21. Security HTTP Response Headers

This section discusses Spring Security's support for adding various security headers to the response.

21.1 Default Security Headers

Spring Security allows users to easily inject the default security headers to assist in protecting their application. The default for Spring Security is to include the following headers:

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

Strict-Transport-Security is only added on HTTPS requests

For additional details on each of these headers, refer to the corresponding sections:

- Cache Control
- Content Type Options
- HTTP Strict Transport Security
- X-Frame-Options
- X-XSS-Protection

While each of these headers are considered best practice, it should be noted that not all clients utilize the headers, so additional testing is encouraged.

You can customize specific headers. For example, assume that want your HTTP response headers to look like the following:

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
```

Specifically, you want all of the default headers with the following customizations:

- X-Frame-Options to allow any request from same domain
- HTTP Strict Transport Security (HSTS) will not be addded to the response

You can easily do this with the following Java Configuration:

```
@EnableWebSecurity
public class WebSecurityConfig extends
                WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        // ...
                        .headers()
                                .frameOptions().sameOrigin()
                                .httpStrictTransportSecurity().disable
        }
}
```

Alternatively, if you are using Spring Security XML Configuration, you can use the following:

```
<http>
        <!-- ... -->

        <headers>
                <frame-options policy="SAMEORIGIN" />
                <hsts disable="true"/>
        </headers>
</http>
```

If you do not want the defaults to be added and want explicit control over what should be used, you can disable the defaults. An example for both Java and XML based configuration is provided below:

If you are using Spring Security's Java Configuration the following will only add Cache Control.

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
	http
	// ...
	.headers()
		// do not use any default headers unless explicitly li
		.defaultsDisabled()
		.cacheControl();
}
}
```

The following XML will only add Cache Control.

```
<http>
	<!-- ... -->

	<headers defaults-disabled="true">
		<cache-control/>
	</headers>
</http>
```

If necessary, you can disable all of the HTTP Security response headers with the following Java Configuration:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
	http
	// ...
	.headers().disable();
}
}
```

If necessary, you can disable all of the HTTP Security response headers with the following XML configuration below:

```
<http>
	<!-- ... -->

	<headers disabled="true" />
```

```
</http>
```

21.1.1 Cache Control

In the past Spring Security required you to provide your own cache control for your web application. This seemed reasonable at the time, but browser caches have evolved to include caches for secure connections as well. This means that a user may view an authenticated page, log out, and then a malicious user can use the browser history to view the cached page. To help mitigate this Spring Security has added cache control support which will insert the following headers into you response.

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
```

Simply adding the <headers> element with no child elements will automatically add Cache Control and quite a few other protections. However, if you only want cache control, you can enable this feature using Spring Security's XML namespace with the <cache-control> element and the headers@defaults-disabled attribute.

```
<http>
	<!-- ... -->

	<headers defaults-disable="true">
		<cache-control />
	</headers>
</http>
```

Similarly, you can enable only cache control within Java Configuration with the following:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
	http
	// ...
	.headers()
		.defaultsDisabled()
		.cacheControl();
}
}
```

If you actually want to cache specific responses, your application can selectively invoke HttpServletResponse.setHeader(String,String) to override the header set by Spring Security. This is useful to ensure things like CSS, JavaScript, and images are properly cached.

When using Spring Web MVC, this is typically done within your configuration. For example, the following configuration will ensure that the cache headers are set for all of your resources:

```
@EnableWebMvc
public class WebMvcConfiguration implements WebMvcConfigurer {

        @Override
        public void addResourceHandlers(ResourceHandlerRegistry regist
                registry
                        .addResourceHandler("/resources/**")
                        .addResourceLocations("/resources/")
                        .setCachePeriod(31556926);
        }

        // ...
}
```

21.1.2 Content Type Options

Historically browsers, including Internet Explorer, would try to guess the content type of a request using content sniffing. This allowed browsers to improve the user experience by guessing the content type on resources that had not specified the content type. For example, if a browser encountered a JavaScript file that did not have the content type specified, it would be able to guess the content type and then execute it.

> There are many additional things one should do (i.e. only display the document in a distinct domain, ensure Content-Type header is set, sanitize the document, etc) when allowing content to be uploaded. However, these measures are out of the scope of what Spring Security provides. It is also important to point out when disabling content sniffing, you must specify the content type in order for things to work properly.

The problem with content sniffing is that this allowed malicious users to use polyglots (i.e. a file that is valid as multiple content types) to execute XSS attacks. For example, some sites may allow users to submit a valid postscript document to a website and view it. A malicious user might create a postscript document that is also a valid JavaScript file and execute a XSS attack with it.

Content sniffing can be disabled by adding the following header to our response:

```
X-Content-Type-Options: nosniff
```

Just as with the cache control element, the nosniff directive is added by default when using the <headers> element with no child elements. However, if you want more control over which headers are added you can use the <content-type-options> element and the headers@defaults-disabled attribute as shown below:

```
<http>
	<!-- ... -->

	<headers defaults-disabled="true">
		<content-type-options />
	</headers>
</http>
```

The X-Content-Type-Options header is added by default with Spring Security Java configuration. If you want more control over the headers, you can explicitly specify the content type options with the following:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
	http
	// ...
	.headers()
		.defaultsDisabled()
		.contentTypeOptions();
}
}
```

21.1.3 HTTP Strict Transport Security (HSTS)

When you type in your bank's website, do you enter mybank.example.com or do you enter https://mybank.example.com? If you omit the https protocol, you are potentially vulnerable to Man in the Middle attacks. Even if the website performs a redirect to https://mybank.example.com a malicious user could intercept the initial HTTP request and manipulate the response (i.e. redirect to https://mibank.example.com and steal their credentials).

Many users omit the https protocol and this is why HTTP Strict Transport Security (HSTS) was created. Once mybank.example.com is added as a HSTS host, a browser can know ahead of time that any request to mybank.example.com should be interpreted as https://mybank.example.com. This greatly reduces the possibility of a Man in the Middle attack occurring.

In accordance with RFC6797, the HSTS header is only injected into HTTPS responses. In order for the browser to acknowledge the header, the browser must first trust the CA that signed the SSL certificate used to make the connection (not just the SSL certificate).

One way for a site to be marked as a HSTS host is to have the host preloaded into the browser. Another is to add the "Strict-Transport-Security" header to the response. For example the following would instruct the browser to treat the domain as an HSTS host for a year (there are approximately 31536000 seconds in a year):

```
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
```

The optional includeSubDomains directive instructs Spring Security that subdomains (i.e. secure.mybank.example.com) should also be treated as an HSTS domain.

As with the other headers, Spring Security adds HSTS by default. You can customize HSTS headers with the <hsts> element as shown below:

```
<http>
        <!-- ... -->

        <headers>
                <hsts
                        include-subdomains="true"
                        max-age-seconds="31536000" />
        </headers>
</http>
```

Similarly, you can enable only HSTS headers with Java Configuration:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {
```

```
@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .httpStrictTransportSecurity()
                        .includeSubdomains(true)
                        .maxAgeSeconds(31536000);
}
}
```

21.1.4 HTTP Public Key Pinning (HPKP)

HTTP Public Key Pinning (HPKP) is a security feature that tells a web client to associate a specific cryptographic public key with a certain web server to prevent Man in the Middle (MITM) attacks with forged certificates.

To ensure the authenticity of a server's public key used in TLS sessions, this public key is wrapped into a X.509 certificate which is usually signed by a certificate authority (CA). Web clients such as browsers trust a lot of these CAs, which can all create certificates for arbitrary domain names. If an attacker is able to compromise a single CA, they can perform MITM attacks on various TLS connections. HPKP can circumvent this threat for the HTTPS protocol by telling the client which public key belongs to a certain web server. HPKP is a Trust on First Use (TOFU) technique. The first time a web server tells a client via a special HTTP header which public keys belong to it, the client stores this information for a given period of time. When the client visits the server again, it expects a certificate containing a public key whose fingerprint is already known via HPKP. If the server delivers an unknown public key, the client should present a warning to the user.

Because the user-agent needs to validate the pins against the SSL certificate chain, the HPKP header is only injected into HTTPS responses.

Enabling this feature for your site is as simple as returning the Public-Key-Pins HTTP header when your site is accessed over HTTPS. For example, the following would instruct the user-agent to only report pin validation failures to a given URI (via the ***report-uri*** directive) for 2 pins:

```
Public-Key-Pins-Report-Only: max-age=5184000 ; pin-sha256="d6qzRu9zOEC
```

A ***pin validation failure report*** is a standard JSON structure that can be captured either by the web application's own API or by a publicly hosted HPKP reporting service, such as, *REPORT-URI*.

The optional includeSubDomains directive instructs the browser to also validate subdomains with the given pins.

Opposed to the other headers, Spring Security does not add HPKP by default. You can customize HPKP headers with the <hpkp> element as shown below:

```
<http>
        <!-- ... -->

        <headers>
                <hpkp
                        include-subdomains="true"
                        report-uri="https://example.net/pkp-report">
                        <pins>
                                        <pin algorithm="sha256">d6qzRu
                                        <pin algorithm="sha256">E9CZ9I
                        </pins>
                </hpkp>
        </headers>
</http>
```

Similarly, you can enable HPKP headers with Java Configuration:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

                @Override
                protected void configure(HttpSecurity http) throws Exc
                                http
                                // ...
                                .headers()
                                                .httpPublicKeyPinning(
                                                                .inclu
                                                                .repor
                                                                .addSh
                }
}
```

21.1.5 X-Frame-Options

Allowing your website to be added to a frame can be a security issue. For example, using clever CSS styling users could be tricked into clicking on something that they were not intending (video demo). For example, a user that is logged into their bank might click a button that grants access to other users. This sort of attack is known as Clickjacking.

Another modern approach to dealing with clickjacking is to use Section 21.1.7, “Content Security Policy (CSP)”.

There are a number ways to mitigate clickjacking attacks. For example, to protect legacy browsers from clickjacking attacks you can use frame breaking code. While not perfect, the frame breaking code is the best you can do for the legacy browsers.

A more modern approach to address clickjacking is to use X-Frame-Options header:

```
X-Frame-Options: DENY
```

The X-Frame-Options response header instructs the browser to prevent any site with this header in the response from being rendered within a frame. By default, Spring Security disables rendering within an iframe.

You can customize X-Frame-Options with the frame-options element. For example, the following will instruct Spring Security to use "X-Frame-Options: SAMEORIGIN" which allows iframes within the same domain:

```
<http>
        <!-- ... -->

        <headers>
                <frame-options
                policy="SAMEORIGIN" />
        </headers>
</http>
```

Similarly, you can customize frame options to use the same origin within Java Configuration using the following:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .frameOptions()
                        .sameOrigin();
}
}
```

21.1.6 X-XSS-Protection

Some browsers have built in support for filtering out reflected XSS attacks. This is by no means foolproof, but does assist in XSS protection.

The filtering is typically enabled by default, so adding the header typically just ensures it is enabled and instructs the browser what to do when a XSS attack is detected. For example, the filter might try to change the content in the least invasive way to still render everything. At times, this type of replacement can become a XSS vulnerability in itself. Instead, it is best to block the content rather than attempt to fix it. To do this we can add the following header:

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

This header is included by default. However, we can customize it if we wanted. For example:

```
<http>
        <!-- ... -->

        <headers>
                <xss-protection block="false"/>
        </headers>
</http>
```

Similarly, you can customize XSS protection within Java Configuration with the following:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .xssProtection()
                        .block(false);
}
}
```

21.1.7 Content Security Policy (CSP)

Content Security Policy (CSP) is a mechanism that web applications can leverage to mitigate content injection vulnerabilities, such as cross-site scripting (XSS). CSP is a declarative policy that provides a facility for web application authors to declare and ultimately inform the client (user-agent) about the sources from which the web application expects to load resources.

> Content Security Policy is not intended to solve all content injection vulnerabilities. Instead, CSP can be leveraged to help reduce the harm caused by content injection attacks. As a first line of defense, web application authors should validate their input and encode their output.

A web application may employ the use of CSP by including one of the following HTTP headers in the response:

- ***Content-Security-Policy***
- ***Content-Security-Policy-Report-Only***

Each of these headers are used as a mechanism to deliver a ***security policy*** to the client. A security policy contains a set of ***security policy directives*** (for example, *script-src* and *object-src*), each responsible for declaring the restrictions for a particular resource representation.

For example, a web application can declare that it expects to load scripts from specific, trusted sources, by including the following header in the response:

```
Content-Security-Policy: script-src https://trustedscripts.example.com
```

An attempt to load a script from another source other than what is declared in the *script-src* directive will be blocked by the user-agent. Additionally, if the ***report-uri*** directive is declared in the security policy, then the violation will be reported by the user-agent to the declared URL.

For example, if a web application violates the declared security policy, the following response header will instruct the user-agent to send violation reports to the URL specified in the policy's *report-uri* directive.

```
Content-Security-Policy: script-src https://trustedscripts.example.com
```

Violation reports are standard JSON structures that can be captured either by the web application's own API or by a publicly hosted CSP violation reporting service, such as, ***REPORT-URI***.

The ***Content-Security-Policy-Report-Only*** header provides the capability for web application authors and administrators to monitor security policies, rather than enforce them. This header is typically used when experimenting and/or developing security policies for a site. When a policy is deemed effective, it can be enforced by using the *Content-Security-Policy* header field instead.

Given the following response header, the policy declares that scripts may be loaded from one of two possible sources.

```
Content-Security-Policy-Report-Only: script-src 'self' https://trusted
```

If the site violates this policy, by attempting to load a script from *evil.com*, the user-agent will send a violation report to the declared URL specified by the *report-uri* directive, but still allow the violating resource to load nevertheless.

Configuring Content Security Policy

It's important to note that Spring Security ***does not add*** Content Security Policy by default. The web application author must declare the security policy(s) to enforce and/or monitor for the protected resources.

For example, given the following security policy:

```
script-src 'self' https://trustedscripts.example.com; object-src https
```

You can enable the CSP header using XML configuration with the <content-security-policy> element as shown below:

```
<http>
        <!-- ... -->

        <headers>
                <content-security-policy
                        policy-directives="script-src 'self' https://t
        </headers>
</http>
```

To enable the CSP *'report-only'* header, configure the element as follows:

```
<http>
        <!-- ... -->
        <headers>
                <content-security-policy
                        policy-directives="script-src 'self' https://t
                        report-only="true" />
        </headers>
</http>
```

Similarly, you can enable the CSP header using Java configuration as shown below:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .contentSecurityPolicy("script-src 'self' https://trus
}
}
```

To enable the CSP *'report-only'* header, provide the following Java configuration:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .contentSecurityPolicy("script-src 'self' https://trus
                .reportOnly();
}
}
```

Additional Resources

Applying Content Security Policy to a web application is often a non-trivial undertaking. The following resources may provide further assistance in developing effective security policies for your site.

An Introduction to Content Security Policy

CSP Guide - Mozilla Developer Network

W3C Candidate Recommendation

21.1.8 Referrer Policy

Referrer Policy is a mechanism that web applications can leverage to manage the referrer field, which contains the last page the user was on.

Spring Security's approach is to use Referrer Policy header, which provides different policies:

```
Referrer-Policy: same-origin
```

The Referrer-Policy response header instructs the browser to let the destination knows the source where the user was previously.

Configuring Referrer Policy

Spring Security ***doesn't add*** Referrer Policy header by default.

You can enable the Referrer-Policy header using XML configuration with the <referrer-policy> element as shown below:

```
<http>
        <!-- ... -->

        <headers>
                <referrer-policy policy="same-origin" />
        </headers>
</http>
```

Similarly, you can enable the Referrer Policy header using Java configuration as shown below:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .referrerPolicy(ReferrerPolicy.SAME_ORIGIN);
}
}
```

21.2 Custom Headers

Spring Security has mechanisms to make it convenient to add the more common security headers to your application. However, it also provides hooks to enable adding custom headers.

21.2.1 Static Headers

There may be times you wish to inject custom security headers into your application that are not supported out of the box. For example, given the following custom security header:

```
X-Custom-Security-Header: header-value
```

When using the XML namespace, these headers can be added to the response using the <header> element as shown below:

```
<http>
        <!-- ... -->

        <headers>
                <header name="X-Custom-Security-Header" value="header-
        </headers>
</http>
```

Similarly, the headers could be added to the response using Java Configuration as shown in the following:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .addHeaderWriter(new StaticHeadersWriter("X-Custom-Sec
}
}
```

21.2.2 Headers Writer

When the namespace or Java configuration does not support the headers you want, you can create a custom `HeadersWriter` instance or even provide a custom implementation of the `HeadersWriter`.

Let’s take a look at an example of using an custom instance of `XFrameOptionsHeaderWriter`. Perhaps you want to allow framing of content for the same origin. This is easily supported by setting the policy attribute to "SAMEORIGIN", but let’s take a look at a more explicit example using the ref attribute.

```
<http>
        <!-- ... -->

        <headers>
                <header ref="frameOptionsWriter"/>
        </headers>
</http>
<!-- Requires the c-namespace.
See https://docs.spring.io/spring/docs/current/spring-framework-refere
-->
<beans:bean id="frameOptionsWriter"
        class="org.springframework.security.web.header.writers.frameop
        c:frameOptionsMode="SAMEORIGIN"/>
```

We could also restrict framing of content to the same origin with Java configuration:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        // ...
        .headers()
                .addHeaderWriter(new XFrameOptionsHeaderWriter(XFrameC
}
}
```

21.2.3 DelegatingRequestMatcherHeaderWriter

At times you may want to only write a header for certain requests. For example, perhaps you want to only protect your log in page from being framed. You could use the `DelegatingRequestMatcherHeaderWriter` to do so. When using the XML namespace configuration, this can be done with the following:

```
<http>
```

```
        <!-- ... -->

        <headers>
                <frame-options disabled="true"/>
                <header ref="headerWriter"/>
        </headers>
</http>

<beans:bean id="headerWriter"
        class="org.springframework.security.web.header.writers.Delegat
        <beans:constructor-arg>
                <bean class="org.springframework.security.web.util.mat
                        c:pattern="/login"/>
        </beans:constructor-arg>
        <beans:constructor-arg>
                <beans:bean
                        class="org.springframework.security.web.header
        </beans:constructor-arg>
</beans:bean>
```

We could also prevent framing of content to the log in page using java configuration:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        RequestMatcher matcher = new AntPathRequestMatcher("/login");
        DelegatingRequestMatcherHeaderWriter headerWriter =
                new DelegatingRequestMatcherHeaderWriter(matcher,new X
        http
        // ...
        .headers()
                .frameOptions().disabled()
                .addHeaderWriter(headerWriter);
}
}
```