Secure Development

HTLM5

אישורים וחתימות

	שם	תפקיד	חתימה
ערך:	שלומי גגולשוילי	מנהל תחום בדיקות ואפליקציה	
אישר:	אברהם זרוק	מנהל מערך אבטחת מידע ממשל זמין	

מעקב שינויים במסמך

מהדורה	תאריך עדכון	עמוד/סעיף עדכון	פרטי העדכון
1.0	19/12/2013	הכל	

All rights reserved, Mimshal zamin, State of Israel

משרד האוצר - התקשוב הממשלתי

ממשל זמין

December 2013

1th Netanel Lorch St. Jerusalem Tel: 02-6664666 Fax: 02-6664650
All rights reserved, Mimshal zamin, State of Israel

Table of contents

All rights reserved, Mimshal zamin, State of Israel

1 General

HTML5 will be the new standard for HTML.
HTML5 is designed to deliver almost everything you want to do online without requiring additional plugins. It does everything from animation to apps, music to movies, and can also be used to build complicated applications that run in your browser.
HTML5 is also cross-platform (it does not care whether you are using a tablet or a smartphone, a netbook, notebook or a Smart TV).
HTML5 can also be used to write web applications that still work when you are not online.
Some of the most interesting new features in HTML5:

- The <canvas> element for 2D drawing
- The <video> and <audio> elements for media playback
- Support for local storage
- New content-specific elements, like <article>, <footer>, <header>, <nav>, <section>
- New form controls, like calendar, date, time, email, url, search

Browser Support for HTML5
HTML5 is not yet an official standard, and no browsers have full HTML5 support.
But all major browsers (Safari, Chrome, Firefox, Opera, and Internet Explorer) continue to add new HTML5 features to their latest versions.

All rights reserved, Mimshal zamin, State of Israel

2 HTML5 Security

The following cheat sheet serves as a guide for implementing HTML 5 in a secure fashion.

Communication APIs

Web Messaging

Web Messaging, also known as Cross Domain Messaging provides a means of messaging between documents from different origins in a way which is generally safer than the multiple hacks used in the past to accomplish this task, however, there are still some recommendations to keep in mind:

- When posting a message, explicitly state the expected origin as the second argument to postMessage rather than * in order to prevent sending the message to an unknown origin after a redirect or some other means of the target window's origin changing.
- The receiving page should always:
  - Check the origin attribute of the sender to verify the data is originating from the expected location.
  - Perform input validation on the data attribute of the event to ensure it's in the desired format.
- Don't assume you have control over data attribute. Single Cross Site Scripting flaw in sending page allows attacker to send messages of any given format.
- Both pages should only interpret the exchanged messages as data. Never evaluate passed messages as code (e.g. via eval()) or insert it to a page DOM (e.g. via innerHTML) as that would create a DOM based XSS vulnerability.
- To assign the data value to an element, instead of using an insecure method like element.innerHTML = data; use the safer option element.textContent = data;
- Check the origin properly exactly to match the FQDN(s) you expect. Note that the following code:

if(message.orgin.indexOf(".owasp.org")!=-1) { / ... */ }*

is very insecure and will not have the desired behavior as www.owasp.org.attacker.com will match.

- If you need to embed external content/untrusted gadgets and allow user-controlled scripts which is highly discouraged, consider use a JavaScript rewriting framework such as Google Caja or check the information on sandboxed frames.
- The receiving IFrame should not accept messages from any domain
- The received message needs to be validated on the client to avoid malicious content being executed.

Cross Origin Resource Sharing

- Validate URLs passed to XMLHttpRequest.open, current browsers allow these URLs to be cross domain and this behavior can lead to code injection by a remote attacker. Pay extra attention to absolute URLs.

All rights reserved, Mimshal zamin, State of Israel

- 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 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 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)
- Keep in mind that CORS does not prevent the requested data from going to an un-authenticated location - it's still important for the server to perform usual CSRF prevention.
- While the RFC recommends a pre-flight request with the OPTIONS verb, current implementations might not perform this request, so it's important that "ordinary" (GET and POST) requests perform any access control necessary.
- 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.
- To mitigate DDoS attacks the Web Application Firewall (WAF) needs to block CORS (Cross-Origin Resource Sharing) requests if they arrive in a high frequency.

WebSockets

- Drop backward compatibility in implemented client/servers and use only protocol versions above hybi-00. Popular Hixie-76 version (hiby-00) and olders are outdated and insecure.
- Recommended version supported in latest versions of all current browsers is RFC 6455 (Supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50 and IE10)
- While it is relatively easy to tunnel TCP services through WebSockets (e.g. VNC, FTP), doing so enables access to these tunneled services for the in-browser attacker in case of a Cross-Site-Scripting attack. These services might also be called directly from a malicious page or program.
- The protocol doesn't handle authorization and/or authentication. Application-level protocols should handle that separately in case sensitive data is being transferred.
- Process the messages received by the websocket as data. Don't try to assign it directly to the DOM nor evaluate as code. If the response is JSON never use the insecure eval() function, use the safe option JSON.parse() instead.
- Endpoints exposed through ws:// protocol are easily reversible to plaintext. Only wss:// (WebSockets over SSL/TLS) should be used for protection against Man-In-The-Middle attacks

All rights reserved, Mimshal zamin, State of Israel

- Spoofing the client is possible outside browser, so WebSockets server should be able to handle incorrect/malicious input. Always validate input coming from the remote site, as it might have been altered.
- When implementing servers, check the Origin: header in Websockets handshake. Though it might be spoofed outside browser, browsers always add the Origin of the page which initiated Websockets connection.
- As WebSockets client in browser is accessible through Javascript calls, all Websockets communication can be spoofed or hijacked through Cross-Site-Scripting. Always validate data coming through WebSockets connection.

Server-Sent Events

- Validate URLs passed to the EventSource constructor, even though only same-origin URLs are allowed.
- As mentioned before, process the messages (event.data) as data and never evaluate the content as HTML or script code.
- Check always the origin attribute of the message (event.origin) to ensure the message is coming from a trusted domain, use a whitelist approach.

Storage APIs

1. Local Storage
   - Also known as Offline Storage, Web Storage. Underlying storage mechanism may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended not to store any sensitive information in local storage.
   - Use the object sessionStorage instead of localStorage if persistent storage is not needed(cookies). sessionStorage object is available only to that window/tab until the window is closed.
   - A single Cross Site Scripting can be used to steal all the data in these objects, so again it's recommended not to store sensitive information in local storage.
   - A single Cross Site Scripting can be used to load malicious data into these objects too, so don't consider objects in these to be trusted.
   - Pay extra attention to “localStorage.getItem” and “setItem” calls implemented in HTML5 page. It helps in detecting when developers build solutions that put sensitive information in local storage, which is a bad practice.
   - Do not store session identifiers in local storage as the data is always accesible by JavaScript. Cookies can mitigate this risk using the httpOnly flag.
   - There is no way to restrict the visibility of an object to a specific path like with the attribute path of HTTP Cookies, every object is shared within an origin and protected with the Same Origin Policy. Avoid host multiple applications on the same origin, all of them would share the same localStorage object, use different subdomains instead.
2. Client-side databases

All rights reserved, Mimshal zamin, State of Israel

- Underlying storage mechanism may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended not to store any sensitive information in local storage.
- If utilized, WebDatabase content on client side can be vulnerable to SQLInjection and needs to have proper validation and parametrization.
- Like Local Storage, a single Cross Site Scripting can be used to load malicious data into a web database too, so don't consider data in these to be trusted either.

Geolocation

- Require user input before calling getCurrentPosition or watchPosition.

Web Workers

- Web Workers are allowed to use XMLHttpRequest object to perform in-domain and Cross Origin Resource Sharing requests. See relevant section of this Cheat Sheet to ensure CORS security.
- While Web Workers don't have access to DOM of the calling page, malicious Web Workers can use excessive CPU for computation, leading to Denial of Service condition or abuse Cross Origin Resource Sharing for further exploitation. Ensure code in all Web Workers scripts is not malevolent. Don't allow creating Web Worker scripts from user supplied input.
- Validate messages exchanged with a Web Worker. Do not try to exchange snippets of Javascript for evaluation e.g. via eval() as that could introduce a DOM Based XSS vulnerability.

Sandboxed frames

- Use the sandbox attribute of an iframe for untrusted content
- The sandbox attribute of an iframe enables restrictions on content within a iframe. The following restrictions are active when the sandbox attribute is set:
    - All markup is treated as being from a unique origin
    - All forms and scripts are disabled
    - All links are prevented from targeting other browsing contexts
    - All features that triggers automatically are blocked
    - All plugins are disabled
- In old versions of user agents where this feature is not supported this attributed will be ignored. Use this feature as an additional layer of protection or check if the browser supports sandboxed frames and only show the untrusted content if supported.
- Apart from this attribute, to prevent Clickjacking attacks and unsolicited framing it is encouraged to use the header X-Frame-Options which supports the deny and same-origin values. Other solutions like framebusting if(window!== window.top) { window.top.location = location; } are not recommended.

Offline Applications

All rights reserved, Mimshal zamin, State of Israel

- Whether the user agent requests permission to the user to store data for offline browsing and when this cache is deleted vary from one browser to the next. Cache poisoning is an issue if a user connects through insecure networks, so for privacy reasons it is encouraged to require user input before sending any manifest file.
- Users should only cache trusted websites and clean the cache after browsing through open or insecure networks.

HTTP Headers to enhance security

1. X-Frame-Options
   - This header can be used to prevent ClickJacking in modern browsers (IE6/IE7 don't support this header(
   - Use the same-origin attribute to allow being framed from urls of the same origin or deny to block all. Example: X-Frame-Options: DENY
2. X-XSS-Protection
   - Enable XSS filter (only works for Reflected XSS(
   - Example: X-XSS-Protection: 1; mode=block
3. Strict Transport Security
   - Force every browser request to be sent over TLS/SSL (this can prevent SSL strip attacks(
   - Use includeSubDomains
   - Example: Strict-Transport-Security: max-age=8640000; includeSubDomains
4. Content Security Policy
   - Policy to define a set of content restrictions for web resources which aims to mitigate web application vulnerabilities such as Cross Site Scripting
   - Example: X-Content-Security-Policy: allow 'self'; img-src *; object-src media.example.com; script-src js.example.com
5. Origin
   - Sent by CORS/WebSockets requests
   - There is a proposal to use this header to mitigate CSRF attacks, but is not yet implemented by vendors for this purpose.

General guidelines

- Don't use obfuscation. The cybercriminals use it to hide their malicious code if it's obfuscated, it's suspect!
- Check for malicious content for XSS protection.
- Use a whitelist approach - Don’t use blacklist.
- Encode any input/output
- Don't use user-controllable parameters
- Use prepared statements, don't compose SQL query from strings
- Data stored via HTTPS -> Access only via HTTPS
- Validate responses
- Only allow the strictly necessary features

All rights reserved, Mimshal zamin, State of Israel

- Don’t trust anything

All rights reserved, Mimshal zamin, State of Israel