Department of Computing, Cork Institute of Technology, Ireland

# Exploiting Security Vulnerabilities present in HTML5 Cross Origin Communication

by

Eoin Carroll

MSc Networking and Security Research Project

Supervised by Byron Treacy

Date: 11th August 2011

## Abstract

The World Wide Web (WWW) has evolved from static web pages in 1990 to dynamic mashups with web2.0 and the desktop like performance goals of HTML5 or web3.0. HTML5 which is the next generation of HTML, is a built on a lot of JavaScript APIs to give it a rich client-side interaction, for features such as cross document client-side communication, as well as multi threaded JavaScript functionality using web workers to give the feel of a desktop like application. This advancement in performance and client-side capability will blur the lines between web applications and desktop applications (Risky, Business, 2010). The single most important security policy enforced by web browsers is the Same Origin Policy (SOP) which prevents scripts in one domain accessing content in another domain, but due to the increased drive for better client-side mashup capabilities, HTML5 has been designed to relax the SOP to allow for more communication on the client-side, to reduce the communication with the server. Even though HTML5 has been designed with security in mind, this research summarizes numerous vulnerabilities, exploits and design weaknesses to show that HTML5 is far from perfect with regards to security. Since the SOP is the single most important browser security policy, and the fact that HTML5 is relaxing this to a Verified Origin Policy (VOP), it is a valuable research area. While there has been research into postMessage API, client-side storage, websockets and miscellaneous type attacks, there has been no single research which takes a holistic view of the APIs and primitives throughout the specification with regards cross origin communication. This research takes a new approach, where we take a bottom up approach to understand the API interaction and the cross origin communication capabilities to determine whether the attack surface has increased with the introduction of HTML5, so that it can be related to mashup functionality. I focus on the postMessage API as all communication using this primitive happens on the client slide unknown to the server, and if the developer does not verify the origin (origin == *) of a message, can increase the attack surface of HTML5 web applications. I take a look at a practical financial mashup to demonstrate how a Cross Site Scripting (XSS) attack can be launched, using the new secure postMessage API. We see that the postMessage API is vulnerable to DOM based XSS attacks and if coupled with CSRF could form a deadly cocktail to exploit web3.0 mashups (Shah, 2010). I conclude by coming up with a set of recommendations for aiding in the secure implementation of HTML5 applications.



## Acknowledgments

I would like to thank my project supervisor Byron Treacy for all his guidance and direction during this project.

# Contents

## Keywords

**HTML** – HyperText Markup Language

**SOP** –Same Origin Policy

**VOP** – Verified Origin Policy

**CORS** – Cross Origin Resource Sharing

**AJAX** – Asynchronous JavaScript and XML

**XHR** –XMLHttpRequest

**XHR2** – XMLHttpRequest 2 (XHR with cross origin capability)

**API** – Application Programmer Interface

**Web1.0** – Static Web pages

**Web2.0** – Dynamic Web pages with simple mashups

**Web3.0** – Web Apps that achieve similar performance as a Desktop Apps

**Mashup** – A website which combines code and data from multiple domains into one application called on Integrator

**WHATWG** –Web Hypertext Application Technology Working Group

**W3C** – World Wide Web Consortium

**OWASP** – Open Web Application Security Project

**JSON** – JavaScript Object Notation

**Iframe** – A frame which isolates content from different origins from its containing document

**XSS** – Cross Site Scripting

**CSRF** – Cross Site Request Forgery

**SQLi** – Sequel Injection

**DOM** – Document Object Model

**RIA** – Rich Internet Application

**Covert Channel** – a channel of communication used for a purpose other than its design intent

**Integrator** – A webpage that combines resources from multiple domains into a single web application

**Gadget** – a self contained API from any domain which is integrated into a web page

**SVG** –Scalable Vector Graphics

# Chapter 1
# Introduction

HTML5 technology will blur the lines between desktop applications and web applications which will change not only the way developers interact with web applications but also the hackers target market (McAfee Labs, 2009). Web based worms will become the future malware threat as scripts will replace binaries in web application attacks since they are platform independent and harder to detect (Risky Business, 2009) (Attack and Defense Labs, 2010a). Due to the new client side primitives such as post-Message and HTML5 client side storage, attacks such as Cross Site Scripting (XSS) will shift from being persistent on the server to persistent on the client (Hanna et al, 2010). Web applications are now becoming one of the most focused areas of security as attacks are being launched at the application layer as opposed the to security mature lower layers. "As web sites take advantage of improved client side technologies, browsers must cope with a growing range of performance, reliability and security issues" (Wright, 2009). Web1.0 was not very interesting from a security perspective as it required no user input since it used static web pages. Web security evolved with the introduction of web2.0 where the end user became interactive with an application supplying input to the server. Web3.0 moves this further along as it blurs the line between what's a desktop and a web application as with HTML5s new powerful APIs web applications can achieve near desktop performance. Desktop applications are platform specific and worms spread machine to machine as they are compiled to a specific architecture where as javascript is platform independent and XSS worms spread from user to user or profile to profile. From a protection perspective a web server gets patched but when there is vulnerability but on the client side it may remain for a long time persistent in client side storage. SQLi attacks have always been on the server but with client-side Sequel lite (SQLite) databases attacks can be injected using XSS.

HTML5 maybe a relatively secure technology but the fact that its functionality is built heavily on JavaScript makes it vulnerable. The client side storage of HTML5 poses the biggest threat as it can be accessed with JavaScript once an attacker can execute a XSS attack payload. The primary attack for HTML5 will be to find a site vulnerable to XSS since all new HTML5 functionality relies primarily on JavaScript (Risky Business, 2010). Apples iphone and ipod don't support flash plug-ins and are pushing hard for the introduction of HTML5 although the use of the audio and video in HTML5 may well take over from the plug-ins attacks surface (Risky Business, 2010) (OWASP, 2011). HTML5 will not completely replace flash plug-ins but will make a significant portion of

its functionality redundant. The future of malware is web based worms and combined with the fact that HTML5 is built primarily on Javascript means that it may well leave web applications susceptible to easier propagation due to the relaxed origin policy and cross domain capabilities.

(Risky Business, 2010)

## 1.1. Browser Security

Due to the rapid growth of the World Wide Web, the browser has become the most popular computer application in use today. However with this, brings a number of security issues, as it has to handle a huge amount of HTTP traffic which may be potentially malicious. One of the main issues with HTTP traffic is that it's the most popular protocol used today on the internet and so its port (80), will never be blocked by any firewall, since it's needed for most businesses and people to communicate. Web Application developers should threat all data received as malicious until proven otherwise by performing proper sanitization and input/output validation.

The browser has two primary security policies: the Same Origin Policy (SOP) and the Navigation Policy, the SOP being the single most important security feature of the browser architecture. Two origins are the same if and only if they have the same scheme/host/port triple e.g. http://example.com and https://example.com have a different port so that are a different origin. The Origin header indicates to the receiver the origin of the browser that sent the request (Barth et al, 2010b). The SOP policy means that JavaScript executing in different domains cannot access the window properties or JavaScript objects in another domain. A JavaScript file can be loaded from any domain, but the SOP is applied to the context of the document that it's executed in, as opposed to the source of the script itself e.g. if web page www.examplea.com loads a script from www.exampleb.com, then the script has access to all of www.examplea.com DOM resources, but not to www.exampleb.com resources. This is necessary as it prevents scripts in one domain from stealing information in another domain.

Prior to the implementation of HTML5 the following techniques could be used to relax the same origin policy:

1. Document.domain
2. Crossdomain.xml

Document.domain is useful for very large websites where orders.example.com and catalog.example.com want to allow scripts access to all sub domains on the website. However if sub domains on this website are sensitive and need protection this would create a serious security issue. Crossdomain.xml is used for plug-in technologies such as, Flash and Silverlight which are Rich Internet Applications (RIAs), the server configures this file to only allow white-listed or all origin headers access to website, similar to Cross Origin Resource Sharing (CORS) in HTML5. The main difference between them is that crossdomain.xml allows access on a per site basis where as CORS gives the ability to define finer granularity by allowing access on a per web page basis (Zalewski, 2011).

Browser security is reliant on the following:

1. Browser architecture and being update to date with latest patches
2. JavaScript
3. Plug-ins
4. Cascade Style Sheets (CSS)

All of the above can lead to vulnerabilities and the ability to launch an exploit, if vulnerable by design or implemented in an insecure manner. The fact that HTML5 relies so heavily on JavaScript, means that it can leave itself open to implementation and JavaScript vulnerabilities (Risky Business, 2010). Browsers are very complex systems and architectures, so the increased no. of JavaScript APIs with HTML5 to allow more client-side control, could potentially increase the attack surface of an application, as there is now much more interaction on the client side and more channels for data flow.

The other important policy is the Navigation Policy, which controls the ability of one frame or window to navigate another. The original permissive policy was insecure as it allowed any frame or window to navigate another. This leads to vulnerabilities as, if an attacker has control of an iframe, he can navigate the parent window or other frame to any website. An improvement over this has been the Descendant policy which only allows a frame to navigate another frame or window if it can draw over it. This means that a parent can navigate a sub or child frame but a sub frame or iframe cannot navigate a parent or other iframe (Barth et al, 2009).

JavaScript is a lightweight scripting language which is platform independent and executes in the security context of the browser sandbox. The main JavaScript feature of interest is its ability to dynamically update the Document Object Model (DOM), so that web pages can be updated on the client side without the need to contact the server for every client-side interaction. The DOM is a tree like structure that contains the contents and structure of a webpage. Access to a webpage's DOM properties or methods, are restricted by the SOP which is enforced by the browser. JavaScript can be invoked in four ways:

1. As a standalone inline script tag <script> which executes in the context of the container document and not the script source
2. As event handlers tied to HTML tags e.g. onfocus= alert("hello);
3. In CSS expression(...) blocks which is allowed in some browsers
4. Special url schemes specified as targets for certain resources or actions (JavaScript:..) in HTML and styles sheets

(Zalewski, 2011)

Once JavaScript code is executed, it has complete access to the DOM of its containing document, which is of course subject to the SOP check by the browser. As mentioned previously, the SOP only allows scripts to interact with scripts and read data from websites in the same origin. However this SOP restriction only applies to reading requests, which means the JavaScript can still use element tags such as <img> and <iframe>, to load scripts from another origin, but they cannot read data or interact with other scripts from their original origin once loaded. Asynchronous JavaScript and XML (AJAX) is not a technology but a collection of technologies which allows a website to make asynchronous requests for data to its originating domain, and again it is governed by the SOP. It operates by sending JavaScript HTTP requests in the background, unknown to the user, which enables a more responsive and richer client-side experience. For example, if you start typing a word into a search engine such as Google, it can use XHR to retrieve data in the form of XML or JavaScript Object Notation (JSON), from the websites origin to use predictive text. This means that the user doesn't need to wait until the full term is entered into the search engine, before the webpage gets updated. The "innerHTML" and document.write DOM properties and methods can then be used by the calling JavaScript function, to update the DOM dynamically, without reloading the entire page. When a response is received using AJAX, JavaScript can simply use the "eval()" construct to evaluate the string received and execute as if it is JavaScript code. The security risk here is that if the response from the server is not sanitized or validated, then arbitrary JavaScript code could be executed which would lead to a Cross Site Scripting (XSS) attack (Zalewski, 2011).

Browser vendors have the challenge of balancing security and performance, on one side there is the need to provide powerful APIs but also need to maintain system security. JavaScript executes in the context of the browser sandbox which restricts the capabilities of its interaction with the client system. While JavaScript does give for a richer client side interaction it can't do the following:

1. Cannot access the file system to read/write arbitrary files such as planting a virus
2. It does not have any raw networking programming capability
3. JavaScript can't open/close new browser windows without a user initiated event
4. It cannot set the file upload property on a form
5. Can only read the contents of files retrieved from another domain

(Flanagan, 2011)

## 1.2. Web Application Security

Web Application attacks can be divided into client and server but with Server-side technology becoming a more mature area of security the shift has focused to the client side. The Open Web Application Security Project (OWASP) maintains a list of the Ten Most Critical Web Application Security Risks – the OWASP Top 10 on a yearly basis. From the OWASP Top 10 2010 the top ten attacks are:

1. Injection

2. Cross Site Scripting (XSS)
3. Broken Authentication and Session Management
4. Insecure Direct Object Reference
5. Cross Site Request Forgery
6. Security Misconfiguration
7. Insecure Cryptographic Storage
8. Failure to Restrict URL access
9. Insufficient Transport Layer Protection
10. Unvalidated Redirects and Forwards

This research focuses on the 3 main client-side attacks which may become more prevalent with the introduction of HTML5;

1. Cross-Site Scripting (XSS)
2. Cross-Site Request Forgery (CSRF)
3. Injection or Sequel Injection (SQLi)

XSS is the most prevalent web application security flaw, which occurs when an application presents user supplied input to the browser without proper input and output validation. It exploits websites trust in a user to supply valid input data. There are three types of:

1. Stored – an attacker injects XSS payload into a backend database of a vulnerable site which gets rendered up to all users browsers who visit the website
2. Reflected – an attacker tricks a user into visiting a malicious link to a website which is vulnerable to XSS. This link contains a XSS payload which is supplied as input to the XSS vulnerable pages and the dynamically generated web page gets reflected back to the user
3. DOM based – this happens when a web application is retrieving data from a 3rd party which is not validated and so a malicious script gets injected into the DOM. This can happen if the application developer failed to sanitize then incoming stream which is malicious JavaScript and so gets interpreted and executed as JavaScript when read by "eval()" or "innerHTML" and injected into the DOM

Detection of XSS is relatively straight forward by performing code reviews to determine an applications inputs and ensuring the code is validating all inputs and outputs sufficiently. This can be done both statically and with automated tools but technologies such as AJAX make XSS much more difficult to detect with automated tools and this will get even worse with Web3.0 such as HTML5 due to cross origin

communication and the lack tool support for HTML5 testing (OWASP, 2011) (Shah, 2010). This is very important for the postMessage API specifically in HTML5 since it's accepting incoming data, it needs to ensure it's properly sanitized and cannot be interpreted and executed as JavaScript. The OWASP XSS Prevention Cheat Sheet gives a guide to XSS prevent.

What can an XSS attack actually achieve?

1. Steal sensitive information
2. Hijack user sessions
3. Compromise browser and underlying system integrity
4. Bigger threat to ecommerce
5. keystroke reading

Once an attacker can launch a XSS attack they have a JavaScript thread running in the users' browser so they have control of a website and users session.

Cross Site Request Forgery (CSRF) happens when an attacker creates a forged HTTP request and then tricks the victim into submitting the request via a malicious link. CSRF is the opposite of XSS whereby it exploits the users trusted in a website. As protection a developer should ensure that each HTTP request is unique and can only be generated by the user. This is a form of session hijacking for example, if a user is currently logged into their online bank then if the attacker can craft a HTTP request to transfer funds, once the victim clicks on this link the transaction will be performed. This is allowed to happen as the user is already logged into the banking application and so appears that they actually made the transaction so there is no repudiation. The best protection for CSRF is to use unique tokens or nonces, which are transmitted with each HTTP request, therefore the attacker would also need the value of the clients' non-predictable nonce value included in his crafted HTTP link. Cookies don't protect as these are automatically sent by the browser and but a nonce can be included in the web application itself by the developer to ensure a random value is sent with each HTTP request from the client. OWASPs CRSF guard can be integrated into an application to automatically include these random tokens or nonces.

CSRF attacks are harder to achieve than XSS attacks as timing is a big issue since the attacker cannot read any responses from the crafted response he enticed the user to click on. If a site is vulnerable to both, XSS and CSRF then its game over, as not only can the attacker inject the CSRF attack through XSS he can also read the response. This is applicable to HTML5 security as if the attacker can find new XSS vulnerabilities then he may well be able to launch CSRF attacks.

SQLi is where an attacker supplies simple text-based attacks that exploit the syntax of the targeted interpret, in this case SQL. It happens when an application sends untrusted data received from a user to a backend database such as SQL, which may allow the attacker access to sensitive information contained within the database. The best protection against SQL is ensuring that all prepared or stored procedures use bind variable which means that the attacker cannot assign values dynamically. This ensures that call commands and untrusted user data input are separated and interpreter differently. All user supplied input should receive input and output validation as well as ensuring the backend database is running with the least privileges so that if an attack does happen it limits the capabilities of the attacker. This attack is very relevant for HTML5 security as it uses client-side storage and also an SQL database (OWASP, 2010a).

## 1.3. APIs and Access Control

"Security Engineering is about building systems to remain dependable in the face of malice, error, or mischance" (Anderson, 2008). The four components required for good security engineering and which can be applied to HTML5 are:

1. Policy
2. Mechanisms
3. Access Control
4. Assurance

Policy is important for HTML5 as the SOP is now being relaxed for certain APIs which is now enforced by the developer as opposed to the browser. This is a shift from Mandatory Access Control (MAC), in the form of the browser policing the SOP without

any control from the developer, to Discretionary Access Control where the developer can control the new relaxed policy or Verified Origin Policy (VOP). The mechanisms are the HTML5 APIs themselves, which are in most cases, are secure but it's the fact that they are built on JavaScript which makes them vulnerable. Access control is also very significant for HTML5 with offline applications and client-side storage, as there must be good access controls built into the application, to ensure that different users on the same machine cannot access another user's information which is stored persistently on the same machine. Assurance is the confidence that a mechanism or control is secure which is also significant for HTML5 since it has been designed with security in mind, people may elude the that fact that they need not be as wary about developing HTML5 applications as opposed to HTML4 or XHTML.

HTML5 now makes web applications more like a desktop applications or operating systems as they can now interact with the file system and network, so developers need to keep the above four engineering principles in mind when designed and implementing HTML5 applications.

The core requirements of any computer system are Confidentiality (stored or transmitted data cannot be sniffed), Integrity (stored or transmitted data cannot be tampered with), and Authentication (verification of source). To ensure HTML5 applications achieve these requirements, the information flow control introduced with HTML5 cross origin communication needs to be understood by developers to prevent poor implementation which will lead to vulnerable web applications.

Application Programmer Interfaces (APIs) are functions which are exposed by applications to allow other parties interoperability and modularization. These APIs themselves maybe secure but the interfaces may not, and this could ultimately lead to an increase in an applications attacks surface. HTML5 APIs are built and rely heavily on JavaScript which could potentially increase the web applications attack surface if the APIs interface is not well understood when implementing. As suggested at the OWASP summit earlier this year, HTML5 may be taking over the attack surface for plug-ins (OWASP, 2011). The most common API failure mode is that transactions that are secure in isolation become insecure in combination e.g. HTML5 webworkers.

One of the biggest programs in computer science is preventing one program from interfering with another. There are typically three techniques used in computing systems for isolating programs:

1. Sandboxing
2. Virtualization

3. Trusted Computing Module (TPM) – show reference

The one of interest here is the sandbox, as this has been implemented by the Google Chrome browser and as a HTML5 iframe attribute. Sandboxing is an application level access control technology which in this case runs in the browser (Anderson, 2008).

## 1.4. Web 2.0 Mashups

While this thesis is not based on mashup development, it is important to understand what a mashup is and the need for mashups with client-side cross origin communication capabilities. A mashup is an application that combines separate APIs and data sources into a single integrated application. A mashup can occur on the server-side or client-side, however, HTML5 aims to develop the capabilities further on the client-side. It involves the integration of Gadgets from different domains into a single origin document called an Integrator. The Gadget is an API which is a self contained application with a JavaScript API. Prior to HTML there were only two options for integrating Gadgets into a mashup, using inline <script> or using <iframe>. The inline script option can be thought of as the all or nothing option, as it gives the script complete access and control to the containing documents DOM, where as the iframe provides complete isolation for Gadgets from other domains so that they cannot access

the integrators DOM properties or methods. Mashups can be divided into the following three categories:

1. Information systems
2. Enterprise systems
3. Web service tracking

However, a new breed of mashup is also emerging called the financial mashup. Mashup developers revealed the following difficulties when developing mashups:

1. Documentation
2. Coding details such as authentication
3. Lack of proper tutorials or examples
4. JavaScript skills needed to integrate the APIs can be a major hindrance

Documentation is vital with regards the API interfaces used in mashups so that developers can understand the interaction and input/outputs for data flow in their mashup. With HTML5 postMessage API, the end user is putting their trust in the Gadget, that it has been securely developed and the verified origin implemented correctly. Developers and users need to be aware of the security risks of mashups to ensure they integrate gadgets securely and reduce risk of code injection or XSS (Zang et al, 2008).

The power in a mashup is not just the data it supplies, but that fact that the received data can be manipulated by the integrator into the mashup main page and presented as they wish. To date this communication between Gadgets supplying the data to the integrator could not happen on the client side due to the SOP restriction other than two workarounds:

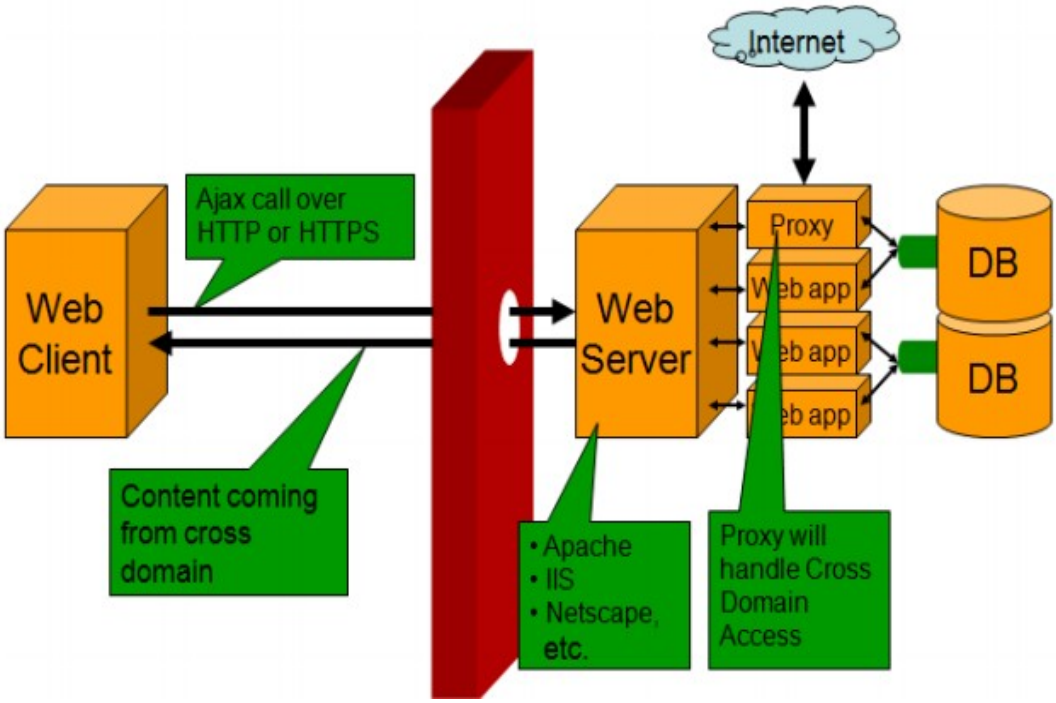
1. Fragment Identifiers (insecure)
2. Proxy Server using JavaScript Object Notation (JSON) – see figure 1.2.1 (insecure)

Mashups rely on AJAX which allows HTTP requests to be done seamlessly in the background unknown to the user but this communication only allows AJAX to read responses from servers in the same origin as the application making the AJAX request.

One of the biggest vulnerabilities with current web 2.0 mashups is the fact that AJAX performs all HTTP requests in the background slightly updating the main page dynamically, without any requirement for the whole page to be reloaded. To achieve this, the main web page reads back the AJAX response using "innerHTML" or "eval()" to update the DOM. If these are not handled correctly, an attacker can inject a JavaScript payload which gets converted to JSON and may execute without any change

to the main page, thereby not notifying the user. This is DOM based XSS in figure 1.2.2, which we will cover later, but is an important point as it the same type of XSS which the postMessage API in HTML5 is susceptible to (Shah, 2010).

**Fig**




**ure 1.2.1** – Web 2.0 Mashup Proxy for cross domain requests (Shah, 2010)

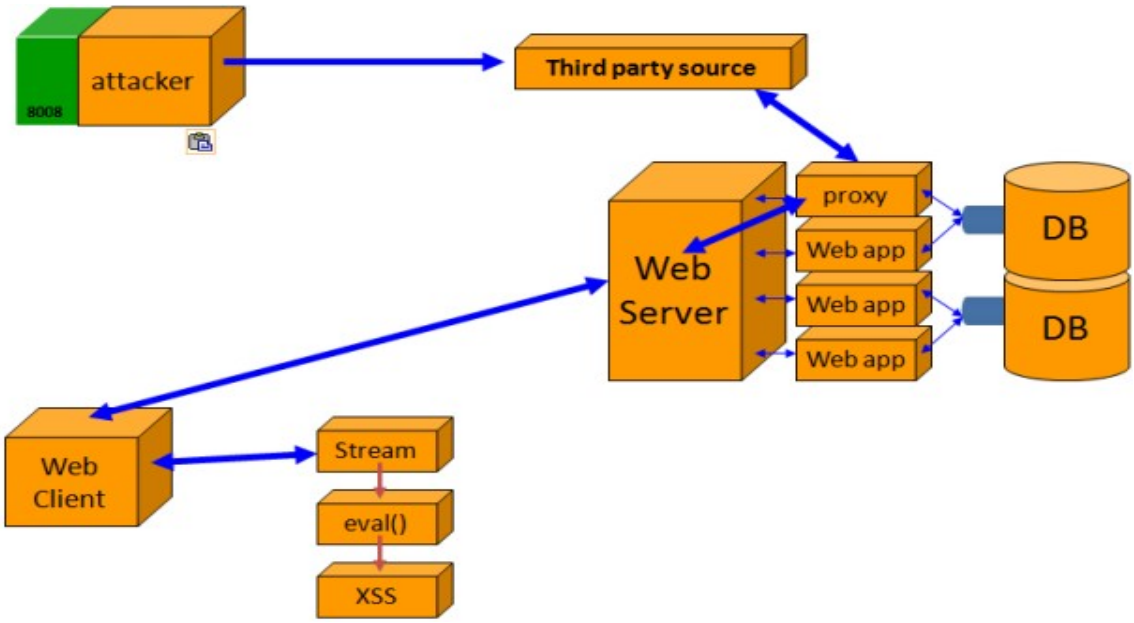
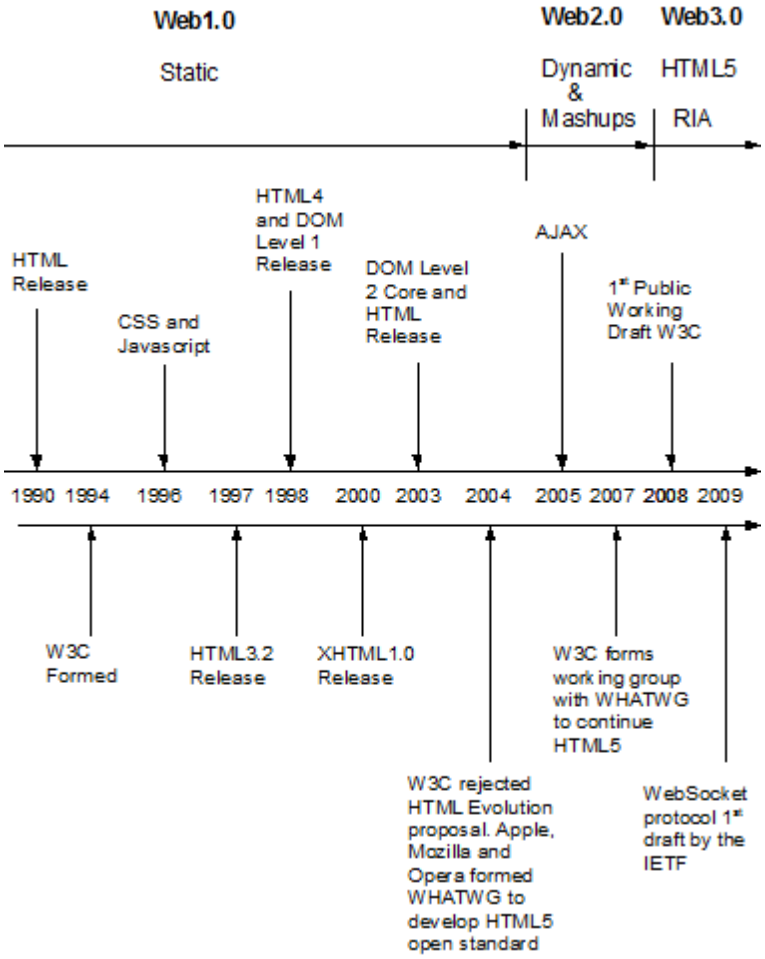



**Figure 1.2.2 -** DOM based XSS (Shah, 2010)

## 1.5. HTML5 Web 3.0 Mashups

**Figure 1.4.1** – Web Evolution from 1.0 to 3.0




(Sutton, 2009) (W3C, 2011b) (Google, 2010a)

The Web2.0 development goal was to develop desktop applications that have the responsiveness, look and tool of traditional desktop Apps. HTML5 is a new technology standard promising to empower browsers to become a suitable platform for developing rich web applications. With the trend towards the web as a platform, browsers have turned into more than standalone applications for accessing the web. HTML5 aims to achieve this client-side performance by introducing new APIs and protocols such as:

1. Web messaging – reduce complexity of current mashups by allowing client side communication between different origins
2. CORS – reduce complexity of current mashups by allowing clients to not just make cross origin request but to read the response also
3. Websocket – for real-time two-way HTTP communication
4. Client side storage and Offline Apps– enables mobile users to work with emails and application offline as well as reducing client to server communication by storing information locally on the client
5. Webworkers – multithreaded JavaScript that speeds up JavaScript performance for better data integration

Client-side mashups may reduce load on the server and give the user better response time and a generally richer experience but this extra communication and relaxed security may well open up more security holes leading to vulnerabilities and exploits in the future. There are three components to a client-side mashup:

1. Process Integration (PI)
2. Data Integration (DI)
3. Data representation

To date direct communication between User Interface (UI) components on the client has not been technically feasible, due to the lack of support or restriction by the SOP. However this can now be bypassed with the HTML5 postMessage API for a pure client –side mashup without the need for proxy server (Aghaee, 2010).

Figure 1.4.2 and

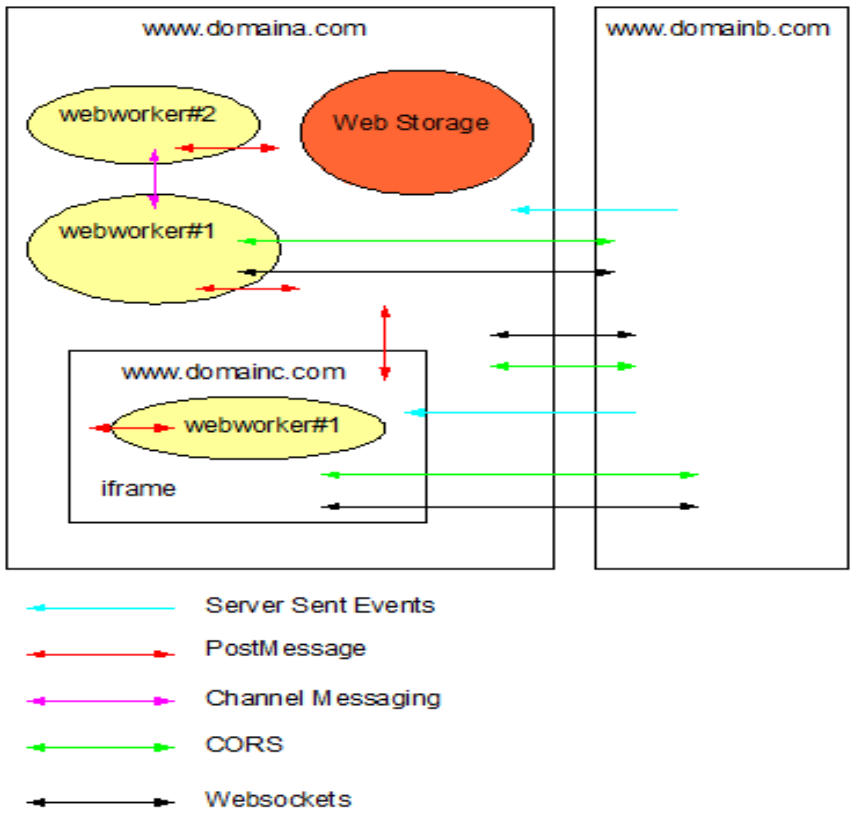


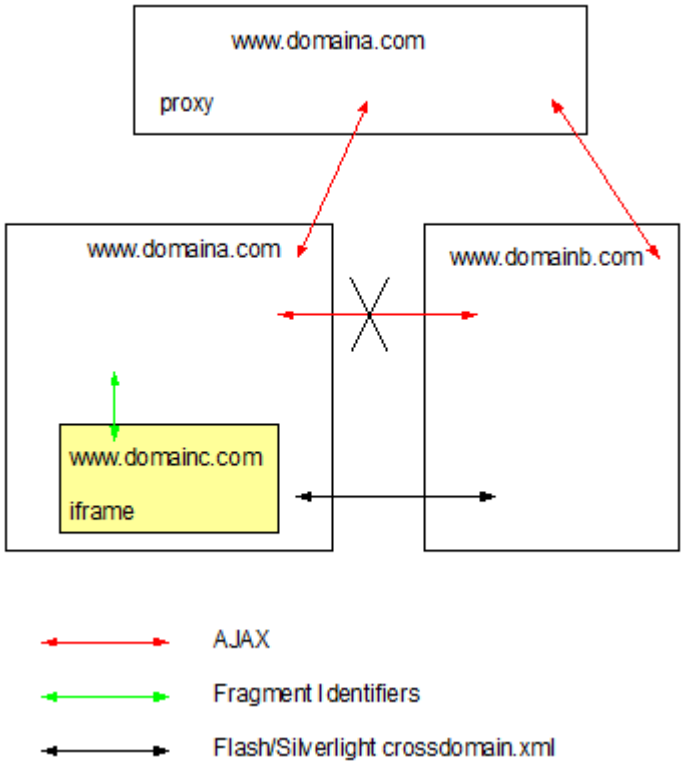

figure 1.4.3 shows the difference in complexity between a HTML5 (web 3.0) mashup and a Web2.0 mashup.

**Figure 1.4.2** – Summary of Cross Origin Communication in HTML5 (Web 3.0)

**Note:** Webstorage is accessed from the main window through the Web Storage API

**Figure 1.4.3** – Summary of Cross Origin Communication in Web 2.0

**Note:** Flash/Silverlight request




permitted cross domain if their origin is permitted in the crossdomain.xml file on the destination server or if the origin check is wild carded to "*" to match any origin. (W3C, 2011b) (Zalewski, 2011)

# CHAPTER 2
# Literature Review

## 2.1. HTML5 Specification

### 2.1.1 What's new

The first working draft of the HTML5 specification was released in January 2008. The core design principles within the specification are:

1. Compatibility with previous technologies such as XML and HTML4
2. Utility to solve real problems and make "Secure by Design"
3. Interoperability with well-defined behaviour and needless complexity
4. Universal access such as media independence

(W3C, 2011b)

The HTML5 Specification is a joint effort between the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C) (WHATWG, 2011) (W3C, 2011b). While Cross Origin Resource Sharing (CORS) is not officially part of the HTML5 specification it is included in this research as it relaxes the SOP and is dependant on many of the HTML5 APIs (W3C, 2011ab) (Google, 2010a). The Internet Engineering Task Force (IETF) is also involved in HTML5, specifically for the development of the websocket protocol. WHATWG is further along on the development side where as the W3C aims to make the specification a official standard to abide by W3C ratification. This research will document the differences between the WHATWG and W3C specifications but focus on the W3C specification as it's the official standard. The W3C specification introduces the following changes on the transition from HTML4\XML to HTML5:

1. Web Messaging (cross document messaging or postMessage API)
2. Client-side Storage (local, session and database storage)
3. Webworkers
4. Websockets
5. iframe sandbox
6. New Tags/Attributes
7. Scalable Vector Graphics (SVG)
8. Server-Sent Events
9. Drag and Drop API
10. File API

**Figure 2.1.1.1** – Summary of HTML5 Specification

| HTML5 Spec | W3C HTML5 Spec | Subsections related to Cross Origin Communication or Javascript APIs | Security sections in Spec |
|---|---|---|---|
| Introduction | Same as WHATWG | | |
| Common Infrastructure | missing HTMLpropertiescollection only | 1. Scripting<br>2. Plug-ins<br>3. Extensibility<br>4. Fetching resources using CORS** | Encrypted HTTP |
| Semantics, structure and APIs of HTML docs | Same as WHATWG | Dynamic markup insertion | Documents |
| Elements of HTML | Missing canvas element 2D only | 1. Scripting<br>2. Forms | Forms constraints security |
| Microdata | * Omitted from W3C | | |
| Loading Web pages | Same as WHATWG | 1. Browsing Contexts<br>2. Window Object<br>3. Origin (relaxed same-origin)<br>4. Session History and Navigation<br>5. Browsing the web<br>6. Offline Web applications | 1. Browsing contexts<br>2. Window Object<br>3. Location Interface |
| Web Application APIs | Same as WHATWG | Scripting | System State and Capabilities |
| User Interaction | Same as WHATWG | Drag and Drop | Risks in Drag and Drop model |
| Video Conferencing | Omitted from W3C | Peer-to-Peer Connections | Data stream considerations |
| Web Workers | * Omitted from W3C HTML5 Spec | 1. Dedicated Web Workers<br>2. Shared Web Workers | |
| Communication | * Omitted from W3C HTML5 Spec | 1. Server-Sent Events<br>2. Websockets<br>3. Cross-document/Web messaging<br>4. Channel messaging | Cross Document/Web messaging |
| Web Storage | * Omitted from W3C HTML5 Spec | 1. Session Storage<br>2. Localstorage<br>3. Database Storage | 1. Localstorage<br>2. DNS Spoofing Attacks<br>3. Cross Directory Attacks<br>4. Implementation risks |
| HTML Syntax | Same as WHATWG HTML5 Spec | | |
| XHTML Syntax | Same as WHATWG | | |
| Rendering | Same as WHATWG | | |
| Obsolete Features | Same as WHATWG | | |
| IANA Consideration | Same as WHATWG | | |
| FileAPI | Under dependencies section | Under dependencies section | |
| CORS | Under dependencies section | Under dependencies section | |
| Geolocation | Omitted from WHATWG | separate W3C spec | |

* Still being worked by W3C but as part of separation specification or draft and not included in their official HTML5 spec
** CORS is not officially part of the HTML5 specification but is mentioned is the spec

Figure 2.1.1.1 above shows a summary of the APIs involved in cross origin communication and new JavaScript APIs, as well as the sections of the specification with security specific statements. The W3C and WHATWG are very similar specifications but since the W3C began working on HTML5 in 2007 (see figure 1.4.1), they made a no. of changes which are summarized in figure 2.1.1.2 below. Most notably the following were split into their own specification or drafts by the W3C, but are still considered part of the HTML5 umbrella:

1. Websockets
2. Web Messaging
3. Client-side Storage
4. Server-Sent Events

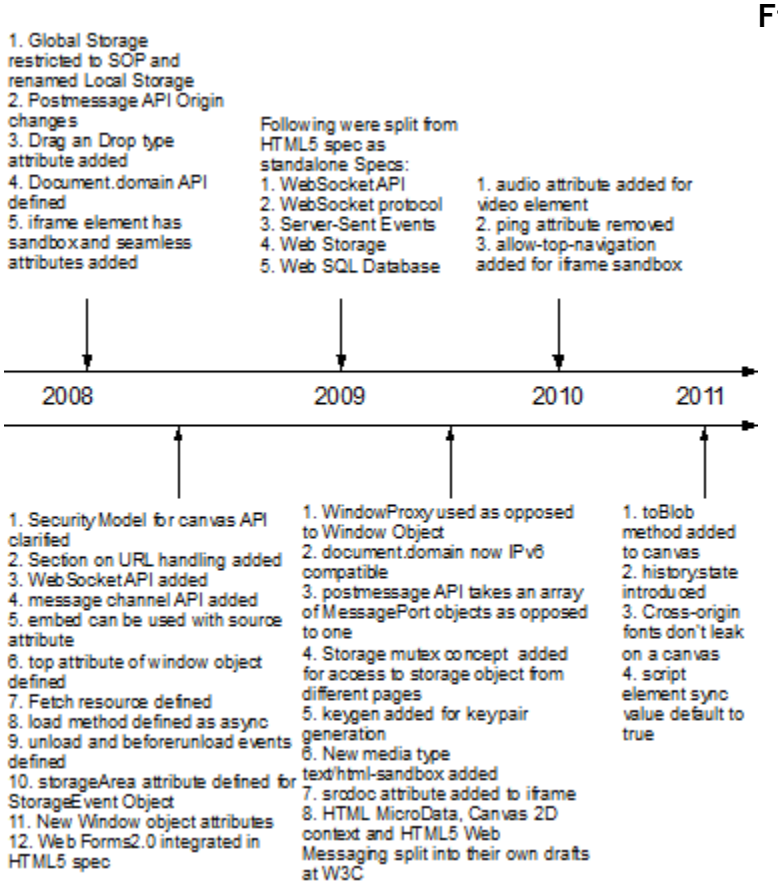
The following major changes were made with regards Webstorage:

1. Globalstorage has been removed and confined to SOP as recommended (Trivero, 2008)

2. Web Database API has been replaced with Indexed DB but both are included in the latest Google Chrome browser revision.

(W3C, 2011b)

**Figure 2.1.1.2** – W3C




HTML5 Spec major changes since 1st draft in Jan-2008

(W3C, 2011b)

The HTML5 spec will not be considered finished until at least two complete implementations of the specification have been released, unlike previous HTML versions where final specification would typically have to be approved by a committee before being implemented (pros and cons to this as security vulnerabilities due to rushed implementation by browser vendors). From my literature review of the HTML5 specification summarized in figure 2.1.1.1 above the following APIs are involved with cross origin communication:

1. Web messaging
2. CORS
3. Drag and Drop
4. Web workers

5. Server Sent events
6. Websockets
7. File API
8. Peer to Peer Communication (WHATWG only)

Specifically the following are the W3C primary APIs which actually request and read cross origin:

1. Web messaging
2. CORS (used by the File API and Server-Sent Events for fetching resources)
3. Websockets
4. Webworkers (can make cross origin connections using websockets and CORS)

For the purpose of this research I will be focusing on web messaging, CORS, web workers and websockets are they are the core cross domain features in the HTML5 specification.

As you can see from figure 2.1.1.3 there are many APIs being worked by the W3C, outside of the HTML5 specification. While Web Messaging, Webworkers, Server-Sent Events, Websockets and Web Storage are actually part of the HTML5 umbrella (W3C, 2011b), the following APIs are standalone specifications or drafts independent of HTML5:

1. Geolocation API
2. File API: Writer
3. File API: Directories and System
4. Network Information API
5. Touch Events System
6. The System Information API
7. Permission for Device Access

**Note:** Not all standalone APIs outside of the HTML5 specification have been listed here but the ones of significance to highlight future trends (W3C, 2011c)

**Figure 2.1.1.3 -**

| W3C Javascript API Drafts | State |
|---|---|
| HTML5 Web Messaging | Last call of draft ended |
| Web Workers | Last call of draft ended |
| Server-Sent Events | Last call of draft ended |
| File API | Working Draft |
| Geolocation API | Working Draft |
| XMLHttpRequest Level 2 | Working Draft |
| Websocket API | Working Draft |
| Web Storage | Working Draft |
| File API:Writer | Working Draft |
| File API: Directories and System | Working Draft |
| Network Information API | Working Draft |
| Touch Events Specification | Working Draft |
| Indexed Database API | Working Draft |
| Permissions for Device API access | Working Draft |
| The System Information API | Working Draft |

Specification Drafts being worked outside of Official HTML5 Spec

(W3C, 2011c)

As you can see from figure 2.1.1.3 above, the W3C is striving towards a lot of new APIs with Operating System like feature so that web applications become more like desktop applications with regards performance and capabilities.

## 2.1.2 Web messaging

Web messaging uses a new HTML5 primitive called the postMessage API to allow developers to pass string messages cross domain between Gadgets and Integrators on client-side mashups. PostMessage is a JavaScript API designed with security in mind to

enable secure inter-frame communications for mashups. This spec defines two mechanisms for communicating between browsing contexts in HTML5 frames:

1. Cross document messaging
2. Channel messaging

Prior to HTML5 it was not possible to communicate between frames residing in different origins due to the SOP enforced by the browser. HTML5 relaxes the SOP so that the postMessage API now uses the VOP which is enforced by the developer and not the browser. PostMessage is a pretty straight forward API with regards operation as it all happens on the client side with no interaction with the server. PostMessage like server-events, Web Sockets, cross-document messaging and channel messaging use the message event to signal when a message has arrived to the window or frame in question. The following events are defined for postMessage:

1. event.data – data of message
2. event.origin – origin of message for server sent events and cross-document messaging (scheme, hostname and port but not fragment identifier or path)
3. event.lastEventid – last event ID string for Server Events
4. event.source – window proxy of source window
5. event.ports – message port array sent with message cross document and channel messaging.

Prior to the implementation of postMessage the communication between iframes was restricted to the same domain by the SOP. While SOP is an important security feature, it prevents pages from different domains from communicating even when those pages are not hostile. This is a messaging technique which was designed in a way that allows documents to communicate with each other regardless of their source domain while preventing XSS attacks. Requires developer to verify the event.origin source but the browser controls this on sending side so can't be spoofed. The developer would use the following piece of code on the sending side to specify the target origin:

windows.postMessage("Hello", "http://www.example.com")

and the iframe gadget would have the following piece of JavaScript on the receiving side to verify the message is from the expected origin:

if(e.origin == "http://www.example.com")

The receiver side of postMessage can also set it's origin verification as follows if (e.origin == "*") which means it will accept messages from any origin.

Channel messaging enables different pieces of code (e.g. running in different browsing contexts) to communicate directly using two way pipes, with a port at each

end. Messages sent in one port are delivered at the other port, and vice versa. Messages are asynchronous and delivered as DOM or message events (W3C, 2011b).

Prior to HTML5 postMessage API, the Gadget was all or nothing by including it in an <iframe> or inline <script>, but now postMessage allows message events to be read and dispatched by a Gadget API using JavaScript.

(Flanagan, 2011)

### 2.1.3 Cross Origin Resource Sharing

AJAX uses XHR to make asynchronous JavaScript HTTP requests from client to server on the same domain. CORS expands this capability by using next generation XHR which is called XHR2 and allows communication across domains. This allows clients to make cross origin requests to websites that reside on different domains. While it's always been possible to make cross origin requests, it has not been possible to read the response due to the SOP enforced by the browser. HTML5 relaxes the SOP and like the postMessage API leaves the verificaition of origin enforcement up to the developer as opposed to the browser. CORS is not as simple an operation as postMessage as it involves cross origin requests from the client-side to server-side as opposed to just client-side as if the case with postMessage. CORS achieves this VOP by using the "Access-Control-Allow-Origin" header. This header is placed in the response from the server and includes the allowed origin, for example if the requesting domain is http:www.example.com then the response from the server would be include "Access-Control-Allow-Origin="http:www.example.com " or like postMessage the verification can be used to allow access from any origin by setting "Access-Control-Allow-Origin="*". CORS uses two types of requests:

1. Simple request
2. Pre-Flight request

(W3C, 2011d)

### 2.1.4 Client-side Storage

Cookies are used for storing small amounts of data with regards session management and user credentials so that websites can track the same user through multiple HTTP requests i.e. state management. Cookies however, are limited to 4kB in size and also need to be transmitted with every request from client to server. HTML5 aims to over come the storage limitation by allowing client side storage where web application developers can store user data and any application dependent data on the client to reduce communication to the server and also allow a richer client side experience. There are three types of client-side storage:

1. Session Storage – data is stored/persists for the life of a browser window or tab
2. Local Storage – data persists across browser sessions
3. WebSQLDatabase – uses an SQLite database

**Note:** WebSQLDatabase has been superseded by IndexedDB but both are currently supported in Google Chrome.

Client side storage access is restricted to the Origin from which the data was saved i.e. there is no relaxation of SOP for this API or primitive. The stage area can be accessed using the JavaScript API as long as the script is executing in the same domain as the origin that stored the data. The origin that is fixed to the storage data is not path specific so is not good for large websites with multiple sub-domains that need to be separated.

(Lubbers, 2010) (W3C, 2011b)

### 2.1.5 Websockets

Traditionally web applications have to send a header with every HTTP request so that receiving server understands the request. This inclusion of the HTTP header with every request increases the data being transmitted and so results in slower communication. This affects web applications when trying to achieve desktop like performance. Websockets is a protocol in development by the IETF to allow web traffic to achieve a secure full-duplex communication channel that operates through a single socket over the web. The initial handshake uses HTTP but then upgrades to the websocket protocol which removes the overhead of the HTTP headers and so only the actual data is transmitted with subsequent requests. Websockets allows two way communications with a remote host across different domains.

(W3C, 2011b)

### 2.1.6 Webworkers

JavaScript is a single threaded scripting language which means it has limited performance capabilities such as speed of execution. Since the introduction of web2.0, the client has become reliant on JavaScript to improve the end users experience. Since it is single threaded this can result in a poor browser experience for the end user if the JavaScript functionality being executed locks up the browser. Webworkers aim to resolve this by adding multi-threaded JavaScript capability, so that JavaScript functionality does not lock up the browser and so can execute silently in the background without interrupting or slowing down the end users experience. The web worker can only communicate with the window from which it was invoked through the

postMessage API i.e. it cannot communicate directly with DOM properties or objects on the window from which it was invoked, as they run in a separate JavaScript context. However they can make use of the Timing API so you can delay the starting or termination of a worker. Webworkers are also confined to only communicating to the windows or iframes within the same origin as enforced by the SOP. Webworkers can however initiate and read CORS and Websocket requests thereby giving them cross origin communication capabilities in this manner but they can the only communicate to the DOM of the same origin.

There are two types of web workers:

1. Dedicated webworkers
2. Shared webworkers

Dedicated Webworkers use the postMessage API from the web messaging to form a single channel connection between the invoking window or another web worker in the same origin. Shared workers use channel messaging from the web messaging API and they allow many workers from the same origin to communicate which is again policed by the SOP enforced by the browser.

Webworkers need to reference an external JavaScript file in order to execute so you must supply the invoking command with an external script located on the domain in question as follows:

```
var w = new Worker('javascriptfile.js')
```

Once the worker has been invoked it can also import a script from the same origin as follows:

```
importscripts('javascriptfile2')
```

Note: Webworkers can't specify an origin in postMessage() when passing information in this channel as is done for web messaging between iframes as webworkers are not permitted to use postMessage cross origin.

(W3C, 2011b) (Flanagan, 2011)

### 2.1.7 Iframe Sandbox

The iframe element has acquired new attributes with the introduction of HTML5:

1. Sandbox
2. Seamless

When the sandbox attribute is enabled is gives an extra set of restrictions or defense against potentially malicious JavaScript. When this is set the content or JavaScript is treated as being from an untrusted domain and so JavaScript is disabled.

Seamless means it executes in same content of containing page much like an inline script so provides no protection.

(W3C, 2011b)

### 2.1.8 Server-Sent Events

This API can open HTTP connections on the same domain or across domains using CORS, so that push events can be received from the server. This means that the client no longer has to waste resources polling the server waiting for a response. There has been no security research documented on Server-Sent Events but as this uses CORS to fetch responses from the Server the same security issues with CORS will apply here also (W3C, 2011b).

### 2.1.9 Drag and Drop API

This API allows a user to initiate a Drag and Drop event with a particular user action such as clicking and moving the mouse. It can be used for moving objects around an area on a web page or application from source to destination data elements. This API doe not apply the SOP restriction on the movement of data.

(W3C, 2011b) (Flanagan, 2011)

### 2.1.10 File API

This API gives system level like functionality, where a file can read and write to/from a web application from the file system on the client. The file system on the client is bound by the SOP of the application but when used with drag and drop, webworkers and postMessage APIs may have the potential for leaking information across domains.

(W3C, 2011c)

## 2.2. HTML5 Vulnerabilities and Exploits

### 2.2.1 Web Messaging and Frame Navigation

The postMessage API is a secure technique for cross origin client side communication when implemented correctly. Prior to HTML5 the only option for client side communication between frames or windows was using the Fragment Identifier Channel which was not designed for this purpose but rather discovered by mashup developers. Alternatively a proxy could be used on the server side to aggregate all requests from user so that the replies appear to be coming from the same domain. The Fragment

Identifier channel uses the window.location.hash to send messages between frames of different origins. This is possible because by placing a message after the location hash does not reload he page e.g.:

frame[0].location = "http://www.example.com/doc#message";

The URL can be different from the containing document URL as it doesn't get reloaded with the location hash in use. The important point to note is that no origin can read the message after the fragment identifier other than the URL placed before the #. This means that this channel does provide confidentiality but lacks authentication since the receiver cannot verify the sender and it's also unreliable. HTML5 have designed the postMessage API to ensure that it provides secure communication between frames. However from work to date, postMessage guarantees authentication but lacks confidentiality when implemented incorrectly. Barth el al (2009) highlighted the fact that the sender should be able to specify the destination frame to ensure that only the intended recipient can access the message, thereby ensuring confidentiality. This has been implemented by the W3C in the HTML5 specification but the problem is that it was designed with Discretionary Access Control (DAC) where the origin of destination is note enforced by the Browser but by the developer e.g.

frame[0]postMessage("hello","http://www.example.com")

However it's still possible to send a message to any destination using the wild card match e.g.:

frame[0]postMessage("hello","*")

Also the receiver doesn't have to specify the origin of the sender e.g.:

if(e.origin == "*)

The W3C specification does state that developers should be careful when implementing the postMessage API for cross origin client side communication but as research shows (Hanna et al, 2010) Google and Facebook have already implemented this incorrectly which can lead to Man in the Middle Attacks (MITM) and code injection such as XSS attacks using frame navigation to form proxies to intercept messages between frames (Hanna et al) (Barth et al, 2009) (W3C, 2011b).

As the postMessage API has the potential to inject malicious code (Hanna et al, 2010) and (Shah, 2010), so there exists potential to launch XSS worms or botnets and as scripted malware is harder to detect than traditional malware (spreading website can be quickly identified due to automated crawlers looking for signatures of browser exploits) since HTML5 based payloads are less likely to be identified since its regular JavaScript running within constraints of the sandbox and does not perform any exploitation on the

browser. Therefore is a more stealth but obviously attacks are confined to browser sandbox unless an attacker can exploit a browser vulnerability (Attack and Defense Labs, 2010a).

With postMessage the source origin is set by the browser so can't be spoofed with guarantees authentication (Barth et al, 2010). The HTML5 specification states that postMessage can be used with a target origin of * and also that the target application should verify the source and validate all input received – we see during our testing that when implemented incorrectly this can lead to DOM based XSS using the postMessage as an attack channel.

Security points from the W3C specification for authors and developers:

1. Authors should check the origin attribute of the postMessage API to verify the source – the source cannot be spoofed as its controlled by the browser but the receiver doesn't have to verify the sender as they can set the origin check to "*" which means accept from any sender
2. After this authors should check that the data received is in the correct format to prevent a possible malicious JavaScript XSS attack
3. Authors should not use wild card (*) for targetOrigin argument in messages containing confidential information

From (Barth et al, 2009), the postMessage API guarantees authentication but may lack confidentiality depending on whether implemented securely or not. Although postMessage is widely believed to be secure, it has been proved to be susceptible to confidentiality attacks and replay attacks due to frame navigation (Hanna et al, 2009).

There are three ways to read back postMessage replies:

1. innerHTML=e.data same as web 2.0 (insecure)
2. textContent=e.data (secure)
3. JSON.parse=e.data (secure)

The first option should not be allowed as postMessage is supposed to be for passing string messages between frames on client side. If JavaScript is injected into postMessage and written to the DOM using "innerHTML" then it opens web app up to DOM based XSS attacks. Cross document messaging may lead to DoS attacks since using many channels can use up client system resources (Flanagan, 2011) (Lubbers, 2010).

### 2.2.2 Cross Site Scripting

XSS attacks have been known for a long time and well documented (Hansen, 2008). As new vectors arise, developers and browser vendors need to update their application or

browser if they are vulnerable. Existing and new elements which have been released with HTML5 contain new event attributes and maybe possible to bypass blacklist XSS filters. In particular, an attribute of considerable interest is the "onfocus" tag with the "autofocus" attribute as when XSS is launched with this it means it can executed without any interaction from the user. With previous versions of HTML5 injected JavaScript required an event such as a mouse click to execute, but with "onfocus" it executes when code injected into the DOM (Attack and Defense Labs, 2010a).

The HTML5 Cheat Sheet has a collection of known HTML5 XSS vectors which were investigated during the testing phase of the project resulting in a successful XSS attack through the postMessage API (Heiderich, 2011)

### 2.2.3 Reverse Shell with CORS

The HTML5 specification does warn to verify the origin of CORS requests and that authors of client side applications should validate the content received from a cross origin resource as it may be malicious. COR enables JavaScript to read contents from different origins using the "responseText" DOM property.

The Origin header is the network equivalent of the origin property found on message events in the postMessage API which allows CORS grant access to public and secure data on a per origin and per page basis as opposed to per origin basis only using crossdomain.xml in web2.0 with plug-ins such as flash and silverlight. For sensitive actions CORS should use a pre-flight request to confirm which HTTP options are supported by the server and whether access is permitted (Lubbers, 2010). The fact that CORS can now make requests cross domain can give rise to CSRF attacks if the application does not have sufficient CSRF protection (W3C, 2011a). If a server has "Access-Control-Allow-Origin="*" set then an attacker can read back requests since this allows any origin access. This can be especially dangerous on internal servers as once attacker has JavaScript executing in domain on victims machine then can read information from internal server and create another remote connection or reverse shell back to his own machine domain (since only attackers server needs to check origin and no check on originating client). The source of CORS request should not be based on the header origin alone as this could be easily spoofed by an attacker or a MITM attack. Origin header check should be combined with session identifier by having the Credentials flag set in a Preflight request such as a cookie to confirm or authenticate the

source client. However if an attacker has launched a successful XSS attack on client then this provides no protection as request will be from authenticated client. The Credentials header used in the Preflight request should never be cached in the interest of saving time as this may be tampered with by attacker and become out of sync with server. Even with the "Access-Control-Allow-Origin="set with whitelisting to match authenticated sites only this can still lead to DoS attacks as the read is denied but the request is still permitted with uses up resources. This was possible with AJAX using XHR but now with a CORS launched continuously from a webworker would lead to a powerful DDoS attack (Attack and Defense Labs, 2010a) (Attack and Defense Labs, 2010b) (Heiderich, 2011).

### 2.2.4 Clickjacking and Drag Drop API

HTML allows any site to frame any URL within an iframe. Clickjacking is done by encapsulating a victim site in a transparent iframe that is put on top of what appears to be a normal page e.g. for login page of a banking website, the attacker could have his own crafted button which is invisibly placed over the banks login button and when the user submits their login details they are actually sent to the attacker as opposed to the bank. Clickjacking attacks are still over looked by major websites. Frame busting refers to code or annotation provided by a web page intended to prevent the web page from being loaded in a sub frame e.g.

```
if(top.location!=location)
top.location=self.location
```

The people you trust may not frame bust! You may take a trusted site in your mashup but it may already have been framed! The Descendant policy states that a frame can navigate only its descendants, meaning an iframe cannot navigate another iframe. Trust is a big issue as partner site that doesn't frame bust can cause the trusting page to be framed by an attacker.

The current protection techniques aside from frame busting code are:

1. X-FRAME-OPTIONS (can't be done on a per page basis
2. Content Security Policy (can't enforce policy site wide)

Only the three sites of the Top-500 survey are using X-FRAME-OPTIONS. Google Chrome introduced a reflective XSS filters called the XSSAuditor, which can be used to circumvent framebusting. The HTML5 sandbox attribute used in Google Chrome can be used to disable JavaScript in the same way as to protect against untrusted gadgets. The iframe sandbox may be thought of as a defense in depth mechanism but there has been

no research to date which has tested its ability to sandbox malicious JavaScript. The Drag and Drop API allows user to drag an element around the webpage and drop at a desired location. This drag operation is defined by adding a draggable attribute to an element and setting an event listener for dragstart. The HTML5 specification does state that the user agents (browser vendor) must not make the data being transferred by the drag operation available to scripts until the drop event has been performed. This is to ensure that no third part domain being crossed in the process can intercept the data. Drag and drop is not restricted by the SOP origin policy and when combined with Clickjacking forms a new powerful attack. It allows an attacker to trick a user into dragging data across domains and then using clickjacking to submit the data into a form controlled by the attacker. Clickjacking is not necessary if a site is already vulnerable to XSS or CSRF. If a sites only defense against clickjacking is framebusting then the new Iframe "sandbox" attribute will counteract the defense as it disable JavaScript which is needed for frame busting code to work.

(Rydstedt et al, 2010) (Stone, 2010) (Attack and Defense Labs, 2010) (W3C, 2011b)

### 2.2.5 Cross Site Request Forgery

When an attacker exploits the websites trust in the user. One shot type attack where attacker abuses users existing session to make a transaction with website currently logged into on attackers behalf but with victims credentials. This type of attack could become persistent with client-side storage capabilities and is much more serious as the attack can be repeated. The SOP has been fundamental security boundary within browsers that prevents content from one origin from interfering with content from another and cross document messaging intentionally relaxes this restriction. One of the biggest risks for a poorly implemented or configured cross domain policy is that it would trivially break any CSRF countermeasures. CSRF countermeasures rely on the SOP to prevent malicious scripts from other domains from accessing secret tokens and content within the target website. If cross origin communication is not implemented securely for example with the postMessage API then it leaves the victim open to CSRF attacks (Hanna et al, 2010) (W3C, 2011a)

### 2.2.7 Cache Poisoning

The Offline Application capability within HTML5 means that applications can store data over long periods of time on the client. Session storage remains as long as the browser or tab remains open but local and database storage remains persistent until deleted by the application. This gives rise to a new attack where by an attacker could

use XSS to gain access to the application cache and poison the cache. This would be a persistent threat as when the application loads up when online again it will load the page from memory. Attacks and Defense labs created a tool called Imposter which can actually perform this attack.
(Attack and Defense Labs, 2010a) (Attack and Defense Labs, 2010b)

### 2.2.8 Client-side RFI

Client side RFI are remote fragment identifiers. Fragment Identifiers are used after a URL to identify an anchor or part of a webpage that a user is currently navigating e.g.
http://www.example.com/#list
The above example would be linked to an element containing "list" in that particular website. This attack requires the attack to get victim to click on a malicious crafted http request or link (Attack and Defense Labs, 2010a).

### 2.2.9 Cross-Site Posting

This is similar attack to Client side RFI but the attacks needs to have AJAX control so would first need to have completed a successful XSS attack. The attacker can redirect all the victims AJAX requests to his website and so steal sensitiveness information from the victim. It may not be possible for the attacker to read the response from the XSS executing on the client but he can get around this by simply sending the data to himself and doesn't need to read any response on the client side (Attack and Defense Labs, 2010a).

### 2.2.10 Network Reconnaissance

Ajax uses XHR which can make cross origin request but is restricted by the SOP so can not read responses. However with XHR2, which is used by CORS, it can now read back cross origin HTTP requests. XHR2 has five readystate statuses and websockets has four. As the status of the readystate property changes with the state of a connection to a service on a particular port, both CORS and websockets can be used to determine the status of a remote port. Attack and Defense Labs proved that depending on the timing of the readystate response for both CORS and websockets, one can determine whether a port is opened or closed. The major limitation of network reconnaissance here is that browsers block connections to well known ports. However, it is possible to do horizontal internal network scans as if a port is open or closed; one can tell whether a

particular IP address is up or not. These new reconnaissance scans are not as reliable as network scanners such as nmap but may be seen in future attacks (Attack and Defense Labs, 2010a)

### 2.2.11 Botnets

While webworkers can only communicate with other webworkers in the same domain using the postMessage API within its container document, it also has the ability to initiate cross domain CORS and websocket requests or connections. Since web workers are multithreaded and operate in the background unknown to the user they could be a prime target for distributed attacks such as to perform password cracking, DDoS, setting up a reverse shell (backdoor) with an attacker or a Botnet. A Botent is a collection of machines under an attacker's control which are used in a master/slave type role to perform tasks on behalf of the attacker. Webworkers seem like the perfect execution model for an attacker's JavaScript as it can run for long periods of time in the browser background unknown to the user and without affecting their browser performance. Webworkers also execute in the context of the browsers sandbox so will evade detection of any anti-malware software since they are not exploiting the browser in any way.

A XSS vulnerability through poor implementation of the postMessage API could allow a webworker to be installed on an victims machine and increase the attack surface since the worker then gives the capabilities to perform cross origin communication using CORS and websockets.

Alternatively, traditional methods such as email spam and persistent XSS on popular websites can also be used. To ensure that the webworker remains executing while the users browser is open and so prolong the attacker's use of their system, tabbed browsing can be used where by multiple blank tabs are opened to reduce the possibility of the victim coming back to the main page and shutting down the webworker.

(Attack and Defense Labs, 2010a)

### 2.2.12 Client-side Storage

Client side storage now allows data to be stored persistently on the client which could possibly lead to persistent threats such as a XSS and CSRF attack. If a website has a XSS vulnerability it can be patched but with client side storage the threat will remain until storage is deleted. Since client side storage can hold much more data (up to 5MB

per origin) than cookies (4kB) then it becomes an obvious target for the attacker to steal possibly sensitive information such as login credentials. It also may create an issue when a PC has been stolen or compromised as once the attack gets into the computer they can retrieve the data on the client. It' possible to launch a one shot attack against a client using a XSS attack which contains SQLi code to extract all information on the three types of data stored on a client:

1. Session
2. Local
3. Database

HTML5CSdump was a script developed to launch this type of attack without any intervention from the user. So once a site is vulnerable to XSS then the attack can take place to retrieve client side storage data. XSS has always been considered a client side problem up until now as data is typically stored on a backend server database. The difference now is if the attack can get a stored XSS attack onto a vulnerable website then he can compromise the data on all clients who access the website – so the attack can be launched from one website to multiple clients. So a vulnerable website now becomes the central point for a distributed attack. Client side storage does restrict access to its data by SOP but this may not be enough to protect against multiple users on the same machine would could lead to compromise of data confidentiality and Integrity. Cross directory attacks may also be possible due to the fact that client-side storage is by origin and not path so all directories from the same origin would have access to the same client side data. The main issue is the fact that a JavaScript interface exists to access the storage which may lead to vulnerabilities with attacks such as XSS since once a site is vulnerable to XSS it has neutralized any protection to SQLi as the script in execution in the same origin as the local storage (Sutton, 2009) (Trivero, 2008).

If data is stored in client side databases and used in code evaluation constructs such as "eval()" then it will can be used to launch persistent XSS attacks and the data resides on the client and will be invoked automatically every time the code evaluation is called on the web page in question. As this all happens on the client side the server has no knowledge so can persist for a long time before the web application developers becomes aware of this issue. A XSS attack is able to write to the client side storage as once it's injected into the victim's browser via vulnerable web page it executes in the security context of the vulnerable web page and so can access the client side storage using the storage APIs since they reside in the same origin. This persistent attack could also happen through a network attack where a user gets exploited by a Man in the Middle

attack at the public wireless access point. The attacker can intercept the victim's packets to intercept malicious JavaScript which in turn gets executed into the victims client storage and all this being oblivious to the server. The client leaves the wireless access point but the persistent threat will remain until the client side database is cleared which doesn't happen until the creating application destroys it. Once the attacker has the persistent XSS injected into client side storage he has complete control over the application (Hanna et al, 2010).

Client –side databases or any client-side storage should not store sensitive information and should use authorization or access controls to retrieve the data. As a general rule "Any piece of info that would not be stored in clear-text on the server side should never be stored on the client-side in any form".

If there is a requirement to store sensitive information then:

1. Use SSL – database is restricted by SOP so can't be accessed by HTTP
2. Unique Database name – prevent attacker enumerating using XSS attack
3. Use input and output encoding on data fetched from database
4. To prevent SQLi then use prepared statements instead of concatenated – don't use variables

(Heiderich, 2011)

### 2.2.13 Websocket

The websockets protocol is being developed by the IETF but the API is being developed by the W3C. CORS sets up HTTP connections but websockets sets up a socket for better real-time communications. Websockets use raw socket access to allow web applications provide functionality that is difficult to provide with only HTTP networking APIs. They use an "in-band" consent protocol where the browser exchanges messages with the server over a socket before handing the socket over to the web application. The current version of the Websocket protocol is vulnerable to proxy cache attack poisoning attacks and has been removed from some browsers but is in the latest version of Google Chrome. The weakness discovered in the websocket protocol is due to the upgrade mechanism in the protocol which is not understood by all transparent proxies and so causes the handshake to succeed even though subsequent traffic over the socket will be misinterpreted by the proxy. A transparent proxy is one which handles requests between a client and server unknown to the client in that it appears they have a direct connection to the server. Barth et al propose the following solution which is being worked by the W3C and IETF:

1. Replace the "Upgrade" mechanism with the more commonly used "Connect" mechanism form the HTTP protocol which will eliminate misinterpretation by transparent proxies
2. Encrypt attacker-controlled portions of the initial handshake message

(Barth et al, 2010a)

### 2.2.14 HTML5 Cheatsheet

An attempt to create a well maintained, informative and categorized cheat sheet to highlight HTML5 as well as other client side and related security issues as well as ways to avoid them (Heiderich, 2011). This builds on previous work by Rsnake with regards XSS vector documentation (Hansen, 2008).

## 2.3. Google Chrome Browser Security

Traditionally browsers run in a single protection domain which does not provide any defense in depth if the browser is exploited due to vulnerability.

Dangers posed to users from a browser come from three factors:

1. The severity of vulnerabilities
2. The window of vulnerability
3. The frequency of exposure

The Google chrome sandbox aims to address the first factor, severity of vulnerabilities by sandboxing its rendering engine. A browser generally places web pages from different sites in different processes but this can be difficult when dealing with iframes in mashups. For now Google chrome sometimes places pages from different origins in the same process. The Google Chrome browser is separated into two modules:

1. Browser Kernel (Operating System Interaction)
2. Rendering Engine (Renders all web pages and JavaScript)

(Figure 2.3.1)

The Rendering Engine acts on web side and the Browser Kernel acts on users' behalf to ensure safe interaction between the web content and the users file system, thus providing a sandbox security (Figure 2.3.2). The main aim of the Chrome sandbox is to provide protection for a users file system in the case of a malicious web page taking advantage of unpatched browser vulnerability i.e. least privilege architecture. One draw back of this is that it runs plug-ins outside of sandbox so users still dependent on vendors supplying secure plug-ins. The Browser uses a sandbox approach, allowing web pages on different tabs to run in separate processes. Further research is being done

to enforce the same SOP by separating different origins into different processes but this is complex as need to take plug-ins into account. While the investigation of the Google Chrome Sandbox it is relevant that a secure design (Barth et al, 2008), can potentially be broken (Vupen, 2008).

**Figure 2.3.1** – The IPC is a secure channel between rendering and kernel engine

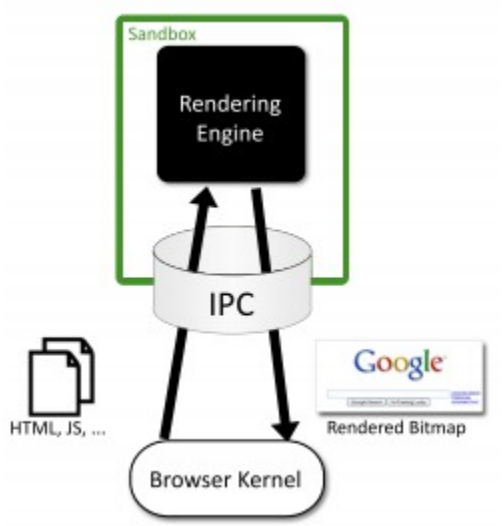



**Figure 2.3.2** – Assignment of tasks between the rendering engine and browser kernel

| Rendering Engine | Browser Kernel |
|---|---|
| HTML parsing | Cookie database |
| CSS parsing | History database |
| Image decoding | Password database |
| JavaScript interpreter | Window management |
| Regular expressions | Location bar |
| Layout | Safe Browsing blacklist |
| Document Object Model | Network stack |
| Rendering | SSL/TLS |
| SVG | Disk cache |
| XML parsing | Download manager |
| XSLT | Clipboard |

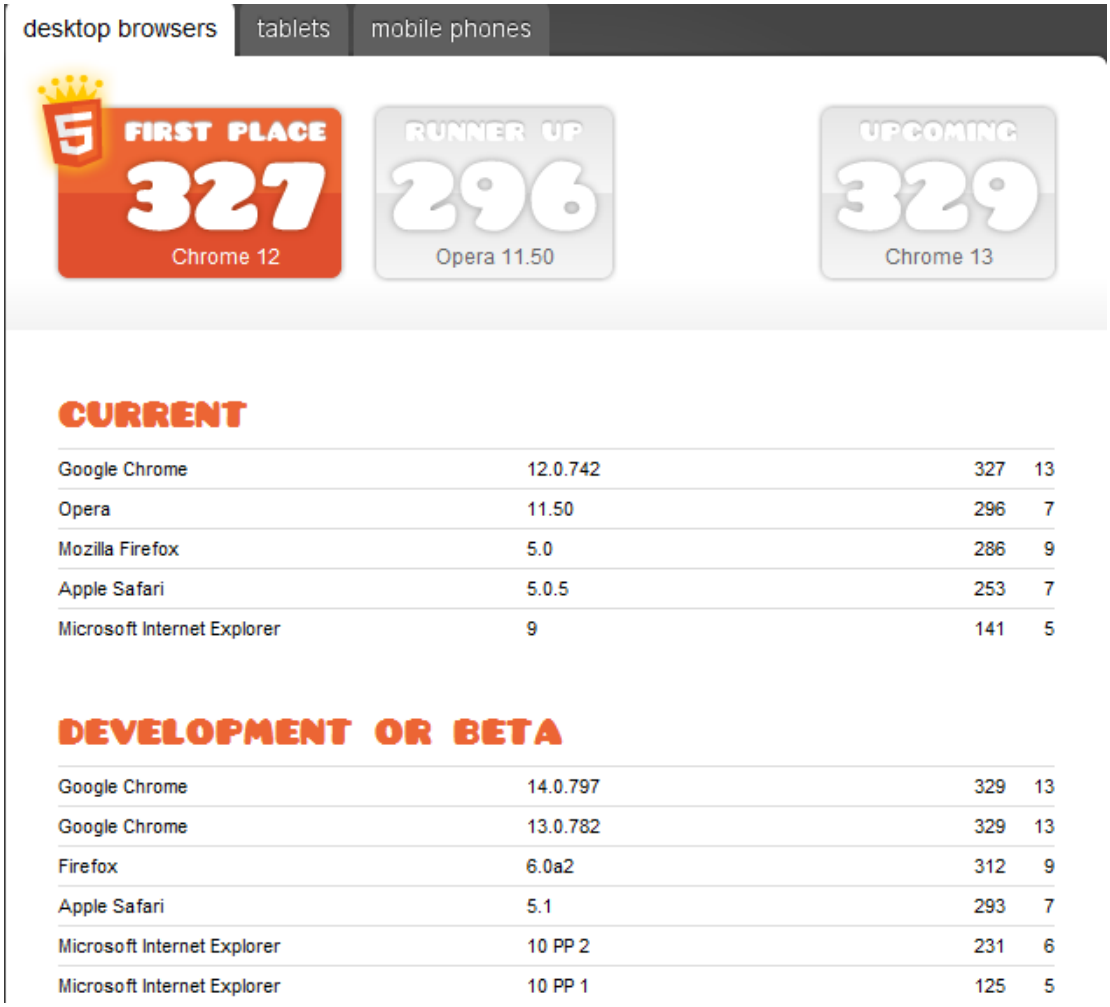
| Both |
|---|
| URL parsing |
| Unicode parsing |

Google is striving towards a revolutionizing the web as an operating system which is event from the release of their new Google Chromium Operating System (Risky Business, 2010).

## 2.4. Google Chrome Support for HTML5

Google Chrome was selected for this project as it's the most advanced browser in keeping up with the evolving HTML5 specification (Leenheer, 2011).

**Figure 2.4.1** – Browser Support for the HTML5 Specification which is ranked by a points system




As the browser updates automatically it can't be stated the exact version that was used throughout this project but it ranged from revision 10-12. Both revisions contained all the HTML5 functionality required for my implementation and testing which included:

1. Web Messaging
2. Websockets
3. CORS
4. Webworkers

**Note:** Google Chrome only supports reading for File API

## 2.5 Literature Review Summary

The HTML5 specification was designed with security in mind to create a Rich Interactive Application with desktop like performance i.e. the next generation of web application Web3.0. It aims to standardise web technologies by removing the need for plug-ins such as Flash and Adobe pdf but this may mean that HTML5 will take over the plug-in attack surface (OWASP, 2011). Even though there are a no. of security

"warnings" and statement throughout the specification as captured in figure 2.1.1.1, there have been many vulnerabilities and exploits documented. While the websocket protocol is a design issue, the majority of other security issues related to implementation issues, but one could argue that they may have been eliminated by higher level APIs, thus removing the security responsibility from the developer. The majority of the attacks summarized from my literature review require XSS to be performed as they require JavaScript executing in the context of a vulnerable webpage on a victims machine. Since XSS is the predominant web application attack, I focuse on a claim by (Hanna et al, 2010) to have suceeded injecting code using postMessage API. When combined with other attacks described in this attack it can increase the attacks surface due to the new JavaScript API capabilities of HTML5.

# CHAPTER 3
# Thesis Goal

This research takes a bottom up approach focusing on the client-side cross origin communication primitives and APIs in HTML5, to determine their use and dependencies throughout the HTML5 specification. While there has been much research into the security of new HTML5 primitives and APIs, there are been no single research that documents all the inter-dependencies of cross origin communication and their vulnerabilities within the HTML5 specification. Specifically this research aims to

determine whether the use of cross document messaging (postMessage API) throughout the HTML5 specification could increase an applications attack surface. As this specification is still a work in progress this thesis is heavily theory based with a practical implementation of a Financial HTML5 mashup to demonstrate how the postMessage API can be abused to provide a covert type channel to launch further attacks such as SQLi and CSRF, which effectively increases the attack surface due to the capabilities of new HTML5 JavaScript APIs. With the knowledge gained from this research I then propose a set of recommendations for both the W3C and developers to aid in secure deployment of HTML5 web applications.

# CHAPTER 4
# Analysis

## 4.1. Summary of Research Findings

If an attacker can get a victim to click on a vulnerable website using reflected or stored XSS then he can get a malicious script executing in the origin of the vulnerable website on the victims machine. Once an attacker has a malicious JavaScript executing on the clients' machine then new HTML5 APIs such as webworkers, websockets and CORS give rise to new attacks and amplify existing attack capabilities such as more powerful DDoS attacks. In section 1.2 I summarized a DOM based XSS attack using web 2.0 where JavaScript code is injected into AJAX and into the DOM using "innerHTML" without proper input validation. Hanna et al 2010, claim to have injected malicious code using the postMessage API but there has been no documented demonstration or other research on this. Applications from Google Connect and Facebook have already shown poor implementation with the postMessage API so I plan to follow the same route with a practical demonstration to determine if it's possible to launch a XSS attack as this has been mostly theoretical from my research findings (Hanna et al, 2010). This attack maybe possible if:

1. The Integrator puts excessive trust in a Gadget and fails to sanitize the incoming postMessage API data from the iframe containing the Gadget
2. Find a HTML5 XSS vector to get malicious JavaScript through Google Chrome XSS filter
3. The postMessage API response can be read using "innerHTML" and injected into DOM

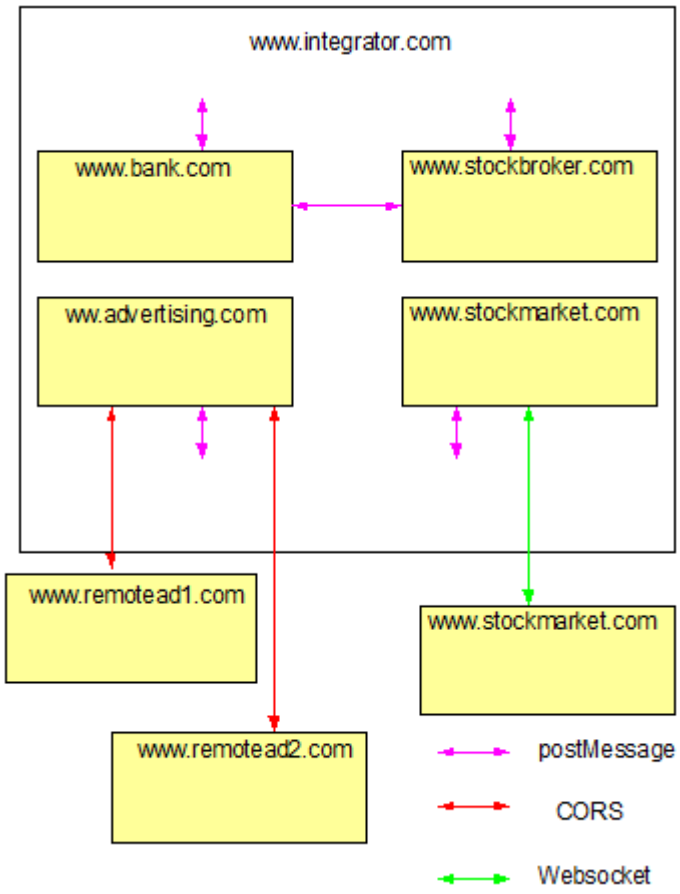
The postMessage API was designed as a string passing communication channel but because it still allows the user to read incoming messages using "innerHTML" as was shown early for web 2.0 using AJAX, it can lead to secure consequences when implemented incorrectly. Once an attacker can successfully inject a XSS attack through postMessage then they can access the client side storage database as this runs in the security context of the containing site. It may then be possible to perform frame navigation, launch webworkers as well as create remote shells (backdoors) back to the attacker using CORS and websockets. While all these attacks can happen by simply supplying the user with a malicious link to click on, this attack does not require any user interaction since the attack is persistent on the Gadget side as I will show and requires no end user interaction. One of the most powerful tools available to an attacker building an XSS exploit is being able to generate requests to the target website from the victims' browser being able to read the responses – cross origin communication using CORS and websockets now gives the attacker this capability (Zalewski, 2008).

## 4.2. Implementation Design

To put this XSS attack using the postMessage API into context and understand its possible severity, I use a financial mashup as an example (DeRyke, 2010).
Most mashups today are those which use Google maps API but enterprise mashups are on the rise as well as ecommerce mashups so this implementation is a good practical example. While websockets and CORS both use cross origin communication, this implementation focuses on the postMessage API, as once a XSS attack can be successfully executed then it can launch numerous attacks using the new powerful JavaScript API capabilities from HTML5. This implementation will be based on the postMessage code injection claim (Hanna et al, 2010), the HTML5 Cheatsheet (Heiderich, 2011), frame navigation (Barth et al, 2009), webworkers and CORS capabilities (Attacks and Defense Labs, 2010a) and DOM based XSS (Shah, 2010).
With Reflected and Stored XSS the user needs to be lured or tricked into clicking on a vulnerable or infected website. However, when DOM based XSS is combined with the new "onfocus" attribute in HTML5 it does not require any user interaction to execute the payload since it happens automatically once injected into the DOM (Heiderich, 2011). Figure 4.2.1 shows a Financial Mashup Design where content from multiple domains is intregated into one domain, called an Integrator, using Gadgets. All communication between APIs and Integrator will use the postMessage API on the client side for communication. Based on fact that, the Integrator could place excessive trust in the Gadgets incoming postMessage string and Frame Navigation, the attacker can

launch client side XSS, SQLi and CSRF (possible if the developer does not provide the correct protection)

**Figure 4.2.1** – Financial Mashup (DeRyke, 2010)



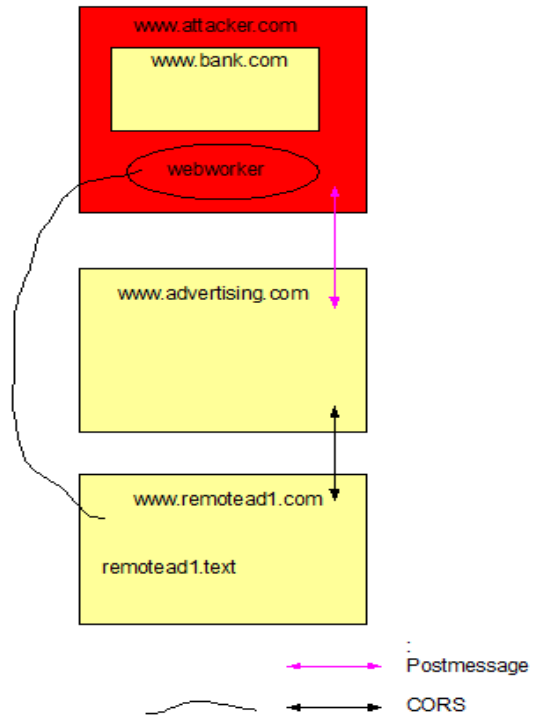

This Financial Mashup provides integrated access to financial and stock information. The mashup contains a Component or Gadget from a bank, an advising component from a brokerage firm, an advertising component and a stockmarket update. The Bank needs to communicate with its own server and with stockbroker for all transactions. The Advertising Gadget retrieves keywords from transactions to help user get latest available deals. The Stockmarket Gadget uses websockets to keep user up to date with the latest stock news real-time. The Advertising Gadget is using CORS to make cross origin requests to remoterad1 and remotead2 for advertising feeds.

This Integrator uses the origin check on postMessage API correctly but fails to use the correct method when writing the postMessage strings from the Gadgets into the Integrator DOM which leads to DOM based XSS in the Integrator. With the SOP, communication between iframes would have been prohibited but with postMessage this communication is now possible. Once a malicious script is successfully passed using postMessage API and executed in context of the Integrator webpage then numerous attacks outlined in Chapter would be possible, such as:

1. Create DoS attacks CORS
2. Create reverse shell or backdoor using CORs or websockets to attackers site
3. Access Client-side storage of Integrator website

A webworker cannot be created by using XSS as it takes an external JavaScript file as input which must be on server of website being attacker. This may be possible using other attacks vectors requiring user interaction but I did not pursue this any further. However to stay inline with the attack methodology in this design I use Frame Navigation in the XSS payload so that once executed will navigate the Bank Gadget to the attackers website which contains the Bank website as an iframe. This means that the victim has no idea this attack is happening since the navigation of an iframe does not show any change in the top level domain location in the browser URL and also since the attackers' webpage has no content it appears that the iframe containing the Bank gadget has not been navigated. Once the iframe containing the Bank website has been navigated to the attacker then the attacker can launch a webworker on the clients' machine. As per Figure 4.2.2 below, the XSS attack can be launch by an attacker compromising an advertisement feed file remotead1.txt on a remote server which gets retrieved by Advertising website using CORS and then passed into the Integrator using postMessage. Or as I performed in this implementation, this XSS payload could simply be a stored XSS attack on Advertising website which gets passed into Integrator also using postMessage.

**Figure 4.2.2** – PostMessage XSS used to




navigate iframe to attackers website

# CHAPTER 5
# Implementation

The main components in this implementation are:

1. Google Chrome Browser version 12.0.742.122 on a standard laptop using Windows XP
2. Backtrack VM image installed on windows XP
3. Virtual hosting setup on apache2 server in Backtrack5
4. Websocket server installed and setup on apache2 server
5. Host file on Windows XP was setup to resolve virtual domain names to backtrack Virtual Machine IP address

Backtrack5 was selected as the test environment as it's a self contained virtual image. Although there's no malicious network activity being performed, its good security practice to perform testing in a Virtual machine. In my implementation, I demonstrate how DOM based XSS can happen so silently with no user interaction or clicking on links required using HTML "onfocus" tag.

The following websites were created:

- www.integrator.com
- www.bank.com
- www.advertising.com
- www.stockbroker.com
- www.stockmarket.com
- www.remotead1.com
- www.remotead2.com
- www.attacker.com

All websites apart from www.integrator .com were placed in iframes and integrated into www.integrator.com as Gadgets, using the postMessage API for communication between Gadgets and the main Integrator on the client side.

A Websocket server was setup on www.stockmarket.com to demonstrate how this can be used for real-time communication.

CORS was used for the connection between www.advertising.com and it's remote cross origin feeds on www.remotead1.com and www.remotead2.com

See Appendix A for specific implementation configuration details.

# Chapter 6
# Testing and Analysis

The goal of testing the mashup from figure 4.2.1 was to determine if a XSS vector can be created to launch through the postMessage API to cause a DOM based XSS attack in the Integrator. None of the existing XSS vectors were possible to inject using postMessage (Hansen, 2008) as the Google Chrome Browser has a XSS filter which can catch these well known vectors. However the HTML5 cheat sheet capitalizes on the fact that HTML5 uses new Elements, Tags and Attributes which may not be detected by current browser of web application XSS filtering capabilities.

Appendix B documents successful XSS vectors I created using the HTML5 cheat sheet as a guide. There is a simple client-side storage XSS vector in Appendix B which shows how client-side data can be retrieved from local storage once an attacker can get a malicious JavaScript to execute within a vulnerable website on the victims machine. The following XSS vector was created which is passed from www.advertising.com when the www.integrator.com website is loaded and successfully navigates the www.bank.com Gadget iframe to the attacker's website of www.attacker.com:

```
var res = '<input type"text" value="" onfocus= "javascript: window.frames[0].location=\'http://www.attacker.com\'" autofocus>';
top.postMessage(res, targetOrigin);
```

Since the www.integrator.com web page does reads the postMessage data using "innerHTML", the injected javascript gets written to the DOM dynamically and executed without any page reload or user knowledge. Once the attacker has his script executing in the www.integrator.com webpage then he can intercept as all the users financial information.

The iframe navigation is not possible from iframe to iframe since Google Chrome implements the Descendant Policy. So therefore it was necessary to first get a foothold in the parent window (www.integrator.com) to allow the www.bank.com iframe to be navigated. This iframe navigation XSS vector could be replaced by a script from (Trivero, 2008) to automate Client side storage attack or SQLi using his script HTML5CSdump.

This implementation proves that if implemented incorrectly the postMessage API can be used to inject malicious JavaScript into the DOM for a DOM based XSS attack. Once the javascript is executing in the context of www.integrator.com then the attacker can create reverse shells back to his own website using CORS or websockets. More importantly the attacker can launch a client-side storage attack to write a persistent XSS and CSRF payload into the client which would get executed every time

the web page was visited by the victim. The key point to note here is that the cross origin capability of postMessage enabled the attack, but once inside the vulnerable website the attacker can then use further cross origin communication capabilities of HTML5 such as CORS and websockets to send data back to himself. This client-side communication or reverse shell (backdoor) was not possible with web2.0 as the SOP restricted this type of communication. Since CORS and websockets may use up system resources and slow down the end users experience the attacker can also use a more stealthy approach using webworkers. The webworkers API has the ability to initiate and read back CORS and websocket connection across domains and since it runs in the background is much stealthier. As webworkers require an external file located on the victims domains server when initiated, it was not possible to inject a webworkers payload through the postMessage API. There maybe scope for further research here such as uploading a webworker file using CORS to target server prior to injecting webworker command. To get around this, a webworker can simply be initiated under www.attacker.com when the www.bank.com iframe has been navigated at load up time per figure 4.2.2. If www.integrator and www.bank.com don't have explicit checks on the source and target origin checks for postMessage then www.attacker.com can form a MITM attack (Hanna et al, 2010). However the real threat of the webworker in this case is that it could be used for a very powerful distributed DoS or password cracking attack (Attack and Defense Labs, 2010a). The SOP prevents javascript from communicating across domains, however with the new capabilities of postMessage, webworkers, CORS and websockets a XSS worm or botnet could be realized. A traditional XSS worm propagates network to network but a XSS worm propagates from profile to profile or user to user. Since javascript executes in the context of the browsers sandbox and the language is platform independent it makes for easy development by malware authors as well as difficult detection. Now that the SOP has been relaxed by HTML5 using the VOP there is the likely that there will be an increase in XSS worm since they are no longer confined to the SOP (Risky Business, 2010).

While websockets and CORS were not tested directly the communication was captured using Wireshark in appendix B. Wireshark log of the websocket connection to www.stockmarket.com shows the upgrade handshake from HTTP to websocket. Wireshark log www.advertising.com shows the cross origin request to www.remotead1.com and www.remotead2.com with the "Access-Control-Allow-Origin=*".

The Google Chrome Developer tool snapshot in Appendix B also shows the different components of www.integrator.com loading including the websocket communication channel.

### Tool used:

Google Chrome Developers Tools and Wireshark

http://code.google.com/chrome/devtools/

http://www.wireshark.org/

# Chapter 7
# Recommendation for Secure Implementation

## 7.1. HTML5 Specification Recommendations

The W3C should take the following into account before making the HTML5 specification an official complete standard in 2022:

1. The HTML5 Specification is over 1000 pages and very complex to read as well as resource taxing so would be better for secure development to:

a. Create a high level functionality section which defines typical scenarios for HTML5 API functionality
b. Create a Cross Origin Communication section similar to figure 1.4.2 so that developers can understand the APIs interaction and dependencies and so understand the Data Flow and attack surface of the their applications to a higher degree – possibly using a UML model. This is very important as HTML5 is relaxing the single most important Browser Security Policy
c. Create a separate security dedicated section in the specification so developers can understand risks and consequences of poor implementation

2. Remove the separation between future drafts (figure 2.1.1.3) the current HTML5 specification
3. Review new APIs and determine whether possible to apply MAC as opposed to DAC and take security responsibility from the developer where by creating higher level APIs which would result in more consistent implementation. Currently seems to be much inconsistency especially with postMessage API (Hanna et al, 2010). PostMessage API could possibly be updated to disallow he use of "innerHTML" to prevent DOM based XSS
4. Develop new Security Testing Tools for HTML5
5. Add HTML5 threat model from European and Network Information Security Agency (ENISA) when published into the HTML5 specification under the new security from point 1c above (OWASP, 2011)
6. Aim to eliminate existing vulnerabilities which are inherent from previous HTML designs
7. Push as much Client side validation into the Browser as opposed to the developer so by creating MAC as opposed to DAC
8. The official release of HTML5 finished specification will not be released until 2022 so focus should be given to consolidate all HTML5 resources in one resource which would include:
   a. Specification
   b. Security and Testing tools (Shah, 2010)
   c. Threats
   d. Data Flow and Access Control

e. Integration capabilities of individual APIs in Mashups

f. Work with OWASP to ensure HTML5 Security is high priority and defenses for threats are well documented

9. Possibly define method for better client-side storage allocation with origin – currently this allocates storage by origin but is not path relative
10. Review potential for monitoring webworkers activity on a client so that malicious activity can be detected
11. Education and Implementation – a secure Design can be counteracted by lack of education and poor implementation

## 7.2. Developer Recommendations

Most of the HTML5 vulnerabilities and attacks can be prevented once all input is properly validated or sanitized i.e. proper protection against XSS. Developers should become very familiar with the OWASP top 10 to ensure they know how to defend against XSS, CSRF and SQLi which will help mitigate the majority of HTML5 attacks published to date. In addition:

1. Review dataflow and channels of communication thoroughly within you're application or mashup so that you understand all input and outputs to prevent against XSS attacks which would ultimately increase your attack surface if using other HTML5 cross origin APIs
2. Review documentation of Gadget APIs thoroughly to understand the interface capabilities and so reduce the risk of a vulnerability within your mashup
3. Don't store sensitive data in client-side databases – always ensure whether storing sensitive data in client-side storage or not to perform output sanitization so hat no JavaScript code can be evaluated and executed since this could lead to a persistent and stealthy XSS or potential CSRF attack as would take place unknown to the server
4. Use proper origin checks with postMessage, CORS and websockets to prevent MITM and session hijacking attacks – avoid wild card checks (*)
5. Don't base access control for CORS just on origin header need to use Credentials also with session management
6. Hanna et al, 2010 recommends an economy of liabilities which is a good approach to reduce the workload in postMessage origin checks and

reduce potential for inconsistent implementation across the same application or other applications

7. Client-side storage currently doesn't provide a path with origin when securing data from a website on a client – developers need to take this into consideration if they have public and sensitive information on the same origin but different paths – ensure authorization and access control on per user basis
8. Even when the origin check is verified for cross origin APIs such as postMessage then content should still be treated as malicious as a trusted Gadget or website may have become compromised. Ensure to use ""text.Content" and "JSON.parser" as opposed to "innerHTML" to prevent DOM based XSS (Lubbers, 2010)
9. Even though websockets is available in some browsers such as Google Chrome it is not safe to use as there is a weakness in the protocol design which leaves it vulnerable to attack (Barth et al, 2010a)
10. Do not use Drag and Drop API if you're website does not have clickjacking protection
11. Do not use iframe sandbox attribute with frame busting code as a clickjacking defense as this does counteracts the protection

# Chapter 8
# Conclusion

HTML5 is the latest HTML specification which has been designed with security in mind but even since its 1st draft release in January 2008 there has been numerous vulnerabilities and exploits documented. There have been design issues with regards the websocket protocol, the Drag and Drop API (no SOP restriction) and one could argue that the postMessage API suffers from design flaw as there is too much security left up to the developer. HTML5 specification is a major revamp of the HTML specification since HTML4 and combined with the large amount of security research documented this project required an extensive literature review. All research to date covers specific parts of the HTML5 specification but there has been no documented research which has reviewed the entire HTML5 specification for all cross origin capabilities to understand the complete attack surface. In this research I have consolidated much dispersed research into a document which also summarizes the complete cross origin interaction in the HTML5 specification which aims to help developers design and build secure HTML5 applications. I've also demonstrated a claim (Hanna et al, 2010) to have injected code into postMessage and proved that this attack can lead to the same DOM based XSS vulnerabilities present in web2.0 using AJAX. The main value gained from this research is the awareness for developers of the potential dangers of HTML5 if you're application is susceptible to XSS and also highlighting recommended guidelines for development. The HTML5 specification is perceived to be secure by design and the postMessage API a secure channel for communication between iframes, which is not the case and this research highlights the fact that XSS is still the prevalent web application weakness but combined with the new powerful HTML5 Javascript APIs will increase you're applications attacks surface.

Although I have not executed a SQLi or CSRF forgery attack these are all possible from my literature review once the payload can be injected via postMessage. Since this payload injection has been proved with the financial mashup implementation using postMessage, all these named attacks can be executed. While I have documented the cross origin communication capabilities of HTML5, I have only tested the postMessage API. There is scope to further test websockets protocol and CORS as well as the iframe sandbox but I focused more on postMessage and this is the channel where from all subsequent attacks can be launched. An interesting area for future research is the iframe sandbox attribute as there has been no documented security research on this. Given the fact that the Google Chrome sandbox which was designed with security in

mind has already claimed to been pwned (Vupen, 2011), the iframe sandbox would be a valuable area of research since it is a defense in depth feature of HTML5. My research could possibly have included more testing using the OWASP testing guide to identify possible new attacks but due to the volume of the literature and no. of documented security issues it was not possible due to the project timeline. With Google driving for a web operating system and the complexity of HTML5 specification there is a real need to review web applications from a security system perspective to account for access control and policies with all the new channels for data flow. The fact that the W3C has more system like APIs in development (figure 2.1.1.3) and the WHATWG is focusing on Peer-to-Peer networking highlights the fact that the web will soon be reflective of an actual operating system. With cloud and mobile computing combined with HTML5, end user authentication and authorization will become an important focal point requiring the W3C to possibly work with the Trusted Computing Group (TCG) to control information flow and access at hardware level using digital signatures.

Since XSS is currently the most prevalent web application attack suggests that developers are not aware of how to ensure their applications have had all inputs validated, and mistakes by Google and Facebook (Hanna et al, 2010) leads one believe that postMessage will fall to the same implementation vulnerabilities with regards XSS attacks as web2.0 Due to the fact that HTML5 relies some heavily on javascript if new primitives such a postMessage are not used securely then this will undoubtedly increase an applications attacks surface

# Primary References


Anderson, R., 2008. *Security Engineering: A Guide to Building Dependable Distributed Systems.* 2nd ed. Indianapolis: Wiley.

Attack and Defense Labs, 2010a. Attacking with HTML5. *Blackhat Abu Dhabi 2010.* Abu Dhabi, United Arab Emirates 10-11 November 2010.

Attack and Defense Labs, 2010b. "HTML5 Security" [online], Available at: http://www.andlabs.org/html5.html [Accessed 02 March 2011]

Barth, A., et al. (2010a) "Transparent Proxies: Threat or Menace?" [Online], Available at: http://www.adambarth.com/experimental/websocket.pdf [Accessed 27 April 2011]

Barth, A., et al, 2009. Secure Frame Communication in Browsers. The 17th SECURITY SYNOPSIUM 2008. SAN Jose, California, USA 28-29 July 2008.

Barth, A., et al, (2008) "The Security Architecture of the Chromium Browser", Available at: http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf [Accessed 17-Oct-2010]

Flanagan, D. (2011) "Javascript: The Definitive Guide", O'Reilly

Hanna, S., et al, (2010) "The Emperor's New APIs: On the (In) Secure Usage of New Client-side Primitives", W2SP 2010: Web2.0 Security and Privacy Conference 2010, [online], Available at: http://w2spconf.com/2010/papers/p03.pdf [Accessed 20-Oct-2010]

Heiderich, M. (2011) "HTML5 Security Cheatsheet Project" [online],
Available at:
http://code.google.com/p/html5security/w/list
[Accessed 18-Feb-2011]

Lubbers, et al, (2010) "Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development", Apress

OWASP, (2011) "OWASP Summit Browser Security Track" [online],
Available at:
https://www.owasp.org/index.php/Category:Summit_2011_Browser_Security_Track_
[Accessed 01 March 2011]

Rydstedt, G., et al, 2010. Busting Frame Busting: A Study of Clickjacking Vulnerabilities on Popular Sites. *W2SP 2010: Web2.0 Security and Privacy Conference 2010.* Oakland, California, United States May 20 2010.

Shah, S., 2010. Hacking Browser's DOM -Exploiting Ajax and RIA. *Blackhat USA 2010*. Nevada, Las Vegas, United States 2009.

Sutton, M., 2009. A Wolf in Sheeps Clothing: The Dangers of Persistent Web Browser. *Blackhat USA 2009*. Las Vegas, Nevada, USA 25-30 July 2009.

Risky Business, 2010. [podcast] 2010. Available at:
http://media.risky.biz/auscert2010/RB2-AC-sutton.mp3
[Accessed 20-September 2010]

Trivero, A., (2008) "Abusing HTML5 Structures Client Side Storage" [online],
Available at:
http://trivero.secdiscover.com/html5whitepaper.pdf
[Accessed 05 November 2010]

WHATWG, (2011) "HTML5 Draft Standard" [online],
Available at:

http://www.whatwg.org/specs/web-apps/current-work

[Accessed 21 October 2010]

W3C, (2011a) "Web Security Wiki" [online],

Available at:

http://www.w3.org/Security/wiki/Main_Page

[Accessed 15 February 2011]

W3C, (2011b) "HTML5 Development Spec" [online],

Available at:

http://dev.w3.org/html5/

[Accessed 07-Jul-2011]

W3C, (2011d) "CORS" [online],

Available at:

http://www.w3.org/TR/cors/

[Accessed 07-Jul-2011]

Zalewski, M., (2011), "The Browser Security Handbook" [online],

Available at:

http://code.google.com/p/browsersec/wiki/Main

[Accessed 17-Oct-2010]

## Secondary References

Aghaee, S., Pautasso, C., 2010 "Mashup Development with HTML5" [online],

Available at:

http://portal.acm.org/citation.cfm?id=1945009

[Accessed 01 July 2011]

Barth, A., (2010b) "The Web Origin Concept" [online],

Available at:

http://tools.ietf.org/html/draft-abarth-origin-09

[Accessed 05 November 2010]

De Ryke, P., et al, 2010. Towards Building Secure Web Mashups.*OWASP Appsec Research 2010.* Stockholm, Sweden 21-24 June 2010.

Google, (2010a) "HTML5 Rocks" [online],

Available at:

http://www.html5rocks.com/

[Accessed 20 November 2010]

Google, (2010b), "pysocket" [online],

Available at:

http://code.google.com/p/pywebsocket/

[Accessed 22 October 2010]

Hansen, R. (2008) "XSS (Cross Site Scripting) Cheat Sheet" [online],

Available at:

http://ha.ckers.org/

[Accessed 19 February 2011]

Leenheer, N. (2011) "The HTML5 Test" [online],

Available at:

http://html5test.com/

[Accessed 20-Feb-2011]

Mansfield-Divine, S., (2010)"Divide and conquer: the threats posed by hybrid apps and HTML 5", Network Security, Volume 2010, Issue 3, March 2010, Pages 4-6 [online],

Available at:

http://www.webvivant.com/feature-divide-and-conquer.html

[Accessed 05 November 2010]

McAfee Labs, (2009) "2010 Threat Predictions" [online],
Available at:
http://www.mcafee.com/us/local_content/white_papers/7985rpt_labs_threat_predict_1209_v2.pdf
[Accessed 17-Oct-2010]

OWASP, (2010a) "OWASP Top 10 for 2010" [online],
Available at:
http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[Accessed 20-Nov-2010]

Ridley, S., 2010. Escaping The Sandbox. *Blackhat Abu Dhabi 2010*. Abu Dhabi, United Arab Emirates 10-11 November 2010.

Stone, P., 2010. Next Generation Clickjacking. *BlackHat Europe 2010*. Barcelona, Spain April 12-15 2010.

Stuttard, D., Pinto, M. (2008) "The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws", Wiley

Trend Micro, (2009) "The Future of Threats and Threat Technologies: how the landscape is changing" [online],
Available at:
http://affinitypartner.trendmicro.com/media/34716/trend_micro_2010_future_threat_report_final.pdf
[Accessed 17 October 2010]

Vupen, (2011) "Google Chrome Pwned by VUPEN aka Sandbox/ASLR/DEP Bypass" [online],
Available at:
http://www.vupen.com/demos/
[Accessed 10 May 2011]

Wright, A., (2009) "Ready for a Web OS" [online],

Available at:

http://portal.acm.org/citation.cfm?id=1610260

[Accessed 20-Nov-2010]

W3C, (2011c)"ALL STANARDS AND DRAFTS" [online],

Available at:

http://www.w3.org/TR/#tr_Javascript_APIs

[Accessed 07-Jul-2011]

Zang, N., et al. (2008) "Mashups: Who? What? Why?" [Online],

Available at:

http://portal.acm.org/citation.cfm?

id=1358826&dl=ACM&coll=DL&CFID=34351625&CFTOKEN=56267493

[Accessed 01 July 2011]

Hickson, I., (2011) "WebSockets Protocol" [online],

available:

http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-10

[accessed on 17-Jul-2011]

# Appendix A Implementation Details

- Components in this implementation were:

1. Google Chrome Browser was installed on a standard laptop using Windows XP

   http://www.google.com/landing/chrome/beta/

2. Backtrack VM image was installed on Windows XP Laptop

   http://www.backtrack-linux.org/backtrack/backtrack-5-release/

3. Virtual hosting setup on apache2 server in Backtrack5

4. Websocket server installed and setup on apache2 server

5. Host file on Windows XP was setup to allow all virtual domain names to be resolved to the Backtrack5 VM IP address
6. CORS Header Files were wild carded on server to allow access from any domain

**Note:** Backtrack5 was selected as the test environment as it's a self contained virtual image. Even though there's no malicious network activity being performed as all communication is being performed between multiple virtually hosted websites on this single localised server implementation, it's good security testing practice.

- The following Virtual Domains were setup on the Apache 2 server using instructions from links provided below:

www.integrator.com

www.bank.com

www.advertising.com

www.stockbroker.com

www.stockmarket.com

www.attacker.com

www.remotead1.com

www.remotead2.com

http://httpd.apache.org/docs/2.0/vhosts/examples.html

https://help.ubuntu.com/10.04/serverguide/C/httpd.html

http://www.debian-administration.org/articles/412

The actual website HTML files were created in /var/www

The actual virtual domain setup was done under /etc/apache2

- CORS: Setup was straight forward as the XHR2 Header only needed to be changed to Access-Control-Allow-Origin "*" in sites files located at /etc/apache2/sites-available
- Websockets – this setup was more difficult and required the following changes:

1. Installed mod_python and py_websocket to allow websocket server using instructions from following links:

http://code.google.com/p/pywebsocket/

http://chewpichai.blogspot.com/2007/08/install-apache2-and-modpython-on-ubuntu.html

http://www.travisglines.com/web-coding/how-to-set-up-apache-to-serve-html5-websocket-applications-with-pywebsocket

(Google, 2010b)

2. Note mod_python had to be installed prior to mod_pywebsocket
3. These install steps lead to a simple websocket echo server where the server returns the same text supplied from the client –a Handler for incoming websockets connections needs to be setup on the websocket server on the domain in question. In this case the websocket server is running on www.stockmarket.com so was setup at

   */home/stockmarket/websocket_handlers*
4. */etc/apache2/httpd.conf* file had to be updated with mod_pywebsocket for websocket module to run with HTTP server
5. To prevent the websocket from timing out every 10 seconds the folllowing file has to be updated:

*/etc/apache2/mods-available/reqtimeout.conf*

- The hosts file was configured on Laptop with Windows XP as follows:

(Files included on Disks provided)

# Appendix B XSS Attack Vectors

## These Scripts vector gets executed without any user interaction as

```
<input type"text" value="" onfocus= "javascript: function eoin(){var carter1 = 'carroll';
alert(carter1);};eoin()" autofocus>
<input type'text' value= onfocus= 'javascript: function eoin(){var carter1 = \'carroll\';
alert(carter1);};eoin()' autofocus>
```

## This script gets executed once victim clicks button - could be used with clickjacking

```
<form><button formaction="javascript:function eoin(){var name ='eoin carroll woz
here';alert(name);}eoin()">X
```

## Script can inject an iframe to allow javascript i.e. disable iframe sandbox

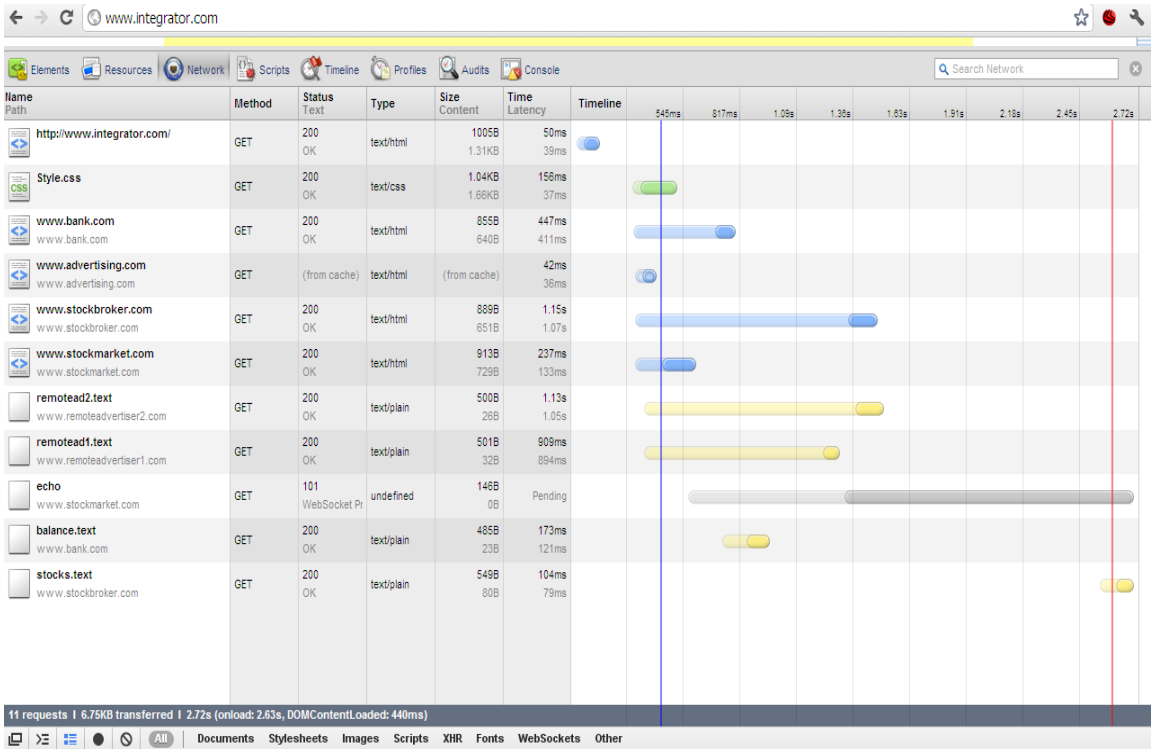
```
<iframe sandbox="allow-same-origin allow-forms allow-scripts"
src="http://example.org/"></iframe>
```

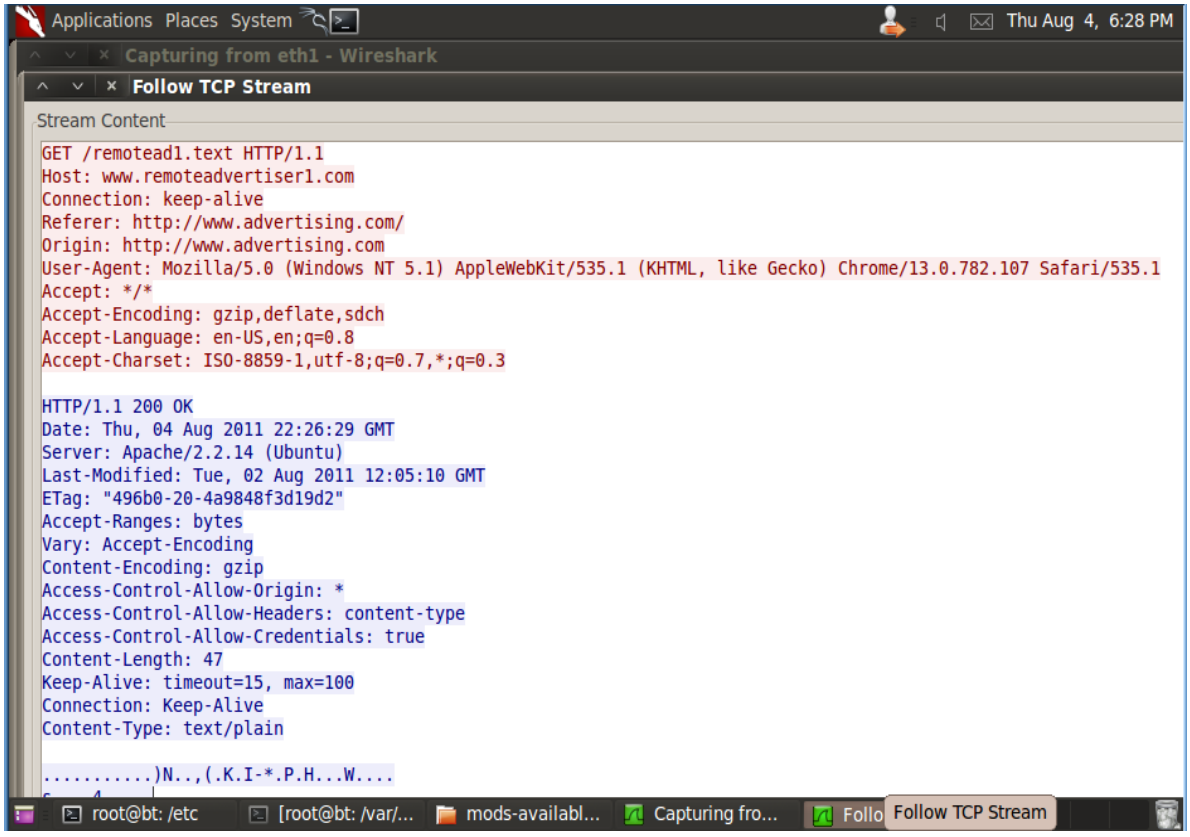
## Script can steal client-side storage once injected using postmessage

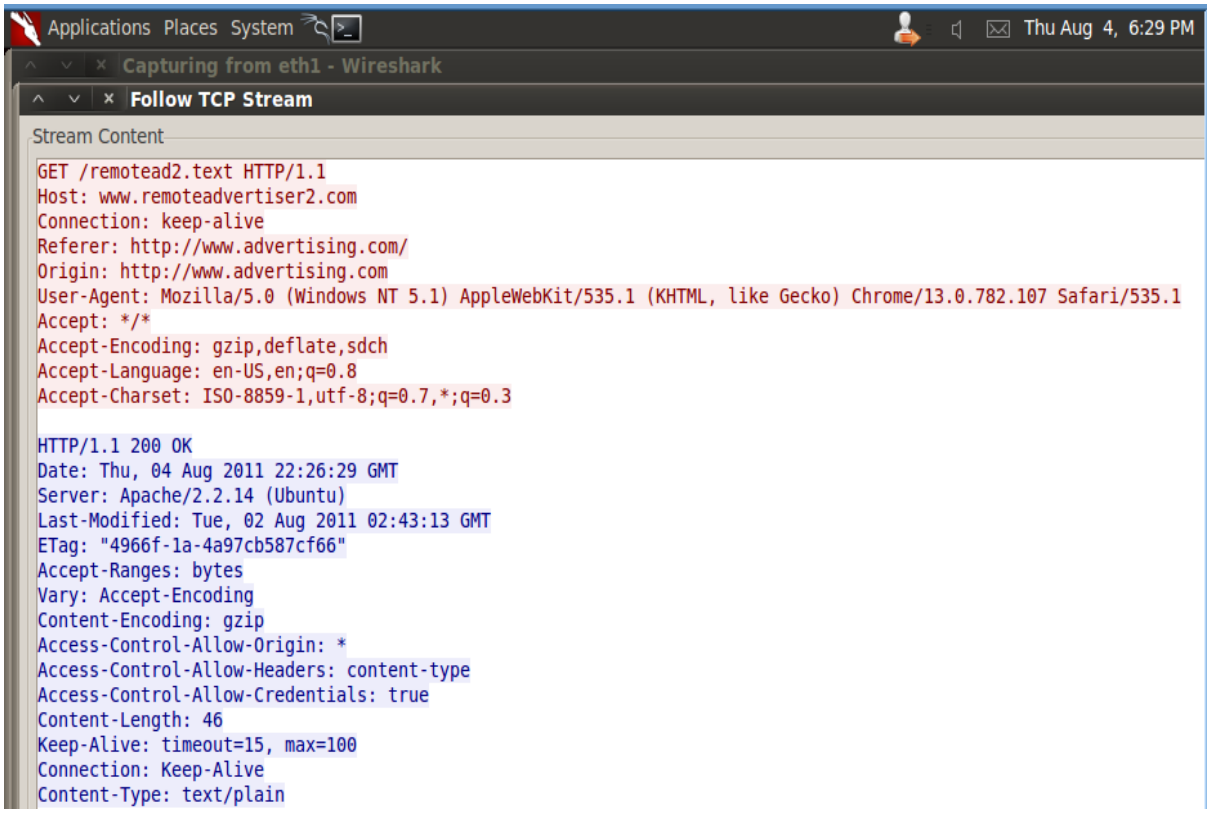
```
<input type"text" value="" onfocus= "javascript: function eoin()
{sessionStorage.lastname='Eoin
```

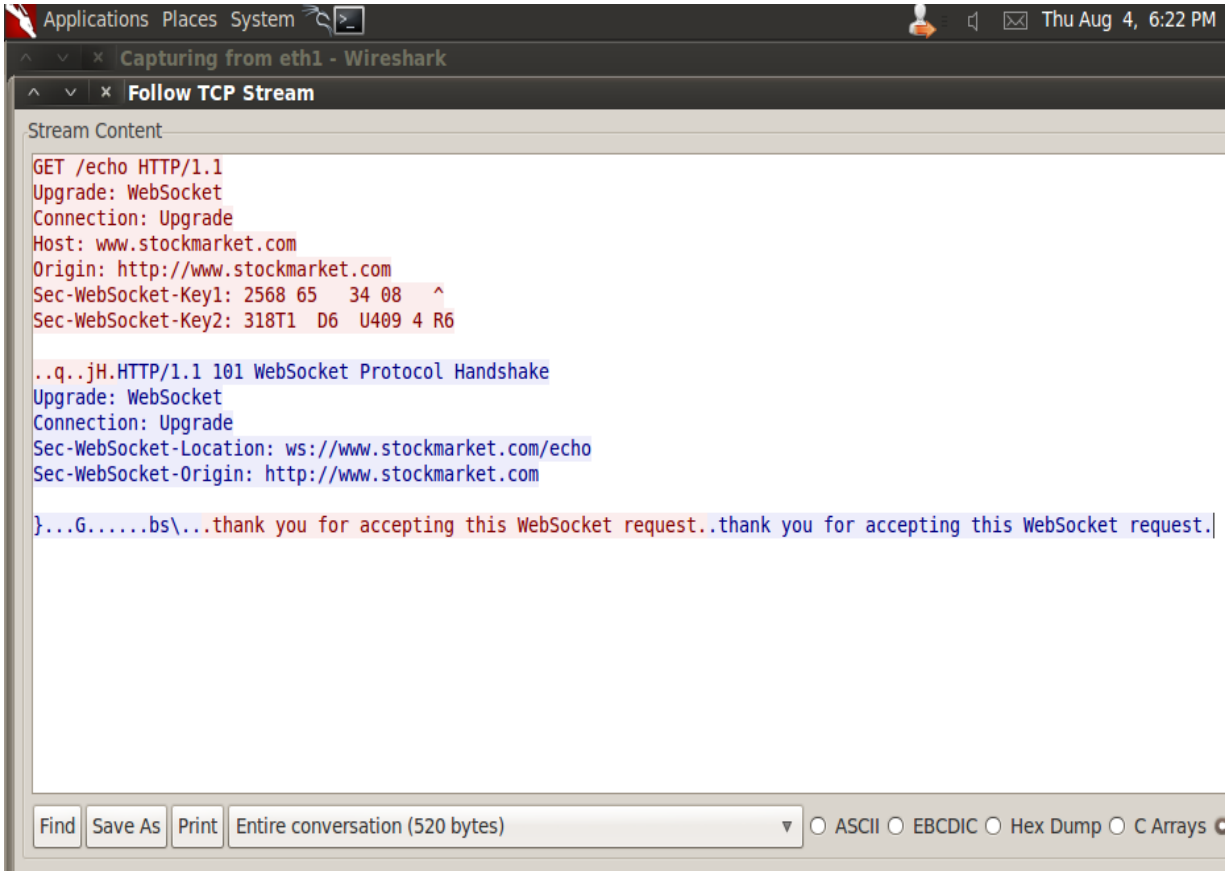
Carroll';document.write(sessionStorage.lastname);window.top.postMessage(sessionStorage.lastname, 'http://www.test.com');};eoin()" autofocus>

# Appendix C Wireshark and Google Chrome Developer Tool




Applications Places System
Thu Aug 4, 6:28 PM
Capturing from eth1 - Wireshark
Follow TCP Stream
Stream Content
GET /remotead1.text HTTP/1.1
Host: www.remoteadvertiser1.com
Connection: keep-alive
Referer: http://www.advertising.com/
Origin: http://www.advertising.com
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/13.0.782.107 Safari/535.1
Accept: */*
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
HTTP/1.1 200 OK
Date: Thu, 04 Aug 2011 22:26:29 GMT
Server: Apache/2.2.14 (Ubuntu)
Last-Modified: Tue, 02 Aug 2011 12:05:10 GMT
ETag: "496b0-20-4a9848f3d19d2"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Credentials: true
Content-Length: 47
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/plain
..........)N..,(.K.I-*.P.H...W....
root@bt: /etc
[root@bt: /var/...
mods-availabl...
Capturing fro...
Follo
Follow TCP Stream


Applications Places System
Thu Aug 4, 6:29 PM
Capturing from eth1 - Wireshark
Follow TCP Stream
Stream Content
GET /remotead2.text HTTP/1.1
Host: www.remoteadvertiser2.com
Connection: keep-alive
Referer: http://www.advertising.com/
Origin: http://www.advertising.com
User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/13.0.782.107 Safari/535.1
Accept: */*
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
HTTP/1.1 200 OK
Date: Thu, 04 Aug 2011 22:26:29 GMT
Server: Apache/2.2.14 (Ubuntu)
Last-Modified: Tue, 02 Aug 2011 02:43:13 GMT
ETag: "4966f-1a-4a97cb587cf66"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Credentials: true
Content-Length: 46
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/plain


Applications Places System
Thu Aug 4, 6:22 PM
Capturing from eth1 - Wireshark
Follow TCP Stream
Stream Content
GET /echo HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: www.stockmarket.com
Origin: http://www.stockmarket.com
Sec-WebSocket-Key1: 2568 65 34 08 ^
Sec-WebSocket-Key2: 318T1 D6 U409 4 R6
..q..jH.HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Location: ws://www.stockmarket.com/echo
Sec-WebSocket-Origin: http://www.stockmarket.com
}...G.....bs\...thank you for accepting this WebSocket request..thank you for accepting this WebSocket request.
Find
Save As
Print
Entire conversation (520 bytes)
ASCII
EBCDIC
Hex Dump
C Arrays