Chapter 22. Secure Application Configuration

One component of successfully delivering a secure web application to your customers is to ensure the web application you are delivering is configured in a way that makes use of built-in browser security mechanisms.

Web applications today are built on a multitude of languages, frameworks, and technologies. However, because the sole method of delivery for a web application is still the browser, learning how to make use of the browser's built-in security mechanisms is essential to good security posture.

In this chapter, we will evaluate and discuss several security technologies implemented by the web browser. You will also learn how to configure them correctly to maximize the security of your web application.

Content Security Policy

Content Security Policy (CSP) is one of the browser's primary security mechanisms for protecting against the most common forms of cyberattacks involving a browser client. If implemented correctly, it is capable of preventing XSS, data injection, phishing, framing, and redirect attacks.

In order to provide a clean developer experience without breaking the internet, CSP was designed to be implemented with a significant amount of configuration options. Because of this, a strong CSP policy differs drastically from a weak CSP policy. It is, in fact, possible to run a fully functioning website without any CSP policy whatsoever, leading to the browser implementing no mitigations against common attacks.

Let's take a deeper look into CSP policies from an implementation perspective so you can learn how to properly configure a CSP policy on your web application in order to allow the browser to implement security mechanisms on behalf of your users.

Implementing CSP

CSP can be implemented on a web application via one of two methods. The most common method is to have your server return a `Content-Security-Policy` header with every request. Note that the `X-Content-Security-Policy` and `X-Webkit-CSP` headers are deprecated and should no longer be used to implement a CSP policy.

Alternatively, you can implement a CSP policy by including a `meta` tag in the `<HEAD></HEAD>` of every HTML page. Such a meta tag would look as follows:

```
<meta
  http-equiv="Content-Security-Policy"
  content="default-src 'self'; img-src data:" />
```

CSP Structure

Regardless of the implementation you choose, the method of configuring your CSP policy is roughly the same. A CSP policy is composed of *directives*, which are separated by semicolons (;). After each directive you may include a configuration option corresponding with that directive, which will then be implemented by the browser.

An example directive would be `script-src scripts.mega-bank.com`, which would permit the website to only execute JavaScript scripts that are sourced from the provided origin `scripts.mega-bank.com`. All other scripts would throw a CSP error in the browser console.

Important Directives

The list of directives supported by CSP varies slightly by browser and expands periodically as browsers update and adhere to the latest version of the CSP specifications. Currently, the CSP specification is maintained by the World Wide Web Consortium (W3C) nonprofit organization, which maintains a variety of web standards.

Here are some of the most important directives, from a security perspective:

`default-src`

This is a fallback for other directives and allow-lists sources from which images, scripts, CSS stylesheets, and other resources can be loaded. Defining this directive prevents your website from being able to execute unintended scripts, thus reducing XSS risk significantly.

`sandbox`

When configured, this directive creates a sandbox on page-load that prevents resources from being able to create pop-ups, execute scripts, or interact with browser plug-ins.

`frame-ancestors`

This directive defines which web pages may embed the current web page. Setting this to `'none'` is often the most secure option, as it prevents other websites from clickjacking by placing user interface elements in front of the current web page and tricking users into clicking on them.

`eval` *and* `inline script`

These functionalities are blocked by default simply by having any CSP policy implemented. This is a huge win because both of these script execution methods are popular attack vectors for XSS attacks. These, however, may be disabled with `'unsafe-inline'` and `'unsafe-eval'`.

`report-uri`

This directive allows you to define an endpoint to which CSP errors are reported for logging.

CSP Sources and Source Lists

Directives in a CSP policy that end with `-src` take an input known as a *source list*—a whitespace-separated list of origins and CSP-specific configuration values. Source lists are used to tell the directive how to operate once loaded into the browser. Several directives are unique to CSP, and it's important to understand how they function:

`*`

This wildcard operator allows any URL *except* `blob` and `file`. For example, `image-src *` allows images from any web-based origin but not from the local filesystem.

`'none'`

This prevents any source from loading. Note that CSP policy including `none` after a directive must have no other sources or the CSP policy will fail to load. For example, `image-src 'none'` is valid, but `image-src 'none' images.mega-bank.com` is not.

`data:`

This allows the loading of base64-encoded images. For example, `img-src data:`.

`https:`

This allows loads provided the resources in the source list all implement HTTPS. For example, `img-src https: mega-bank.com`.

`'self'`

This refers to the current origin of the loaded page. If a web page loads to *test.mega-bank.com/123*, any images available from *test.mega-bank.com/** will be capable of loading provided the CSP policy is set to `img-src 'self'`.

Strict CSP

Sometimes you are building a more complex web application and need the capacity to load inline script. However, you want to avoid the pitfalls of enabling inline script to all scripts because it is one of the most common XSS attack vectors.

With CSP, it is possible to enable inline scripts securely with a little bit of effort. This form of CSP configuration is often called *strict CSP* because it provides additional security rules for scripts that run in the browser to prevent against XSS, but it does not limit functionality.

There are two methods of implementing a strict CSP policy. The first method is a *hash-based* strict CSP, and the second is a *nonce-based* strict CSP. Both of these methods require that common XSS sinks implement either a randomized nonce or a SHA-256 hash that will be verified prior to each script execution.

A simple nonce-based strict CSP implementation looks as follows:

```
Content-Security-Policy:
  script-src 'nonce-{RANDOM}' 'strict-dynamic';
```

When the CSP policy is set to `'strict-dynamic'` with `'nonce-{RANDOM}'` the browser will enforce that inline scripts provide a new attribute, `nonce`. In order to adhere to this, inline scripts will look as follows:

```
<script src="..." nonce="123">alert()</script>
```

The `nonce` value is created at runtime, and each of your scripts is loaded in with the correct `nonce` value. Prior to script execution, the browser checks to ensure the nonce in the script attribute matches the predefined value. If true, the script execution continues (dynamic scripts loaded from the top-level script with the correct nonce may also load). If false, script execution fails and a CSP error is thrown in the console.

The hash-based approach operates similarly, but instead of using randomized nonces, it makes use of SHA-256 hashes. In the hash case, a hash

of every single inline script is added to the CSP directive `script-src` source list. When an inline script attempts to execute, it is hashed and compared against the source list. If it fails to meet that check, an error is thrown in the console and the script fails to execute.

Nonce-based strict CSP is ideal for scenarios where every web page is rendered on the server, allowing the new nonces to be created on every page load.

Hash-based CSP is better for applications that need to be cached (e.g., make use of a CDN) because the collision rate for SHA-256 hashes is so low that the probability of two scripts colliding (a malicious script creating the same hash as a nonmalicious script) is somewhere in the ballpark of 1/43,000,000,000. Because CDNs and caches often last quite a while, a hacker could craft an inline script payload matching the current nonce. However, it's extremely unlikely they could craft an inline script payload that hashes identically to a nonmalicious script in the same page.

Example Secure CSP Policy

The following is a secure-by-default CSP policy that can be used as a starter policy prior to further customizations. It provides an example implementation of the nonce strategy for strict CSP on script sources, blocks frame-ancestors to prevent clickjacking attacks, enforces HTTPS on images while allowing base64 image loads, presents a reporting URI for CSP errors, and provides a default `'self'` as a fallback for source lists. The code to implement this CSP is as follows:

```
Content-Security-Policy
default-src: 'self';
script-src: 'self' 'nonce-jgoj23j2o3j2oij26jk2nkn26kjh23' 'strict-dynamic';
frame-ancestors: 'none';
img-src: data: https:;
report-uri: https://reporting.megabank.com
```

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) is a browser-implemented security mechanism that is often confused with CSP. While CSP allows a developer to choose which scripts are allowed to be *executed* in the browser, CORS is capable of blocking scripts at an earlier phase prior to the script ever reaching the JavaScript execution context in the browser.

CORS is important in part because two of the primary methods of performing network requests within JavaScript (the only browser-supported programming language) are `fetch` and `XMLHTTPRequest`. Both of these APIs respect a concept called Same Origin Policy (SOP), which stipulates that a web application should only be able to make network calls within its own (same) origin unless defined in a CORS policy.

It is important to note that without SOP, a `fetch` or `XMLHTTPRequest` from in-browser JavaScript code could perform privileged operations against another website (not the one in which the code executed). This is because the browser automatically attaches cookies to network requests, meaning if the user is authenticated on an ecommerce store in tab #2, tab #1 could make purchases against that ecommerce store if SOP did not exist (this is known as a Cross-Site Request Forgery or CSRF attack).

CORS gives us a configuration solution that enables developers to allow *specific* cross-origin requests to complete without significant degradation of secure posture (as would occur by disabling SOP).

Types of CORS Requests

There are two methods by which a CORS policy can be checked. These are often called the *simple* CORS method and the *preflight* CORS method.

The specification maintained by the WHATWG organization says while the simple version of the check is sufficient for the most common network requests, preflight should be used for any type of unusual or state-changing request.

Simple CORS Requests

A simple request can be either a POST, HEAD or GET request, but it must follow several conditions outlined in the specification. First, a simple request may only contain a few select headers. These are noted in the documentation as “CORS-safelisted headers”:

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Content-Type`

Next, the `Content-Type` header must only contain one of the following headers: `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`.

Finally, the network request originating from the browser may not make use of advanced network request features like event listeners attached to the `XMLHTTPRequest` object.

Should all of these conditions be met, the browser will attach a single header to the network request: `origin: <source-origin>`. If the requested server is configured correctly, it will return an HTTP status code 200 with another header attached: `access-control-allow-origin: <source-origin>`.

If the origin contained within the header provided by the browser and the header provided by the server do not match, the response from the server will not be passed back to script execution and the browser console will throw a `CORS Policy` error.

Preflighted CORS Requests

Any request that does not meet the criteria to be performed as a simple request will be performed as a preflighted request.

In a preflighted request, the browser will send a preflight check, which is simply an HTTP Options request formatted to include the following

headers:

- `origin: <source-origin>`
- `access-control-request-method: <permitted HTTP method>`
- `access-control-request-headers: <permitted HTTP headers>`

Similarly to the simple request, if any of these checks fail on the preflight request, the browser network stack will not pass the response to the script execution context and will instead throw a `CORS Policy` error.

The design theory behind preflighted requests is that they offer more security by refusing to pass any relevant data or operations to the server prior to determining if the server and browser are permitted to communicate via matching CORS headers.

Implementing CORS

While the browser determines what `origin`, `access-control-request-method`, and `access-control-request-headers` to attach to an outbound network request, it is still up to the application developer to correctly configure a server to match the inbound CORS network requests with appropriate responses.

Most modern server software packages offer simple CORS middleware, often abstracting away most of the CORS configuration. In the server-side JavaScript ecosystem, this is often done with the `cors` npm module:

```
const express = require('express');
const app = express();
const cors = require('cors');

const corsOptions = {
  origin: 'https://mega-bank.com'
};

app.get('/users/:id', cors(corsOptions), function (req, res, next) {
  res.sendStatus(200);
});

app.listen(443, function () {
```

```
    console.log('listening on port 443')
});
```

Like many other configuration-based utilities in the browser, CORS accepts multiple mechanisms for defining allowed origins—including wildcards (`*`). It's important to always remember the principle of least privilege and grant the server-side CORS policy the minimum scope of origins rather than the maximum. For that reason, allowlist only the specific origins you need for your application to function.

Headers

Modern browsers allow various types of security configuration to be performed at the header level, as we saw in both CSP and CORS. Beyond these landmark security mechanisms that are configured via HTTP headers, there are a multitude of additional header-based security mechanisms that offer a bit of security improvement with very little implementation difficulty. The following are several important, but often disregarded, security mechanisms that every modern web application should consider making use of.

Strict Transport Security

The HTTP Strict Transport Security (HSTS) header informs the browser that a web page should only be loaded over HTTPS and that future HTTP requests should be "upgraded" to HTTPS. HSTS is the preferred mechanism for forcing requests to make use of SSL/TLS (HTTPS) because `301 redirects` require an initial HTTP request, which can be vulnerable to man-in-the-middle attacks.

HSTS is easily implemented as a header, using the following syntax:

```
Strict-Transport-Security: max-age=<expire-time>; includeSubDomains; preload
```

The parameters are as follows:

`max-age`

Time in seconds for the browser to remember the HSTS configuration.

`includeSubDomains`

Apply HSTS policy to all subdomains of current origin.

`preload`

An optional parameter not included in the specification. Google currently runs a preload list to which your application can be submitted. Firefox and Chrome browsers will check the preload list prior to loading a web page and load the most recent HSTS policy if it exists.

Cross-Origin-Opener Policy (COOP)

Modern browsers maintain a "browsing context." This context includes data about the current page, iframes, and tab data. The sharing of this browsing context can be modified via the `Cross-Origin-Opener-Policy` security header.

This header can be implemented via the syntax: `Cross-Origin-Opener-Policy: <config>` where the config parameter can be `unsafe-none`, `same-origin-allow-popups`, or `same-origin`. The parameters are as follows:

`unsafe-none`

Permits the sharing of browsing context with the page or document that opened the current page (default value if header is not defined)

`same-origin-allow-popups`

Permits the sharing of browsing context with pop-ups originating from the current page

`same-origin`

Restricts sharing of browser context to only pages of the same origin

Despite being a relatively new and unknown security mechanism, setting the `Cross-Origin-Opener-Policy` header to `same-origin` prevents new tabs, windows, or other browsing contexts from being able to navigate back to the opening context. This is a large security boon because it prevents unintended information leakage.

Cross-Origin-Resource-Policy (CORP)

In 2018, security researchers discovered that by using side-channel timing attacks against modern CPU architectures, it was possible to read the memory and state from another tab via JavaScript. This exploit is known as Spectre, and it still affects Intel CPUs in use today.

The seldom-used but highly effective mitigation for this type of attack is the `Cross-Origin-Resource-Policy` header. This header, which is supported by Firefox, Edge, and Chrome web browsers, allows a developer to tell the browser to further restrict where a resource (e.g., document, window) can be accessed within the web browser.

The configuration values for this header are:

`same-origin`

: Restricts read to same-origin (protocol/scheme, domain, port). This is the most secure option.

`same-site`

: Restricts reads to same-site (top-level domain) only. This is less secure than `same-origin` because it permits subdomains to access resources sent by the server.

`cross-origin`

: Allows reads from any origin. This is the least secure configuration option.

Although this is an opt-in technology, the CORP header instructs the browser to enforce memory sharing at a lower level than possible

through JavaScript—making it a good addition to your security headers on any web application.

Headers with Security Implications

While less complex to understand and implement than the aforementioned headers, the following headers offer bits of additional security to your web application:

`X-Content-Type-Options`

When set to `nosniff`, prevents browsers from guessing MIME types (aka *mime sniffing*), which can prevent browsers from attempting to convert nonexecutable file types to executable file types.

`Content-Type`

One of the most common headers, can be used to indicate the type of content being sent in an HTTP request, e.g., `Content-Type: text/html; charset=UTF-8`. The primary security benefit is that it allows the client to interpret the content correctly.

Legacy Security Headers

In the past, individual headers were often used for enabling security mechanisms on the browser. However, in the modern era many of these headers have been rolled into features in CSP or enabled by default in the browser.

The following "legacy" headers are listed with their modern equivalents, for applications that need to keep said functionality but wish to implement it in a forward-looking manner:

`X-Frame-Options`

CSP `frame-ancestors`

`X-XSS-Protection`

Removed. CSP blocks the most common XSS sinks by default. Browsers no longer scan for reflected XSS.

`Expect-CT`

Removed. All TLS certificates generated after May 2018 support certificate transparency requirements (CT) by default. All previous TLS certificates are now expired due to the 39 month maximum age.

`Referrer-Policy`

Modern browsers no longer send all referrer information by default, but legacy browsers will send `origin`, `path`, and `query string` to the same site, and `origin` to other sites. This can be disabled in legacy browsers by setting the header: `Referrer-Policy: strict-origin-when-cross-origin`.

`X-Powered-By`

Some web servers attach this header by default so the client will know which server sent a response. This allows for easy fingerprinting of servers and versions, a type of information disclosure that makes attacks easier for hackers. This is disabled by default on most modern servers, but if you make use of a legacy version of your server software, make sure to disable this header.

`X-Download-Options`

Supported only by Internet Explorer, this header allowed a developer to prevent downloads from being run in memory with Internet Explorer context. As Internet Explorer is a deprecated browser, this header is usually no longer needed.

Cookies

Despite some modern web applications making use of alternative methods of storing and transmitting session information and tokens, most web applications still use cookies as the primary way of authenticating a user on a per-request basis. Because of this, it's important to understand how

you can lock down and harden the ways in which the browser handles your tokens to prevent leakage or modification.

Creating and Securing Cookies

Cookies are typically set via the HTTP header `Set-Cookie`. This header takes in a set of configurations including the content of the cookie data, which is a key:value pair separated by the equals (`=`) sign. After the cookie data, a set of attributes can be included that allow the developer to configure how the cookies are handled in the browser.

A common example of the creation of a cookie is as follows:

```
Set-Cookie: auth_token=abc123; Secure;
```

In the preceding example, a cookie is set via HTTP header that will henceforth be passed to and from the browser and server on every HTTP request. This cookie has the key `auth_token` and the value `abc123`. It also includes an attribute flag `Secure`, which tells the browser to only send the cookie if the website is loaded over HTTPS. This is done to prevent man-in-the-middle attacks.

The primary method of securing cookies is via attribute flags, of which a significant number allow the developer to restrict insecure mechanisms of the browser, thereby improving the security and integrity of your cookies. Of these attributes, the most powerful security flags are as follows:

`Path`

By default the browser will send all cookies on every network request that occurs for the current domain (e.g., *mega-bank.com*). Including a `path=/website` attribute would restrict the browser from sending cookies on any network request to the current domain that does not include */website* in its path.

`Secure`

The `secure` attribute should be attached to any cookie that includes sensitive data. It prevents the cookie from being sent over unencrypted network calls in order to prevent man-in-the-middle attacks from stealing the data within a cookie. It can be set via `Secure;`.

`Expires`

Forces the browser to discard the cookie after a set date specified by the `HTTP-date` timestamp, e.g., `Expires: Wed, 20 Dec 2025 01:01:00 GMT`.

`HttpOnly`

This attribute prevents JavaScript code from being able to read the cookies. This means that even if your website is compromised via an XSS attack that allows a hacker to run JavaScript against your users' browsers, the JavaScript should not be able to access the session cookies and hence should not be able to make privileged network calls on behalf of the hacked user. It can be set via `HttpOnly;`.

`SameSite`

Accepts three values: `Lax`, `Strict`, or `None`. Used to prevent CSRF sites by preventing cookies from being sent alongside cross-site requests. It should usually be set to `Strict`, which only allows a cookie to be sent from the site that generated it.

The `domain` attribute should not be used unless absolutely required, as it will loosen the restrictions put on cookies to only be sent to the current domain. An odd side effect of using the `domain` attribute on your cookies is that it changes the default cookie security model from `my-domain.com` to `*.my-domain.com` simply by being present. This means if you make use of the `domain` attribute for specifying additional domains for which your cookies can be sent, you will also be sending cookies to all of your subdomains. This can have a significant number of security implications spanning as far as easy account takeovers (ATO) should your subdomains contain user-developed script.

Testing Cookies

Typically cookies are tested via interception tools that allow a developer or security engineer to implement their own man-in-the-middle attack against a development or staging environment in order to intercept and read their application's cookies. Various tools, often called HTTP proxies, are available for interception. Several of these tools are specialized for security testing, including the very popular Burp Suite and the open source competitor ZAP.

When using either of these tools with an application that only communicates over HTTP, you will need to configure the local application with a TLS certificate used for testing and provide a decryption key to your HTTP interception tool.

Cookies can also be tested in the browser development tools provided with Firefox, Safari, Chrome, or Edge web browsers. Doing so is quite simple and can be done via console or UI.

To view cookies on the Chrome web browser, perform the following steps:

1. Press F12 to open the browser Developer tools.
2. Go to the Application tab in the console.
3. Expand the Cookies dropdown under the Storage section.
4. Select the websites for which you wish to view the cookies.

Inside of the Developer tools JavaScript console, you can also type `document.cookie` to see the cookies for the currently loaded main document of a web page.

Framing and Sandboxing

Occasionally when developing a web application, you may run into an edge case where you want to run code from another developer within the same page as your first-party code. This is a common use case in websites that allow in-depth user customization, the classic example being MySpace, which allowed deep customization of user profiles. Another

case may be the partnership between your company and another company, both producing related web-based software.

In either of these cases, allowing third-party code to run in your website is by default a huge risk. But these risks can be mitigated to some extent if the third-party code is *sandboxed* prior to being executed inside of your web application. In this section, we will discuss the pros and cons of several methods of sandboxing potentially malicious third-party code.

Traditional Iframe

If implemented correctly, the browser's `iframe` element can allow a developer to securely use code from another developer within the same page as first-party code. Iframes create a separate DOM and JavaScript context, which are sandboxed to prevent leakage to the main frame. Iframe escapes are considered P0 (Critical Risk) exploits and are rapidly resolved via browser vendors, which means Google (Chrome), Microsoft (Edge), and Mozilla (Firefox) are funding the security of this mechanism rather than your company.

The downsides to using an iframe are significant and as a result, it is not the appropriate sandboxing tool for every situation, despite being one of the easiest. Iframes don't meld well with most UIs because they carry their own DOM and must be configured identically to the primary window. As a result, iframes require code duplication, which means that if the UI of the main frame changes a color, it must be changed in the frame CSS as well. This leads to annoyances during development and takes additional time. Also on the UI front, iframes are difficult to scale up and down as if they were a primary DOM element. The iframe object requires unique CSS formatting that adds complexity.

Beyond these limitations, iframe communication with the main window is limited to the `POSTMessage` API. This means that any subframe-to-main-frame communication requires additional development time.

Finally, because the iframe bootstraps its own DOM and JavaScript context and requires additional code for configuration, it may present signifi-

cant performance degradation when multiple iframes are used in the same page.

These limitations aside, iframes are easy to implement. A simple iframe sandbox would appear as follows:

```
<iframe src="https://other-website.com"></iframe>
```

Because the preceding iframe makes use of a separate origin from the main window, *https://mega-bank.com*, it gains the benefits of both iframe isolation and SOP isolation mechanisms.

An even more secure implementation is as follows:

```
<iframe src="https://other-website.com" sandbox></iframe>
```

Do note the `sandbox` attribute attached to the `iframe` element. The `sandbox` attribute forces the iframe to use several additional layers of isolation, including:

- Treats content as separate origin (see the discussion of SOP in "Cross-Origin Resource Sharing")
- Blocks form submissions
- Blocks script execution
- Disables APIs
- Prevents links from accessing other browsing contexts
- Prevents the use of plug-ins
- Prevents the iframe from viewing the parent browsing context
- Disables autoplay

The iframe should be your first line of defense when considering sandboxing. Nonetheless, it is important to always weigh its limitations and know what your other options are.

Web Workers

Web workers do not offer the same level of isolation as an iframe, but they do offer more isolation than executing third-party code in the same execution context as first-party code. Web workers are separate isolated threads run inside of the browser that are capable of same-origin script execution but without the capacity to interact with the DOM.

Creating a web worker is simple on supported browsers:

```
if (window.Worker) {
 // create Web Worker and have it run code from code.js
 const myWorker = new Worker('code.js');
 // destroy Web Worker
 myWorker.terminate()
}
```

Because web workers have their own JavaScript execution context, they cannot access variables and data from the main JavaScript execution context of the page unless it is explicitly passed via `postMessage` or `MessageChannel` APIs. However, web workers are capable of both in-domain and cross-origin HTTP requests via the `XMLHTTPRequest` API, which makes them less secure than a separate-origin iframe.

Subresource Integrity

Subresource integrity is not an isolation technique per se, but instead a method of verifying that code from a third party has not been modified.

If iframes do not fit your needs for integrating with a third-party JavaScript vendor, your team could manually review the third-party JavaScript and then generate a based64-encoded SHA-256, SHA-384, or SHA-512 hash representing the content of that JavaScript code.

After generating a hash representing the content of the third-party JavaScript code, you can load that code into your browser with an `integrity` attribute that includes the previously generated hash:

```
<script src="other-website.com/stuff.js"
        integrity="sha384-NmJhYmViMzNiMmU1NzllMDMyODdl"
        crossorigin="anonymous"></script>
```

On load, the browser will perform an HTTP GET to *other-website.com/stuff.js* to obtain the script. Prior to loading it into the JavaScript execution context, though, it will compare the hash of the resource with the hash provided in the `integrity` attribute. If the integrity hash does not match the hash of the file loaded from the third-party website, the browser will deem the file modified since it was last reviewed, throw a console error, and refuse to load the JavaScript.

Shadow Realms

Shadow Realms is, at the time of this writing, an upcoming JavaScript language feature that aims to get rid of some of the pitfalls of iframe isolation while maintaining JavaScript sandboxing between different developers. Shadow Realms introduces a new method of executing JavaScript in an isolated manner by creating a new execution context with its own global objects, intrinsics, and built-ins (standard objects and functions).

Shadow Realms can be tested today by implementing one of the publicly available shims—JavaScript-based code blocks that emulate the API that is planned to be released to all major web browsers by 2025.

Creating a Shadow Realm is easy and can be done entirely in JavaScript:

```
const shadowRealm = new ShadowRealm();

// imports code that executes within its own environment.
const doSomething = await shadowRealm.importValue('./file.js', 'redDoSomething')

// This call chains to the shadowRealm's redDoSomething
doSomething();
```

Shadow Realms is designed to be easy to implement and doesn't interfere with your UI. It takes up less memory and is capable of *synchronous code execution*. This means you can write code in your main execution context

that executes directly after your Shadow Realm's code. This is not possible in an iframe, as iframe-to-main-window communication is by default *asynchronous*.

Preparing for and supporting the release of Shadow Realms may be the best method of sandboxing JavaScript for next-generation browsers. Shadow Realms is currently a stage 3/4 TC39 proposal, which means it will be added to the JavaScript programming language once it is capable of meeting Test262 (compatibility) tests with all major web browsers.

Summary

Web application security is largely about writing secure code and deploying web applications in a secure manner. But as the ecosystem for developing and deploying web applications evolves, more and more third-party code, tools, and technologies become standard.

As an adaptive security engineer, it's important to note that your web application's security is only as good as its weakest link. Often that weakest link is the browser, network stack, or a third-party integration. Properly configuring your web application to mitigate these additional risks to the best of your ability gives your web application a chance at being deployed, utilized, and maintained as securely as possible.