Global Information Assurance Certification Paper

AJAX versus WebSocket a comparison of security implications

GIAC (GWAPT) Gold Certification

Author: Sven Thomassin, sven.thomassin@be.pwc.com

Advisor: Antonios Atlasis

Accepted: July 25th 2014

Abstract

Today, more and more web applications make use of asynchronous client-server communication technologies like AJAX and WebSockets. They play an important role in improving the user experience and decreasing delays during content loading. The security implications of asynchronous client-server communication technologies have been examined and discussed in the literature but a comparative study of AJAX and WebSockets is lacking. This paper offers the community a comparison of the security implications of each technology. We will examine AJAX and WebSockets from a high-level viewpoint, before studying the specific AJAX and WebSocket security implications. The paper provides a self-contained, in-depth, comparative overview of AJAX and WebSocket security implications and is of particular value to web application testers as it highlights important security implications which must be examined.

1. Introduction

Today's web application content is loaded to the web browser by means of the HyperText Transfer Protocol (HTTP). HTTP is a stateless Layer 7 protocol for transferring data objects (e.g. HTML pages, JavaScript and CSS files and images) by means of HTTP messages. Two types of HTTP messages exist, being requests sent by the client and responses returned by the server. Prior to exchanging HTTP messages a connection is set up with the server when a client requests data objects. This connection can be reused to retrieve additional data objects. (IETF, 1999) (Wang, Salim, & Moskovits, 2013)

In the early days of the Internet web application content was loaded synchronously, which meant that the complete web page was reloaded upon each client request. This paradigm shifted with Web 2.0. Web 2.0 brought applications with an interactive and rich end-user experience to the browser (O'Reilly, 2005). This enhanced user experience is made possible by web application techniques like AJAX (Asynchronous JavaScript and XML) and WebSockets. Both techniques provide means to asynchronously load web application content to the web browser. In practice this means that only parts of the active web application have to be refreshed instead of loading the complete web page.

This paper will focus on AJAX and HTML5 WebSockets and how they are used to build web applications that support asynchronous communication. Other techniques like WebRTC or streaming will not be discussed. In chapter 2 we will elaborate on how AJAX and WebSockets can be implemented to support asynchronous communication. Chapter 3 will cover the security implications related to AJAX and WebSockets. In chapter 4 we will compare the security implications of both technologies. Additionally, we will discuss the important attention points which should be taken into account when using either AJAX or WebSockets. In chapter 5 we will draw our conclusions.

Sven Thomassin, sven.thomassin@be.pwc.com

2. Asynchronous client-server communication

When surfing through a web application that relies on synchronous client-server communication the user has to wait for data objects to be returned by the server before the web page is completely visible. Figure 1 graphically demonstrates that the user has to wait for the server's response after each page request.

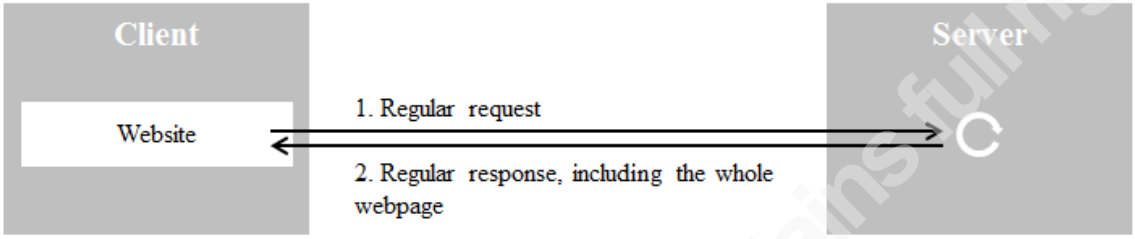


Figure 1 – Synchronous communication

Through asynchronous client-server communication the number of data exchanges between the client and the server can be reduced by adding a layer between the client and the server on the client side. In the following paragraphs we discuss AJAX and WebSockets which allow the implementation of asynchronous communication in practice.

2.1. AJAX

AJAX, or Asynchronous JavaScript and XML, is a way of combining standards to exchange data between the client and the server (W3Cschools, 2014d). In an AJAX application, a JavaScript based AJAX engine is placed between the client and the server on the client side, as depicted in Figure 2 and Figure 3. The exchanged data is typically serialized as JSON or XML to prevent the exchange of the webpage in its entirety. The AJAX engine handles the data exchange and places the retrieved data in the web application's DOM. Furthermore, the AJAX engine can contain business logic to decide whether each user interaction should result in a request to the server.

Figure 3Figure 2 depicts how asynchronous requests are handled which require an interaction with the server to obtain data. (Garrett, 2005) (OpenAjax Alliance, 2014)

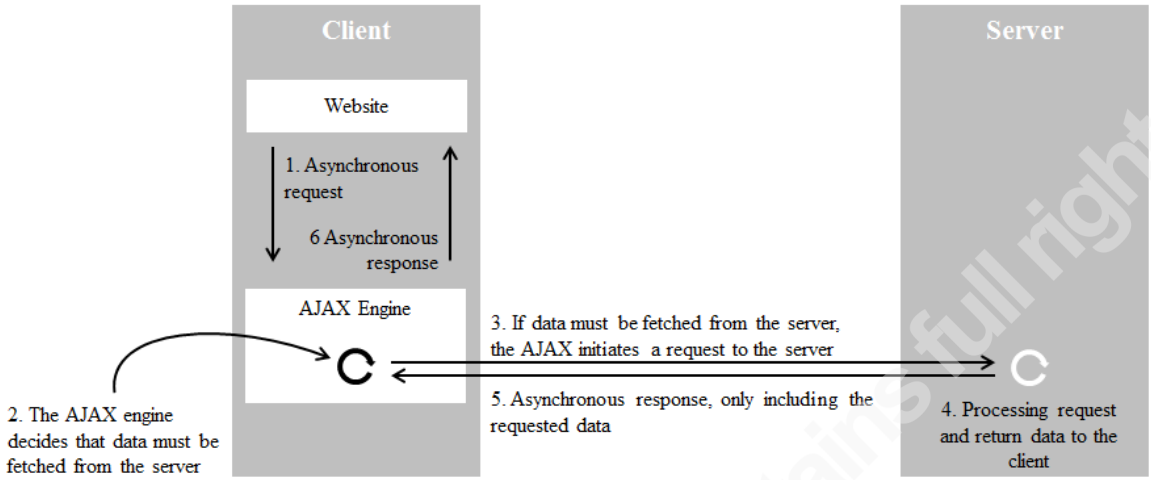


Figure 2 – Asynchronous communication, the AJAX engine requests data from the server

Figure 3 depicts how asynchronous requests are handled which can be processed by the client-side business logic instead of requesting server interaction to obtain data.

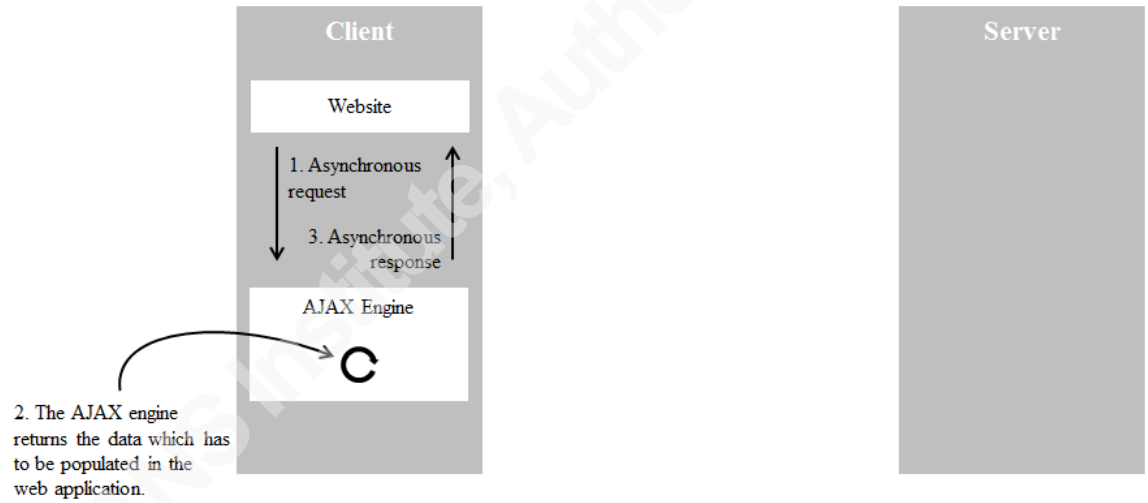


Figure 3 – Asynchronous communication, the AJAX engine returns the data without server interaction

The JavaScript XMLHttpRequest object, or the ActiveXObject for older browsers, is used to exchange data with the server. The open() and the send() methods are respectively used to open a connection with the server and to send data to the server. The data retrieved from the server can be obtained by the responseText or responseXML property and will be placed in the DOM by the AJAX engine. (W3C Schools, 2014a) (W3C Schools, 2014b) (W3C Schools, 2014c)

The solution described above does not foresee in a full duplex communication channel, but relies on requests introduced by the end-user. To simulate this behavior a technique called polling can be used. In this technique the AJAX engine automates client-side requests to obtain information from the server in a near real-time fashion. Long polling can be used to further decrease the time delay to obtain data in real-time. In this case periodical requests are sent while the connection

remains open until a server response is received or until the connection is timed out. (Wang, Salim, & Moskovits, 2013)

2.2. WebSockets

The unscalable approach of polling, and long-polling, leads to unnecessary network latency which can be prevented by using WebSockets. WebSockets offer a bidirectional communication channel via one socket that allows the client and the server to initiate data transfers. As such the burden on servers can be reduced allowing them to support more concurrent connections. (Wang, Salim, & Moskovits, 2013) (IETF, 2011) (Kaazing Corporation, n.d.)

To better examine the security implications of using WebSockets we set up our own local, RFC 6455 compliant, WebSocket echo server in an Ubuntu 12.10 virtual machine. After some research we found a simple, working WebSocket server using Python Tornado (Gaitatzis, 2013). Tornado is framework and asynchronous networking library in Python and can be found on http://www.tornadoweb.org. More information on how the WebSocket server was set up and configured can be found in Appendix A.

2.2.1. WebSocket opening handshake

Our WebSocket server echoes each WebSocket request. To make use of this service, the connection must be upgraded to a WebSocket connection. The first step in the opening handshake is initiated by the client who sends an Upgrade header with its request. Moreover, a Sec-WebSocket-Key header is sent with the request, so the server can prove that it received a valid opening handshake and avoid accepting connections from non-WebSocket clients.

```
GET / HTTP/1.1
Host: ***:8889
User-Agent: ***
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Sec-WebSocket-Version: 13
Origin: null
Sec-WebSocket-Key: EP21sUrYzIJxfVsNO6LVzA==
Connection: keep-alive, Upgrade
Upgrade: websocket
```

Hereafter the server responds with an informational header (101 Web Socket Protocol Handshake) to open the Web Socket connection. The response contains, besides the upgrade header, the Sec-WebSocket-Accept header which is the server confirmation for initiating the WebSocket connection (IETF, 2011):

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: yi7RhLS933PPEbrKWadwmVBrm4M=
```

Upon completion of the handshake all client-server communication is done over the WebSocket protocol. The WebSocket requests and responses are initiated and processed by JavaScript. Besides the Sec-WebSocket-Key and Sec-WebSocket-Accept headers, RFC 6455 registers 3 other new HTTP header fields for WebSockets (IETF, 2011):

- The **Sec-WebSocket-Extensions** header is used to agree on protocol extensions.
- The **Sec-WebSocket-Protocol** is used to define the sub protocol of the connection.
- The **Sec-WebSocket-Version** is an indication of the protocol version of the connection, allowing the server to correctly interpret the opening handshake.

3. AJAX and WebSocket security implications

3.1. AJAX specific security implications

As in a normal web application, AJAX communication can be intercepted by means of an intercepting proxy. Hence, AJAX is vulnerable to the attack vectors described below. Additionally, one must not only review the intercepted traffic but should also review the source code of AJAX applications to, for example, reveal hidden URL's to which AJAX calls can be made.

3.1.1. Encryption

The open() method of the XMLHttpRequest object requires the definition of the URL that has to be visited. Since AJAX traffic is sent over HTTP, all communication between the client and the server should be encrypted by using HTTPS. If not, all data exchanged between the client and the server could be intercepted and potentially altered mid-air.

Based solely on the URL of the website, one cannot infer whether the same URI scheme is used for the AJAX calls. Suppose that:

- A website is served over HTTPS, whilst AJAX calls are made over HTTP.
- Cookies are set without a secure flag and are used to track users.

As a consequence cookies could be leaked once an AJAX call is made, facilitating a malicious user's job in hijacking legitimate users' sessions. Hence, a tester should verify whether HTTP or HTTPS is used for AJAX calls, for example by means of an intercepting proxy or by checking the JavaScript source.

3.1.2. HTTP Request Method used

The open() method requires the HTTP Request method to be used. Generally speaking, the GET or POST methods are used for making requests. The POST method is more resilient from a security viewpoint, since parameters that are sent along a

GET request can be read by all intermediate devices between the client and the server. Moreover, the URL, including the parameters, is stored in the browser history.

3.1.3. Input validation and output encoding

User input cannot be trusted, thus it is important to properly validate it prior to processing. Otherwise it would be possible to introduce an XSS vulnerability in the application by injecting JSON pairs or injecting HTML or XML elements in the exchanged data. In the latter a malicious user could inject multiple nested XML entities to create an XML bomb to overload the XML parser on the client or server side (SoapUI, 2014). As a consequence he could impact the availability of the application.

An XSS vulnerability could also be introduced in case output encoding is forgotten. Typically, in an AJAX based application, the application is populated by means of the innerHtml method, which in contrast to the innerText, does not encode dangerous characters e.g. <, >, “, ‘, ` and = (OWASP, 2013). Hence the server should HTML encode these characters when returning them to the browser to prevent the execution of scripts.

3.1.4. Client side business logic

Since user input cannot be trusted it is good practice to handle all business logic server side, with the client side assisting in performing parts of the business logic to avoid unnecessary client-server communications. Placing too much reliance in client-side business logic could undermine the application security model. As a malicious user could investigate the AJAX engine to understand and circumvent the client side business logic.

3.1.5. Origin related security concepts

The same origin policy (SOP) prevents that scripts from different origins can access each other’s content, which makes it difficult to perform cross-origin communication. There are some workarounds to the SOP restrictions, like Cross-Origin Resource Sharing (CORS) or the usage of <script> elements since the SOP does not apply for this element. Hence, it is important to consider whether third party scripts can be included in the application as well as to investigate which cross-origin requests these scripts make. (De Rijck, 2012)

When a request is made from within a certain origin, the HTTP Origin header is sent with the request. The server can respond with different Access-Control-Allow-headers, like the Access-Control-Allow-Origin or the Access-Control-Allow-Methods header. When testing AJAX-based web applications, the usage of these response headers should be verified by intercepting client-server communication with an intercepting proxy.

Specifically, the value of the Access-Control-Allow-Origin should be verified. It is good practice to specify a set of domains, instead of using null or the asterisk as wildcard, because these will grant access to any origin. In case a request is made from a disallowed origin, the server should respond with no CORS headers in the response. Otherwise valuable information like allowed origins can be leaked.

CORS is no watertight security measure for servers to prevent unintended access to server resources as it only limits the origins that could access the server resource. Servers should not protect confidential information by means of CORS headers as origin headers can easily be altered. A malicious user could for example use a proxy to intercept the HTTP request headers and modify them prior to forwarding the request to the server.

3.1.6. Authentication & authorization

As AJAX communication happens over HTTP, the server-side application can take care of synchronous and asynchronous client requests. As such, user access control can be handled by the application. Thus, the same authentication module can be used to check whether a user is authenticated when receiving a synchronous or asynchronous request. Hence, it is important that the session information is sent along each request.

Upon authentication, the server-side application must ensure whether the authenticated user is authorized to access the requested information (OWASP, 2014). Hence, during a penetration test the authorization module should be investigated to understand who can access the requested data objects. It is important to note that vertical and horizontal privilege escalations should be tested when different authorization levels are used.

3.2. WebSocket specific security implications

One must keep in mind that a WebSocket connection provides a bidirectional communication channel via one socket. As such scalability issues, like the C10k problem, should be addressed on server level (Kegel, 2014). As the above statement concerns more server configuration it also implies that DoS testing against a WebSocket Server should be taken into consideration when performing a web application test. Further examination of the C10k problem is out of scope of this paper.

3.2.1. Encryption

Asynchronous client-server communication that contains sensitive data must be encrypted by means of the wss:// URI scheme instead of the regular ws:// URI scheme (ethicalhack3r, 2013).

Since cookies are exchanged with the server during the WebSocket opening handshake, similar security implications as described in paragraph 3.1.1 can be applied to WebSockets.

Sven Thomassin, sven.thomassin@be.pwc.com

3.2.2. Input validation and output encoding

The same security implications apply for WebSockets as discussed in the input validation and output encoding section of the AJAX specific security implications – user input cannot be trusted. Hence user input should be validated prior to being processed. Similarly, the server-side application must ensure that user input is properly encoded prior to populating the web application.

3.2.3. Client side business logic

As in AJAX application, the client side business logic should be reduced to prevent malicious users investigating the JavaScript source code which handles the client-server communication.

3.2.4. Origin related security concepts

A malicious user could initiate a Denial-of-Service attack against the WebSocket server when all connections are accepted, (Shema, Shekyan, & Toukharian, 2012). For example, suppose that a website is vulnerable to stored cross-site scripting. A malicious user could exploit this vulnerability to setup a WebSocket connection when a legitimate user visits the vulnerable page. In case multiple users visit the vulnerable page the availability of the WebSocket server could be negatively influenced as all users will set up a WebSocket connection due to the Stored XSS.

The use of the origin header can make it difficult to accomplish such an attack. The origin header is sent along with the WebSocket upgrade request. Before completing the handshake, the server could check the origin header to decide whether to accept or refuse WebSocket connections. (Wang, Salim, & Moskovits, 2013)

The origin header, however, should not be relied on since this header can easily be spoofed by a browser plugin or an intercepting proxy, or even natively in the source code of non-browser applications. A more effective measure against Denial-of-Service attacks would be authenticating application users, as well checking whether they are authorized to access the WebSocket service.

3.2.5. Authentication & authorization

Data leakage via the WebSocket service should be prevented by serving the data over an encrypted channel after the user is authenticated. RFC 6455 states that the WebSocket protocol does not prescribe a particular authentication mechanism to identify or authenticate users (IETF, 2011). As such, client authentication could take place at the application level prior to or even after the application layer protocol upgrade. For example by means of a cookie based authentication mechanism or by means of a protocol, like XMPP, that allows to authenticate users upon the WebSocket upgrade (XMPP.org, 2009). The examination of XMPP is out of scope of this paper, but we will describe the cookie based authentication mechanism in more detail below.

Before upgrading to the WebSocket protocol, authentication could take place by means of, for example, a form based authentication mechanism that prompts for the client's credentials. A session cookie will be set upon a successful authentication attempt by the client. This session cookie is sent with all subsequent requests, to the website including the request to upgrade to WebSockets. During the opening handshake, the WebSocket server should check whether this cookie is linked to an authenticated user.

This approach places the security controls concerning authentication and cookie management back in the web application. The authentication mechanism must ensure that user credentials are sent over an encrypted channel, strong password requirements are enforced and user account information is not leaked. Additionally, the authentication mechanism must be protected against brute forcing attacks, while the cookie management implementation must ensure that a session cookie is set upon successful authentication. This means that the appropriate flags, i.e. HttpOnly and secure, must be set accordingly. The session cookie value must have high entropy to avoid that a malicious user would start guessing the cookie values in an automated fashion. The latter can for example be tested by means of the Burp Sequencer module.

As described in the AJAX specific security implications, the likelihood of performing a vertical or horizontal privilege escalation upon authentication should be verified as well.

3.2.6. WebSocket tunneling

TCP services, like a database or a remote desktop connection, can be tunnelled over WebSocket, refer to tools such as sstur/node-websocket-tunnel and wstunnel. A malicious user could obtain access to these tunnelled services, in case of an XSS attack (Heroku dev center, 2014), and mask the exfiltration of information over a WebSocket connection. Hence, during a penetration test one must ensure that unwanted tunnelling is not possible, while verifying which services could be tunnelled over the WebSocket connection. We will not further elaborate on this security implication as this is dependent on the WebSocket server used and requires an in-depth research in this specific domain.

4. Ajax & WebSocket Security: A comparison

The table below provides an overview of potential security vulnerabilities related to AJAX and WebSocket.

	AJAX	WebSocket
Encryption	The client server communication should go over an encrypted channel such as HTTPS.	The wss:// URI scheme has to be used to send the client-server communication over an encrypted channel
HTTP Request Methods used	The preferred method is POST, especially in case parameters are sent with the request.	Not applicable to WebSockets as the WebSocket protocol does not rely on request methods.
Input validation and output encoding	User input should be validated before being processed. This user input should be sanitized or encoded to named entities before being returned by server. Hence, one must ensure that characters like <, >, “, ‘, ` and = are not allowed in user input. But if this is not possible they should be properly encoded when returned by the server.	
Client side business logic	The JavaScript source code should be investigated to understand the application business logic on the client side.	
Origin related security concepts	AJAX calls should only be allowed from a preset list of origins. Moreover, the allowed origins should not be leaked.	The WebSocket service should only be served to a set of allowed origins. Moreover, the allowed origins should not be leaked.
Authentication & authorization	If relevant, one must ensure that AJAX calls can only be answered when the user is authenticated and has the required authorization level.	It is important to check whether authentication and authorization verifications are made during the opening handshake.
WebSocket tunneling	This technique does not apply to AJAX	Verify whether the WebSocket server allows tunneling of TCP services.

AJAX and WebSocket implementations are only as safe as the web application that implements their features. Suppose that a web application is prone to a cross-site scripting attack, while the AJAX or WebSocket features are securely implemented. A malicious user could, for example, abuse this XSS vulnerability to alter the

application's behavior to undermine the encryption used or even bypass the implemented access controls.

In general, similar security implications can be drawn for AJAX and WebSockets. Both require the use of input validation and output encoding. While the former ensures user input is not trusted, the latter mitigates any damage otherwise caused by returning malicious user input to a client. Client-side business logic implications are also similar for both technologies. AJAX and WebSockets rely heavily on client-side business logic, requiring an analysis of the JavaScript source code to verify which business logic controls are performed on the client-side and which on the server-side. Furthermore, both AJAX and WebSockets implementations should verify the request origin to ensure that their features are only served to a set of allowed origins.

However, the security implications for each technology differ when it comes to encryption. In both cases, encryption measures should be used to avoid data leakage in transit. While AJAX must use the HTTPS URI scheme, WebSockets must use the wss:// URI scheme. The configuration of access controls also differs. For AJAX, access checks can be performed for each request, whereas for WebSockets they can either be performed during the opening handshake or in-transit with e.g. XMPP.

The security implications are different when it comes to HTTP request methods and service tunneling. The former is only relevant for AJAX while the latter only for WebSockets.

Sven Thomassin, sven.thomassin@be.pwc.com

5. Conclusion

In this paper we discussed how asynchronous client-server communication differs from regular communication methods. We further elaborated how asynchronous communication is implemented in practice by means of AJAX and WebSockets and how the two differ from each other from an implementation and a security viewpoint.

In essence we can conclude that the security implications that are present in classic web applications can be extrapolated to websites using AJAX and WebSocket features. Specifically encryption, input validation and output encoding, client business logic vulnerabilities and authentication and authorization issues are relevant security implications. Additionally, the importance of origin related security implications may not be forgotten, as these could help to prevent that AJAX calls and WebSocket connections are made by unwanted origins. As CORS will not protect these features from unauthorized access, the application should rely instead on the authentication and authorization controls in place.

Security implications related to the HTTP Request Methods are only relevant for AJAX based applications. Specifically, data sent via a GET request will be leaked to intermediate devices between the client and the server. WebSockets adds another security implication namely WebSocket tunneling. A malicious user could exploit this feature to exfiltrate information over WebSocket connections.

During a web application penetration test one must ensure that the web application using AJAX and WebSockets is not prone to classic vulnerabilities such as for those stated in the OWASP Top 10. Otherwise secure AJAX and WebSocket implementations may still be vulnerable via the website using these features.

Sven Thomassin, sven.thomassin@be.pwc.com

6. References

De Rijck, P. (2012). HTML5 Security. Leuven.

ethicalhack3r. (2013, August 30). *Security Testing HTML5 WebSockets*. Retrieved June 23, 2014, from ethicalhack3r: http://www.ethicalhack3r.co.uk/security-testing-html5-websockets

Gaitatzis, T. (2013, September 19). *WebSocket server with Python Tornado*. Retrieved May 29, 2014, from Tony Gaitatzis: http://tonygaitatzis.tumblr.com/post/61729708977/websocket-server-with-python-tornado

Garrett, J. J. (2005, February 18). *Ajax: A New Approach to Web Applications*. Retrieved February 20, 2014, from Adavaptive Path: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/

Heroku dev center. (2014, April 21). *WebSocket Security*. Retrieved June 23, 2014, from Heroku dev center: https://devcenter.heroku.com/articles/websocket-security

IETF. (1999, June). *RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1*. Retrieved Febuari 21, 2014, from IETF: https://tools.ietf.org/html/rfc2616

IETF. (2011, December). *RFC 6455 - The WebSocket Protocol*. Retrieved February 21, 2014, from IETF: https://tools.ietf.org/html/rfc6455

Kaazing Corporation. (n.d.). *About HTML5 WebSockets*. Retrieved May 2, 2014, from Websocket.org: http://www.websocket.org/aboutwebsocket.html

Kegel, D. (2014, February 2). *The C10K problem*. Retrieved May 28, 2014, from Dan Kegel's Web Hostel: http://www.kegel.com/c10k.html

OpenAjax Alliance. (2014, March 22). *Introducing Ajax and OpenAjax*. Retrieved March 22, 2014, from OpenAjax Alliance: http://www.openajax.org/whitepapers/Introducing%20Ajax%20and%20OpenAjax.php

O'Reilly, T. (2005, September 30). *What Is Web 2.0*. Retrieved March 11, 2014, from O'Reilly Media: http://oreilly.com/pub/a/web2/archive/what-is-web-20.html?page=1

Sven Thomassin, sven.thomassin@be.pwc.com

OWASP. (2013, January 22). *OWASP AJAX Security Guidelines*. Retrieved June 25, 2014, from OWASP: https://www.owasp.org/index.php/OWASP_AJAX_Security_Guidelines

OWASP. (2014, April 4). *HTML5 Security Cheat Sheet*. Retrieved June 23, 2014, from OWASP: https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet

Shema, M., Shekyan, S., & Toukharian, V. (2012). Hacking with WebSockets. Retrieved July 15, 2014, from http://media.blackhat.com/bh-us-12/Briefings/Shekyan/BH_US_12_Shekyan_Toukharian_Hacking_Websocket_Slides.pdf

SoapUI. (2014). *XML Bomb*. Retrieved July 15, 2014, from SoapUI: http://www.soapui.org/Security/xml-bomb.html

W3C Schools. (2014a). *AJAX - Create an XMLHttpRequest Object*. Retrieved June 23, 2014, from w3cschools.com: http://www.w3schools.com/ajax/ajax_xmlhttprequest_create.asp

W3C Schools. (2014b). *AJAX - Send a Request To a Server*. Retrieved June 23, 2014, from w3cschools.com: http://www.w3schools.com/ajax/ajax_xmlhttprequest_send.asp

W3C Schools. (2014c). *AJAX - Server Response*. Retrieved June 23, 2014, from w3cschools.com: http://www.w3schools.com/ajax/ajax_xmlhttprequest_response.asp

W3Cschools. (2014d). *Ajax Tutorial*. Retrieved May 27, 2014, from w3cschools: http://www.w3schools.com/ajax/default.ASP

Wang, V., Salim, F., & Moskovits, P. (2013). *The Definitive Guide to HTML5 WebSocket.* Apress.

XMPP.org. (2009, September 25). *Simple Authentication and Security Layer*. Retrieved June 2, 2014, from XMPP Standards Foundation: http://xmpp.org/protocols/urn:ietf:params:xml:ns:xmpp-sasl/

Appendix A. Setting up and configuring a simple WebSocket echo server

We configured the WebSocket server and client be following these steps:

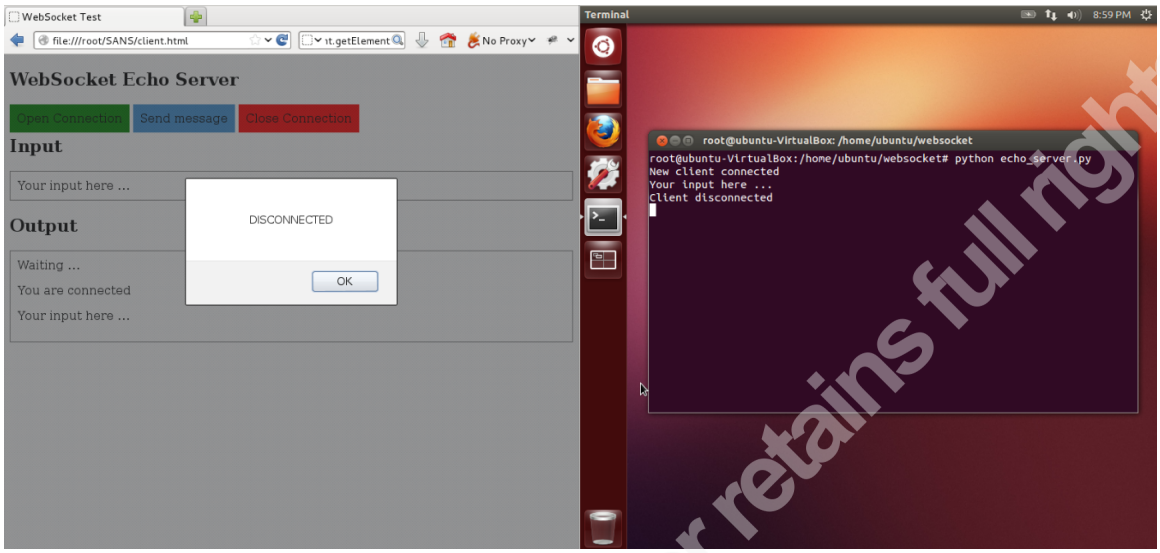
Step 1. Install Tornado library: $ easy_install tornado
Step 2. We slightly modified the WebSocket server Python script to be more verbose.
Step 3. Run the WebSocket server
Step 4. Create a client side page to test the WebSocket echo server. This page provides functionality to open a WebSocket connection, send a message and close the connection. It provides an input field to send your text to the server and an output field which contains the server response.
Step 5. Browse to client.html to test the WebSocket echo server

We used, and slightly modified, the code of Tony Gaitatzis for our WebSocket echo server (Gaitatzis, 2013):

```
#!/usr/bin/python
import tornado.web
import tornado.websocket
import tornado.ioloop

# This is our WebSocketHandler - it handles the messages
# from the tornado server
class WebSocketHandler(tornado.websocket.WebSocketHandler):
  # the client connected
        def open(self):
                print "New client connected"
                self.write_message("You are connected")

        # the client sent the message
        def on_message(self, message):
                print(message)
                self.write_message(message)

        # client disconnected
        def on_close(self):
                print "Client disconnected"

# start a new WebSocket Application
# use "/" as the root, and the
# WebSocketHandler as our handler
application = tornado.web.Application([
  (r"/", WebSocketHandler),
])

# start the tornado server on port 8889
if __name__ == "__main__":
  application.listen(8889)
  tornado.ioloop.IOLoop.instance().start()
```

Sven Thomassin, sven.thomassin@be.pwc.com

The image below depicts the client and server side user interface.



Sven Thomassin, sven.thomassin@be.pwc.com