Understanding and Exploiting HTML5

image courtesy: https://www.w3.org/

v0.5

2017

Rishu Ranjan

TABLE OF CONTENTS

Introduction

- **HTML** stands for **Hypertext Markup Language** which is the standard markup language used for development of web pages and web applications.
- A markup language is a computer language that uses tags to define elements within a document. It is human-readable, meaning markup files contain standard words, rather than typical programming syntax.
- Web browsers receive HTML documents from a web server or from local storage and render them into multimedia web pages.
- **HTML5** is the fifth and current major version of Hyper Text Markup Language.

History

- From 1991 to 1999, HTML developed from version 1 to version 4. In year 2000, the World Wide Web Consortium (**W3C**) recommended XHTML 1.0. The XHTML syntax was strict, and the developers were forced to write valid and "well-formed" code.
- In 2004, W3C's decided to close down the development of HTML, in favor of XHTML and in the same year, Web Hypertext Application Technology Working Group(**WHATWG**) was formed. The WHATWG wanted to develop HTML, consistent with how the web was used, while being backward compatible with older versions of HTML.
- In 2004 - 2006, the WHATWG gained support by the major browser vendors.
- In 2006, W3C announced that they would support WHATWG and after 2 years, the first HTML5 public draft was released.
- In 2012, WHATWG and W3C decided on a separation: WHATWG wanted to develop HTML as a "Living Standard". A living standard is always updated and improved. New features can be added, but old functionality cannot be removed. In the same year, the WHATWG HTML5 Living Standard was published in 2012, and is continuously updated.
- W3C wanted to develop a definitive HTML5 and XHTML standard.
- The W3C HTML5 recommendation was released 28 October 2014.
- W3C also published an HTML 5.1 Candidate Recommendation on 21 June 2016.

What is New in HTML5?

- The DOCTYPE declaration for HTML5 is very simple:
  <!DOCTYPE html>
- The character encoding (charset) declaration is also very simple:
  <meta charset="UTF-8">
  Note: The default character encoding in HTML5 is UTF-8.

New HTML5 Elements

The most interesting new HTML5 elements are:

- New semantic elements like <header>, <footer>, <article>, and <section>.
- New attributes of form elements like **number**, **date**, **time**, **calendar**, **and range**.
- New graphic elements: <svg> and <canvas>.
- New multimedia elements: <audio> and <video>.

New HTML5 API's

The most interesting new API's in HTML5 are:

- HTML Geolocation
- HTML Drag and Drop
- HTML Local Storage
- HTML Application Cache
- HTML Web Workers
- HTML SSE

HTML5 Security Issues

- HTML5 is latest Version of the **Hypertext Markup Language** . But, it has multiple security issues due to the fact that HTML5 applications heavily utilize JavaScript libraries to facilitate dynamic interaction on the client side, a number of new attack surfaces, such as Cross-Site Scripting(**XSS**) and Cross-Site Request Forgery(**CSRF**) vulnerabilities, are opened.
- Most of the vulnerabilities in HTML5 are related to developers using features insecurely and storing sensitive information on the client side. If client side information is stolen, it is difficult to track back the attack since everything is happening on the client side.
- Along with that, by default, HTML5 offers very little or no protection at all against attacks such as SQL injection, XSS, and CSRF.
- We will go through the following list of features in HTML5 and understand what type of vulnerabilities can be introduced if not used properly:

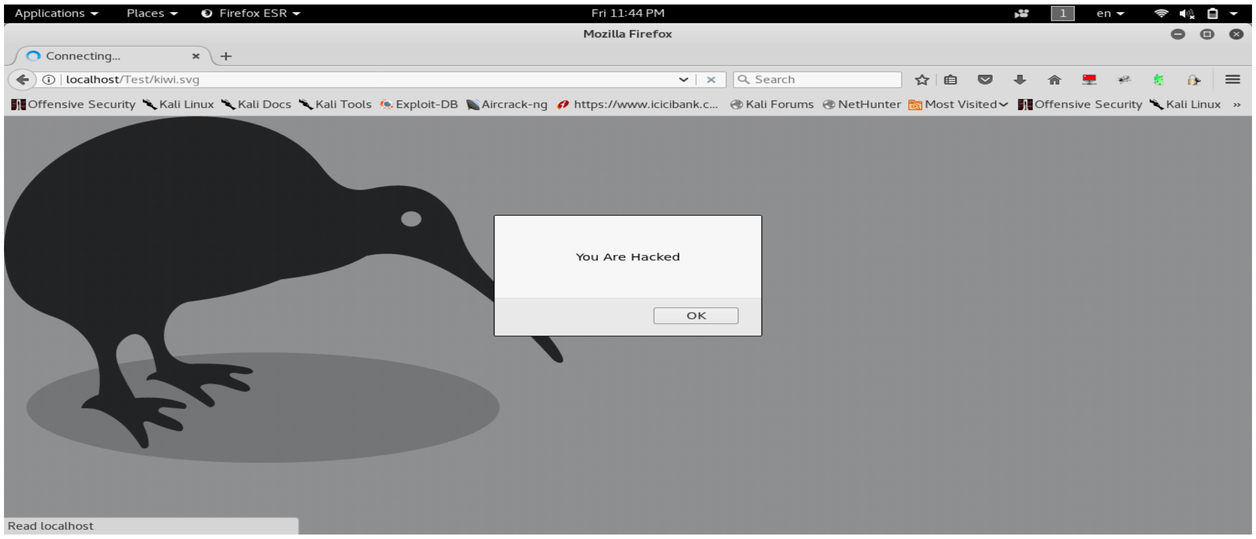
    - Scalable Vector Graphic(SVG)
    - Cross Origin Resource Sharing(CORS)
    - Sandboxed Iframes
    - WEB Storage

Scalable Vector Graphic(SVG)

- **Scalable Vector Graphics** is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation.
- HTML5 allows SVG graphics to be inserted in documents and they could be created inline within the document using the **<svg>** tag. If we are adding the SVG graphic to the document using the **<img>** tag then malicious JavaScript inside the svg file would not be executed unless the user saves the file/image and views it as a standalone in his browser.

Example

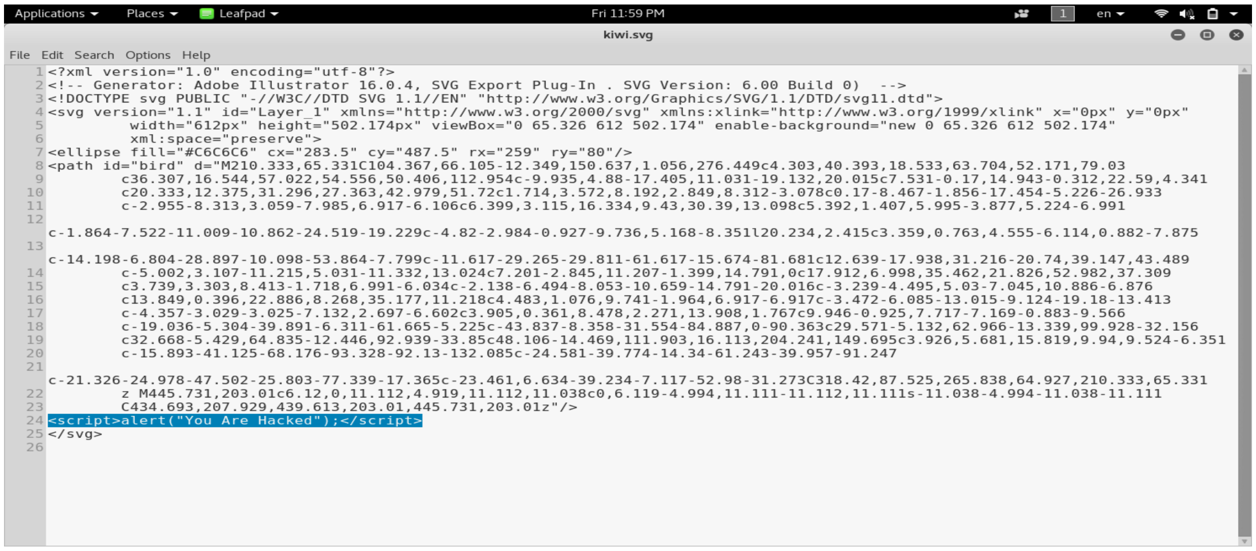
A Svg image is uploaded and accessed from the web browser



You Are Hacked

OK

The content of kiwi.svg file below:



```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 16.0.4, SVG Export Plug-In . SVG Version: 6.00 Build 0)  -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
	 width="612px" height="502.174px" viewBox="0 65.326 612 502.174" enable-background="new 0 65.326 612 502.174"
	 xml:space="preserve">
<ellipse fill="#C6C6C6" cx="283.5" cy="487.5" rx="259" ry="80"/>
<path id="bird" d="M210.333,65.331C104.367,66.105-12.349,150.637,1.056,276.449c4.303,40.393,18.533,63.704,52.171,79.03
	c36.307,16.544,57.022,54.556,50.406,112.954c-9.935,4.88-17.405,11.031-19.132,20.015c7.531-0.17,14.943-0.312,22.59,4.341
	c20.333,12.375,31.296,27.363,42.979,51.72c1.714,3.572,8.192,2.849,8.312-3.078c0.17-8.467-1.856-17.454-5.226-26.933
	c-2.955-8.313,3.059-7.985,6.917-6.106c6.399,3.115,16.334,9.43,30.39,13.098c5.392,1.407,5.995-3.877,5.224-6.991

c-1.864-7.522-11.009-10.862-24.519-19.229c-4.82-2.984-0.927-9.736,5.168-8.351l20.234,2.415c3.359,0.763,4.555-6.114,0.882-7.875

c-14.198-6.804-28.897-10.098-53.864-7.799c-11.617-29.265-29.811-61.617-15.674-81.681c12.639-17.938,31.216-20.74,39.147,43.489
	c-5.002,3.107-11.215,5.031-11.332,13.024c7.201-2.845,11.207-1.399,14.791,0c17.912,6.998,35.462,21.826,52.982,37.309
	c3.739,3.303,8.413-1.718,6.991-6.034c-2.138-6.494-8.053-10.659-14.791-20.016c-3.239-4.495,5.03-7.045,10.886-6.876
	c13.849,0.396,22.886,8.268,35.177,11.218c4.483,1.076,9.741-1.964,6.917-6.917c-3.472-6.085-13.015-9.124-19.18-13.413
	c-4.357-3.029-3.025-7.132,2.697-6.602c3.905,0.361,8.478,2.271,13.908,1.767c9.946-0.925,7.717-7.169-0.883-9.566
	c-19.036-5.304-39.891-6.311-61.665-5.225c-43.837-8.358-31.554-84.887,0-90.363c29.571-5.132,62.966-13.339,99.928-32.156
	c32.668-5.429,64.835-12.446,92.939-33.85c48.106-14.469,111.903,16.113,204.241,149.695c3.926,5.681,15.819,9.94,9.524-6.351
	c-15.893-41.125-68.176-93.328-92.13-132.085c-24.581-39.774-14.34-61.243-39.957-91.247

c-21.326-24.978-47.502-25.803-77.339-17.365c-23.461,6.634-39.234-7.117-52.98-31.273C318.42,87.525,265.838,64.927,210.333,65.331
	z M445.731,203.01c6.12,0,11.112,4.919,11.112,11.038c0,6.119-4.994,11.111-11.112,11.111s-11.038-4.994-11.038-11.111
	C434.693,207.929,439.613,203.01,445.731,203.01z"/>
<script>alert("You Are Hacked");</script>
</svg>
```

Security Issue

- If you uploads any file that can use for XSS (HTML,SWF,etc) the content type will change to "text/plain; charset=us-ascii" . But for images it will stay the same . so if you upload SVG with JS content it will work fine !
- The "Content-Type: image/svg+xml; charset=us-ascii" header will make this attack works. Just upload the svg file to the site .

Remediation

The way browsers handle SVG files is terrible. If you're serving SVG files that your users can upload, **only allow them to be served as text/plain**.

Cross Origin Resource Sharing(CORS)

- **Cross-origin resource sharing** is a mechanism that allows restricted resources (e.g. fonts) on a web page to be requested from another domain outside the domain from which the first resource was served. A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy.
- Due to the limitation that two pages on different origins could not communicate even if they had a mutual trust relationship with each other, a new standard called CORS was introduced with HTML5, which allowed cross domain http requests, and more importantly, cross domain AJAX requests.

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

```
Origin: http://testcorsa.com
```

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

```
Access-Control-Allow-Origin: http://www.testcorsa.com
```

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

```
Access-Control-Allow-Origin: *
```

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

Remediation

- Validate URLs passed to XMLHttpRequest.open. Current browsers allow these URLs to be cross domain; this behavior can lead to code injection by a remote attacker. Pay extra attention to absolute URLs.
- Ensure that URLs responding with Access-Control-Allow-Origin: * do not include any sensitive content or information that might aid attacker in further attacks. Use the Access-Control-Allow-Origin header only on chosen URLs that need to be accessed cross-domain. Don't use the header for the whole domain.
    - Allow only selected, trusted domains in the Access-Control-Allow-Origin header. Prefer whitelisting domains over blacklisting or allowing any domain (do not use * wildcard nor blindly return the Origin header content without any checks).
- Discard requests received over plain HTTP with HTTPS origins to prevent mixed content bugs.
- Don't rely only on the Origin header for Access Control checks. Browser always sends this header in CORS requests, but may be spoofed outside the browser. Application-level protocols should be used to protect sensitive data.

Sandboxed Iframes

- The **HTML <iframe> element** represents a nested browsing context, effectively embedding another HTML page into the current page.
- By default, a normal iframe loads all the contents from the destination, including HTML, CSS, and JavaScripts.
- With the advent of sandboxed iframes, we are allowed to specify the content that should be loaded to the iframe. When the sandbox attribute is specified in an iframe tag, there is an extra set of restrictions applied to it.

Example
The following is an example of an iframe with a sandbox attribute.

```
<iframe sandbox src="http://test-iframes.com/"></ iframe>
```

By embedding a website with iframe using the sandbox attribute, it will prevent the JavaScript from being executed; therefore, it is considered a security feature.
If you would like to allow scripts from an iframe to be executed, you would need to set the value of the sandbox attribute to allow-scripts:

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

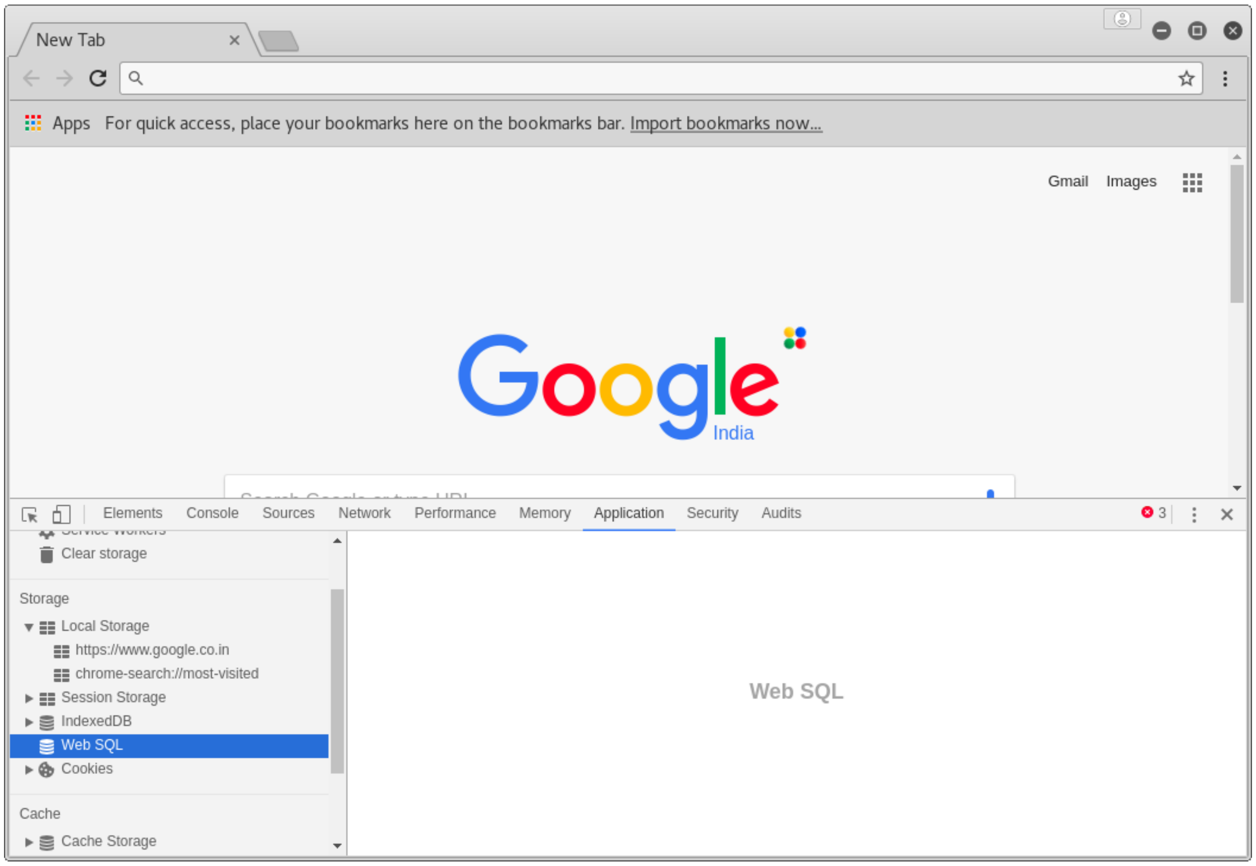
Security Concerns
Though the sandboxed iframe was implemented as a security feature for JavaScript, there are a few security concerns attached to it. One of the major concerns is the use of a frame busting code such as:

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

The code above would prevent the website from being loaded into an iframe; it is a very commonly known practice across websites. However, the problem is that with a sandboxed iframe, we can prevent the JavaScripts from being executed, and the above code would not work.

Remediation
Therefore, it is recommended to use X-frame-options and to set it to either same origin or deny

WEB Storage


New Tab
Apps For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...
Gmail Images
Google India
Elements Console Sources Network Performance Memory Application Security Audits
Clear storage
Storage
Local Storage
https://www.google.co.in
chrome-search://most-visited
Session Storage
IndexedDB
Web SQL
Cookies
Cache
Cache Storage
Web SQL

WebSQL has the sole purpose of creating offline applications to allow a larger amount of storage upon the client side. We have three core methods with WebSQL:

1. openDatabase - Used for accessing existing database or accessing new ones.
2. transaction –Allows you to control transactions. Transaction is used to perform SQL operations; it contains a set of operations that are considered to be a single operation separate from other transactions.
3. executeSQL -Used to execute a SQL query.

Security Concerns

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

SQL Injection

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

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

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

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

Secure Statement

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

In this case, we are utilizing a parameterized query, so the user supplied input would not be executed by an SQL query. With SQL injection on the client side, we are limited to INSERTING, UPDATING, and DELETING the data from/to the database. It’s not possible for an attacker to retrieve the data from the database by exploiting a client side SQL injection vulnerability. The impact of a client side SQL Injection depends upon the nature of the data being stored inside the database.

Note
On November 2010, the W3C announced Web SQL Database (relational SQL database) as a deprecated specification. A new standard Indexed Database API or IndexedDB (formerly WebSimpleDB) is actively developed, which provides key/value database storage and methods for performing advanced queries.

References

Introduction- https://en.wikipedia.org/wiki/HTML5

History and What is New in HTML5?- https://www.w3schools.com/html/html5_intro.asp

HTML5 Security Issues-
1. https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet
2. https://media.blackhat.com/bh-eu-12/shah/bh-eu-12-Shah_HTML5_Top_10-WP.pdf
3. http://resources.infosecinstitute.com/general-html5-security/#gref
4. https://www.rafaybaloch.com/2017/06/html5-modern-day-attack-and-defence.html
5. https://www.owasp.org/images/0/03/Mario_Heiderich_OWASP_Sweden_The_image_that_called_me.pdf
6. https://en.wikipedia.org/wiki/Cross-origin_resource_sharing