Home
About OWASP
Acknowledgements
Advertising
AppSec Events
Books
Brand Resources
Chapters
Donate to OWASP
Downloads
Funding
Governance
Initiatives
Mailing Lists
Membership
Merchandise
News
Community portal
Presentations
Press
Projects
Video
Volunteer

Reference
Activities
Attacks
Code Snippets
Controls
Glossary
How To...
Java Project
.NET Project
Principles
Technologies
Threat Agents
Vulnerabilities

Language

Tools

HTML5 Security Cheat Sheet

Last revision (mm/dd/yy): **04/7/2014**

Introduction

[hide]
1 Introduction
2 Communication APIs
  2.1 Web Messaging
  2.2 Cross Origin Resource Sharing
  2.3 WebSockets
  2.4 Server-Sent Events
3 Storage APIs
  3.1 Local Storage
  3.2 Client-side databases
4 Geolocation
5 Web Workers
6 Sandboxed frames
7 Offline Applications
8 Progressive Enhancements and Graceful Degradation Risks
9 HTTP Headers to enhance security
  9.1 X-Frame-Options
  9.2 X-XSS-Protection
  9.3 Strict Transport Security
  9.4 Content Security Policy
  9.5 Origin
10 Authors and Primary Editors
  10.1 Other Cheatsheets

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 that 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 that it's in the desired format.
- Don't assume you have control over the `data` attribute. A single Cross Site Scripting flaw in the sending page allows an 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. For more information see DOM based XSS Prevention Cheat Sheet.
- To assign the data value to an element, instead of using a 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 using a JavaScript rewriting framework such as Google Caja or check the information on sandboxed frames.

Cross Origin Resource Sharing

- 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).
- Keep in mind that CORS does not prevent the requested data from going to an unauthenticated 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.

WebSockets

- Drop backward compatibility in implemented client/servers and use only protocol versions above hybi-00. Popular Hixie-76 version (hiby-00) and older are outdated and insecure.
- The 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's 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 the `ws://` protocol are easily reversible to plain text. Only `wss://` (WebSockets over SSL/TLS) should be used for protection against Man-In-The-Middle attacks.
- Spoofing the client is possible outside a browser, so the 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 the Websockets handshake. Though it might be spoofed outside a browser, browsers always add the Origin of the page that initiated the Websockets connection.
- As a WebSockets client in a browser is accessible through JavaScript calls, all Websockets communication can be spoofed or hijacked through Cross Site Scripting. Always validate data coming through a 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.
- Always check the origin attribute of the message (`event.origin`) to ensure the message is coming from a trusted domain. Use a whitelist approach.

Storage APIs

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. 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.

Client-side databases

- 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.
- Underlying storage mechanisms 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 the client side can be vulnerable to SQL injection and needs to have proper validation and parameterization.
- Like Local Storage, a single Cross Site Scripting can be used to load malicious data into a web database as well. Don't consider data in these to be trusted.

Geolocation

- The Geolocation RFC recommends that the user agent ask the user's permission before calculating location. Whether or how this decision is remembered varies from browser to browser. Some user agents require the user to visit the page again in order to turn off the ability to get the user's location without asking, so for privacy reasons, it's recommended to 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:
  1. All markup is treated as being from a unique origin.
  2. All forms and scripts are disabled.
  3. All links are prevented from targeting other browsing contexts.
  4. All features that triggers automatically are blocked.
  5. All plugins are disabled.

It is possible to have a fine-grained control over `iframe` capabilities using the value of the `sandbox` attribute.

- In old versions of user agents where this feature is not supported, this attribute 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

- Whether the user agent requests permission to the user to store data for offline browsing and when this cache is deleted varies 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.

Progressive Enhancements and Graceful Degradation Risks

- The best practice now is to determine the capabilities that a browser supports and augment with some type of substitute for capabilities that are not directly supported. This may mean an onion-like element, e.g. falling through to a Flash Player if the <video> tag is unsupported, or it may mean additional scripting code from various sources that should be code reviewed.

HTTP Headers to enhance security

X-Frame-Options

- This header can be used to prevent ClickJacking in modern browsers.
- 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`
- For more information on Clickjacking Defense please see the Clickjacking Defense Cheat Sheet.

X-XSS-Protection

- Enable XSS filter (only works for Reflected XSS).
- Example: `X-XSS-Protection: 1; mode=block`

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

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

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.

Authors and Primary Editors

First	Last	Email
Mark	Roxberry	mark.roxberry [at] owasp.org
Krzysztof	Kotowicz	krzysztof [at] kotowicz.net
Will	Stranathan	will [at] cltnc.us
Shreeraj	Shah	shreeraj.shah [at] blueinfy.net
Juan Galiana	Lara	jgaliana [at] owasp.org

Other Cheatsheets

Developer Cheat Sheets (Builder)

- Authentication Cheat Sheet
- Choosing and Using Security Questions Cheat Sheet
- Clickjacking Defense Cheat Sheet
- C-Based Toolchain Hardening Cheat Sheet
- Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet
- Cryptographic Storage Cheat Sheet
- DOM based XSS Prevention Cheat Sheet
- Forgot Password Cheat Sheet
- **HTML5 Security Cheat Sheet**
- Input Validation Cheat Sheet
- JAAS Cheat Sheet
- Logging Cheat Sheet
- .NET Security Cheat Sheet
- Password Storage Cheat Sheet
- Pinning Cheat Sheet
- Query Parameterization Cheat Sheet
- Ruby on Rails Cheatsheet
- REST Security Cheat Sheet
- Session Management Cheat Sheet
- SQL Injection Prevention Cheat Sheet
- Transport Layer Protection Cheat Sheet
- Unvalidated Redirects and Forwards Cheat Sheet
- User Privacy Protection Cheat Sheet
- Web Service Security Cheat Sheet
- XSS (Cross Site Scripting) Prevention Cheat Sheet

Assessment Cheat Sheets (Breaker)

- Attack Surface Analysis Cheat Sheet
- XSS Filter Evasion Cheat Sheet
- REST Assessment Cheat Sheet

Mobile Cheat Sheets

- IOS Developer Cheat Sheet
- Mobile Jailbreaking Cheat Sheet

OpSec Cheat Sheets (Defender)

- Virtual Patching Cheat Sheet

Draft Cheat Sheets

- OWASP Top Ten Cheat Sheet
- Access Control Cheat Sheet
- Application Security Architecture Cheat Sheet
- Business Logic Security Cheat Sheet
- PHP Security Cheat Sheet
- Secure Coding Cheat Sheet
- Secure SDLC Cheat Sheet
- Threat Modeling Cheat Sheet
- Web Application Security Testing Cheat Sheet
- Grails Secure Code Review Cheat Sheet
- IOS Application Security Testing Cheat Sheet
- Key Management Cheat Sheet
- Insecure Direct Object Reference Prevention Cheat Sheet
- Content Security Policy Cheat Sheet

Category: Cheatsheets