ATTACK & DEFENSE
labs

Attacking with HTML5

By,

Lavakumar Kuppan
www.andlabs.org
October 18, 2010

Introduction:

HTML5 is redefining the ground rules for future Web Applications by providing a rich set of new features and by extending existing features and APIs. HTML5 Security is still an unexplored region because HTML5 features are not yet adopted by web applications (apart from experimental support) and it is assumed that until that happens the end users have nothing to worry about.

This paper would prove this assumption wrong by discussing a range of attacks that can be carried out on web users 'right now' even on websites that do not support or intend to support HTML5 in the near future. Browser vendors have been trying to outdo each other in supporting the latest features defined in the HTML5 spec. This has exposed the users of these browsers to the attacks that would be discussed in this paper.

The initial sections of this paper cover attacks and research that have been published by me and other researchers earlier this year. The latter sections covers attacks that are completely new and exclusive.

The list of attacks covered:

1) Cross-site Scripting via HTML5
2) Reverse Web Shells with COR
3) Clickjacking via HTML5
    a. Text-field Injection
    b. IFRAME Sandboxing
4) HTML5 Cache Poisoning
5) Client-side RFI
6) Cross-site Posting
7) Network Reconnaissance
    a. Port Scanning
    b. Network Scanning
    c. Guessing user's Private IP
8) HTML5 Botnets
    a. Botnet creation
        i. Reaching out to victims
        ii. Extending execution life-time
    b. Botnets based attacks
        i. DDoS attacks
        ii. Email spam
        iii. Distributed Password Cracking

Cross-site Scripting via HTML5:

HTML5 introduces new elements that contain event attributes and new event attributes for existing tags. These event attributes can be used for executing JavaScript by bypassing blacklist based filters designed blocking Cross-site Scripting attacks.

A filter that only looks for known malicious tags can be bypassed using the new HTML5 Audio and Video tags.

Example:

```
<video onerror="javascript:alert(1)"><source>

<audio onerror="javascript:alert(1)"><source>
```

Filters that block '<' and '>' can prevent tag injection in most cases. XSS could still be possible in such cases if the attacker is able to inject script inside an existing event attribute or add a new event attribute. A filter that blocks existing event attributes can be bypassed by the new event attributes added in HTML5 like 'onforminput' and 'onformchange'.

Example:

```
<form id=test onforminput=alert(1)> <input> </form> <button form=test onformchange=alert(2)>X
```

Apart from aiding in bypassing filters some new features can be used to automate script execution like the HTML5 'autofocus' attribute. When this attribute is set an element automatically gets focus.

Cases where data injection is possible inside the attribute section of an Input tag is common. In such cases traditionally the injected JavaScript would be placed inside the 'onmouseover' or 'onclick' tag and user interaction would be required to execute the script. With HTML5 the 'onfocus' tag can be used for injecting script and then by setting the autofocus attribute we can trigger the script execution automatically.

Example:

Before HTML5:

```
<input type="text" value="-->Injecting here" onmouseover="alert('Injected value')">
```

With HTML5:

```
<input type="text" value="-->Injecting here" onfocus="alert('Injected value')" autofocus>
```

Mario Heiderich maintains a list of all such new HTML5 vectors in the HTML5 Security CheatSheet [1].

Reverse Web Shells with COR:

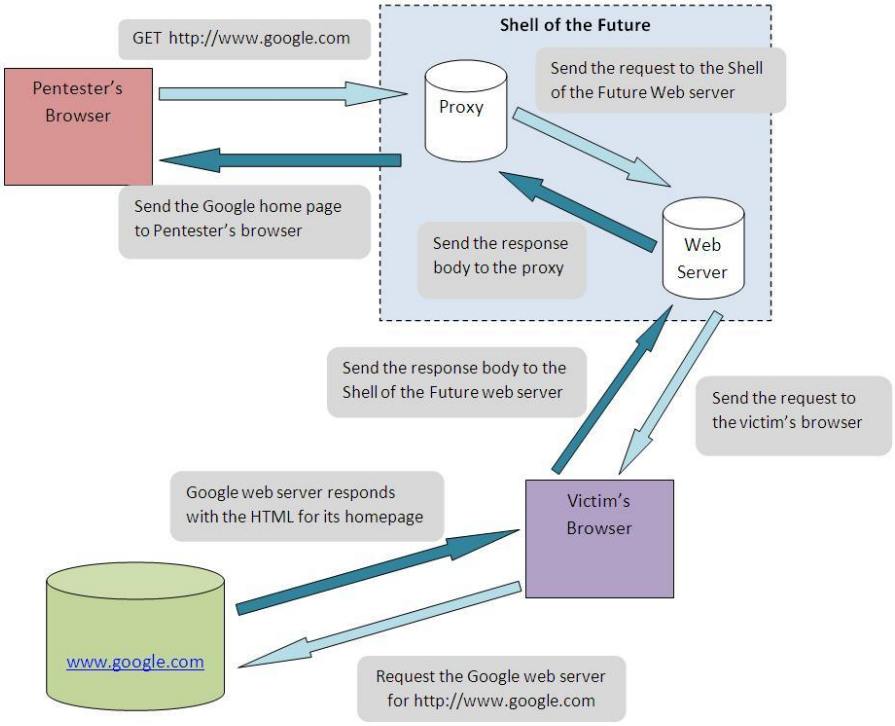
HTML5's Cross Origin Request allows browsers to make cross domain Ajax calls from a.com to b.com and read the response as long as b.com allows it. This feature can be used to tunnel HTTP traffic over cross domain Ajax calls and set-up a browser equivalent of a reverse shell.

By doing this an attacker can hijack a victim's session using XSS even if anti-session hijacking measure like Http-Only cookie and Session ID-IP address binding are used.

Once the JavaScript payload is injected to the victim's browser either through Cross-site Scripting or by convincing the victim to paste the scripting in the browser's address bar, the script starts talking to the attacker's server through Cross Origin Requests. Using this connection the attacker can browse the victim's affected session by tunneling his requests through the victim's browser.

Earlier this year I had released an open source tool named 'Shell of the Future'[2] which is an implementation of this idea. It is extremely easy to use as it automates the entire attack and comes with two default JavaScript payloads.

Architecture of Shell of the Future



Clickjacking via HTML5:

Text-field Injection:

ClickJacking can be used to submit forms across domains by bypassing the CSRF protection. Though it is very easy to click links or buttons through ClickJacking, populating the Input fields of the target form is relatively harder to do.
HTML5's Drag and Drop API can be used to fill the target forms simply by convincing the victim to perform a Drag and Drop action. The attacker's site can camouflage the attack as a game that requires the player to drag and drop items while invisibly attacker controlled data is populated in to the input fields of the target form.

Example:

```
<div draggable="true" ondragstart="event.dataTransfer.setData('text/plain', 'Evil data')">
<h3>DRAG ME!!</h3>
</div>
```

This method was introduced by Paul Stone at BlackHat Europe 2010[3] .

IFRAME Sandboxing:

There is a general misconception that including Framebusting code in each page of the site is the best way to defend against Clickjacking attacks. This approach appears to be the most popular solution as well even though the OWASP guidelines[4] clearly mention its disadvantages.
If a websites' only defense against ClickJacking attacks is FrameBusting then this protection can be bypassed in a few different ways. One of them is to use the IFRAME 'sandbox' attribute which is part of HTML5.

Example:

```
<iframe src="http://www.victim.site" sandbox></iframe>
```

Setting this attribute disables JavaScript within the iframe. Since 'framebusting' relies on JavaScript, this attributes effectively neutralizes the defense. Popular sites like eBay, WordPress, PayPal rely only on 'framebusting' for protection and are hence open to this attack.

HTML5 Cache Poisoning:

HTML5 introduces a new type of caching system called as the Application Cache or the programmable cache. While traditional caches are meant to improve page load times, the Application Cache is designed to enable Offline web browsing. Hence this cache is more persistent than traditional cache.

By poisoning HTML5 caches an attacker can have his cached pages alive for longer durations and use it to steal the user's credentials[5].

Earlier this year I released a new version of Imposter[6] that can be used to poison HTML5 caches.

Client-side RFI:

Websites that perform Ajax requests to URLs mentioned in the location hash and include the response in the HTML of the page can be exploited using COR. By getting the victim to click on a link that includes the URL of an attacker controlled page in the Location hash, it is possible to perform client-side RFI resulting in a Cross-site Scripting attack. This attack was discovered by Matt Austin in July this year[7]. Mobile.facebook.com and other many other websites including the JQuery library was found to be vulnerable to this.

Cross-site Posting:

This is a variation of the Client-side RFI attack discussed earlier. If the URL of the Ajax request can be controlled by an attacker, like in the case of location hash then an attacker can redirect legitimate requests to his site and steal sensitive session information. Even though the response of the Ajax request might not be processed by the requesting site this attack would work.

Network Reconnaissance:

Cross domain XMLHttpRequests and WebSockets can be used for performing reliable port scans. The latest version of Firefox, Chrome and Safari support both these features and can be used for intranet reconnaissance.

Cross domain XHR has five possible readystate statuses and WebSocket has four possible readystate statuses. When a new connection is made to any service the status of the readystate property changes based on the state of the connection. This transition between different states can be used to determine if the remote port to which the connection is being made is either open, closed or filtered.

Port Scanning:

When a WebSocket or COR connection is made to a specific port of an IP address in the internal network the initial state of WebSocket is readystate 0 and for COR its readystate 1. Depending on the status of the remote port, these initial readystate statuses change sooner or later. The below table shows the relation between the status of the remote port and the duration of the initial readystate status. By observing how soon the initial readystate status changes we can identify the status of the remote port.

Behavior based on port status:

Port Status	WebSocket (ReadyState 0)	COR (ReadyState 1)
Open (application type 1&2)	< 100 ms	< 100 ms
Closed	~1000 ms	~1000 ms
Filtered	> 30000 ms	> 30000 ms

There are some limitations to performing port scans this way. The major limitation is that all browser's block connections to well known ports and so they cannot be scanned. The other limitation is that these are application level scans unlike the socket level scans performed by tools like nmap. This means that based on the nature of the application listening on a particular port the response and interpretation might vary.

There are four types of responses expected from applications:

1. **Close on connect:** Application terminates the connection as soon as the connection is established due to protocol mismatch.
2. **Respond & close on connect:** Similar to type-1 but before closing the connection it sends some default response
3. **Open with no response:** Application keeps the connection open expecting more data or data that would match its protocol specification.
4. **Open with response:** Similar to type-3 but sends some default response on connection, like a banner or welcome message

The behavior of WebSockets and COR for each of these types is shown in the table below.

Behavior based on application type:

Application Type	WebSocket (ReadyState 0)/ COR (ReadyState 1)
Close on connect	< 100 ms
Respond & close on connect	< 100 ms
Open with no response	> 30000 ms
Open with response	< 100 ms (FF & Safari) | > 30000 ms (Chrome)

Network Scanning:

The port scanning technique can be applied to perform horizontal network scans of internal networks. Since both an open port and a closed port can be accurately identified, horizontal scans can be made for ports like 3389 that would be unfiltered in the personal firewalls of most corporate systems.

Identification of an open or closed port would indicate that a particular IP address is up.

Guessing User's Private IP Address

Most home user's connected to WiFi routers are given IP addresses in the 192.168.x.x range. And the IP address of the router is often 192.168.x.1 and they almost always have their administrative web interfaces running on port 80 or 443.

These two trends can be exploited to guess the private IP address of the user in two steps:

Step 1: Identify the user's subnet

This can be done by scanning port 80 and/or 443 on the IP addresses from 192.168.0.1 to 192.168.255.1. If the user is on the 192.168.3.x subnet then we would get a response for 192.168.3.1 which would be his router and thus the subnet can be identified.

Step 1: Identify the IP address

Once the subnet is identified we scan the entire subnet for a port that would be filtered by personal firewalls, port 30000 for example. So we iterate from 192.169.x.2 to 192.168.x.254, when we reach the IP address of the user we would get a response (open/closed) because the request is generated from the user's browser from within his system and so his personal firewall does not block the request.

HTML5 Botnets:

Every time a user clicks on a link he is giving a remote website an opportunity to execute code (JavaScript) on his machine. The window of this opportunity is widened by the concept of tabbed browsing. Most users have multiple open tabs and most tabs remain open through the entirety of their browsing session which could stretch for hours.

This enables an external entity to utilize the user's processing power and bandwidth for his malicious needs. Spammers, especially on sites like Twitter, have been able to get thousands of users to click on their links in very short durations. But JavaScript has serious performance constraints and is severely handicapped by the browser's sandbox which has limited such abuse.

HTML5 introduced WebWorkers which is a threading model for JavaScript. This lets any website start a background JavaScript thread unknown to the user and execute code without slowing down or making the browser unresponsive.

Botnet Creation:

An HTML5 botnet would include thousands of systems that have the attacker controlled page open on their browsers for an extended duration allowing continued execution of the attacker's JavaScript.

There are two phases in building such a botnet:

1) Reaching out to victims
2) Extending execution lifetime

Reaching out to victims:

This involves getting the victim to visit an attacker controlled website. This can be done in a number of different ways:

1) Email spam
2) Trending topics on Twitter
3) Persistent XSS on popular websites, forums etc
4) Search Engine Poisoning
5) Compromised websites

These are methods used by current JavaScript malware authors to attack victims to their website and can draw thousands of victms. While traditional malware spreading website can be quickly identified due to automated crawlers looking for signatures of browser exploits, HTML5 based payloads are less likely to be identified since its regular JavaScript working within the constraints of the sandbox and does not perform any exploitation against the browsers.

Extending execution lifetime:

Once a victim visits the attacker controlled page it is essential to keep this page open in the victim's browser for as long as possible. This can be done by using a combination of Clickjacking and Tabnabbing.

When the page is loaded, it would contain an invisible link with the target attribute set to '_blank'. This link is always placed under the mouse pointer using the 'document.onmousemove' event handler. This way, when the victim clicks anywhere on the page a new tab opens and grabs the victim's attention. With multiple tabs open the likelihood of the victim coming back to the main tab and closing it is reduced.

To add to this effect Tabnabbing can be used to refresh the page after the user leaves it, to update the favicon and appearance to seem similar to popular websites like YouTube, Google or Facebook so that the page blends in with the other tabs the victim would usually have open.

Botnets based attacks:

The following attacks can be performed using an HTML5 botnet:

1) Application-level DDoS attacks
2) Email Spam
3) Distributed password cracking

DDoS Attacks:

Application-level DDoS attacks on websites are becoming increasingly common with even sites like Twitter being affected. Usually these attacks involve large number for HTTP requests to specific sections of the website that could potentially be resource intensive for the server to process.

Background JavaScript threads that were started using WebWorkers can send cross domain XMLHttpRequests even though the remote website does not support it. The Cross Origin Request security restriction is only on reading the response.

A website that does not support Cross Origin requests will also process these request thereby creating load on the server. A simple request like http://www.target.site/search_product.php?product_id=% when sent in large numbers can create serve performance issues on the server.

A browser can send surprisingly large of GET requests to a remote website using COR from WebWorkers. During tests it was found that around 10,000 requests/minute can be sent from

a single browser. With even a very small botnet of just 600 zombies we would be sending around 100,000 requests/sec, depending on the nature of the page being requested this could be enough to bring a website down.

Email Spam:

Spam mails are largely sent using open-relay mail servers and botnet zombies. Though it would not be possible to a regular open-relay mail server from JavaScript still it would be possible to send such spam mails through the web equivalent of open-relay mails servers.

Many websites have feedback sections which ask the user to enter their name, email ID, subject and feedback. Once these are entered and the form is submitted, the server would craft this in the form of an email, with hard-coded from and to mail addresses and send it to the internal mail server.

Poorly designed websites would contain the from and to mail addresses in hidden form fields on the browser and by overwriting them to external addresses it should be possible to send mails with spoofed addresses if the company's mail server is also configured to operate in an open-relay mode.

Since only GET requests can be sent through COR, the feedback form should either be sending all data in QueryString or it should be be differentiating between QueryString and POST parameters. Alternatively if it is JSP page then HTTP Parameter Pollution can be used to submit forms over GET.

Distributed Password Cracking and Ravan:

Password cracking has always been a task assigned for programs written in native code with performance enhancement by writing some sections in Assembly. With its relatively slower execution rate JavaScript has never been considered for performing such resource-intensive tasks.

Things however have changed, JavaScript engines in modern browser are becoming increasingly fast and the concept of WebWorkers allows creation of dedicated background threads for the purpose of password cracking. During our tests it has been possible to observe password guessing rates of 100,000 MD5 hashes/second in JavaScript.

This figure is still slow compared to native code which can easily loop through a few million MD5 hashes/second on a machine with similar configuration. The JavaScript approach has been found to be on an average about 100-115 times slower than that of native code but it more than makes up for it in scalability. ~100 machines running the JavaScript password cracking program can match the cracking rate of one machine running a similar program written in native code.

As shown in the previous sections it would be very easy to build a botnet of a few thousand zombies executing our JavaScript password cracker in the background. Even with 1000 zombies our cracking rate would be equivalent to that of having 10 machines of similar configurations running a password cracked written in native code. An effective botnet creation effort could potentially get hundreds of thousands of such zombies to crack password hashes providing unimaginable computing capability.

Ravan:

Ravan, is a JavaScript distributed password cracker that uses HTML5 WebWorkers to perform password cracking in background JavaScript threads. The current implementation supports salted MD5 and SHA hashes.

At present it cracks password in pure brute force mode, there is another mode being worked on which would be an intelligent variation of brute force attack where combinations that have statistically been proven to be used more often by users are tried before the less probable combinations.

Though plain MD5 and SHA-1 hashes can be cracked at eye-blinking speeds using Rainbow tables, salted-hashes are resistant to Rainbow table based attacks and brute force is often the only option left.

Ravan is a web based tool that has three components:

Master: The browser that submits the hash

Worker: The browsers that performing the cracking

Web Interface: The central medium that proxies messages between the master and the workers

When a master submits a hash on the web interface for cracking, a unique hash ID is generated by Ravan along with an URL containing this ID.

This URL must be sent to all the workers that would be performing the actual cracking from their browsers.

The cracking process is broken down in to smaller chunks of work. Each chunk or slot refers to 20 million combinations.

As new workers join they are allotted certain slots of work and on completion and submission of results more slots are allotted to them. The process of co-coordinating the slots amongst the workers and monitoring the cracking process is done in JavaScript in the master's browser.

The web interface acts as proxy between the master and the workers by relaying messages both ways.

This system is meant to be used for legitimate reasons and requires the explicit permission of workers before beginning execution on their machine.

References:

1. HTML5 Security CheatSheet - http://code.google.com/p/html5security/
2. Shell of the Future - http://www.andlabs.org/tools.html#sotf
3. Next Generation Clickjacking - http://www.contextis.co.uk/resources/white-papers/clickjacking/Context-Clickjacking_white_paper.pdf
4. OWASP ClickJacking Guide - http://www.owasp.org/index.php/Clickjacking
5. Chrome and Safari users open to stealth HTML5 AppCache attack - http://blog.andlabs.org/2010/06/chrome-and-safari-users-open-to-stealth.html
6. Imposter - http://www.andlabs.org/tools.html#imposter
7. Hacking Facebook with HTML5 - http://m-austin.com/blog/?p=19