# 19. Cross Site Request Forgery (CSRF)

This section discusses Spring Security's Cross Site Request Forgery (CSRF) support.

## 19.1 CSRF Attacks

Before we discuss how Spring Security can protect applications from CSRF attacks, we will explain what a CSRF attack is. Let's take a look at a concrete example to get a better understanding.

Assume that your bank's website provides a form that allows transferring money from the currently logged in user to another bank account. For example, the HTTP request might look like:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

Now pretend you authenticate to your bank's website and then, without logging out, visit an evil website. The evil website contains an HTML page with the following form:

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
        name="amount"
        value="100.00"/>
<input type="hidden"
        name="routingNumber"
        value="evilsRoutingNumber"/>
<input type="hidden"
        name="account"
        value="evilsAccountNumber"/>
<input type="submit"
        value="Win Money!"/>
</form>
```

You like to win money, so you click on the submit button. In the process, you have unintentionally transferred $100 to a malicious user. This happens because, while the evil website cannot see your cookies, the cookies associated with your bank are still sent along with the request.

Worst yet, this whole process could have been automated using JavaScript. This means you didn't even need to click on the button. So how do we protect ourselves from such attacks?

## 19.2 Synchronizer Token Pattern

The issue is that the HTTP request from the bank's website and the request from the evil website are exactly the same. This means there is no way to reject requests coming from the evil website and allow requests coming from the bank's website. To protect against CSRF attacks we need to ensure there is something in the request that the evil site is unable to provide.

One solution is to use the Synchronizer Token Pattern. This solution is to ensure that each request requires, in addition to our session cookie, a randomly generated token as an HTTP parameter. When a request is submitted, the server must look up the expected value for the parameter and compare it against the actual value in the request. If the values do not match, the request should fail.

We can relax the expectations to only require the token for each HTTP request that updates state. This can be safely done since the same origin policy ensures the evil site cannot read the response. Additionally, we do not want to include the random token in HTTP GET as this can cause the tokens to be leaked.

Let's take a look at how our example would change. Assume the randomly generated token is present in an HTTP parameter named _csrf. For example, the request to transfer money would look like this:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&_csrf=<secure-random>
```

You will notice that we added the _csrf parameter with a random value. Now the evil website will not be able to guess the correct value for the _csrf parameter (which must be explicitly provided on the evil website) and the transfer will fail when the server compares the actual token to the expected token.

## 19.3 When to use CSRF protection

When should you use CSRF protection? Our recommendation is to use CSRF protection for any request that could be processed by a browser by normal users. If you are only creating a service that is used by non-browser clients, you will likely want to disable CSRF protection.

### 19.3.1 CSRF protection and JSON

A common question is "do I need to protect JSON requests made by javascript?" The short answer is, it depends. However, you must be very careful as there are CSRF exploits that can impact JSON requests. For example, a malicious user can create a CSRF with JSON using the following form:

```
<form action="https://bank.example.com/transfer" method="post" enctype="text/plai
<input name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsA
<input type="submit"
        value="Win Money!"/>
</form>
```

This will produce the following JSON structure

```
{ "amount": 100,
"routingNumber": "evilsRoutingNumber",
"account": "evilsAccountNumber",
"ignore_me": "=test"
}
```

If an application were not validating the Content-Type, then it would be exposed to this exploit. Depending on the setup, a Spring MVC application that validates the Content-Type could still be exploited by updating the URL suffix to end with ".json" as shown below:

```
<form action="https://bank.example.com/transfer.json" method="post" enctype="text
<input name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsA
<input type="submit"
        value="Win Money!"/>
</form>
```

### 19.3.2 CSRF and Stateless Browser Applications

What if my application is stateless? That doesn't necessarily mean you are protected. In fact, if a user does not need to perform any actions in the web browser for a given request, they are likely still vulnerable to CSRF attacks.

For example, consider an application uses a custom cookie that contains all the state within it for authentication instead of the JSESSIONID. When the CSRF attack is made the custom cookie will be sent with the request in the same manner that the JSESSIONID cookie was sent in our previous example.

Users using basic authentication are also vulnerable to CSRF attacks since the browser will automatically include the username password in any requests in the same manner that the JSESSIONID cookie was sent in our previous example.

## 19.4 Using Spring Security CSRF Protection

So what are the steps necessary to use Spring Security's to protect our site against CSRF attacks? The steps to using Spring Security's CSRF protection are outlined below:

- Use proper HTTP verbs
- Configure CSRF Protection
- Include the CSRF Token

### 19.4.1 Use proper HTTP verbs

The first step to protecting against CSRF attacks is to ensure your website uses proper HTTP verbs. Specifically, before Spring Security's CSRF support can be of use, you need to be certain that your application is using PATCH, POST, PUT, and/or DELETE for anything that modifies state.

This is not a limitation of Spring Security's support, but instead a general requirement for proper CSRF prevention. The reason is that including private information in an HTTP GET can cause the information to be leaked. See RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's for general guidance on using POST instead of GET for sensitive information.

### 19.4.2 Configure CSRF Protection

The next step is to include Spring Security's CSRF protection within your application. Some frameworks handle invalid CSRF tokens by invaliding the user's session, but this causes its own problems. Instead by default Spring Security's CSRF protection will produce an HTTP 403 access denied. This can be customized by configuring the AccessDeniedHandler to process `InvalidCsrfTokenException` differently.

As of Spring Security 4.0, CSRF protection is enabled by default with XML configuration. If you would like to disable CSRF protection, the corresponding XML configuration can be seen below.

```
<http>
        <!-- ... -->
        <csrf disabled="true"/>
</http>
```

CSRF protection is enabled by default with Java Configuration. If you would like to disable CSRF, the corresponding Java configuration can be seen below. Refer to the Javadoc of csrf() for additional customizations in how CSRF protection is configured.

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

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

## 19.4.3 Include the CSRF Token

### Form Submissions

The last step is to ensure that you include the CSRF token in all PATCH, POST, PUT, and DELETE methods. One way to approach this is to use the `_csrf` request attribute to obtain the current `CsrfToken`. An example of doing this with a JSP is shown below:

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
        method="post">
<input type="submit"
        value="Log out" />
<input type="hidden"
        name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
</form>
```

An easier approach is to use the csrfInput tag from the Spring Security JSP tag library.

If you are using Spring MVC `<form:form>` tag or Thymeleaf 2.1+ and are using

> `@EnableWebSecurity`, the `CsrfToken` is automatically included for you (using the `CsrfRequestDataValueProcessor`).

### Ajax and JSON Requests

If you are using JSON, then it is not possible to submit the CSRF token within an HTTP parameter. Instead you can submit the token within a HTTP header. A typical pattern would be to include the CSRF token within your meta tags. An example with a JSP is shown below:

```
<html>
<head>
        <meta name="_csrf" content="${_csrf.token}"/>
        <!-- default header name is X-CSRF-TOKEN -->
        <meta name="_csrf_header" content="${_csrf.headerName}"/>
        <!-- ... -->
</head>
<!-- ... -->
```

Instead of manually creating the meta tags, you can use the simpler csrfMetaTags tag from the Spring Security JSP tag library.

You can then include the token within all your Ajax requests. If you were using jQuery, this could be done with the following:

```
$(function () {
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(e, xhr, options) {
        xhr.setRequestHeader(header, token);
});
});
```

As an alternative to jQuery, we recommend using cujoJS's rest.js. The rest.js module provides advanced support for working with HTTP requests and responses in RESTful ways. A core capability is the ability to contextualize the HTTP client adding behavior as needed by chaining interceptors on to the client.

```
var client = rest.chain(csrf, {
token: $("meta[name='_csrf']").attr("content"),
name: $("meta[name='_csrf_header']").attr("content")
});
```

The configured client can be shared with any component of the application that needs to make a request to the CSRF protected resource. One significant different between rest.js and jQuery is that only requests made with the configured client will contain the CSRF token, vs jQuery where *all* requests will include the token. The ability to scope which requests receive the token helps guard against leaking the CSRF token to a third party. Please refer to the rest.js reference documentation for more information on rest.js.

### CookieCsrfTokenRepository

There can be cases where users will want to persist the `CsrfToken` in a cookie. By default the `CookieCsrfTokenRepository` will write to a cookie named `XSRF-TOKEN` and read it from a header named `X-XSRF-TOKEN` or the HTTP parameter `_csrf`. These defaults come from AngularJS

You can configure `CookieCsrfTokenRepository` in XML using the following:

```
<http>
        <!-- ... -->
        <csrf token-repository-ref="tokenRepository"/>
</http>
<b:bean id="tokenRepository"
        class="org.springframework.security.web.csrf.CookieCsrfTokenRepository"
        p:cookieHttpOnly="false"/>
```

The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` to improve security.

You can configure `CookieCsrfTokenRepository` in Java Configuration using:

```
@EnableWebSecurity
public class WebSecurityConfig extends
                WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .csrf()
                                .csrfTokenRepository(CookieCsrfTokenRepository.wi
        }
}
```

The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` (by using `new CookieCsrfTokenRepository()` instead) to improve security.

## 19.5 CSRF Caveats

There are a few caveats when implementing CSRF.

### 19.5.1 Timeouts

One issue is that the expected CSRF token is stored in the HttpSession, so as soon as the HttpSession expires your configured `AccessDeniedHandler` will receive a InvalidCsrfTokenException. If you are using the default `AccessDeniedHandler`, the browser will get an HTTP 403 and display a poor error message.

One might ask why the expected `CsrfToken` isn't stored in a cookie by default. This is because there are known exploits in which headers (i.e. specify the cookies) can be set by another domain. This is the same reason Ruby on Rails no longer skips CSRF checks when the header X-Requested-With is present. See this webappsec.org thread for details on how to perform the exploit. Another disadvantage is that by removing the state (i.e. the timeout) you lose the ability to forcibly terminate the token if it is compromised.

A simple way to mitigate an active user experiencing a timeout is to have some JavaScript that lets the user know their session is about to expire. The user can click a button to continue and refresh the session.

Alternatively, specifying a custom `AccessDeniedHandler` allows you to process the `InvalidCsrfTokenException` any way you like. For an example of how to customize the `AccessDeniedHandler` refer to the provided links for both xml and Java configuration.

Finally, the application can be configured to use CookieCsrfTokenRepository which will not expire. As previously mentioned, this is not as secure as using a session, but in many cases can be good enough.

### 19.5.2 Logging In

In order to protect against forging log in requests the log in form should be protected against CSRF attacks too. Since the `CsrfToken` is stored in HttpSession, this means an HttpSession will be created as soon as `CsrfToken` token attribute is accessed. While this sounds bad in a RESTful / stateless architecture the reality is that state is necessary to implement practical security. Without state, we have nothing we can do if a token is compromised. Practically speaking, the CSRF token is quite small in size and should have a negligible impact on our architecture.

A common technique to protect the log in form is by using a JavaScript function to obtain a valid CSRF token before the form submission. By doing this, there is no need to think about session timeouts (discussed in the previous section) because the session is created right before the form submission (assuming that CookieCsrfTokenRepository isn't configured instead), so the user can stay on the login page and submit the username/password when he wants. In order to achieve this, you can take advantage of the `CsrfTokenArgumentResolver` provided by Spring Security and expose an endpoint like it's described on here.

### 19.5.3 Logging Out

Adding CSRF will update the LogoutFilter to only use HTTP POST. This ensures that log out requires a CSRF token and that a malicious user cannot forcibly log out your users.

One approach is to use a form for log out. If you really want a link, you can use JavaScript to have the link perform a POST (i.e. maybe on a hidden form). For browsers with JavaScript that is disabled, you can optionally have the link take the user to a log out confirmation page that will perform the POST.

If you really want to use HTTP GET with logout you can do so, but remember this is generally not recommended. For example, the following Java Configuration will perform logout with the URL /logout is requested with any HTTP method:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .logout()
                                .logoutRequestMatcher(new AntPathRequestMatcher("
        }
}
```

### 19.5.4 Multipart (file upload)

There are two options to using CSRF protection with multipart/form-data. Each option has its tradeoffs.

- Placing MultipartFilter before Spring Security
- Include CSRF token in action

Before you integrate Spring Security's CSRF protection with multipart file upload, ensure that you can upload without the CSRF protection first. More information about using multipart forms with Spring can be found within the 17.10 Spring's multipart (file upload) support section of the Spring reference and the MultipartFilter javadoc.

#### Placing MultipartFilter before Spring Security

The first option is to ensure that the `MultipartFilter` is specified before the Spring Security filter. Specifying the `MultipartFilter` before the Spring Security filter means that there is no authorization for invoking the `MultipartFilter` which means anyone can place temporary files on your server. However, only authorized users will be able to submit a File that is processed by your application. In general, this is the recommended approach because the temporary file upload should have a negligble impact on most servers.

To ensure `MultipartFilter` is specified before the Spring Security filter with java configuration, users can override beforeSpringSecurityFilterChain as shown below:

```
public class SecurityApplicationInitializer extends AbstractSecurityWebApplicatio

        @Override
        protected void beforeSpringSecurityFilterChain(ServletContext servletCont
                insertFilters(servletContext, new MultipartFilter());
        }
}
```

To ensure `MultipartFilter` is specified before the Spring Security filter with XML configuration, users can ensure the <filter-mapping> element of the `MultipartFilter` is placed before the springSecurityFilterChain within the web.xml as shown below:

```
<filter>
        <filter-name>MultipartFilter</filter-name>
        <filter-class>org.springframework.web.multipart.support.MultipartFilter</
</filter>
<filter>
```

```
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filte
</filter>
<filter-mapping>
        <filter-name>MultipartFilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

### Include CSRF token in action

If allowing unauthorized users to upload temporariy files is not acceptable, an alternative is to place the `MultipartFilter` after the Spring Security filter and include the CSRF as a query parameter in the action attribute of the form. An example with a jsp is shown below

```
<form action="./upload?${_csrf.parameterName}=${_csrf.token}" method="post" encty
```

The disadvantage to this approach is that query parameters can be leaked. More genearlly, it is considered best practice to place sensitive data within the body or headers to ensure it is not leaked. Additional information can be found in RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's.

## 19.5.5 HiddenHttpMethodFilter

The HiddenHttpMethodFilter should be placed before the Spring Security filter. In general this is true, but it could have additional implications when protecting against CSRF attacks.

Note that the HiddenHttpMethodFilter only overrides the HTTP method on a POST, so this is actually unlikely to cause any real problems. However, it is still best practice to ensure it is placed before Spring Security's filters.

# 19.6 Overriding Defaults

Spring Security's goal is to provide defaults that protect your users from exploits. This does not mean that you are forced to accept all of its defaults.

For example, you can provide a custom CsrfTokenRepository to override the way in which the `CsrfToken` is stored.

You can also specify a custom RequestMatcher to determine which requests are protected by CSRF (i.e. perhaps you don't care if log out is exploited). In short, if Spring Security's CSRF protection doesn't behave exactly as you want it, you are able to customize the behavior. Refer to the Section 43.1.18, "<csrf>" documentation for details on how to make these customizations with XML and the `CsrfConfigurer` javadoc for details on how to make these customizations when using Java configuration.


| Prev | Up | Next |
| --- | --- | --- |
| 18. Remember-Me Authentication | Home | 20. CORS |