

# HTML5 SECURITY ASPECTS AND SECURING YOUR WEB APPLICATION

Pranshu Sharma , Satish Anand Arjunan

## I. 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 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.origin.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 (hybi-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.

## II. 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 accessible 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.

## III. 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`.

## IV. 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.

#### V. SANDBOX 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.

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.

#### VI. 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.

#### VII. PROGRESSIVE ENHANCEMENT AND GRACEFUL DEGRADATION RISK

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.

#### VIII. SECURING WEB COMMUNICATION: CERTIFICATES, SSL AND HTTPS://

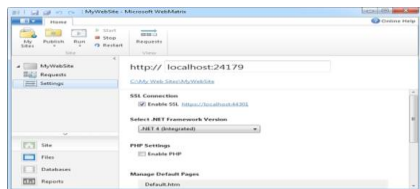
If your website lets users enter sensitive information in the browser and send it to the server, you really should secure that communication. Sensitive information might be a password, a social security number, or a credit card number - basically, anything that no one should know except the user and your website. To establish secure communications between the user and your website, you need an SSL certificate. A certificate does two things:

- It verifies that your site belongs to you. This assures users that they're sending sensitive information to you and not to a malicious user whose site *looks* like yours.
- It encrypts the communications between the user's browser and the web server where your site is hosted. (This is the "SSL" part of "SSL certificate," which stands for "secure sockets layer" and which is the security protocol used to encrypt the communication.) Encryption prevents people from eavesdropping and reading the sensitive information.

As a user, you've probably used an SSL certificate many times. Any time you access a website using the protocol `https://` instead of `http://`, you're using SSL, such as when you buy something online or do your banking online. If you've noticed this, you've also noticed that you as a user don't have to do anything to use SSL and `https://` -- the verification and encryption all happen automatically.

When you create a website, though, it's up to you to get a certificate and to make sure your site is

configured to use it. Certificates are sold by various companies and are valid for a specified time. (Search for "SSL certificate" on the web.) Hosting providers often offer SSL certificates to go along with their hosting plans. To get a certificate, in addition to paying the fee, you must provide information about your company or yourself that helps the certificate authority (CA) validate who you are and then code this information into the certificate. When the process is finished, the CA sends you some files that constitute the certificate. After you've acquired a certificate, it has to be installed on the server where your website is hosted. If you're using a hosting provider, you don't have direct access to the server and therefore can't install the certificate yourself. Instead, the hosting company typically provides a configuration utility or dashboard that lets you send them the certificate (if they didn't issue it themselves) and install it on their servers. If you are managing your own server,



you need to install the certificate yourself. (If you're using IIS 7, see [How to Set Up SSL on IIS 7](#) on the IIS.net website.)

## IX. TESTING SSL IN WEBMATRIX

When you're creating a website in WebMatrix, you want to be able to test that sensitive pages are working properly using SSL. To help with this, WebMatrix includes a *self-signed certificate* that's created by IIS Express. A self-signed certificate performs the encryption that certificates do, but it has not been verified by a CA. It's therefore useful for testing to make sure that encryption is working. However, it's not suitable for use with a live site, because users would see an error message that warns them that the certificate is not authenticated. To use the self-signed certificate in WebMatrix, in

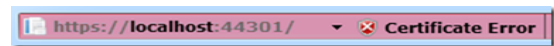


the **Site** navigation pane, click **Settings**. Under **SSL Connection**, select **Enable SSL**.

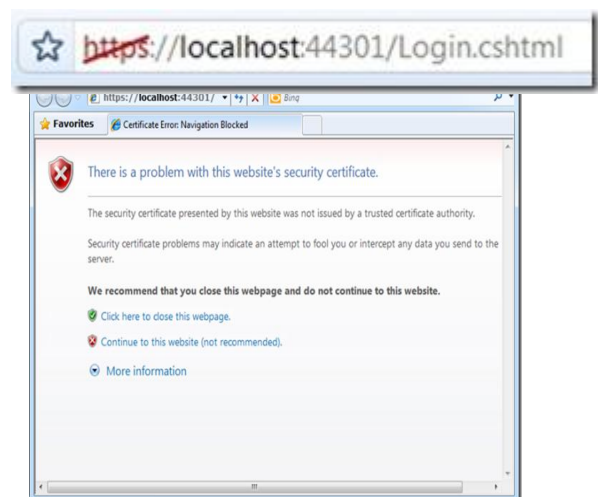
Although SSL is enabled, when you run a page from WebMatrix using IIS Express, the page will

automatically use the *http://* protocol. To verify that the *https://* protocol is using the self-signed certificate, click **Run** and then choose a browser option with HTTPS.

Because you're testing using a self-signed certificate, the browser will display an error. For example, here's what Internet Explorer displays:



Because you're just testing in WebMatrix, it's safe



to go ahead and click **Continue to this website** (or whatever similar option you see in other browsers). The browser will often indicate that you're using a non-trusted certificate. In Internet Explorer, the address bar is shown with a red background and "Certificate Error" is displayed:

In Google Chrome, the address bar shows the protocol (*https://*) with a line through it:

However, when you later deploy your website to the server where your real certificate is installed, you won't see this error any more.

## REFERENCES

- [1] [www.owasp.org](http://www.owasp.org)
- [2] [www.microsoft.com](http://www.microsoft.com)