International Journal of Computer Applications (0975 – 8887)
Volume 82 – No 6, November 2013

An Empirical Study of HTML5 Websockets and their Cross Browser Behaviour for Mixed Content and Untrusted Certificates

Achin Kulshrestha

ABSTRACT

Websockets allow a full duplex connection to be made over a single socket between the client and the server. Today, Websockets is a finished standard and has greatly helped modern web applications to achieve real time communication without any overhead of sending HTTP headers with every request. This research provides an overview of the Websocket protocol and API, and focuses on the state of Websocket security. The research also aims to explicate behavior of different browser implementations of Websockets when delivering mixed content (ws/https) and the browser response when an untrusted certificate is encountered while making a secure Websocket connection. The crux of this paper is to analyze at the grassroots security concerns pertaining to Websockets and discuss best practices for secure deployment.

Keywords

HTML5, HTTP, Mixed Content, Security, Websockets.

1. INTRODUCTION

Normally when a user accesses a URL, the browser sends a HTTP request to the corresponding web server hosting the page. The web server processes the request and sends back the response. Once the browser has received the response, it is rendered on the DOM. However, today is the age of information accessed in “real-time” whether it is sports updates, stock prices or movie ticket sales and it is necessary that this information has to be updated on the user’s client device without him/her refreshing the browser page. Previously, attempts to provide data in real time was achieved by methods of long-polling and server side push technologies such as Comet. Since, all these technologies involved HTTP request and response headers to be sent with each polling request, it was an overhead to send unnecessary header data with each request leading to increase in latency. Websockets provided an approach to carry out a full duplex communication without any overhead of http headers and that translates to some serious performance improvements, especially for applications requiring fast real-time updates. Simply put, HTTP was designed to be a stateless protocol and not for real-time, full-duplex communication. And that is when Websockets were brought into existence.

2. THE WEBSOCKET PROTOCOL AT THE GRASSROOTS

HTML5 Websockets allow up streaming and down streaming connections with the server over a single TCP connection and therefore place less overhead on servers, as the same machine can support concurrent connections. The Websocket protocol was designed to accommodate the existing infrastructure on which Web functions. The Websocket connection uses the same ports as HTTP (80) and HTTPS (443) that provides it the ability to traverse firewalls and proxies without any problems.

The RFC6455 [1] defines the specification of the Websocket protocol standard. As part of the design principle to keep it compatible with the existing state of web, the protocol specification defines that the Websocket connection would start with a handshake which marks a protocol switch from HTTP to Websocket. The browser sends a Websocket request to the server (ws:// or wss://), indicating to the server that it wants to switch protocols from HTTP to Websocket. The server gets to know about this through the Upgrade header which is sent along with the connection request:

GET ws://example.com/?encoding=text HTTP/1.1

Origin: http://example.com

Cookie: __ASWE1241

Connection: Upgrade

Host: example.com

Sec-Websocket-Key: KJsdnkjwqel12ee454==

Upgrade: Websocket Sec-Websocket-Version: 13

If the server also has the knowledge of Websocket protocol, it agrees upon switching the protocol from HTTP to Websocket.

HTTP/1.1 101 Websocket Protocol Handshake

Connection: Upgrade

Server: Websocket-Server

Upgrade: Websocket

Access-Control-Allow-Origin: http://Websocket.org

Access-Control-Allow-Credentials: true

Sec-Websocket-Accept: hAKDSL/WEKAldmQWmasdsAS=

Access-Control-Allow-Headers: content-type

After the handshake is complete the HTTP connection is replaced by the Websocket connection over the same TCP channel. Once the connection has been established, a full duplex communication is created between the client and the server and exchange of data frames continues.

2.1 Websocket Javascript API

Almost all Modern browsers support Websockets. A new webscoket connection can be made to the Websocket server by calling the Websocket constructor

var connection = new Websocket('ws://example.org')

One can also use wss://, which is the secure socket variant to ws:// in the same way https is to http.

var connection = new Websocket('wss://secure.example.org’);

If your connection is accepted and established by the server, then an onopen event is fired on the client’s side. On the client side it can be handled as follows:

connection.onopen = function(){ console.log('Connection is now open!');}

If the connection is refused by the server, or for some other reason is closed, then the onclose event is fired.

connection.onclose = function(){console.log('Connection has been closed');}

The connection can also be explicitly closed.

connection.close();

In case of any errors, errors can be handled using the onerror event.

connection.onerror = function(error){console.log('Error ' + error);

Sending and Receiving Messages

Once a connection is successfully opened to the server, messages can be sent and received from the server. The .send() method of the browser Websocket API can be used to create a connection object.

connection.send(‘hello world');

Should the client receive a message from the server, it raises the onmessage event which can be handled

```
connection.onmessage = function(e){
  var server_message = e.data;
 }
```

JSON objects can also be sent to the server rather than simple messages. However, they should be serialized to a string, like so:

```
var message = {
'name': 'deadbeef',
'job': ‘roller coster’
   };
```

2.2 Websocket Frames

After a successful handshake, the application and the Websocket server may exchange Websocket messages. A message is composed as a sequence of one or more message fragments or data “frames.”

Each frame includes information such as(Shown in Fig 1):

- Frame length
- Type of message (binary or text) in the first frame in the message
- A flag (FIN) indicating whether this is the last frame in the message

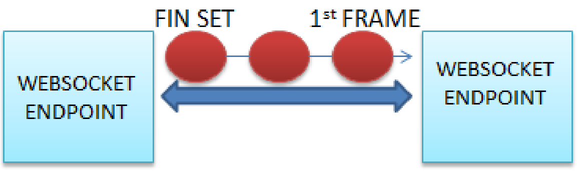

Fig 1 Websocket Frames

3. BROWSER BEHAVIOUR FOR MIXED CONTENT – WEBSOCKETS

When using SSL, the connection between the two endpoints (the web server and the browser) are encrypted and hence protected from sniffers and MITM attacks. However, if there is any content inside the HTTPS page that is being fetched through regular, cleartext HTTP then the connection is only partially encrypted as the non-https content can be easily sniffed and modified by anyone who can act as a man in the middle. So even though the browser would designate the website as secure by showing gray padlock or green padlock in the address bar, it actually isn’t. As per RFC6455 [6], if the application origin is secure that is HTTPS, then if an insecure Websocket connection is attempted then a security exception has to be thrown. Mixed content handling by Websockets differs slightly for different browsers. Following is an analysis for different browser behaviours for dealing with mixed content.

3.1 BROWSER BEHAVIOUR - HTTPS AND INSECURE WEBSOCKETS(WS://)

Table 1. Browser Analysis – Mixed Content

Tests	**Chrome 27 on Windows**	**Safari 5.1.7**	**Firefox 21**
Is a Websocket insecure (ws://) connection allowed from a HTTPS origin which made the request?	Yes, chrome allows a ws:// connection to be made from an https:// origin	Yes, Safari allows the connection	No, it is not allowed by default. A setting has to be changed “network.Websocket.allowInsecureFromHTTPS = True.”
By default, is any security warnings showed if mixed connections (HTTPS/WS://) are made?	No, chrome doesn’t show any Websocket related warnings	No error or warning is shown	Yes, Firefox shows an error “SecurityError: The operation is insecure.”

Table 2. Browser Analysis – Other Browsers Variants

Tests	**Chrome 25 on Android (Jelly Bean)**	**Firefox on Android**	**IE10 10.0**
Is a Websocket insecure (ws://) connection allowed from a HTTPS origin which made the request?	Yes, chrome allows a ws:// connection to be made from an https:// origin	No, it is not allowed by default.	Not allowed.
By default, is any security warnings showed if mixed connections (HTTPS/WS://) are made?	No	N/A	Yes, Error is shown” SCRIPT5022: SecurityError”

4. BROWSER BEHAVIOR FOR UNTRUSTED CERTIFICATES – SECURE WEBSOCKET CONNECTIONS

Following is an analysis of different browser behaviours when untrusted certificates are encountered while making WSS connection.

Table 3 Browser Analysis – Untrusted Certificates

Tests	Chrome 27 on Windows	Firefox 21	Safari 5.1.7
Is a connection allowed to be made using an untrusted certificate?	No, the connection is dropped, certificate has to be added to the trusted list	No, the connection is dropped, certificate has to be added to the trusted list	No, the connection is dropped, certificate has to be added to the trusted list
Does the browser show any error for an untrusted certificate when making a wss connection?	No	No	No

Table 4 Browser Analysis – Other browser Variants

Tests	Chrome on Android (Jelly Bean)	Firefox on Android	IE10 10.0
Is a connection allowed to be made using an untrusted certificate?	No, the connection is dropped, certificate has to be added to the trusted list	No, the connection is dropped, certificate has to be added to the trusted list	No, the connection is dropped, certificate has to be added to the trusted list
Does the browser show any error for an untrusted certificate when making a wss connection?	No	No	Yes, Error shown is “SCRIPT12038: Websocket Error: Network Error 12038, The host name in the certificate is invalid or does not match or certificate is invalid”

5. THE STATE OF WEBSOCKET SECURITY

The RFC 6455 of the Websocket protocol states that “The Websocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code.”

Although the Websocket protocol differs from HTTP in various ways, Most of the issues relevant to HTTP based web application such as MITM, authentication and authorization are relevant to Websockets as well. Also unlike HTTP, Websocket messages do not include HTTP headers. This may affect the behavior of web proxies and firewalls as most of them do some amount of packet inspection by identifying the headers. Security issues that may arise while using Websocket protocol are described in more detail in the following sections.

5.1 Websocket handshake

The most critical part of a Websocket connection is the handshake and most of the times; the real purpose of the handshake is misinterpreted by application developers. A Websocket handshake just defines the establishment of a mutual agreement between the client and the server; it is not at all intended to prove trust or identity. For authentication/authorization of end points either the implicit authentication mechanisms like Basic authentication or cookies can be employed. Another option is use a challenge response kind of mechanism to prove authenticity.

5.2 Websockets and the Same Origin Policy

The Websocket API allows making request cross domain to any server. This is a useful mechanism for app developers as it allows for communication between two completely different services. The parameter which allows a server to decide to whether allow/disallow the connection request is the Origin header. The data frames of Websockets do not include the HTTP headers, so the origin header is sent to the server during the Switch protocol and upgrade HTTP request. The onus lies with the server to verify the origin header. Despite this verification check, it is always possible for the attacker to spoof the origin header and connect to the server. However, verification is worth the risk as it protects the server against Cross Site Request Forgery attacks [2].

The Origin header can be considered analogous to the X-Requested-With [3] header used while making AJAX requests. Web browsers send a header of X-Requested-With: XMLHttpRequest which is used to identify AJAX requests made by a browser and those made directly. However, this header can be easily set by non-browser clients. In a nutshell, the Origin header can be used to differentiate Websocket requests from different locations and hosts they should be used a medium of authenticating the source.

5.3 Denial of Service Attacks using Websockets

The slowloris attack [4] allowed keeping many connections open to the target web server and holding them open as long as possible. This feat is accomplished by sending partial http requests to the web server. Since every Websocket connection is persistent, slowloris can be applied to it as well. Moreover, the specification says that for the client, only one handshake is required per origin and multiple Websocket connections can be made. Therefore any server which doesn’t have effective thread management capabilities would be susceptible to these kinds of attacks.

5.4 Websockets Data frame security

Websockets have been made to solve connection problems only. The specification does not provide any information about data validation and authentication. A Websocket server should never blindly trust any data coming from the client. A defensive strategy [5] for handling data frames have to be kept in mind while consuming any data on the server side. These include:

- Verify payload lengths to avoid buffer overflows and under runs
- Avoid resource exhaustion. For example, allocation of memory without checking the size of the input buffer.
- Out of sequence data
- Client sending messages in wrong order.

If any invalid data is received, the connection should be closed after sending close frames. The same approach applies to the client as well for any data that is being received from the server.

6. WEBSOCKET IMPLEMENTATION CONSIDERATION

6.1 Restrict the browser's resources using CSP

Content Security policy [7] is an added layer of security which acts as a blacklisting/Whitelisting for resources loaded by web applications. On the web, the CSP policy can be implemented via a HTTP header or a meta element. Similar to HTTP, a Websocket endpoint is defined by a URL which means origin-based security can be applied. With the help of CSP, we can restrict all script resources to be loaded from authorized sources. Also Websockets and XHR connections can be restricted using the connect-src attribute.

- script-src 'self' – Load scripts only from the same origin
- script-src https://example.com – allow scripts to be loaded from a particular origin
- connect-src wss://example.com

6.2 Connect with WSS:// scheme

The Websocket communication channel can be encrypted the same way TLS is used to encrypt HTTP, using certificates. A WSS communication begins with an establishment of a TLS handshake and then the protocol is upgraded to Websockets. A variety of attacks against Websockets become impossible such as MITM if the transport is secured.

6.3 The client is not trustworthy

It is possible that Websocket connections are established outside the purview of a browser. So the server side Websocket implementation should be robust enough to handle arbitrary data. Injection attacks are just as possible over Websockets as they are over HTTP. Also verify the origin header matches the expected value.

6.4 Validate data from the server

Equal validation has to be applied to any data received from the server. All data has to be encoded in proper format before inserting to the DOM and no code has to be evaluated directly. In most of the contemporary web application, JSON is used for sending/receiving data. Use JSON.parse [8] to parse the data securely.

6.5 HTML5 Security Cheat Sheet

Below are the general guidelines for the safe deployment of a web application utilizing WebSockets. The list is based on the guidelines placed by the HTML5 Security Cheat Sheet [9]

- Recommended protocol version is versions above hybi-00. Popular Hixie-76 version (hiby-00) and older are outdated and insecure.
- Recommended version supported in latest versions of all current browsers is RFC 6455 (Supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50 and IE10).
- XSS vulnerabilities may exist and usage of TCP services through WebSockets such as VNC, FTP can act as a backdoor.
- Websockets don't handle authorization and/or authentication. Application-level protocols should handle that separately as per the need.
- Process the messages received by the websocket as data. It should not be assigned to DOM directly or evaluated. For JSON response never use the insecure eval() function, use the safe option JSON.parse() instead.
- Use Secure Websockets WSS://
- Spoofing the client is possible outside browser, so WebSockets server should be able to handle incorrect/malicious input. Validation of input coming from the remote site is important, as it might have been altered.
- When implementing servers, check the Origin: header in Websockets handshake. Though it might be spoofed outside browser, browsers always add the Origin of the page which initiated Websockets connection. This can also help in Prevent CSRF attacks.

7. CONCLUSIONS

Websockets is a relatively new technology and being a modern protocol, cross origin communication is embedded right inside them. While we can always make sure that we are communicating with trusted clients and servers, but Websocket does provide the flexibility to communicate between parties on any domain. This has made Websockets a very powerful piece of technology for real time application. At the same time it is important that security risks that Websockets are considered while developing applications and security mechanisms pertaining to authentication and authorization are put in place. Most importantly, Websockets have been designed to solve communication problems not security problems.

8. REFERENCES

[1] I. Fette and A. Melnikov, 2011, The WebSocket Protocol RFC 6455 Websocket Specification, Internet Engg. Task Force, URL: http://tools.ietf.org/html/rfc6455

[2] Jussi-Pekka Erkkilä, The Websocket security analysis, Aalto University School of Science, 2012,URL

"http://juerkkil.iki.fi/files/writings/Websocket2012.pdf", pp 2-3

[3] Adam Barth, Collin Jackson and John C. Mitchell, Robust Defenses for Cross-Site Request Forgery, CCS, 2008, pp 6-7

[4] Slowloris attack and tool, http://ckers.org/slowloris/

[5] Mike Shema, Using HTML5 Websockets Securely, URL "http://deadliestwebattacks.files.wordpress.com/2013/03/asec-f41-mike-shema.pdf", 2013

[6] The Web Socket API, W3c Working Draft http://dev.w3.org/html5/Websockets/#the-Websocket-interface, 2009

[7] Joel Weinberger, Adam Barth, Dawn Song, Towards Client-side HTML Security Policies URL "https://www.usenix.org/legacy/event/hotsec11/tech/final_files/Weinberger.pdf4", pp 3-4, CCS 2008

[8] Json.parse, msdn, http://msdn.microsoft.com/en-us/library/ie/cc836466(v=vs.94).aspx.

[9] Websockets- HTML5 Security Cheat Sheet, URL https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet.