IJSES
Volume 8, Issue 10, pp. 58-61, 2024.
International Journal of Scientific Engineering and Science
ISSN (Online): 2456-7361

Effective Strategies to Protect Web Applications from CSRF Attacks

Okhonko Pylyp
Application Security Engineer, Tential
Rockville, Maryland, United States

Abstract— *This paper discusses effective strategies for protecting web applications from CSRF (Cross-Site Request Forgery) attacks. The mechanisms of action of CSRF attacks, their potential threats and methods of their implementation are analyzed. The main focus is on security methods, including the use of CSRF tokens, checking the Origin and Referer headers, as well as configuring the SameSite attribute for cookies. Recommendations on the use of two-factor authentication and the implementation of middleware for token management are provided. The principles of the Same Origin Policy and the CORS (Cross-Origin Resource Sharing) mechanism, which provide additional levels of protection, are also considered. The work highlights the need for an integrated approach and continuous security monitoring, since using one method does not guarantee complete protection*.

Keywords— *CSRF attacks, web applications, protection, CSRF tokens, SameSite, two-factor authentication, single origin policy, CORS, web security*.

I. INTRODUCTION

In recent years, the development of web applications has become an integral part of the digital infrastructure used across various fields, including e-commerce, social networks, and government services. Along with the increase in the number of users and the expansion of web application functionality, there has been a significant rise in threats related to data security and user protection. One such threat is CSRF (Cross-Site Request Forgery) attacks, which allow attackers to perform unauthorized actions on behalf of users without their knowledge or consent.

The relevance of this topic is due to the fact that, despite the significant attention paid to web application security, CSRF attacks remain a serious threat.

The aim of this paper is to explore modern strategies for protecting web applications from CSRF attacks, analyze their effectiveness, and develop recommendations for their implementation.

1. Mechanism of CSRF Attacks

In 2015, CSRF attacks were included in the OWASP list of the most critical vulnerabilities (OWASP is an open web application security project created and supported by the non-profit organization OWASP Foundation), ranking in eighth position. However, in 2017, this type of threat was no longer included in the updated list. This may create the illusion that the vulnerability has lost its relevance, but that is not the case. According to research conducted by Positive Technologies [1] as part of penetration testing and security assessments of web applications, the majority of them remain vulnerable to CSRF attacks. Unlike other vulnerabilities that arise due to programming errors, CSRF is related to the inherent functioning of web servers and browsers. Most websites that use a typical architecture are by default susceptible to this threat.

CSRF (Cross-Site Request Forgery) represents cross-site request forgery. The mechanism of this attack relies on the use of cookies. The term "CSRF" was introduced by Peter Watkins in 2001. Cookies are data elements exchanged between a client and server, which the server sends to the client in a specific format. The browser stores this data on the user's device and, when necessary, sends it back to the server in the HTTP request header. When a user clicks on a specially crafted link created by an attacker, a hidden request may be sent to the server on the user's behalf, executing a malicious action. However, for the attack to succeed, the user must be logged into the target site, and the site must not require confirmation of an action that cannot be ignored or forged.

Although CSRF may resemble XSS (Cross-Site Scripting) attacks, there is a fundamental difference between them. Both types of attacks use web application users as attack vectors; however, CSRF can be combined with XSS or other methods, such as redirects, forming a separate class of vulnerabilities.

The main threat of CSRF attacks lies in the fact that they exploit the normal behavior of browsers and the HTTP protocol, making them difficult to detect. For example, loading images from another site is common practice, and the browser cannot distinguish whether attackers are trying to load an image or perform a hidden malicious action on the target site [1].

To successfully carry out a CSRF attack, several conditions must be met:

- Authentication via cookies: The attack is possible only if user authentication depends on cookies or Basic HTTP authentication.
- Predictability of request parameters: The values of the parameters in the requests must be easily guessable by the attacker.
- Presence of certain vulnerable functions in the application: The application must contain functions that could be of interest to an attacker, such as actions with high privilege levels or changes to user data [2].

Next, we will consider the main misconceptions associated with CSRF vulnerabilities in systems.

Forging HTTP requests is a new security threat. This statement is not true. Issues related to message data forgery

have been discussed since the late 1980s. For instance, as early as 1988, theoretical works on this topic appeared. Practical attention to this vulnerability has also been traced on security forums such as Bugtraq, starting from at least 2000. The term "CSRF" was first introduced by Peter Watkins in 2001 [4].

CSRF and XSS belong to the same category. Although CSRF and XSS exploit client-side vulnerabilities in web applications, they represent different threat categories. In the case of CSRF, user behavior and browser-server interaction are exploited, which distinguishes this vulnerability from XSS, although there are situations where both vulnerabilities can act in tandem. However, it is essential to understand that CSRF is an independent vulnerability that can exist without the presence of XSS or other types of attacks [4].

CSRF is a rare and difficult-to-execute vulnerability. However, in practice, research conducted by companies like Positive Technologies shows that most web applications are vulnerable to this threat. Unlike many other threats, CSRF is not caused by code errors but by standard functionality inherent to most servers and browsers. Thus, websites with typical architecture are by default vulnerable to this attack [4].

2. Use of Authentication Tokens (CSRF Token) as a Protection Method

Certain types of HTTP requests are vulnerable to CSRF attacks, especially those that modify the server or contain critical data. POST, PUT, DELETE, and PATCH requests are particularly at risk:

- POST requests are used to send data to the server to create or modify resources. They are often used for form submissions, creating database entries, or conducting financial transactions, making them vulnerable to CSRF attacks due to the potential for changing the system's state.
- PUT requests are intended to update existing resources on the server, which may include changing user data, updating product information, or system settings.
- DELETE requests are used to remove resources from the server, such as user accounts, files, or database records.
- PATCH requests are used for partial updates to resources, such as modifying specific fields in a record or configurations, also making them a target for CSRF attacks.

Other types of HTTP requests, such as GET, HEAD, OPTIONS, and TRACE, are less susceptible to CSRF attacks, as they generally do not modify the server's state and are mainly used for retrieving metadata or checking resource availability.

The generation of CSRF tokens is a key element in protecting against cross-site request forgery. These signals must be sufficiently complex and random to prevent attackers from guessing. The effective transmission of CSRF tokens is a critical aspect of ensuring the security of web applications:

- Transmission through hidden form fields. One of the most secure methods of transmitting CSRF tokens is by including them in hidden fields within HTML forms submitted via POST. This ensures that tokens are transmitted only at the time of form submission, preventing them from being exposed in the URL.
- Transmission via URL. This method is less secure because query strings in the URL can be logged by the server or passed to third parties through the HTTP Referer header. Additionally, such tokens may be visible in the browser's address bar, increasing the risk of exposure.
- Use of custom headers. Some applications prefer to transmit CSRF tokens through custom request headers, providing additional security since browsers typically do not allow cross-domain transmission of such headers.

After generating CSRF tokens, it is important to ensure their proper storage and validation:

- Storage of tokens on the server. CSRF tokens must be stored in the user's session data on the server so that each token can be associated with a specific user.
- Validation process. When the server receives a request requiring CSRF token validation, it retrieves the token from the request and compares it with the one stored in the user's session. If the tokens match, the request is considered valid; otherwise, the server rejects the request.

To protect a website from CSRF attacks, the following methods are recommended, as described in Table 1.

TABLE 1. Methods of protecting websites from CSRF attacks [5].

Method name	Description
Use of CSRF tokens	Tokens must be generated by the server and included in every form submitted by the user. If the token does not match the expected value, the request should be rejected.
Checking Origin and Referer headers	This method ensures that requests come from trusted domains.
Limiting the use of dangerous HTTP methods	Web applications should limit the use of methods such as POST, PUT, DELETE, and PATCH to minimize the risk of data modification on the server.
SameSite attribute for cookies	This attribute restricts the sending of cookies only in requests from the same site, reducing the risk of CSRF attacks.
Regular updates and monitoring	It is important to install updates and patches in a timely manner, as well as monitor website activity to detect and prevent threats in time.

These measures together create a multi-layered defense that effectively counters CSRF attacks.

3. Additional Security Measures and Best Practices

The Same Origin Policy (SOP) is an important security mechanism used in web browsers. It defines "origin" as a combination of elements such as the scheme (protocol), domain name, and port.

If two web resources use the same scheme, domain, and port, they are considered to have the same origin. Otherwise, if even one of these elements differs, the resources are treated as belonging to different origins (cross-origin). For example, websites with URLs http://site.store.com and http://api.store.com will be perceived as different origins, despite both belonging to the same domain.

The main rule of this policy states that scripts running on one website can interact with data from another website only if both sites share the same origin. For instance, if you need to display data from http://api.store.com on http://site.store.com

Volume 8, Issue 10, pp. 58-61, 2024. ISSN (Online): 2456-7361

via a GET request, the browser may block this action due to CORS policy.

CORS (Cross-Origin Resource Sharing) is a mechanism that allows web resources to provide access to their data from other origins, thereby bypassing SOP restrictions. This mechanism categorizes requests into simple and complex. Simple requests are those that perform safe actions and do not modify server data, such as requests using the GET and HEAD methods. These requests are not blocked by the browser.

For complex requests, such as DELETE, the browser first sends a preflight request using the OPTIONS method to determine whether the specific type of request is allowed for the particular origin. If the server's response contains the necessary headers, such as Access-Control-Allow-Origin, Access-Control-Allow-Methods, and Access-Control-Allow-Headers, the main request will be executed. Otherwise, the browser will block it.

Configuring CORS for simple requests is to send the Access-Control-Allow-Origin header in response to the message. For complex requests, more detailed configuration is required, including specifying allowable methods and headers in the response to the preflight request. It is important to remember that for security purposes, it is advisable to avoid using the wildcard value "*" in the Access-Control-Allow-Origin header, as this allows any origin to access the data [6].

The REST (Representational State Transfer) architecture stipulates that GET requests should be used solely for retrieving data or resources, while server state changes should be performed using other methods, such as PUT, POST, or DELETE. It is important to note that each HTTP method has its own purpose: GET is used for data retrieval, POST for updates, PUT for creation, and DELETE for removal. However, in situations where the method is not obvious, additional security measures should be applied to minimize risks. For example, to prevent unwanted actions, it is essential to always use GET only for data retrieval.

Double Cookie Submission Method:

An alternative to synchronization tokens is the double cookie submission method. Upon visiting the site before authentication, a value is generated and stored as a cookie in the user's browser. Then, any action sent from the client must include this value as a hidden form field. If the value in the form and the cookie match, the server accepts the request. Otherwise, the request is rejected. This method requires additional precautions, such as cookie encryption or the use of HMAC to enhance security.

In some cases, adding CSRF tokens or modifying the interface to improve security may be difficult. In such situations, custom request headers can be used to protect AJAX or API endpoints. This method relies on the Same Origin Policy (SOP), which limits the ability to execute requests with custom headers only to the same origin from which the JavaScript was executed. It is important to remember that browsers by default block cross-origin requests with such headers. Methods like POST, PUT, PATCH, and DELETE, which modify the system's state, must include a CSRF token in the request. An example implementation using the Axios library shows how to automatically add tokens to the headers of every AJAX request to ensure security.

One of the most effective ways to protect against CSRF attacks is by using proven solutions such as token management modules. For example, the **csurf** library allows for the automatic generation and management of CSRF tokens, providing a high level of security for your application [7].

```
// server.js
var cookieParser = require('cookie-parser')
var csrf = require('csurf')
var bodyParser = require('body-parser')
var express = require('express')

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()

// parse cookies
// we need this because "cookie" is true in csrfProtection
app.use(cookieParser())

app.get('/form', csrfProtection, function (req, res) {
  // pass the csrfToken to the view
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function (req, res) {
  res.send('data is being processed')
})
```

Next, the csrfToken should be set as the value of a hidden input field with the name **_csrf**.

```
<form action="/process" method="POST">
  <input type="hidden" name="_csrf" value="{{csrfToken}}">

  Favorite color: <input type="text" name="favoriteColor">
  <button type="submit">Submit</button>
</form>
```

HttpOnly Cookies: Cookies with the HttpOnly attribute are protected from being accessed through JavaScript scripts, making them less vulnerable to cross-site scripting (XSS) attacks. In the event of a successful XSS attack, attackers will not be able to access these cookies, as JavaScript cannot read or use them.

Combined Use: These two attributes are often used together to enhance the security of a web resource. For example, they can be combined with two-factor authentication, creating a multi-layered security system.

Content Security Policy (CSP): Content Security Policy (CSP) is an important tool for protecting web applications from various threats, such as XSS and the injection of malicious data. This mechanism allows administrators to control the sources of content that can be loaded and executed on a page, thus limiting the possibility of attacks. CSP can also be used to restrict the use of protocols, for example, allowing content to be loaded only through HTTPS, further enhancing the security of the application [8].

II. Conclusion

In conclusion, it can be stated that effective protection of web applications from CSRF attacks requires the integration of various methods, each aimed at minimizing the risk of vulnerability exploitation. Key methods, such as the use of CSRF tokens and the configuration of the SameSite policy, combined with other measures like two-factor authentication and header validation, help create a multi-layered security system. It is important to note that continuous updates and system monitoring are crucial elements in maintaining the security of web applications, especially in the face of evolving new types of attacks.

References

1. Laundev I. S. Methods of protecting web applications from CSRF attacks / I. S. Laundev. // A young scientist. — 2022. — № 8 (403). — Pp. 4-7.
2. Agrawal S. Mitigating Cross-Site Request Forgery (CSRF) Attacks Using Reinforcement Learning and Predictive Analytics //Applied Research in Artificial Intelligence and Cloud Computing. – 2023. – Vol. 6. – No. 9. – pp. 17-30.
3. Tokarev D. I. Modern methods of protection against CSRF attacks //Bulletin of the Student Scientific Society of the Donetsk National University The founders: Donetsk National University. – 2022. – vol. 1. – no. 14. – pp. 202-206.
4. Padalets A.M., Kulikova A. S. Methods of protection against network attacks such as cross-site request forgery //Innovative technologies in the training of modern professional personnel: experience, problems. - 2020. – pp. 91-95.
5. Agrawal S. Mitigating Cross-Site Request Forgery (CSRF) Attacks Using Reinforcement Learning and Predictive Analytics //Applied Research in Artificial Intelligence and Cloud Computing. – 2023. – Vol. 6. – No. 9. – pp. 17-30.
6. Darmawan I. et al. Json web token penetration testing on cookie storage with csrf techniques //2021 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS). – IEEE, 2021. – pp. 1-5.
7. Padalets A.M., Kulikova A. S. Methods of protection against network attacks such as cross-site request forgery //Innovative technologies in the training of modern professional personnel: experience, problems. - 2020. – pp. 91-95.
8. Shutko N. A. Theoretical concepts of protecting Web applications from vulnerabilities //Bulletin of Science. – 2022. – T. 4. – №. 11 (56). – Pp. 253-269.