CONTENT SECURITY POLICY BEST PRACTICES

Jake Meredith – jmeredith[at]isecpartners[dot]com

iSEC Partners, Inc
123 Mission Street, Suite 1020
San Francisco, CA 94105
https://www.isecpartners.com

July 14, 2013

Abstract

Content Security Policy is an HTTP header that provides client-side defense-in-depth against content injection attacks. This document describes the nuances of Content Security Policy, provides guidance on testing and deploying, and proposes a list of best practices for its secure use.

1 INTRODUCTION

Content Security Policy 1.0 is a mechanism that allows developers to whitelist the locations from which applications can load resources. It does this through use of directives which restrict various web resources for the web application. The main purpose behind Content Security Policy is to mitigate content injection vulnerabilities. Cross-site scripting (XSS) vulnerabilities tend to be the most prevalent and harmful form of these vulnerabilities. Web applications can use this feature by supplying a `Content-Security-Policy` HTTP header that specifies valid script locations. If the web application is attempting to load a script, the client browser matches the source of the script against the Content Security Policy directives before loading it. Content Security Policy extends beyond script origins, allowing web applications to control the source of images, frames, styles, and other elements. It is important to note that CSP is a second-line defense mechanism. Application authors should always take standard precautions to prevent injections such as output encoding of user-controlled content.

Content Security Policy can be deployed in a `Report-Only` mode. In this mode, the web application's behavior does not change, but any violations of the policy are reported to a specified URI for logging. This makes it possible to test different policies or directives before deploying them.

Content Security Policy is a very granular control system and can therefore be easily implemented incorrectly. The best practices will enable server administrators and developers to avoid common Content Security Policy deployment mistakes and help secure their environment. This paper will describe the details of Content Security Policy, give implementation specifics, and then propose a list of best practices for its use.

2 CONTENT SECURITY POLICY

Content Security Policy is defined by a list of semicolon (;) delimited directives. A directive is defined by specifying a resource and a list of allowable URIs. This list of URIs defines a whitelist for the only locations

from which the client browser can load resources for the page. Content Security Policy has numerous directives, schemes, and keywords that can be defined as part of the header. Since CSP is enforced by the web application client, we will end this section by summarizing how the most popular web browsers support CSP.

2.1 DIRECTIVES

2.1.1 DEFAULT-SRC

The `default-src` directive is exactly as it sounds: a directive that specifies the default rules governing resources when a resource-specific directive is not defined. If other directives (see below) are not explicitly defined, then the `default-src` directive will be used for that resource. A typical policy using this directive would be:

```
Content-Security-Policy: default-src 'self'
```

This policy contains one directive, `default-src`, which permits all resources to be loaded only from the applications own origin, see section 2.13.1 for details on the keyword `self`.

If any resource- specific directive is defined in the Content-Security-Policy header after the `default-src`, that directive will overwrite the `default-src` directive for that particular resource completely. The following example demonstrates this behavior:

```
Content-Security-Policy: default-src 'none'; script-src js.csp.com;
```

The `default-src` directive specifies that no resources are permitted; see section 2.13.2 for details on the keyword `none`. The subsequent `script-src` directive overwrites the `default-src` directive for script resources, such as JavaScript. The end result of this directive would allow the web application to load scripts from js.csp.com and forbid all other resources.

2.1.2 SCRIPT-SRC

The `script-src` directive controls the allowable locations from where an application is allowed to load scripts. This directive may be given multiple sources separated by a space as permitted locations. A simple policy using this directive would be:

```
Content-Security-Policy: default-src 'self'; script-src scripts.csp.com
```

This policy permits the client browser to load scripts from script.csp.com and all other resources from the host domain.

There are a two other directives that can apply to `script-src` control on script source locations: `unsafe-inline` and `unsafe-eval`. The `unsafe-inline` directive instructs the browser to execute inline scripts. The `unsafe-eval` directive allows the browser to execute certain JavaScript functions otherwise disallowed by Content Security Policy. These directives exist in order to facilitate adoption of Content Security Policy. The removal of inline scripts and unsafe methods is a non-trivial task detailed in section 3.1.2 and 3.1.3, so these directives allow more applications to begin using and benefitting from Content Security Policy before they can be completely stripped of inline scripts or unsafe methods. An example of their basic use is:

```
Content-Security-Policy: default-src 'self'; script-src scripts.csp.com 'unsafe-inline'
'unsafe-eval'
```

This policy would have the same effect as the previous policy except that inline scripts would be allowed and the client browser would not prohibit the dangerous JavaScript methods, such as `eval` and `Funtion`.The

`unsafe-inline` and `unsafe-eval` directives have to be applied to either a `script-src` or `default-src` directive in order to provide functionality. The inclusion of these directives limits the effectiveness of preventing Cross-site Scripting.

2.1.3 OBJECT-SRC

The security model of the most common plugins allows full DOM access, so restricting object resources is as important to preventing content injection as restricting script resources. The `object-src` directive controls the source locations of embedded elements, applets, and plugins. This directive may be given a list of sources from which to load object resources. A simple policy using this directive would be:

```
Content-Security-Policy: default-src 'none'; object-src plugins.csp.com
```

This policy begins by specifying a default policy where no resources are allowed. The second part of the policy allows the client browser to load object data from plugins.csp.com, overwriting the default policy for this particular resource. The end result of this policy is that object data is the only permitted resource on this page and can only be loaded from plugins.csp.com.

2.1.4 STYLE-SRC

The `style-src` directive dictates the locations from which the client browser can load stylesheets. As style, can be specified inline, Content Security Policy applies similar restrictions on style as the `script-src` directive does with scripts. If a web application requires inline style the policy must specify `unsafe-inline`. The same security concerns occur in the use of `unsafe-inline` for style as they do for scripts. Many web applications depend upon inline style and externalizing these styles is non-trivial, see sections 3.1.2 and 3.1.3 for detailed descriptions. A policy involving the `style-src` is:

```
Content-Security-Policy: default-src 'none'; style-src *.csp.com 'unsafe-inline'
```

This policy first declares a default policy of `none` which disallows all resources. The `style-src` directive lists any subdomain of csp.com and specifies the allowance of inline style.

2.1.5 IMAGE-SRC

The `image-src` directive limits the locations from which the client browser can load image resources. This directive takes a list of allowed image sources. An example of its use is:

```
Content-Security-Policy: default-src 'self'; image-src img.csp.com images.csp.com
```

This policy dictates a standard `default-src` directive and then specifies two valid locations for images to be loaded from.

2.1.6 MEDIA-SRC

The `media-src` directive allows video and audio sources for the web application. A simple example of its use would be:

```
Content-Security-Policy: default-src 'none'; media-src videos.csp.com media.other.com
```

This policy only permits media sources from the two stated domains and disallows all other resources.

2.1.7 FRAME-SRC

The `frame-src` directive permits frames from certain URIs to be present in the application, it does not control the ability of the application to be framed on other pages (for this look to X-Frame-Options). Many web applications use frames to bring in additional features from external sources. An example would be:

```
Content-Security-Policy: default-src 'self'; frame-src youtube.com
```

This directive would allow frames from youtube.com but all other resources would have to come from its own origin.

2.1.8 FONT-SRC

The `font-src` directive controls access to font resources, such as Google Web Fonts. A simple example of its usage is:

```
Content-Security-Policy: default-src 'self'; font-src
https://themes.googleusercontent.com;
```

This directive would allow fonts to come from Google Web Fonts and all other resources could be accessed only from the origin.

2.1.9 CONNECT-SRC

The `connect-src` directive permits connections, via XML HTTP Request(XHR)/WebSockets/EventSource, to other origins. An example of this is:

```
Content-Security-Policy: default-src 'none'; connect-src xhr.csp.com
```

This directive would only allow outgoing connections (such as XHR) to xhr.csp.com and would not permit any other resources to be loaded into the application.

2.1.10 REPORT URI

Content Security Policy has the ability to not only block offending resources, but also to report the offense to the server through the use of a `report-uri` directive. To use the `report-uri` directive, simply add the directive to the end of the policy followed by location for the reports to go:

```
Content-Security-Policy: default-src 'self'; script-src js.csp.com; report-uri
reports.csp.com/cspReport.cgi
```

The violation reports will be sent via POST as a JSON blob that will look like this:

```
{
	"csp-report": {
		"document-uri": "http://csp.com/index.html",
		"referrer": "http://notorigin.com",
		"blocked-uri": "http://notorigin.com/attack.js",
		"violated directive": "script-src 'none'" ,
		"original-policy": "default-src 'self'; script-src 'none'; report-uri
		/uri_parser"
	}
}
```

Table 1 report-uri

The parts of the JSON blob are fairly self-explanatory and give a lot of information about the violation.

2.1.11 SANDBOX

When the `sandbox` directive is used, the browser will treat the page as though it were loaded into an iframe with the `sandbox` attribute. This will effectively remove this frame from associating with the main application in terms of the Same Origin Policy. This will prevent the page from performing certain actions, such as submitting forms and many others. This attribute is best applied to pages that contain untrusted content and those which do not need to submit information.

The sandbox directive has several keywords to alter its restrictions. If the sandbox directive is stated without any additional values:

```
Content-Security-Policy: sandbox
```

The header will enforce a Same Origin Policy, prevent popups, and prevent plugin and script execution on the associated page. All of these restrictions can be removed by adding additional parameters to the directive. The additional parameters are: allow-forms, allow-same-origin, allow-scripts, and allow-top-navigation. Each of these values can be added to the sandbox attribute in order to fit an application's needs. An example of the use of sandbox to only restrict scripts is:

```
Content-Security-Policy: sandbox allow-forms allow-same-origin allow-top-navigation
```

2.2 SCHEME

```
The scheme indicates a particular protocol or type of data that will be transmitted. Examples
of schemes are http: or javascript:. If the scheme is added to a directive, only that type of
protocol or data will be allowed to be processed for the resource. An example of using the
scheme to force the use of HTTPS for all resources in a web application is:
       Content-Security-Policy: default-src https:;
```

One pitfall to be aware of is that if the `default-src` has a defined scheme and then another resource specific directive is defined afterwards, it must also specify all schemes it applies to, including those already defined in `default-src`. An example of this is:

```
Content-Security-Policy: default-src https:; script-src https://partner-site.com
```

A mistake would be to only put partner-site.com for the `script-src` directive assuming that the `default-src` will permit only HTTPS traffic. Any resource specific directive overwrites the `default-src` directive for that particular resource.

2.3 KEYWORDS

2.3.1 SELF

When specifying a directive there are a few keywords that can be used to simplify definitions. The `self` keyword aliases the origin of the application. To match self, the resource specified must be loaded from the same host, with the same protocol using the same port. For example, the application is hosted on https://csp.com with a Content Security Policy of:

```
Content-Security-Policy: default-src 'self';
```

This table will show whether a resource can be loaded from the specified domain according to this `Content-Security-Policy` header.

URL	Outcome	Reason
https://csp.com/test.js	Success	Same protocol and host
https://csp.com/dir/test.js	Success	Same protocol and host
http://csp.com/test.js	Failure	Different protocol
https://test.csp.com/test.js	Failure	Different host
https://www.csp.com/dir/test.js	Failure	Different host
https://csp.com:8443/test.js	Failure	Different port

Table 2 Same Origin

2.3.2 NONE

The `none` keyword defines no sources are allowed in the application. This keyword can be an effective way to begin a `default-src` directive in a `Content-Security-Policy` header. It sets a restrictive policy that disallows all resources by default, making permitted resources explicit. Example:

```
Content-Security-Policy: default-src 'none'
```

This header prevents any resources from being loaded.

2.4 BROWSER SUPPORT

Since CSP is still in a draft mode, there is no standardized HTTP header name yet and implementation varies between web browsers. The following table summarizes CSP support offered by the most popular web browsers at time of writing

Browser	Header Name	Fully supported since version	Features supported
Firefox	`Content-Security-Policy`	23.0	All
Chrome	`Content-Security-Policy`	25.0	All
IE	`X-Content-Security-Policy`	Not fully supported	`sandbox` directive only
Safari	`X-Webkit-CSP`	6.0	All
Opera	`Content-Security-Policy`	15.0	All
Android Browser	Not Supported	N/A	None
iOS Safari	`X-Webkit-CSP`	6.0	All
Blackberry Browser	Not Supported	N/A	None

Table 3 Browser Support

For readability purposes, the examples provided document only set the `Content-Security-Policy` header. When implementing CSP in web applications the recommendation is to currently use only the standard header. Using the prefixed header can be useful if the application is specifically targeting the `sandbox` directive in IE. Behavior with the prefixed header could be unintended because of its different syntax and different behavior in Firefox.

3 Implementing Content Security Policy

Implementing Content Security Policy should be done in steps. The first step towards full implementation will be to find out exactly what resources the current web application is loading. The most convenient way to do this will be to deploy CSP in the report only mode, using the `Content-Security-Policy-Report-Only` header.

3.1.1 Content-Security-Policy-Report-Only

Content Security Policy comes with a handy HTTP header called `Content-Security-Policy-Report-Only`. This header can be defined in exactly the same way as a standard `Content-Security-Policy` header but will only report the violations, without preventing their execution. Using the `Report-Only` header will not affect the behavior of the application in any way. This allows for a consequence free way of finding out how the web application will fare under various Content Security Policy configurations. In order to find out exactly what types of resources the web application is currently accessing, try creating a `Content-Security-Policy-Report-Only` header as:

```
Content-Security-Policy-Report-Only: default-src 'none'; report-uri /violation_parser;
```

This will send a violation report to "/violation_parser" for any resources the web application loads, but will not prevent those resources from loading. This particular policy will generate a report for EVERY resource the application loads, so it will create a very large amount of data.

In order to use this data, create a simple database that has a hook for the parser URI and collects all of the violations for a prescribed amount of time. Use the database to collate these reports and figure out exactly what resources the web application uses. Once the data has been collected decide for each resource that was accessed whether or not it should be allowed to be accessed. For instances, find all the scripts legitimately loaded by the application and add their location to the list of permitted script locations, using the `script-src`. If it is evident that only scripts from a certain subdomain are being loaded, then the script-src directive should list only that subdomain as a valid location. Each resource that is to be permitted needs to have a corresponding directive in the Content Security Policy.

Once all of the data has been combed through and directives written, update the `Content-Security-Policy-Report-Only` header with all of the new directives. Using a similar database system, do another round of data gathering using the new header and verify that any violations coming through are expected. Repeat this process as many times as is necessary to feel confident about the Content Security Policy implementation not breaking the application's usability when converted to an enforceable header.

3.1.2 Playing the Inline Script Game

Content Security Policy depends on having specific URI's for valid resource locations. This allows the web application to control the resources that are loaded and prevent malicious content injection. Due to inline

scripts not having an external URI, there is no way for Content Security Policy to ensure their validity and they should not be used. All style attributes and tags must also be externalized to allow Content Security Policy to control their usage, as they can be a target of content injection as well. Removing all inline scripts and style is, in general, non-trivial.

Assume a `Content-Security-Policy` header of:

```
Content-Security-Policy: script-src 'self'
```

This would prevent the use of inline scripts and only allow script resource to load from the origin, csp.com. On the index page of csp.com is a small inline script:

```
<script> alert('Welcome to CSP!')</script>
```

index.html

This script would be flagged as a violation and will not be permitted. In order to maintain functionality this script can be externalized. Create a Javascript file called alert.js with the contents of the `<script>` tag inside a function

```
function welcome()
{
     alert("Welcome to CSP!");
}
```

alert.js

and place the file within the origin, csp.com/alert.js. On the index page of csp.com simply include the Javascript file

```
<script src='alert.js'></script>
```

index.html

And call the welcome() function

```
<script>welcome()</script>
```

index.html

Another very common use of inline scripts is the *onclick()* functionality, to externalize these the *addEventListener()* can be used. A common web application inline script such as:

```
<a href="#" onClick="alert('you clicked me')">Click Me</a>
```

html file

can be replaced with *addEventListener()* calls:

```
function someEvent() {
alert("you clicked me");
}
var obj = document.getElementById("someElementId");
obj.addEventListener("click", someEvent);
```

events.js

```
<script src='events.js'</script>
<a href="#" id="someElementId">Click Me</a>
```

html file

The restriction on inline scripts by Content Security Policy can be avoided by including the `unsafe-inline` directive. This would greatly weaken Content Security Policy's power to prevent XSS but can be a simple way to use Content Security Policy without substantial upfront investment in bringing the application code into compliance with Content Security Policy. The use of `unsafe-inline` can allow the development team time to remove inline scripts and inline CSS while still taking advantage of the other security features offered by Content Security Policy[1].

3.1.3 Evaluating your functions

In order to take advantage of the most secure form of Content Security Policy, several functions need to be eliminated from the code base. These functions are dangerous in that they can cause a web application to execute user input as JavaScript code:

- Javascript operator and function *eval()*
- *Function()* constructor
- *setTimeout()* method without a function as the first argument
- *setInterval()* method without a function as the first argument

The removal of these functions from an established codebase can be non-trivial. When `Content-Security-Policy` is defined, the former two functions will throw a security exception, while the latter will return 0. This restriction can also be avoided by including the `unsafe-eval` directive but would significantly weaken a lot of the protection that Content Security Policy provides. Just like the `unsafe-inline` directive, the `unsafe-eval` directive could be used in the Content Security Policy declaration in order to give the development team time to convert from using the above functions.

3.1.4 Converting to regular Content Security Policy

Once the `Content-Security-Policy-Report-Only` has been established and the violation reporting is happening as desired, the `Content-Security-Policy` header can be used without negatively affecting the application. Simply copy the approved directives from the Content-Security-Policy-Report-Only header and add them to a Content-Security-Policy header. Any violations of the directives will now be blocked.

The `report-uri` can either be different for the `Content-Security-Policy` header or the same as the `Report-Only` header, but it is important that the violation reports are logged. Any violations will be representative of malicious intent, mistakes in the header, or mistakes in the application code. It is important to monitor these violation reports to determine the reason.

3.1.5 Iterative Content Security Policy

The best way to use the `Content-Security-Policy-Report-Only` header after the Content Security Policy has been established is to adjust and analyze the `Content-Security-Policy-Report-Only` header to find even more strict and secure ways to use the directives. Having a report only feature allows for testing new directives and trying out new ways to make the application more secure. For instance, a good starter header might be something as simple as:

[1] http://nmatatal.blogspot.com/2013/01/removing-inline-javascript-for-csp.html

```
Content-Security-Policy: default-src 'self' *.csp.com unsafe-inline; report-uri
'/reporting'
```

This will only allow resources from the application's origin and from subdomains. But if, after experimenting with the `Content-Seucrity-Policy-Report-Only` header, it is found that the application only loads scripts from scripts.csp.com, then a more secure policy would be:

```
Content-Security-Policy: default-src 'self' *.csp.com unsafe-inline; script-src
scripts.csp.com unsafe-inline; report-uri '/reporting'
```

This would be the same as the previous header for all resources except scripts which can now only load from the scripts.csp.com. Now, after the development team has worked very hard to remove all inline scripts, the `Content-Security-Policy-Report-Only` header can be modified to be:

```
Content-Security-Policy-Report-Only: default-src 'self'; script-src scripts.csp.com;
report-uri '/reporting'
```

This header removes the unsafe-inline directive from both the `default-src` and `script-src` directives and instructs browsers to report any violations of the more restrictive policy. After any remaining issues are corrected, the main `Content-Security-Policy` header can be updated to reflect the changes made to the `Content-Security-Policy-Report-Only` header;

```
Content-Security-Policy: default-src 'self' *.csp.com; script-src scripts.csp.com;
report-uri 'reporting'
```

Using this process the `Content-Security-Policy-Report-Only` header can be used to iteratively increase the security of the web application.

4 Best Practices

The best practices listed below are a list of rules to follow that would create the most secure Content Security Policy and deploying it in a disciplined fashion that doesn't break existing functionality. Not all of these are trivial, and for many legacy applications the upfront refactoring investment required to comply with these recommendations will be prohibitive. This is not a list of things all web applications **must** do, but a list of practices that web applications should work toward in order to improve their security. There are also a couple real-world examples of companies implementing Content Security Policy, Twitter, Github.

4.1 Support standard Content Security Policy header

A Content Security Policy helps to detect and mitigate certain types of attacks, including XSS and other injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware. Web applications should have a `Content-Security-Policy` header on HTTP responses. There are 3 variations of the header right now: `X-WebKit-CSP`, `Content-Security-Policy`, and `X-Content-Security-Policy`. Even though it would seem natural to support all headers, the behavior of the non-standard headers can vary, so it is recommended to support only the standard header unless a specific functionality is targetted.

4.2 Avoid the unsafe-inline directive

Content Security Policy allows the server to control the locations from which the application expects to load resources. If the `unsafe-inline` directive is specified then Content Security Policy has no way to control the

injection of inline scripts. Authors should not include `unsafe-inline` in their Content Security Policy if they wish to protect their web application from XSS.

4.3 AVOID THE UNSAFE-EVAL DIRECTIVE

Content Security Policy prevents the use of specific Javascript functions such as *eval()*, which can execute user submitted data as code. The `unsafe-eval` directive will disable this feature and allow the use of the unsafe functions. Adding this directive bypasses one of the most important aspects of Content Security Policy and should not be used.

4.4 NEVER SPECIFY A WILDCARD (*) AS THE DEFAULT POLICY

Content Security Policy allows wildcards through the use of '*'. A Content Security Policy containing a bare '*' as the default policy will not be able to mitigate the risk of script or content injection. This would be the same as not including the `Content-Security-Policy` header, and would provide no protection.

4.5 NEVER SPECIFY A WILDCARD (*) WHEN REFERENCING TOP LEVEL DOMAINS, E.G. *.COM

Content Security Policy allows wildcards through the use of '*'. The misuse of a wildcard can be very dramatic and the server could allow access to more than intended. A directive of `default-src *.com` would allow anyone with a .com domain to have their content included within the application resources.

4.6 ALWAYS SPECIFY A DEFAULT-SRC DIRECTIVE

There are several Content Security Policy directives that a user agent must enforce: `script-src`, `object-src`, `style-src`, `image-src`, `media-src`, `frame-src`, `font-src`, and `connect-src`. Since some of these may be unneeded or not considered, the most secure way to establish a Content Security Policy is to first set the `default-src` to `'self'` or `'none'` and then build up the other directives as needed. This will make the web application "secure by default" with additional directives serving as exceptions added in order to maintain functionality. Not supplying a `default-src` directive means that the application is by default insecure except where specified.

4.7 ALWAYS SPECIFY A REPORT-URI DIRECTIVE

Adding a reporting URI allows the web application to log violations against the defined Content Security Policy. Without this directive, the administrators/developers have no way of detecting flaws in their web application or Content Security Policy configuration. Reporting is also very important for the initial phase of setting up Content Security Policy. Analysis of the `report-uri` of the `Content-Security-Policy-Report-Only` header can inform as to what resources are currently being loaded and from which locations. This is vital in determining the Content Security Policy for an application.

4.8 NO RESOURCE-SPECIFIC DIRECTIVE MAY INDUCE A LOWER SECURITY SCHEME THAN THE DEFAULT-SRC DIRECTIVE

If a `default-src` directive has a defined scheme and resource-specific directives are defined after `default-src`, the resource-specific directive must specify the same scheme or one of higher security. The resource-specific directives, when declared after a `default-src` directive, will overwrite the `default-src` directive

completely for that particular resource. Care needs to be taken when schemes are defined to not induce lower security restrictions for specific resources. An example of improper use:

```
Content-Security-Policy: default-src https:; frame-src frame.csp.com; //WRONG
```

This header would restrict all resources except frames from being passed over HTTPS, but would allow frames to be loaded over an HTTP connection. There are rare cases where this is the intended functionality, but generally this is not advisable.

4.9 All Content Security Policies should use the report-only functionality to observe and detect any needed changes in the Content Security Policy

A Content Security Policy configuration is only as secure as the directives used to define it. Once the `Content-Security-Policy` header is established, using the `Content-Security-Policy-Report-Only` header will allow for testing different directives to see how the current policy can be trimmed or modified to create the most restrictive policy possible while still allowing all resources to be obtained. The `Content-Security-Policy-Report-Only` header should be used in tandem with the `Content-Security-Policy` header to monitor the application for any resource violations and to secure the application.

The report-only mode should also be used to test significant changes in the Content Security Policy configuration before adding to the Content-Security-Policy header in order to maintain site functionality.

4.10 Never use a path within the resource location

According to Content Security Policy 1.0 client browsers should ignore any paths specified a directive source location. If a path is specified in the source location and the author of the policy does not realize that it will be dropped during processing, the directive may include resource locations that are not secure or not under the author's control. An example of including an unsupported path:

```
Content-Security-Policy: default-src csp.com/secure/;
```

This header specifies a path "/secure" in the csp.com domain as the source of all resources for the application. Content Security Policy 1.0 does not support paths in this manner, so the header would actually allow all resources from the csp.com domain instead of only from the "/secure" path.

Note: Content Security Policy 1.1 may support the use of paths, so this recommendation may change in the future.

5 Conclusion

Content Security Policy allows for fine-grain access control over resource origin for web applications. The implementation of a basic form of Content Security Policy can be easily accomplished on any website. The `Content-Security-Policy-Report-Only` header gives the developers/administrators a process for adding Content Security Policy to a web application without breaking the functionality. The removal of inline script, inline style, and insecure functions can be a large task, so the Content Security Policy can be configured to allow these until the application code can be properly modified. If the prerequisite work is done, Content Security Policy can mitigate common content injection vulnerabilities. The best practices proposed above give the developers a guideline for designing a secure policy and avoiding some common missteps in the process.

REFERENCES

1. W3C, Content Security Policy 1.0, http://www.w3.org/TR/CSP/

2. M. West, An Introduction to Content Security Policy, http://www.html5rocks.com/en/tutorials/security/content-security-policy/

3. http://caniuse.com/contentsecuritypolicy