapping Research Questions to Research Methods. *Advances in Information Systems Research*, *274*, 29–41.

Kaazing.com. (2014). What is WebSocket? » Event-Driven Web Communications. Retrieved October 2, 2014, from http://kaazing.com/websocket/

Karapanos, N., & Capkun, S. (2014). On the Effective Prevention of TLS Man-in-the-Middle Attacks in Web Applications. San Diego, CA: USENIX. Retrieved from https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/karapanos

Karlsson, M. (2014). AngularJS Sandbox bypass. Retrieved September 23, 2014, from http://avlidienbrunn.se/angular.txt

Kaufman, C., Perlman, R., & Speciner, M. (2002). Active vs. Passive Attacks. In *Network Security: Private Communication in a Public World* (2nd. ed.). Retrieved from http://www.informit.com/articles/article.aspx?p=27375&seqNum=7

Kizza, J. M. (2013). *Guide to Computer Network Security*. Springer Science & Business Media. Retrieved from http://www.google.fi/books?hl=en&lr=&id=sbA_AAAAQBAJ&pgis=1

Klein, A. (2005). DOM Based Cross Site Scripting or XSS of the Third Kind]. In *Web Application Security Consortium*. Retrieved from http://www.webappsec.org/projects/articles/071105.shtml

Koch, R. (2013). *On WebSockets in penetration testing*. Vienna University of Technology. Retrieved from http://packetstorm.wowhacker.com/papers/attack/AC07815487.pdf

Krebs, B. (20AD). Home Depot: 56M Cards Impacted, Malware Contained — Krebs on Security. Retrieved September 19, 2014, from http://krebsonsecurity.com/2014/09/home-depot-56m-cards-impacted-malware-contained/

Lawson, B. (2013). Interview with Ian Hickson, HTML editor. Retrieved October 29, 2014, from http://html5doctor.com/interview-with-ian-hickson-html-editor/

Lekies, S., Stock, B., & Johns, M. (2013). 25 million flows later. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* (pp. 1193–1204). New York, New York, USA: ACM Press. http://doi.org/10.1145/2508859.2516703

Loreto, S., Saint-Andre, P., Salsano, S., & Wilkins, G. (2011). 6202: RFC Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. Retrieved October 30, 2014, from http://www.hjp.at/doc/rfc/rfc6202.html

Lubbers, P., Albers, B., & Salim, F. (2010). Pro HTML5 Programming. Retrieved from http://link.springer.com/content/pdf/10.1007/978-1-4302-3865-2.pdf

Marlinspike, M. (2009). SSLStrip. *Thoughtcrime Labs.[Online]*. Retrieved from http://scholar.google.fi.pc124152.oulu.fi:8080/scholar?cluster=14495761226387244782&hl=en&as_sdt=2005&sciodt=0,5#0

Meucci, M., & Muller, A. (2014). *OWASP Testing Guide 4.0*. Retrieved from https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf

Mozilla Developer Network. (2014). WebSockets. Retrieved October 30, 2014, from https://developer.mozilla.org/en/docs/WebSockets

ONF. (2015a). OpenFlow-enabled SDN and Network Functions Virtualization. Retrieved February 27, 2015, from https://www.opennetworking.org/solution-brief-openflow-enabled-sdn-and-network-functions-virtualization

ONF. (2015b). Software-Defined Networking (SDN) Definition. Retrieved February 27, 2015, from https://www.opennetworking.org/sdn-resources/sdn-definition

OUSPG. (2015). radamsa. OUSPG. Retrieved from https://github.com/ouspg/radamsa

OWASP. (2010). OWASP SCP Quick Reference Guide v2. Retrieved January 14, 2015, from https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf

OWASP. (2014a). Cross-site Scripting (XSS). Retrieved October 10, 2014, from https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

OWASP. (2014b). Web Application Firewall. Retrieved October 6, 2014, from https://www.owasp.org/index.php/Web_Application_Firewall

Pant, R. (2014). A Cumulative Security Metric for an Information Network. *International Journal of Application or Innovation in Engineering & Management*, *3*(4), 221–224. Retrieved from http://www.ijaiem.org/volume3issue4/IJAIEM-2014-04-29-075.pdf

Ponemon Institute. (2014). Ponemon 2014 SSH Security Vulnerability Report.

Proskuryakov, A. (2009). Bug 32246: Multiple connection attempts to a WebSocket server should not be allowed. Retrieved February 25, 2015, from https://bugs.webkit.org/show_bug.cgi?id=32246#c4

Ronacher, A. (2012). Websockets 101. Retrieved November 24, 2014, from http://lucumr.pocoo.org/2012/9/24/websockets-101/

Santander, M. (2010). *Measuring effectiveness in Information Security Controls*. Retrieved from http://www.sans.org/reading-room/whitepapers/basics/measuring-effectiveness-information-security-controls-33398

Schmidt, M. (2011). HTML5 Web Security 1.0. Retrieved from http://media.hacking-lab.com/hlnews/HTML5_Web_Security_v1.0.pdf

Schneider, C. (2013). Cross-Site WebSocket Hijacking (CSWSH). Christian Schneider. Retrieved from http://www.christian-schneider.net/CrossSiteWebSocketHijacking.html

Schneier, B. (2007). Schneier on Security: In Praise of Security Theater. Retrieved October 9, 2014, from https://www.schneier.com/essays/archives/2007/01/in_praise_of_securit.html

Schneier, B. (2008). Psychology of Security. In *Progress in Cryptology–AFRICACRYPT 2008* (pp. 50–79). Springer Berlin Heidelberg. Retrieved from https://www.schneier.com/paper-psychology-of-security.pdf

Spears, J. L. (2006). Defining Information Security. Las Vegas, Nevada. Retrieved from http://www.isy.vcu.edu/~gdhillon/Old2/Old/secconf/pdfs/11.pdf

Takanen, A., Demott, J., & Miller, C. (2008). *Fuzzing for software security testing and quality assurance*.

Techterms.com. (2010). Character Encoding Definition. Retrieved February 10, 2015, from http://techterms.com/definition/characterencoding

Techterms.com. (2014). Socket Definition. Retrieved October 2, 2014, from http://www.techterms.com/definition/socket

The Home Depot. (2014). *The Home Depot Completes Malware Elimination and Enhanced Encryption of Payment Data in All U.S. Stores*. Retrieved from http://media.corporate-ir.net/media_files/IROL/63/63646/HD_Data_Update_II_9-18-14.pdf

Viswanathan, M. (2005). *Measurement error and research design*. Sage.

von Solms, R., & van Niekerk, J. (2013). From information security to cyber security. *Computers & Security*, *38*, 97–102. http://doi.org/10.1016/j.cose.2013.04.004

W3C. (2011). *The WebSocket API*. *World Wide Web Consortium*. Retrieved from http://www.w3.org/TR/2011/WD-websockets-20110419/

Wang, V., Salim, F., & Moskovits, P. (2013). *The Definitive Guide to HTML5 WebSocket*. Retrieved from http://www.apress.com/9781430247401

Whitman, M., & Mattord, H. (2011). *Principles of Information Security*. Cengage Learning.

Wichers, D. (2014). Input Validation Cheat Sheet - OWASP. Retrieved November 3, 2014, from https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

Wichers, D., & OWASP. (2013). The Top 10 Most Critical Web Application Security Risks. Retrieved October 6, 2014, from https://www.owasp.org/images/1/17/OWASP_Top-10_2013--AppSec_EU_2013_-_Dave_Wichers.pdf

Zhang, H. (2012). Preparing for HTML5 Capabilities and Threats. *ISACA Journal*, *6*(24). Retrieved from http://www.isaca.org/Journal/Past-Issues/2012/Volume-6/Documents/12v6-Preparing-for-HTML5.pdf