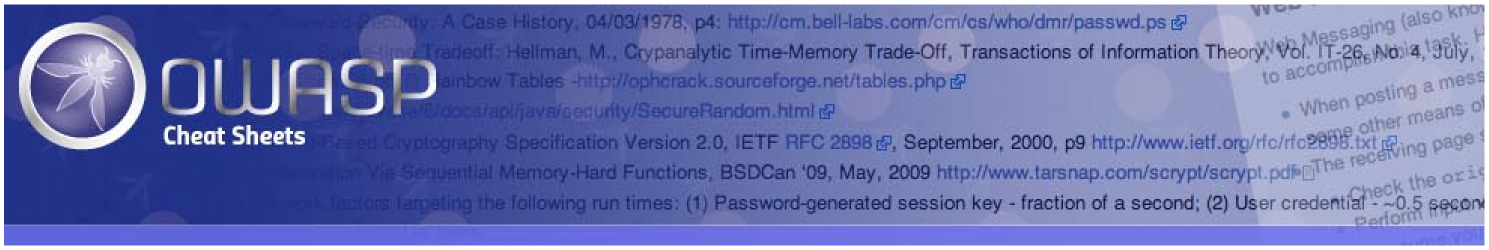
Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

From OWASP



Last revision (mm/dd/yy): **02/2/2018**

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site for which the user is currently authenticated. The impact of a successful CSRF attack is limited to the capabilities exposed by the vulnerable application. For example, this attack could result in a transfer of funds, changing a password, or purchasing an item in the user's context. In effect, CSRF attacks are used by an attacker to make a target system perform a function via the target's browser without knowledge of the target user, at least until the unauthorized transaction has been committed.

Impacts of successful CSRF exploits vary greatly based on the privileges of each victim. When targeting a normal user, a successful CSRF attack can compromise end-user data and their associated functions. If the targeted end user is an administrator account, a CSRF attack can compromise the entire web application. Sites that are more likely to be attacked by CSRF are community websites (social networking, email) or sites that have high dollar value accounts associated with them (banks, stock brokerages, bill pay services). Utilizing social engineering, an attacker can embed malicious HTML or JavaScript code into an email or website to request a specific 'task URL'. The task then executes with or without the user's knowledge, either directly or by utilizing a Cross-Site Scripting flaw (ex: Samy MySpace Worm).

For more information on CSRF, please see the OWASP Cross-Site Request Forgery (CSRF) page.

Warning: No Cross-Site Scripting (XSS) Vulnerabilities

Cross-Site Scripting is not necessary for CSRF to work. However, any cross-site scripting vulnerability can be used to defeat token, Double-Submit cookie, referer and origin based CSRF defenses. This is because an XSS payload can simply read any page on the site using a XMLHttpRequest and obtain the generated token from the response, and include that token with a forged request. This technique is exactly how the MySpace (Samy) worm (http://en.wikipedia.org/wiki/Samy_(XSS)) defeated MySpace's anti-CSRF defenses in 2005, which enabled the worm to propagate. XSS cannot defeat challenge-response defenses such as Captcha, re-authentication or one-time passwords. It is imperative that no XSS vulnerabilities are present to ensure that CSRF defenses can't be circumvented. Please see the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws.

General Recommendations For Automated CSRF Defense

We recommend two separate checks as your standard CSRF defense that does not require user intervention. This discussion ignores for the moment deliberately allowed cross origin requests (e.g., CORS). Your defenses will have to adjust for that if that is allowed.

1. Check standard headers to verify the request is same origin
2. AND Check CSRF token

Each of these is discussed next.

Verifying Same Origin with Standard Headers

There are two steps to this check:

1. Determining the origin the request is coming from (source origin)
2. Determining the origin the request is going to (target origin)

Both of these steps rely on examining an HTTP request header value. Although it is usually trivial to spoof any header from a browser using JavaScript, it is generally impossible to do so in the victim's browser during a CSRF attack, except via an XSS vulnerability in the site being attacked with CSRF. More importantly for this recommended Same Origin check, a number of HTTP request headers can't be set by JavaScript because they are on the 'forbidden' headers list (https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name). Only the browsers themselves can set values for these headers, making them more trustworthy because not even an XSS vulnerability can be used to modify them.

The Source Origin check recommended here relies on three of these protected headers: Origin, Referer, and Host, making it a pretty strong CSRF defense all on its own.

Identifying Source Origin

To identify the source origin, we recommend using one of these two standard headers that almost all requests include one or both of:

- Origin Header
- Referer Header

Checking the Origin Header

If the Origin header is present, verify its value matches the target origin. The Origin HTTP Header (https://wiki.mozilla.org/Security/Origin) standard was introduced as a method of defending against CSRF and other Cross-Domain attacks. Unlike the Referer, the Origin header will be present in HTTP requests that originate from an HTTPS URL. If the Origin header is present, then it should be checked to make sure it matches the target origin.

This defense technique is specifically proposed in section 5.0 of Robust Defenses for Cross-Site Request Forgery (https://seclab.stanford.edu/websec/csrf/csrf.pdf). This paper proposes the creation of the Origin header and its use as a CSRF defense mechanism.

There are some situations where the Origin header is not present.

- Internet Explorer 11 does not add the Origin header on a CORS request (http://stackoverflow.com/questions/20784209/internet-explorer-11-does-not-add-the-origin-header-on-a-cors-request) across sites of a trusted zone. The Referer header will remain the only indication of the UI origin.
- Following a 302 redirect cross-origin (http://stackoverflow.com/questions/22397072/are-there-any-browsers-that-set-the-origin-header-to-null-for-privacy-sensitiv). In this situation, the Origin is not included in the redirected request because that may be considered sensitive information you don't want to send to the other origin. But since we recommend rejecting requests that don't have both Origin and Referer headers, this is OK, because the reason the Origin header isn't there is because it is a cross-origin redirect.

Checking the Referer Header

If the Origin header is not present, verify the hostname in the Referer header matches the target origin. Checking the Referer is a commonly used method of preventing CSRF on embedded network devices because it does not require any per-user state. This makes Referer a useful method of CSRF prevention when memory is scarce or server-side state doesn't exist. This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state which is required to keep track of a synchronization token.

In both cases, just make sure the target origin check is strong. For example, if your site is "site.com" make sure "site.com.attacker.com" doesn't pass your origin check (i.e., match through the trailing / after the origin to make sure you are matching against the entire origin).

What to do when Both Origin and Referer Headers Aren't Present

If neither of these headers is present, which should be VERY rare, you can either accept or block the request. **We recommend blocking**, particularly if you aren't using a random CSRF token as your second check. You might want to log when this happens for a while and if you basically never see it, start blocking such requests.

Identifying the Target Origin

You might think its easy to determine the target origin, but its frequently not. The first thought is to simply grab the target origin (i.e., its hostname and port #) from the URL in the request. However, the application server is frequently sitting behind one or more proxies and the original URL is different from the URL the app server actually receives. If your application server is directly accessed by its users, then using the origin in the URL is fine and you're all set.

Determining the Target Origin When Behind a Proxy

If you are behind a proxy, there are a number of options to consider:

1. Configure your application to simply know its target origin
2. Use the Host header value
3. Use the X-Forwarded-Host header value

Its your application, so clearly you can figure out its target origin and set that value in some server configuration entry. This would be the most secure approach as its defined server side so is a trusted value. However, this can be problematic to maintain if your application is deployed in many different places, e.g., dev, test, QA, production, and possibly multiple production instances. Setting the correct value for each of these situations can be difficult, but if you can do it, that's great.

If you would prefer the application figure it out on its own, so it doesn't have to be configured differently for each deployed instance, we recommend using the Host family of headers. The Host header's purpose is to contain the target origin of the request. But, if your app server is sitting behind a proxy, the Host header value is most likely changed by the proxy to the target origin of the URL behind the proxy, which is different than the original URL. This modified Host header origin won't match the source origin in the original Origin or Referer headers.

However, there is another header called X-Forwarded-Host, whose purpose is to contain the original Host header value the proxy received. Most proxies will pass along the original Host header value in the X-Forwarded-Host header. So that header value is likely to be the target origin value you need to compare to the source origin in the Origin or Referer header.

Verifying the Two Origins Match

Once you've identified the source origin (from either the Origin or Referer header), and you've determined the target origin, however you choose to do so, then you can simply compare the two values and if they don't match you know you have a cross-origin request.

CSRF Specific Defense

Once you have verified that the request appears to be a same origin request so far, we recommend a second check as an additional precaution to really make sure. This second check can involve custom defense mechanisms using CSRF specific tokens created and verified by your application or can rely on the presence of other HTTP headers depending on the level of rigor/security you want.

There are numerous ways you can specifically defend against CSRF. We recommend using one of the following (in ADDITION to the check recommended above):

1. Synchronizer (i.e.,CSRF) Tokens (requires session state)

Approaches that do require no server side state:

2. Double Cookie Defense
3. Encrypted Token Pattern
4. Custom Header - e.g., X-Requested-With: XMLHttpRequest

These are listed in order of strength of defense. So use the strongest defense that makes sense in your situation.

Synchronizer (CSRF) Tokens

- Any state changing operation requires a secure random token (e.g., CSRF token) to prevent CSRF attacks
- Characteristics of a CSRF Token
  - Unique per user session

    - Large random value
    - Generated by a cryptographically secure random number generator
- The CSRF token is added as a hidden field for forms or within the URL if the state changing operation occurs via a GET
- The server rejects the requested action if the CSRF token fails validation

In order to facilitate a "transparent but visible" CSRF solution, developers are encouraged to adopt the Synchronizer Token Pattern (http://www.corej2eepatterns.com/Design/PresoDesign.htm). The synchronizer token pattern requires the generating of random "challenge" tokens that are associated with the user's current session. These challenge tokens are then inserted within the HTML forms and links associated with sensitive server-side operations. When the user wishes to invoke these sensitive operations, the HTTP request should include this challenge token. It is then the responsibility of the server application to verify the existence and correctness of this token. By including a challenge token with each request, the developer has a strong control to verify that the user actually intended to submit the desired requests. Inclusion of a required security token in HTTP requests associated with sensitive business functions helps mitigate CSRF attacks as successful exploitation assumes the attacker knows the randomly generated token for the target victim's session. This is analogous to the attacker being able to guess the target victim's session identifier. The following synopsis describes a general approach to incorporate challenge tokens within the request.

When a Web application formulates a request (by generating a link or form that causes a request when submitted or clicked by the user), the application should include a hidden input parameter with a common name such as "CSRFToken". The value of this token must be randomly generated such that it cannot be guessed by an attacker. Consider leveraging the java.security.SecureRandom class for Java applications to generate a sufficiently long random token. Alternative generation algorithms include the use of 256-bit BASE64 encoded hashes. Developers that choose this generation algorithm must make sure that there is randomness and uniqueness utilized in the data that is hashed to generate the random token.

```
<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWE...
wYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ...
MGYwMGEwOA==">
…
</form>
```

In general, developers need only generate this token once for the current session. After initial generation of this token, the value is stored in the session and is utilized for each subsequent request until the session expires. When a request is issued by the end-user, the server-side component must verify the existence and validity of the token in the request as compared to the token found in the session. If the token was not found within the request or the value provided does not match the value within the session, then the request should be aborted, token should be reset and the event logged as a potential CSRF attack in progress.

To further enhance the security of this proposed design, consider randomizing the CSRF token parameter name and/or value for each request. Implementing this approach results in the generation of per-request tokens as opposed to per-session tokens. Note, however, that this may result in usability concerns. For example, the "Back" button browser capability is often hindered as the previous page may contain a token that is no longer valid. Interaction with this previous page will result in a CSRF false positive security event at the server. Regardless of the approach taken, developers are encouraged to protect the CSRF token the same way they protect authenticated session identifiers, such as the use of TLS.

Existing Synchronizer Implementations

Synchronizer Token defenses have been built into many frameworks so we strongly recommend using them first when they are available. External components that add CSRF defenses to existing applications also exist and are recommended. OWASP has two:

1. For Java: OWASP CSRF Guard
2. For PHP and Apache: CSRFProtector Project

Disclosure of Token in URL

Many implementations of synchronizer tokens include the challenge token in GET (URL) requests as well as POST requests. This is often implemented as a result of sensitive server-side operations being invoked as a result of embedded links in the page or other general design patterns. These patterns are often implemented without knowledge of CSRF and an understanding of CSRF prevention design strategies. While this control does help mitigate the risk of CSRF attacks, the unique per-session token is being exposed for GET requests. CSRF tokens in GET requests are potentially leaked at several locations: browser history, HTTP log files, network appliances that make a point to log the first line of an HTTP request, and Referer headers if the protected site links to an external site.

In the latter case (leaked CSRF token due to the Referer header being parsed by a linked site), it is trivially easy for the linked site to launch a CSRF attack on the protected site, and they will be able to target this attack very effectively, since the Referer header tells them the site as well as the CSRF token. The attack could be run entirely from JavaScript, so that a simple addition of a script tag to the HTML of a site can launch an attack (whether on an originally malicious site or on a hacked site). Additionally, since HTTPS requests from HTTPS contexts will not strip the Referer header (as opposed to HTTPS to HTTP requests) CSRF token leaks via Referer can still happen on HTTPS Applications.

The ideal solution is to only include the CSRF token in POST requests and modify server-side actions that have state changing affect to only respond to POST requests. This is in fact what the RFC 2616 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1) requires for GET requests. If sensitive server-side actions are guaranteed to only ever respond to POST requests, then there is no need to include the token in GET requests.

In most JavaEE web applications, however, HTTP method scoping is rarely ever utilized when retrieving HTTP parameters from a request. Calls to "HttpServletRequest.getParameter" will return a parameter value regardless if it was a GET or POST. This is not to say HTTP method scoping cannot be enforced. It can be achieved if a developer explicitly overrides doPost() in the HttpServlet class or leverages framework specific capabilities such as the AbstractFormController class in Spring.

For these cases, attempting to retrofit this pattern in existing applications requires significant development time and cost, and as a temporary measure it may be better to pass CSRF tokens in the URL. Once the application has been fixed to respond to HTTP GET and POST verbs correctly, CSRF tokens for GET requests should be turned off.

ASP.NET, MVC and .NET Web Pages

.NET Web Forms provides comprehensive CSRF defense using ViewState. For more information visit the following resources.

- https://msdn.microsoft.com/en-us/library/ms972969.aspx#securitybarriers_topic2
- http://software-security.sans.org/developer-how-to/developer-guide-csrf

ASP.NET MVC also provides mechanisms for providing CSRF defenses: See:

- https://github.com/aspnet/Antiforgery (code) - http://www.dotnetcurry.com/aspnet/1343/aspnet-core-csrf-antiforgery-token (how to use)
- https://docs.microsoft.com/en-us/aspnet/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages
- https://docs.microsoft.com/en-us/aspnet/web-api/overview/security/preventing-cross-site-request-forgery-csrf-attacks

Double Submit Cookie

If storing the CSRF token in session is problematic, an alternative defense is use of a double submit cookie. A double submit cookie is defined as sending a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value match.

When a user authenticates to a site, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session id. The site does not have to save this value in any way, thus avoiding server side state. The site then requires that every transaction request include this random value as a hidden form value (or other request parameter). A cross origin attacker cannot read any data sent from the server or modify cookie values, per the same-origin policy. This means that while an attacker can force a victim to send any value he wants with a malicious CSRF request, the attacker will be unable to modify or read the value stored in the cookie. Since the cookie value and the request parameter or form value must be the same, the attacker will be unable to successfully force the submission of a request with the random CSRF value.

As an example, the Direct Web Remoting (DWR) (http://directwebremoting.org) Java library version 2.0 has CSRF protection built in that implements the double cookie submission transparently.

When the UI and the function service reside in different hosts, the Double Submit Cookie guard turns difficult to implement because the UI body and the Set-Cookie response header will be generated as part of different requests to different processes. The Set-Cookie response header would need to be induced by a request from the client javascript. In that case, making sure that both the UI and the service request came from a client serviced by the same UI origin appears as difficult as the original issue.

Encrypted Token Pattern

The Encrypted Token Pattern leverages an encryption, rather than comparison, method of Token-validation. After successful authentication, the server generates a unique Token comprised of the user's ID, a timestamp value and a nonce (http://en.wikipedia.org/wiki/Cryptographic_nonce), using a unique key available only on the server. This Token is returned to the client and embedded in a hidden field. Subsequent AJAX requests include this Token in the request-header, in a similar manner to the Double-Submit pattern. Non-AJAX form-based requests will implicitly persist the Token in its hidden field. On receipt of this request, the server reads and decrypts the Token value with the same key used to create the Token. Inability to correctly decrypt suggest an intrusion attempt. Once decrypted, the UserId and timestamp contained within the token are validated to ensure validity; the UserId is compared against the currently logged in user, and the timestamp is compared against the current time.

On successful Token-decryption, the server has access to parsed values, ideally in the form of claims (http://en.wikipedia.org/wiki/Claims-based_identity). These claims are processed by comparing the UserId claim to any potentially stored UserId (in a Cookie or Session variable, if the site already contains a means of authentication). The Timestamp is validated against the current time, preventing replay attacks. Alternatively, in the case of a CSRF attack, the server will be unable to decrypt the poisoned Token, and can block and log the attack.

This pattern exists primarily to allow developers and architects protect against CSRF without session-dependency. It also addresses some of the shortfalls in other stateless approaches, such as the need to store data in a Cookie, circumnavigating the Cookie-subdomain and HTTPONLY issues.

Protecting REST Services: Use of Custom Request Headers

Adding CSRF tokens, a double submit cookie and value, encrypted token, or other defense that involves changing the UI can frequently be complex or otherwise problematic. An alternate defense which is particularly well suited for AJAX endpoints is the use of a custom request header. This defense relies on the same-origin policy (SOP) (https://en.wikipedia.org/wiki/Same-origin_policy) restriction that only JavaScript can be used to add a custom header, and only within its origin. By default, browsers don't allow JavaScript to make cross origin requests.

A particularly attractive custom header and value to use is:

- X-Requested-With: XMLHttpRequest

because most JavaScript libraries already add this header to requests they generate by default. Some do not though. For example, AngularJS used to, but doesn't anymore. Their rationale and how to add it back is here (https://github.com/angular/angular.js/commit/3a75b1124d062f64093a90b26630938558909e8d).

If this is the case for your system, you can simply verify the presence of this header and value on all your server side AJAX endpoints in order to protect against CSRF attacks. This approach has the double advantage of usually requiring no UI changes and not introducing any server side state, which is particularly attractive to REST services. You can always add your own custom header and value if that is preferred.

This defense technique is specifically discussed in section 4.3 of Robust Defenses for Cross-Site Request Forgery (https://seclab.stanford.edu/websec/csrf/csrf.pdf). However, bypasses of this defense using Flash were documented as early as 2008 and again as recently as 2015 by Mathias Karlsson to exploit a CSRF flaw in Vimeo (https://hackerone.com/reports/44146). But, we believe that the Flash attack can't spoof the Origin or Referer headers so by checking both of them we believe this combination of checks should prevent Flash bypass CSRF attacks. (NOTE: If anyone can confirm or refute this belief, please let us know so we can update this article)

Alternate CSRF Defense: Require User Interaction

Sometimes its easier or more appropriate to involve the user in the transaction in order to prevent unauthorized transactions (forged or otherwise).

The following are some examples of challenge-response options:

- Re-Authentication (password or stronger)
- One-time Token
- CAPTCHA

While challenge-response is a very strong defense against CSRF (assuming proper implementation), it does impact user experience. For applications in need of high security, tokens (transparent) and challenge-response should be used on high risk functions.

An example of a transparent use of security tokens that doesn't impact the user experience is the use of transaction IDs. Imagine a user wants to initiate a transaction, like send money via PayPal.

- In Step 1: the user provides the email address and the amount of money they want to send.
  - The server responds with a confirmation dialog that includes a unique transaction ID for this money transfer.
- In Step 2: the user confirms the proposed transaction.
  - The confirmation request includes the transaction ID generated in step 1. If the server verifies the transaction ID as part of this step, this prevents CSRF against this transaction flow without using a CSRF specific defense token. The transaction ID as part of the confirm essentially serves as the CSRF defense if the attacker cannot predict this value, The transaction ID 'should' be random/unguessable. Ideally, not just the next transaction number in a growing by one list of transaction IDs.

This is just one example of site's behavior that might naturally protect certain state changing events against CSRF attacks.

Alternate CSRF Defense: SameSite cookie attribute

SameSite cookie attribute prevents the browser from sending this cookie along with cross-site requests. The main goal is mitigate the risk of cross-origin information leakage, and provides some protection against cross-site request forgery attacks.

Currently, only Chrome based browsers supports this attribute.

Source: SameSite dedicated article.

Example of cookies using this attribute:

```
Set-Cookie: JSESSIONID=xxxxx; SameSite=Strict

Set-Cookie: JSESSIONID=xxxxx; SameSite=Lax
```

Prevention Measures That Do NOT Work

For more information on prevention measures that do NOT work, please see the OWASP CSRF article.

Personal Safety CSRF Tips for Users

Since CSRF vulnerabilities are reportedly widespread, it is recommended to follow the following best practices to mitigate risk. These include:

- Logoff immediately after using a Web application
- Do not allow your browser to save username/passwords, and do not allow sites to “remember” your login
- Do not use the same browser to access sensitive applications and to surf the Internet freely (tabbed browsing).
- The use of plugins such as No-Script makes POST based CSRF vulnerabilities difficult to exploit. This is because JavaScript is used to automatically submit the form when the exploit is loaded. Without JavaScript the attacker would have to trick the user into submitting the form manually.

Integrated HTML-enabled mail/browser and newsreader/browser environments pose additional risks since simply viewing a mail message or a news message might lead to the execution of an attack.

Implementation example

The following JEE web filter provides an example of implementation of concepts described into this article.

It implements the following concepts, precisely the ones focusing on stateless approach because the following project,OWASP CSRFGuard (https://github.com/aramrami/OWASP-CSRFGuard), cover the stateful approach.

- Verifying same origin with standard headers.
- CSRF specific defenses:
  - Double submit cookie (stateless).
  - Leverage SameSite cookie attribute.

Source are located here (https://github.com/righettod/poc-csrf) and provides a runnable POC.

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletResponseWrapper;
import javax.xml.bind.DatatypeConverter;
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.security.SecureRandom;
import java.util.Arrays;

/**
 * Filter in charge of validating each incoming HTTP request about Headers and CSRF token.
 * It is called for all requests to backend destination.
 *
 * We use the approach in which:
 * - The CSRF token is changed after each valid HTTP exchange
 * - The custom Header name for the CSRF token transmission is fixed
 * - A CSRF token is associated to a backend service URI in order to enable the support for multiple parallel Ajax request from the same application
 * - The CSRF cookie name is the backend service name prefixed with a fixed prefix
 *
 * Here for the POC we show the "access denied" reason in the response but in production code only return a generic message !
 *
 * @see "https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet"
 * @see "https://wiki.mozilla.org/Security/Origin"
 * @see "https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie"
 * @see "https://chloe.re/2016/04/13/goodbye-csrf-samesite-to-the-rescue/"
 */
@WebFilter("/backend/*")
public class CSRFValidationFilter implements Filter {

    /**
     * JVM param name used to define the target origin
     */
    public static final String TARGET_ORIGIN_JVM_PARAM_NAME = "target.origin";

    /**
     * Name of the custom HTTP header used to transmit the CSRF token and also to prefix
     * the CSRF cookie for the expected backend service
     */
    private static final String CSRF_TOKEN_NAME = "X-TOKEN";

    /**
     * Logger
     */
    private static final Logger LOG = LoggerFactory.getLogger(CSRFValidationFilter.class);

    /**
     * Application expected deployment domain: named "Target Origin" in OWASP CSRF article
     */
    private URL targetOrigin;

    /***
     * Secure generator
     */
    private final SecureRandom secureRandom = new SecureRandom();


    /**
     * {@inheritDoc}
     */
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpReq = (HttpServletRequest) request;
        HttpServletResponse httpResp = (HttpServletResponse) response;
        String accessDeniedReason;

        /* STEP 1: Verifying Same Origin with Standard Headers */
        //Try to get the source from the "Origin" header
        String source = httpReq.getHeader("Origin");
        if (this.isBlank(source)) {
            //If empty then fallback on "Referer" header
            source = httpReq.getHeader("Referer");
            //If this one is empty too then we trace the event and we block the request (recommendation of the article)...
            if (this.isBlank(source)) {
                accessDeniedReason = "CSRFValidationFilter: ORIGIN and REFERER request headers are both absent/empty so we block the request !";
                LOG.warn(accessDeniedReason);
                httpResp.sendError(HttpServletResponse.SC_FORBIDDEN, accessDeniedReason);
                return;
            }
        }

        //Compare the source against the expected target origin
        URL sourceURL = new URL(source);
        if (!this.targetOrigin.getProtocol().equals(sourceURL.getProtocol()) || !this.targetOrigin.getHost().equals(sourceURL.getHost())
        || this.targetOrigin.getPort() != sourceURL.getPort()) {
            //One the part do not match so we trace the event and we block the request
            accessDeniedReason = String.format("CSRFValidationFilter: Protocol/Host/Port do not fully matches so we block the request! (%s != %s) ",
                this.targetOrigin, sourceURL);
            LOG.warn(accessDeniedReason);
            httpResp.sendError(HttpServletResponse.SC_FORBIDDEN, accessDeniedReason);
            return;
        }

        /* STEP 2: Verifying CSRF token using "Double Submit Cookie" approach */
        //If CSRF token cookie is absent from the request then we provide one in response but we stop the process at this stage.
        //Using this way we implement the first providing of token
        Cookie tokenCookie = null;
        if (httpReq.getCookies() != null) {
            String csrfCookieExpectedName = this.determineCookieName(httpReq);
            tokenCookie = Arrays.stream(httpReq.getCookies()).filter(c -> c.getName().equals(csrfCookieExpectedName)).findFirst().orElse(null);
        }
        if (tokenCookie == null || this.isBlank(tokenCookie.getValue())) {
            LOG.info("CSRFValidationFilter: CSRF cookie absent or value is null/empty so we provide one and return an HTTP NO_CONTENT response !");
            //Add the CSRF token cookie and header
            this.addTokenCookieAndHeader(httpReq, httpResp);
            //Set response state to "204 No Content" in order to allow the requester to clearly identify an initial response providing the initial CSRF token
            httpResp.setStatus(HttpServletResponse.SC_NO_CONTENT);
        } else {
            //If the cookie is present then we pass to validation phase
            //Get token from the custom HTTP header (part under control of the requester)
            String tokenFromHeader = httpReq.getHeader(CSRF_TOKEN_NAME);
            //If empty then we trace the event and we block the request
            if (this.isBlank(tokenFromHeader)) {
                accessDeniedReason = "CSRFValidationFilter: Token provided via HTTP Header is absent/empty so we block the request !";
                LOG.warn(accessDeniedReason);
                httpResp.sendError(HttpServletResponse.SC_FORBIDDEN, accessDeniedReason);
            } else if (!tokenFromHeader.equals(tokenCookie.getValue())) {
                //Verify that token from header and one from cookie are the same
                //Here is not the case so we trace the event and we block the request
                accessDeniedReason = "CSRFValidationFilter: Token provided via HTTP Header and via Cookie are not equals so we block the request !";
                LOG.warn(accessDeniedReason);
                httpResp.sendError(HttpServletResponse.SC_FORBIDDEN, accessDeniedReason);
            } else {
                //Verify that token from header and one from cookie matches
                //Here is the case so we let the request reach the target component (ServiceServlet, jsp...) and add a new token when we get back the bucket
                HttpServletResponseWrapper httpRespWrapper = new HttpServletResponseWrapper(httpResp);
                chain.doFilter(request, httpRespWrapper);
                //Add the CSRF token cookie and header
                this.addTokenCookieAndHeader(httpReq, httpRespWrapper);
            }
        }
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        //To easier the configuration, we load the target expected origin from an JVM property
        //Reconfiguration only require an application restart that is generally acceptable
        try {
            this.targetOrigin = new URL(System.getProperty(TARGET_ORIGIN_JVM_PARAM_NAME));
        } catch (MalformedURLException e) {
            LOG.error("Cannot init the filter !", e);
            throw new ServletException(e);
        }
        LOG.info("CSRFValidationFilter: Filter init, set expected target origin to '{}'.", this.targetOrigin);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void destroy() {
        LOG.info("CSRFValidationFilter: Filter shutdown");
    }

    /**
     * Check if a string is null or empty (including containing only spaces)
     *
     * @param s Source string
     * @return TRUE if source string is null or empty (including containing only spaces)
     */
    private boolean isBlank(String s) {
        return s == null || s.trim().isEmpty();
    }

    /**
     * Generate a new CSRF token
     *
     * @return The token a string
     */
    private String generateToken() {
        byte[] buffer = new byte[50];
        this.secureRandom.nextBytes(buffer);
        return DatatypeConverter.printHexBinary(buffer);
    }

    /**
     * Determine the name of the CSRF cookie for the targeted backend service
     *
     * @param httpRequest Source HTTP request
     * @return The name of the cookie as a string
     */
    private String determineCookieName(HttpServletRequest httpRequest) {
        String backendServiceName = httpRequest.getRequestURI().replaceAll("/", "-");
        return CSRF_TOKEN_NAME + "-" + backendServiceName;
    }

    /**
     * Add the CSRF token cookie and header to the provided HTTP response object
     *
     * @param httpRequest  Source HTTP request
     * @param httpResponse HTTP response object to update
     */
    private void addTokenCookieAndHeader(HttpServletRequest httpRequest, HttpServletResponse httpResponse) {
        //Get new token
        String token = this.generateToken();
        //Add cookie manually because the current Cookie class implementation do not support the "SameSite" attribute
        //We let the adding of the "Secure" cookie attribute to the reverse proxy rewriting...
        //Here we lock the cookie from JS access and we use the SameSite new attribute protection
        String cookieSpec = String.format("%s=%s; Path=%s; HttpOnly; SameSite=Strict", this.determineCookieName(httpRequest), token, httpRequest.getRequestURI());
        httpResponse.addHeader("Set-Cookie", cookieSpec);
        //Add cookie header to give access to the token to the JS code
        httpResponse.setHeader(CSRF_TOKEN_NAME, token);
    }
}
```

Authors and Primary Editors

Dave Wichers - dave.wichers[at]owasp.org
Paul Petefish - https://www.linkedin.com/in/paulpetefish
Eric Sheridan - eric.sheridan[at]owasp.org
Dominique Righetto - dominique.righetto[at]owasp.org

Other Cheatsheets

V - T - E (https://www.owasp.org/index.php?title=Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet&action=edit) **Cheat Sheets** [Collapse]

Developer / Builder 3rd Party Javascript Management · Access Control · AJAX Security Cheat Sheet · Authentication (ES) · Bean Validation Cheat Sheet · Choosing and Using Security Questions · Clickjacking Defense · Credential Stuffing Prevention Cheat Sheet · **Cross-Site Request Forgery (CSRF) Prevention** · Cryptographic Storage · C-Based Toolchain Hardening · Deserialization · DOM based XSS Prevention · Forgot Password · HTML5 Security · HTTP Strict Transport Security · Injection Prevention Cheat Sheet · Injection Prevention Cheat Sheet in Java · JSON Web Token (JWT) Cheat Sheet for Java · Input Validation · JAAS · LDAP Injection Prevention · Logging · Mass Assignment Cheat Sheet · .NET Security · OS Command Injection Defense Cheat Sheet · OWASP Top Ten · Password Storage · Pinning · Query Parameterization · REST Security · Ruby on Rails · Session Management · SAML Security · SQL Injection Prevention · Transaction Authorization · Transport Layer Protection · Unvalidated Redirects and Forwards · User Privacy Protection · Web Service Security · XSS (Cross Site Scripting) Prevention · XML External Entity (XXE) Prevention Cheat Sheet

Assessment / Breaker Attack Surface Analysis · REST Assessment · Web Application Security Testing · XML Security Cheat Sheet · XSS Filter Evasion

Mobile Android Testing · IOS Developer · Mobile Jailbreaking

OpSec / Defender Virtual Patching

Draft and Beta Application Security Architecture · Business Logic Security · Content Security Policy · Denial of Service Cheat Sheet · Grails Secure Code Review · Insecure Direct Object Reference Prevention · IOS Application Security Testing · Key Management · PHP Security · Regular Expression Security Cheatsheet · Secure Coding · Secure SDLC · Threat Modeling · Vulnerability Disclosure

All Pages In This Category