*The goal of this document is to help operational teams with creating secure web applications. All Mozilla sites and deployments are expected to follow the recommendations below. Use of these recommendations by the public is strongly encouraged. The Enterprise Information Security (Infosec) team maintains this document as a reference guide.*

**Table of Contents**

1. [Cheat Sheet](https://infosec.mozilla.org/guidelines/web_security#web-security-cheat-sheet)
2. [Transport Layer Security (TLS/SSL)](https://infosec.mozilla.org/guidelines/web_security#transport-layer-security-tlsssl)
   1. [HTTPS](https://infosec.mozilla.org/guidelines/web_security#https)
   2. [HTTP Strict Transport Security](https://infosec.mozilla.org/guidelines/web_security#http-strict-transport-security)
   3. [HTTP Redirections](https://infosec.mozilla.org/guidelines/web_security#http-redirections)
   4. [HTTP Public Key Pinning](https://infosec.mozilla.org/guidelines/web_security#http-public-key-pinning)
   5. [Resource Loading](https://infosec.mozilla.org/guidelines/web_security#resource-loading)
3. [Content Security Policy](https://infosec.mozilla.org/guidelines/web_security#content-security-policy)
4. [contribute.json](https://infosec.mozilla.org/guidelines/web_security#contributejson)
5. [Cookies](https://infosec.mozilla.org/guidelines/web_security#cookies)
6. [Cross-origin Resource Sharing](https://infosec.mozilla.org/guidelines/web_security#cross-origin-resource-sharing)
7. [CSRF Prevention](https://infosec.mozilla.org/guidelines/web_security#csrf-prevention)
8. [Referrer Policy](https://infosec.mozilla.org/guidelines/web_security#referrer-policy)
9. [robots.txt](https://infosec.mozilla.org/guidelines/web_security#robotstxt)
10. [Subresource Integrity](https://infosec.mozilla.org/guidelines/web_security#subresource-integrity)
11. [X-Content-Type-Options](https://infosec.mozilla.org/guidelines/web_security#x-content-type-options)
12. [X-Frame-Options](https://infosec.mozilla.org/guidelines/web_security#x-frame-options)
13. [X-XSS-Protection](https://infosec.mozilla.org/guidelines/web_security#x-xss-protection)
14. [Version History](https://infosec.mozilla.org/guidelines/web_security#version-history)

**Web Security Cheat Sheet**

| **Guideline** | **Security Benefit** | **Implementation Difficulty** | **Order†** | **Requirements** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| [HTTPS](https://infosec.mozilla.org/guidelines/web_security#https) | **MAXIMUM** | **MEDIUM** |  | Mandatory | Sites should use HTTPS (or other secure protocols) for all communications |
| [Public Key Pinning](https://infosec.mozilla.org/guidelines/web_security#http-public-key-pinning) | **LOW** | **MAXIMUM** | -- | Mandatory for maximum risk sites only | Not recommended for most sites |
| [Redirections from HTTP](https://infosec.mozilla.org/guidelines/web_security#http-redirections) | **MAXIMUM** | **LOW** | 3 | Mandatory | Websites must redirect to HTTPS, API endpoints should disable HTTP entirely |
| [Resource Loading](https://infosec.mozilla.org/guidelines/web_security#resource-loading) | **MAXIMUM** | **LOW** | 2 | Mandatory for all websites | Both passive and active resources should be loaded through protocols using TLS, such as HTTPS |
| [Strict Transport Security](https://infosec.mozilla.org/guidelines/web_security#http-strict-transport-security) | **HIGH** | **LOW** | 4 | Mandatory for all websites | Minimum allowed time period of six months |
| [TLS Configuration](https://infosec.mozilla.org/guidelines/web_security#https) | **MEDIUM** | **MEDIUM** | 1 | Mandatory | Use the most secure Mozilla TLS configuration for your user base, typically [Intermediate](https://wiki.mozilla.org/Security/Server_Side_TLS#Intermediate_compatibility_.28default.29) |
| [Content Security Policy](https://infosec.mozilla.org/guidelines/web_security#content-security-policy) | **HIGH** | **HIGH** | 10 | Mandatory for new websites Recommended for existing websites | Disabling inline script is the greatest concern for CSP implementation |
| [Cookies](https://infosec.mozilla.org/guidelines/web_security#cookies) | **HIGH** | **MEDIUM** | 7 | Mandatory for all new websites Recommended for existing websites | All cookies must be set with the Secure flag, and set as restrictively as possible |
| [contribute.json](https://infosec.mozilla.org/guidelines/web_security#contributejson) | **LOW** | **LOW** | 9 | Mandatory for all new Mozilla websites Recommended for existing Mozilla sites | Mozilla sites should serve contribute.json and keep contact information up-to-date |
| [Cross-origin Resource Sharing](https://infosec.mozilla.org/guidelines/web_security#cross-origin-resource-sharing) | **HIGH** | **LOW** | 11 | Mandatory | Origin sharing headers and files should not be present, except for specific use cases |
| [Cross-site Request Forgery Tokenization](https://infosec.mozilla.org/guidelines/web_security#csrf-prevention) | **HIGH** | **UNKNOWN** | 6 | Varies | Mandatory for websites that allow destructive changes Unnecessary for all other websites Most application frameworks have built-in CSRF tokenization to ease implementation |
| [Referrer Policy](https://infosec.mozilla.org/guidelines/web_security#referrer-policy) | **LOW** | **LOW** | 12 | Recommended for all websites | Improves privacy for users, prevents the leaking of internal URLs via Refererheader |
| [robots.txt](https://infosec.mozilla.org/guidelines/web_security#robotstxt) | **LOW** | **LOW** | 14 | Optional | Websites that implement robots.txt must use it only for noted purposes |
| [Subresource Integrity](https://infosec.mozilla.org/guidelines/web_security#subresource-integrity) | **MEDIUM** | **MEDIUM** | 15 | Recommended‡ | ‡ Only for websites that load JavaScript or stylesheets from foreign origins |
| [X-Content-Type-Options](https://infosec.mozilla.org/guidelines/web_security#x-content-type-options) | **LOW** | **LOW** | 8 | Recommended for all websites | Websites should verify that they are setting the proper MIME types for all resources |
| [X-Frame-Options](https://infosec.mozilla.org/guidelines/web_security#x-frame-options) | **HIGH** | **LOW** | 5 | Mandatory for all websites | Websites that don't use DENY or SAMEORIGIN must employ clickjacking defenses |
| [X-XSS-Protection](https://infosec.mozilla.org/guidelines/web_security#x-xss-protection) | **LOW** | **MEDIUM** | 13 | Mandatory for all new websites Recommended for existing websites | Manual testing should be done for existing websites, prior to implementation |

† Suggested order that administrators implement the web security guidelines. It is based on a combination of the security impact and the ease of implementation from an operational and developmental perspective.

**Transport Layer Security (TLS/SSL)**

Transport Layer Security provides assurances about the confidentiality, authentication, and integrity of all communications both inside and outside of Mozilla. To protect our users and networked systems, the support and use of encrypted communications using TLS is mandatory for all systems.

**HTTPS**

Websites or API endpoints that only communicate with modern browsers and systems should use the [Mozilla modern TLS configuration](https://wiki.mozilla.org/Security/Server_Side_TLS#Modern_compatibility).

Websites intended for general public consumption should use the [Mozilla intermediate TLS configuration](https://wiki.mozilla.org/Security/Server_Side_TLS#Intermediate_compatibility_(default)).

Websites that require backwards compatibility with extremely old browsers and operating systems may use the [Mozilla backwards compatible TLS configuration](https://wiki.mozilla.org/Security/Server_Side_TLS#Old_backward_compatibility). This is not recommended, and use of this compatibility level should be noted in your risk assessment.

**Compatibility**

| **Configuration** | **Oldest compatible clients** |
| --- | --- |
| Modern | Firefox 27, Chrome 22, Internet Explorer 11, Opera 14, Safari 7, Android 4.4, Java 8 |
| Intermediate | Firefox 1, Chrome 1, Internet Explorer 7, Opera 5, Safari 1, Internet Explorer 8 (XP), Android 2.3, Java 7 |
| Backwards Compatible (Old) | Internet Explorer 6 (XP), Java 6 |

**See Also**

* [Mozilla Server Side TLS Guidelines](https://wiki.mozilla.org/Security/Server_Side_TLS)
* [Mozilla Server Side TLS Configuration Generator](https://mozilla.github.io/server-side-tls/ssl-config-generator/) - generates software configurations for the three levels of compatibility

**HTTP Strict Transport Security**

HTTP Strict Transport Security (HSTS) is an HTTP header that notifies user agents to only connect to a given site over HTTPS, even if the scheme chosen was HTTP. Browsers that have had HSTS set for a given site will transparently upgrade all requests to HTTPS. HSTS also tells the browser to treat TLS and certificate-related errors more strictly by disabling the ability for users to bypass the error page.

The header consists of one mandatory parameter (max-age) and two optional parameters (includeSubDomains and preload), separated by semicolons.

**Directives**

* max-age: how long user agents will redirect to HTTPS, in seconds
* includeSubDomains: whether user agents should upgrade requests on subdomains
* preload: whether the site should be included in the [HSTS preload list](https://hstspreload.org/)

max-age must be set to a minimum of six months (15768000), but longer periods such as two years (63072000) are recommended. Note that once this value is set, the site must continue to support HTTPS until the expiry time has been reached.

includeSubDomains notifies the browser that all subdomains of the current origin should also be upgraded via HSTS. For example, setting includeSubDomains on domain.mozilla.com will also set it on host1.domain.mozilla.com and host2.domain.mozilla.com. Extreme care is needed when setting the includeSubDomains flag, as it could disable sites on subdomains that don’t yet have HTTPS enabled.

preload allows the website to be included in the [HSTS preload list](https://hstspreload.org/), upon submission. As a result, web browsers will do HTTPS upgrades to the site without ever having to receive the initial HSTS header. This prevents downgrade attacks upon first use and is recommended for all high risk websites. Note that being included in the HSTS preload list requires that includeSubDomains also be set.

**Examples**

*# Only connect to this site via HTTPS for the two years (recommended)*

Strict-Transport-Security: max-age**=**63072000

*# Only connect to this site and subdomains via HTTPS for the next two years and also include in the preload list*

Strict-Transport-Security: max-age**=**63072000; includeSubDomains; preload

**See Also**

* [MDN on HTTP Strict Transport Security](https://developer.mozilla.org/docs/Web/Security/HTTP_strict_transport_security)
* [RFC6797: HTTP Strict Transport Security (HSTS)](https://tools.ietf.org/html/rfc6797)

**HTTP Redirections**

Websites may continue to listen on port 80 (HTTP) so that users do not get connection errors when typing a URL into their address bar, as browsers currently connect via HTTP for their initial request. Sites that listen on port 80 should only redirect to the same resource on HTTPS. Once the redirection has occured, [HSTS](https://infosec.mozilla.org/guidelines/web_security#http-strict-transport-security) should ensure that all future attempts go to the site via HTTP are instead sent directly to the secure site. APIs or websites not intended for public consumption should disable the use of HTTP entirely.

Redirections should be done with the 301 redirects, unless they redirect to a different path, in which case they may be done with 302 redirections. Sites should avoid redirections from HTTP to HTTPS on a different host, as this prevents HSTS from being set.

**Examples**

*# Redirect all incoming http requests to the same site and URI on https, using nginx*

**server** {

**listen** 80;

**return** 301 https://$host$request\_uri;

}

*# Redirect for site.mozilla.org from http to https, using Apache*

<**VirtualHost** \*:80>

**ServerName** site.mozilla.org

**Redirect** permanent / https://site.mozilla.org/

</**VirtualHost**>

**HTTP Public Key Pinning**

[Maximum risk](https://infosec.mozilla.org/guidelines/risk/standard_levels#standard-risk-levels-definition-and-nomenclature) sites must enable the use of HTTP Public Key Pinning (HPKP). HPKP instructs a user agent to bind a site to specific root certificate authority, intermediate certificate authority, or end-entity public key. This prevents certificate authorities from issuing unauthorized certificates for a given domain that would nevertheless be trusted by the browsers. These fradulent certificates would allow an active attacker to MitM and impersonate a website, intercepting credentials and other sensitive data.

Due to the risk of knocking yourself off the internet, HPKP must be implemented with extreme care. This includes having backup key pins, testing on a non-production domain, testing with Public-Key-Pins-Report-Only and then finally doing initial testing with a very short-lived max-age directive. Because of the risk of creating a self-denial-of-service and the very low risk of a fraudulent certificate being issued, it is *not recommended* for the majority websites to implement HPKP.

**Directives**

* max-age: how long the user agent the keys will be pinned; the site must use a cert that meets these pins until this time expires
* includeSubDomains: whether user agents should pin all subdomains to the same pins

Unlike with HSTS, what to set max-age is highly individualized to a given site. A longer value is more secure, but screwing up your key pins will result in your site being unavailable for a longer period of time. Recommended values fall between 15 and 120 days.

**Examples**

*# Pin to DigiCert, Let's Encrypt, and the local public-key, including subdomains, for 15 days*

Public-Key-Pins: max-age**=**1296000; includeSubDomains; pin-sha256**=**"WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18=";

pin-sha256**=**"YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg="; pin-sha256**=**"P0NdsLTMT6LSwXLuSEHNlvg4WxtWb5rIJhfZMyeXUE0="

**See Also**

* [About Public Key Pinning](https://noncombatant.org/2015/05/01/about-http-public-key-pinning/)
* [The HPKP Toolset](https://scotthelme.co.uk/hpkp-toolset/) - helpful tools for generating key pins

**Resource Loading**

All resources — whether on the same origin or not — should be loaded over secure channels. Secure (HTTPS) websites that attempt to load active resources such as JavaScript insecurely will be blocked by browsers. As a result, users will experience degraded UIs and “mixed content” warnings. Attempts to load passive content (such as images) insecurely, although less risky, will still lead to degraded UIs and can allow active attackers to deface websites or phish users.

Despite the fact that modern browsers make it evident that websites are loading resources insecurely, these errors still occur with significant frequency. To prevent this from occuring, developers should verify that all resources are loaded securely prior to deployment.

**Examples**

*<!-- HTTPS is a fantastic way to load a JavaScript resource -->*

<script src="https://code.jquery.com/jquery-1.12.0.min.js"></script>

*<!-- Attempts to load over HTTP will be blocked and will generate mixed content warnings -->*

<script src="http://code.jquery.com/jquery-1.12.0.min.js"></script>

*<!-- Although passive content won't be blocked, it will still generate mixed content warnings -->*

<img src="http://very.badssl.com/image.jpg">

**See Also**

* [MDN on Mixed Content](https://developer.mozilla.org/en-US/docs/Security/MixedContent)

**Content Security Policy**

Content Security Policy (CSP) is an HTTP header that allows site operators fine-grained control over where resources on their site can be loaded from. The use of this header is the best method to prevent cross-site scripting (XSS) vulnerabilities. Due to the difficulty in retrofitting CSP into existing websites, CSP is mandatory for all new websites and is strongly recommended for all existing high-risk sites.

The primary benefit of CSP comes from disabling the use of unsafe inline JavaScript. Inline JavaScript – either reflected or stored – means that improperly escaped user-inputs can generate code that is interpreted by the web browser as JavaScript. By using CSP to disable inline JavaScript, you can effectively eliminate almost all XSS attacks against your site.

Note that disabling inline JavaScript means that *all* JavaScript must be loaded from <script> src tags . Event handlers such as *onclick* used directly on a tag will fail to work, as will JavaScript inside <script> tags but not loaded via src. Furthermore, inline stylesheets using either <style> tags or the style attribute will also fail to load. As such, care must be taken when designing sites so that CSP becomes easier to implement.

**Implementation Notes**

* Aiming for default-src: https: is a great first goal, as it disables inline code and requires https.
* For existing websites with large codebases that would require too much work to disable inline scripts, default-src: https: 'unsafe-inline' is still helpful, as it keeps resources from being accidentally loaded over http. However, it does not provide any XSS protection.
* It is recommended to start with a reasonably locked down policy such as default-src 'none'; img-src 'self'; script-src 'self'; style-src 'self' and then add in sources as revealed during testing.
* In lieu of the preferred HTTP header, pages can instead include a <meta http-equiv=“Content-Security-Policy”content=“…”> tag. If they do, it should be the first <meta> tag that appears inside <head>.
* Care needs to be taken with data: URIs, as these are unsafe inside script-src and object-src (or inherited from default-src).
* Similarly, the use of script-src 'self' can be unsafe for sites with JSONP endpoints. These sites should use a script-srcthat includes the path to their JavaScript source folder(s).
* Unless sites need the ability to execute plugins such as Flash or Silverlight, they should disable their execution with object-src 'none'.
* Sites should ideally use the report-uri directive, which POSTs JSON reports about CSP violations that do occur. This allows CSP violations to be caught and repaired quickly.
* Prior to implementation, it is recommended to use the Content-Security-Policy-Report-Only HTTP header, to see if any violations would have occured with that policy.

**Examples**

*# Disable unsafe inline/eval, only allow loading of resources (images, fonts, scripts, etc.) over https*

*# Note that this does not provide any XSS protection*

Content-Security-Policy: default-src https:

*<!-- Do the same thing, but with a <meta> tag -->*

<meta http-equiv="Content-Security-Policy" content="default-src https:">

*# Disable the use of unsafe inline/eval, allow everything else except plugin execution*

Content-Security-Policy: default-src **\***; object-src 'none'

*# Disable unsafe inline/eval, only load resources from same origin except also allow images from imgur*

*# Also disables the execution of plugins*

Content-Security-Policy: default-src 'self'; img-src 'self' https://i.imgur.com; object-src 'none'

*# Disable unsafe inline/eval and plugins, only load scripts and stylesheets from same origin, fonts from google,*

*# and images from same origin and imgur. Sites should aim for policies like this.*

Content-Security-Policy: default-src 'none'; font-src 'https://fonts.googleapis.com';

img-src 'self' https://i.imgur.com; object-src 'none'; script-src 'self'; style-src 'self'

*# Pre-existing site that uses too much inline code to fix*

*# but wants to ensure resources are loaded only over https and disable plugins*

Content-Security-Policy: default-src https: 'unsafe-eval' 'unsafe-inline'; object-src 'none'

*# Don't implement the above policy yet; instead just report violations that would have occured*

Content-Security-Policy-Report-Only: default-src https:; report-uri /csp-violation-report-endpoint/

*# Disable the loading of any resources and disable framing, recommended for APIs to use*

Content-Security-Policy: default-src 'none'; frame-ancestors 'none'

**See Also**

* [An Introduction to Content Security Policy](http://www.html5rocks.com/en/tutorials/security/content-security-policy/)
* [Content Security Policy Playground](http://www.cspplayground.com/)
* [Content Security Policy Level 2 Standard](https://www.w3.org/TR/CSP2/)
* [Google CSP Evaluator](https://csp-evaluator.withgoogle.com/)
* [Using the frame-ancestors directive to prevent framing](https://infosec.mozilla.org/guidelines/web_security#x-frame-options)

**contribute.json**

contribute.json is a text file placed within the root directory of a website that describes what it is, where its source exists, what technologies it uses, and how to reach support and contribute. contribute.json is a Mozilla standard used to describe all active Mozilla websites and projects.

Its existence can greatly speed up the process of bug triage, particularly for smaller websites with just a handful of maintainers. It further assists security researchers to find testable websites and instructs them on where to file their bugs against. As such, contribute.json is mandatory for all Mozilla websites, and must be maintained as contributors join and depart projects.

Require subkeys include name, description, bugs, participate (particularly irc and irc-contacts), and urls.

**Examples**

{

"name": "Bedrock",

"description": "The app powering www.mozilla.org.",

"repository": {

"url": "https://github.com/mozilla/bedrock",

"license": "MPL2",

"tests": "https://travis-ci.org/mozilla/bedrock/"

},

"participate": {

"home": "https://wiki.mozilla.org/Webdev/GetInvolved/mozilla.org",

"docs": "http://bedrock.readthedocs.org/",

"mailing-list": "https://www.mozilla.org/about/forums/#dev-mozilla-org",

"irc": "irc://irc.mozilla.org/#www",

"irc-contacts": [

"someperson1",

"someperson2",

"someperson3"

]

},

"bugs": {

"list": "https://bugzilla.mozilla.org/describecomponents.cgi?product=www.mozilla.org",

"report": "https://bugzilla.mozilla.org/enter\_bug.cgi?product=www.mozilla.org",

"mentored": "https://bugzilla.mozilla.org/buglist.cgi?f1=bug\_mentor&o1=isnotempty

&query\_format=advanced&bug\_status=NEW&product=www.mozilla.org&list\_id=10866041"

},

"urls": {

"prod": "https://www.mozilla.org",

"stage": "https://www.allizom.org",

"dev": "https://www-dev.allizom.org",

"demo1": "https://www-demo1.allizom.org"

},

"keywords": [

"python",

"less-css",

"django",

"html5",

"jquery"

]

}

**See Also**

* [The contribute.json Standard](https://www.contributejson.org/)

**Cookies**

All cookies should be created such that their access is as limited as possible. This can help minimize damage from cross-site scripting (XSS) vulnerabilities, as these cookies often contain session identifiers or other sensitive information.

**Directives**

* Name: Cookie names may be either be prepended with either \_\_Secure- or \_\_Host- to prevent cookies from being overwritten by insecure sources
  + Use \_\_Host- for all cookies needed only on a specific domain (no subdomains) where Path is set to /
  + Use \_\_Secure- for all other cookies sent from secure origins (such as HTTPS)
* Secure: All cookies must be set with the Secure flag, indicating that they should only be sent over HTTPS
* HttpOnly: Cookies that don’t require access from JavaScript should be set with the HttpOnly flag
* Expiration: Cookies should expire as soon as is necessary: session identifiers in particular should expire quickly
  + Expires: Sets an absolute expiration date for a given cookie
  + Max-Age: Sets a relative expiration date for a given cookie (not supported by IE <8)
* Domain: Cookies should only be set with this if they need to be accessible on other domains, and should be set to the most restrictive domain possible
* Path: Cookies should be set to the most restrictive path possible, but for most applications this will be set to the root directory
* SameSite: Forbid sending the cookie via cross-origin requests (such as from <img> tags, etc.), as a strong [anti-CSRF measure](https://infosec.mozilla.org/guidelines/web_security#csrf-prevention)
  + SameSite=Strict: Only send the cookie when site is directly navigated to
  + SameSite=Lax: Send the cookie when navigating to your site from another site

**Examples**

*# Session identifier cookie only accessible on this host that gets purged when the user closes their browser*

Set-Cookie: MOZSESSIONID**=**980e5da39d4b472b9f504cac9; Path**=**/; Secure; HttpOnly

*# Session identifier for all mozilla.org sites that expires in 30 days using the \_\_Secure- prefix*

*# This cookie is not sent cross-origin, but is sent when navigating to any Mozilla site from from another site*

Set-Cookie: \_\_Secure-MOZSESSIONID**=**7307d70a86bd4ab5a00499762; Max-Age**=**2592000; Domain**=**mozilla.org; Path**=**/; Secure; HttpOnly; SameSite**=**Lax

*# Sets a long-lived cookie for the current host, accessible by Javascript, when the user accepts the ToS*

*# This cookie is sent when navigating to your sent from another site, such as by clicking a link*

Set-Cookie: \_\_Host-ACCEPTEDTOS**=**true; Expires**=**Fri, 31 Dec 9999 23:59:59 GMT; Path**=**/; Secure; SameSite**=**Lax

*# Session identifier used for a secure site, such as bugzilla.mozilla.org. It isn't sent from cross-origin*

*# requests, nor is it sent when navigating to bugzilla.mozilla.org from another site. Used in conjunction with*

*# other anti-CSRF measures, this is a very strong way to defend your site against CSRF attacks.*

Set-Cookie: \_\_Host-BMOSESSIONID**=**YnVnemlsbGE**=**; Max-Age**=**2592000; Path**=**/; Secure; HttpOnly; SameSite**=**Strict

**See Also**

* [RFC 6265 (HTTP Cookies)](https://tools.ietf.org/html/rfc6265)
* [HTTP Cookie Prefixes](https://tools.ietf.org/html/draft-west-cookie-prefixes)
* [Same-site Cookies](https://tools.ietf.org/html/draft-west-first-party-cookies-07)

**Cross-origin Resource Sharing**

Access-Control-Allow-Origin is an HTTP header that defines which foreign origins are allowed to access the content of pages on your domain via scripts using methods such as XMLHttpRequest. crossdomain.xml and clientaccesspolicy.xml provide similar functionality, but for Flash and Silverlight-based applications, respectively.

These should not be present unless specifically needed. Use cases include content delivery networks (CDNs) that provide hosting for JavaScript/CSS libraries and public API endpoints. If present, they should be locked down to as few origins and resources as is needed for proper function. For example, if your server provides both a website and an API intended for XMLHttpRequest access on a remote websites, *only* the API resources should return the Access-Control-Allow-Origin header. Failure to do so will allow foreign origins to read the contents of any page on your origin.

**Examples**

*# Allow any site to read the contents of this JavaScript library, so that subresource integrity works*

Access-Control-Allow-Origin: **\***

*# Allow https://random-dashboard.mozilla.org to read the returned results of this API*

Access-Control-Allow-Origin: https://random-dashboard.mozilla.org

*<!-- Allow Flash from https://random-dashboard.mozilla.org to read page contents -->*

<cross-domain-policy xsi:noNamespaceSchemaLocation="http://www.adobe.com/xml/schemas/PolicyFile.xsd">

<allow-access-from domain="random-dashboard.mozilla.org"/>

<site-control permitted-cross-domain-policies="master-only"/>

<allow-http-request-headers-from domain="random-dashboard.mozilla.org" headers="\*" secure="true"/>

</cross-domain-policy>

*<!-- The same thing, but for Silverlight-->*

**<?xml version="1.0" encoding="utf-8"?>**

<access-policy>

<cross-domain-access>

<policy>

<allow-from http-request-headers="\*">

<domain uri="https://random-dashboard.mozilla.org"/>

</allow-from>

<grant-to>

<resource path="/" include-subpaths="true"/>

</grant-to>

</policy>

</cross-domain-access>

</access-policy>

**See Also**

* [MDN on HTTP access control (CORS)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS)
* [Adobe on Setting crossdomain.xml](https://www.adobe.com/devnet/adobe-media-server/articles/cross-domain-xml-for-streaming.html)
* [Microsoft on Setting clientaccesspolicy.xml](https://msdn.microsoft.com/library/cc197955%28v=vs.95%29.aspx)

**CSRF Prevention**

Cross-site request forgeries are a class of attacks where unauthorized commands are transmitted to a website from a trusted user. Because they inherit the users cookies (and hence session information), they appear to be validly issued commands. A CSRF attack might like this:

<!-- Attempt to delete a user's account -->

<img src="https://accounts.mozilla.org/management/delete?confirm=true">

When a user visits a page with that HTML fragment, the browser will attempt to make a GET request to that URL. If the user is logged in, the browser will provide their session cookies and the account deletion attempt will be successful.

While there are a variety of mitigation strategies such as Origin/Referrer checking and challenge-response systems (such as [CAPTCHA](https://en.wikipedia.org/wiki/CAPTCHA)), the most common and transparent method of CSRF mitigation is through the use of anti-CSRF tokens. Anti-CSRF tokens prevent CSRF attacks by requiring the existence of a secret, unique, and unpredictable token on all destructive changes. These tokens can be set for an entire user session, rotated on a regular basis, or be created uniquely for each request. Although [SameSite](https://infosec.mozilla.org/guidelines/web_security#Cookies) cookies are the best defense against CSRF attacks, they are not yet fully supported in all browsers and should be used in conjection with other anti-CSRF defenses.

**Examples**

*<!-- A secret anti-CSRF token, included in the form to delete an account -->*

<input type="hidden" name="csrftoken" value="1df93e1eafa42012f9a8aff062eeb1db0380b">

*# Server-side: set an anti-CSRF cookie that JavaScript must send as an X header, which can't be done cross-origin*

Set-Cookie: CSRFTOKEN**=**1df93e1eafa42012f9a8aff062eeb1db0380b; Path**=**/; Secure; SameSite**=**Strict

*// Client-side, have JavaScript add it as an X header to the XMLHttpRequest*

**var** token **=** readCookie(CSRFTOKEN); *// read the cookie*

httpRequest.setRequestHeader('X-CSRF-Token', token); *// add it as an X-CSRF-Token header*

**See Also**

* [Wikipedia on CRSF Attacks and Prevention](https://en.wikipedia.org/wiki/Cross-site_request_forgery#Prevention)
* [OWASP CSRF Prevention Cheat Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)

**Referrer Policy**

When a user navigates to a site via a hyperlink or a website loads an external resource, browsers inform the destination site of the origin of the requests through the use of the HTTP Referer (sic) header. Although this can be useful for a variety of purposes, it can also place the privacy of users at risk. HTTP Referrer Policy allows sites to have fine-grained control over how and when browsers transmit the HTTP Referer header.

In normal operation, if a page at <https://example.com/page.html> contains <img src="https://not.example.com/image.jpg">, then the browser will send a request like this:

**GET** /image.jpg **HTTP/**1.1

Host: not.example.com

Referer: https://example.com/page.html

In addition to the privacy risks that this entails, the browser may also transmit internal-use-only URLs that it may not have intended to reveal. If you as the site operator want to limit the exposure of this information, you can use HTTP Referrer Policy to either eliminate the Referer header or reduce the amount of information that it contains.

**Directives**

* no-referrer: never send the Referer header
* same-origin: send referrer, but only on requests to the same origin
* strict-origin: send referrer to all origins, but only the URL sans path (e.g. <https://example.com/>)
* strict-origin-when-cross-origin: send full referrer on same origin, URL sans path on foreign origin

**Notes**

Although there are other options for referrer policies, they do not protect user privacy and limit exposure in the same way as the options above.

no-referrer-when-downgrade is the default behavior for all current browsers, and can be used when sites are concerned about breaking existing systems that rely on the full Referrer header for their operation.

Please note that support for Referrer Policy is still in its infancy. Chrome currently only supports no-referrer from the directives above, and Firefox awaits full support with Firefox 52.

**Examples**

*# On example.com, only send the Referer header when loading or linking to other example.com resources*

Referrer-Policy: same-origin

*# Only send the shortened referrer to a foreign origin, full referrer to a local host*

Referrer-Policy: strict-origin-when-cross-origin

*# Disable referrers for browsers that don't support strict-origin-when-cross-origin*

*# Uses strict-origin-when-cross-origin for browsers that do*

Referrer-Policy: no-referrer, strict-origin-when-cross-origin

*<!-- Do the same, but with a meta tag -->*

<meta http-equiv="Referrer-Policy" content="no-referrer, strict-origin-when-cross-origin">

*<!-- Do the same, but only for a single link -->*

<a href="https://mozilla.org/" referrerpolicy="no-referrer, strict-origin-when-cross-origin">

**See Also**

* [Referrer Policy standard](https://w3c.github.io/webappsec-referrer-policy/#referrer-policy-same-origin)
* [MDN on Referrer Policy](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy)

**robots.txt**

robots.txt is a text file placed within the root directory of a site that tells robots (such as indexers employed by search engines) how to behave, by instructing them not to index certain paths on the website. This is particularly useful for reducing load on your website, though disabling the indexing of automatically generated content. It can also be helpful for preventing the pollution of search results, for resources that don’t benefit from being searchable.

Sites may optionally use robots.txt, but should only use it for these purposes. It should not be used as a way to prevent the disclosure of private information or to hide portions of a website. Although this does prevent these sites from appearing in search engines, it does not prevent its discovery from attackers, as robots.txt is frequently used for reconnaisance.

**Examples**

*# Stop all search engines from archiving this site*

User-agent: **\***

Disallow: /

*# Using robots.txt to hide certain directories is a terrible idea*

User-agent: **\***

Disallow: /secret/admin-interface

**See Also**

* [About robots.txt](http://www.robotstxt.org/robotstxt.html)

**Subresource Integrity**

Subresource integrity is a recent W3C standard that protects against attackers modifying the contents of JavaScript libraries hosted on content delivery networks (CDNs) in order to create vulnerabilities in all websites that make use of that hosted library.

For example, JavaScript code on jquery.org that is loaded from mozilla.org has access to the entire contents of everything of mozilla.org. If this resource was successfully attacked, it could modify download links, deface the site, steal credentials, cause denial-of-service attacks, and more.

Subresource integrity locks an external JavaScript resource to its known contents at a specific point in time. If the file is modified at any point thereafter, supporting web browsers will refuse to load it. As such, the use of subresource integrity is mandatory for all external JavaScript resources loaded from sources not hosted on Mozilla-controlled systems.

Note that CDNs must support the Cross Origin Resource Sharing (CORS) standard by setting the Access-Control-Allow-Originheader. Most CDNs already do this, but if the CDN you are loading does not support CORS, please contact Mozilla Information Security. We are happy to contact the CDN on your behalf.

**Directives**

* integrity: a cryptographic hash of the file, prepended with the hash function used to generate it
* crossorigin: should be anonymous to inform browsers to send anonymous requests without cookies

**Examples**

*<!-- Load jQuery 2.1.4 from their CDN -->*

<script src="https://code.jquery.com/jquery-2.1.4.min.js"

integrity="sha384-R4/ztc4ZlRqWjqIuvf6RX5yb/v90qNGx6fS48N0tRxiGkqveZETq72KgDVJCp2TC"

crossorigin="anonymous"></script>

*<!-- Load AngularJS 1.4.8 from their CDN -->*

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"

integrity="sha384-r1y8TJcloKTvouxnYsi4PJAx+nHNr90ibsEn3zznzDzWBN9X3o3kbHLSgcIPtzAp"

crossorigin="anonymous"></script>

*# Generate the hash myself*

$ curl -s https://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js | \

openssl dgst -sha384 -binary | \

openssl base64 -A

r1y8TJcloKTvouxnYsi4PJAx+nHNr90ibsEn3zznzDzWBN9X3o3kbHLSgcIPtzAp

**See Also**

* [SRI Hash Generator](https://www.srihash.org/) - generates <script> tags for you, and informs you if the CDN lacks CORS support
* [Subresource Integrity W3C Standard](https://w3c.github.io/webappsec-subresource-integrity/)

**X-Content-Type-Options**

X-Content-Type-Options is a header supported by Internet Explorer, Chrome and Firefox 50+ that tells it not to load scripts and stylesheets unless the server indicates the correct MIME type. Without this header, these browsers can incorrectly detect files as scripts and stylesheets, leading to XSS attacks. As such, all sites must set the X-Content-Type-Options header and the appropriate MIME types for files that they serve.

**Examples**

*# Prevent browsers from incorrectly detecting non-scripts as scripts*

X-Content-Type-Options: nosniff

**See Also**

* [Microsoft on Reducing MIME Type Security Risks](https://msdn.microsoft.com/en-us/library/gg622941%28v=vs.85%29.aspx)

**X-Frame-Options**

X-Frame-Options is an HTTP header that allows sites control over how your site may be framed within an iframe. Clickjacking is a practical attack that allows malicious sites to trick users into clicking links on your site even though they may appear to not be on your site at all. As such, the use of the X-Frame-Options header is mandatory for all new websites, and all existing websites are expected to add support for X-Frame-Options as soon as possible.

Note that X-Frame-Options has been superceded by the Content Security Policy’s frame-ancestors directive, which allows considerably more granular control over the origins allowed to frame a site. As frame-ancestors is not yet supported in IE11 and older, Edge, Safari 9.1 (desktop), and Safari 9.2 (iOS), it is recommended that sites employ X-Frame-Options in addition to using CSP.

Sites that require the ability to be iframed must use either Content Security Policy and/or employ JavaScript defenses to prevent clickjacking from malicious origins.

**Directives**

* DENY: disallow allow attempts to iframe site (recommended)
* SAMEORIGIN: allow the site to iframe itself
* ALLOW-FROM *uri*: deprecated; instead use CSP’s frame-ancestors directive

**Examples**

*# Block site from being framed with X-Frame-Options and CSP*

Content-Security-Policy: frame-ancestors 'none'

X-Frame-Options: DENY

*# Only allow my site to frame itself*

Content-Security-Policy: frame-ancestors 'self'

X-Frame-Options: SAMEORIGIN

*# Allow only framer.mozilla.org to frame site*

*# Note that this blocks framing from browsers that don't support CSP2+*

Content-Security-Policy: frame-ancestors https://framer.mozilla.org

X-Frame-Options: DENY

**See Also**

* [MDN on X-Frame-Options](https://developer.mozilla.org/en-US/docs/Web/HTTP/X-Frame-Options)
* [CSP standard on ‘frame-ancestors’](https://www.w3.org/TR/CSP2/#directive-frame-ancestors)
* [OWASP Clickjacking Defense Cheat Sheet](https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet)

**X-XSS-Protection**

X-XSS-Protection is a feature of Internet Explorer and Chrome that stops pages from loading when they detect reflected cross-site scripting (XSS) attacks. Although these protections are largely unnecessary in modern browsers when sites implement a strong Content Security Policy that disables the use of inline JavaScript ('unsafe-inline'), they can still provide protections for users of older web browsers that don’t yet support CSP.

New websites should use this header, but given the small risk of false positives, it is only recommended for existing sites. This header is unnecessary for APIs, which should instead simply return a restrictive Content Security Policy header.

**Examples**

*# Block pages from loading when they detect reflected XSS attacks*

X-XSS-Protection: 1; mode**=**block

**Version History**

| **Date** | **Editor** | **Changes** |
| --- | --- | --- |
| April, 2018 | April | Added SameSite cookies and syntax highlighting |
| June, 2017 | April | Moved cookie prefixes to no longer be experimental |
| November, 2016 | April | Added Referrer Policy, tidied up XFO examples |
| October, 2016 | April | Updates to CSP recommendations |
| July, 2016 | April | Updates to CSP for APIs, and CSP’s deprecation of XFO, and XXSSP |
| February, 2016 | April | Initial document creation |