Chapter 29. Defending Against CSRF Attacks

In Part II we built Cross-Site Request Forgery (CSRF) attacks that took advantage of a user's authenticated session in order to make requests on their behalf. We built CSRF attacks with `<a></a>` links, via `<img></img>` tags, and even via HTTP POST using web forms. We saw how effective and dangerous CSRF-style attacks are against an application because they function at both an elevated privilege level and often are undetectable by the authenticated user.

In this chapter, we will learn how to defend our codebase against such attacks, and mitigate the probability that our users will be put at risk for any type of attack that targets their authenticated session.

Header Verification

Remember the CSRF attacks we built using `<a></a>` links? In that discussion, the links were distributed via email or another website entirely separate from the target.

Because the origin of many CSRF requests is separate from your web application, we can mitigate the risk of CSRF attacks by checking the origin of the request. In the world of HTTP, there are two headers we are interested in when checking the origin of a request: `referer` and `origin`. These headers are important because they cannot be modified programmatically with JavaScript in all major browsers. As such, a properly implemented browser's `referrer` or `origin` header has a low chance of being spoofed:

Origin header

The `origin` header is sent only on HTTP POST requests. It is a simple header that indicates where a request originated. Unlike `referer`, which appears on all requests (regardless of HTTP or HTTPS). An `origin` header looks like: `Origin: https://www.mega-bank.com:80`.

Referer header

The `referer` header is set on all requests and also indicates where a request originated from. The only time this header is not present is when the referring link has the attribute `rel=noreferer` set. A `referer` header looks like: `Referer: https://www.mega-bank.com:80`.

When a POST request is made to your web server—for example, `https://www.mega-bank.com/transfer` with params `amount=1000` and `to_user=123`—you can verify that the location of these headers is the same as your trusted origins from which you run your web servers. Here is a node implementation of such a check:

```
const transferFunds = require('../operations/transferFunds');
const session = require('../util/session');

const validLocations = [
 'https://www.mega-bank.com',
 'https://api.mega-bank.com',
 'https://portal.mega-bank.com'
 ];

const validateHeadersAgainstCSRF = function(headers) {
 const origin = headers.origin;
 const referer = headers.referer;
 if (!origin || referer) { return false; }
 if (!validLocations.includes(origin) ||
     !validLocations.includes(referer)) {
       return false;
     }
  return true;
};

const transfer = function(req, res) {
 if (!session.isAuthenticated) { return res.sendStatus(401); }
```

```
  if (!validateHeadersAgainstCSRF(req.headers)) { return res.sendStatus(401); }

  return transferFunds(session.currentUser, req.query.to_user, req.query.amount);
};

module.exports = transfer;
```

Whenever possible, check both headers. If neither header is present, it is safe to assume that the request is not standard and should be rejected.

These headers are a first line of defense, but there is a case where they will fail. Should an attacker get an XSS on an allowlisted origin of yours, they can initiate the attack from your own origin, appearing to come from your own servers as a legitimate request.

This case is even more worrisome if your website allows user-generated content to be posted. In this case, validating headers to ensure that they come from your own web servers may not be beneficial at all. As such, it is best to employ multiple forms of CSRF defense, with header verification being a starting point rather than a full-fledged solution.

CSRF Tokens

The most powerful form of defense against CSRF attacks is the *anti-CSRF token*, often just called a CSRF token (see Figure 29-1). CSRF tokens defend against CSRF attacks in a very simple way and can be implemented in a number of ways to fit your current application architecture with ease. Most major websites rely on CSRF tokens as their primary defense against CSRF attacks.

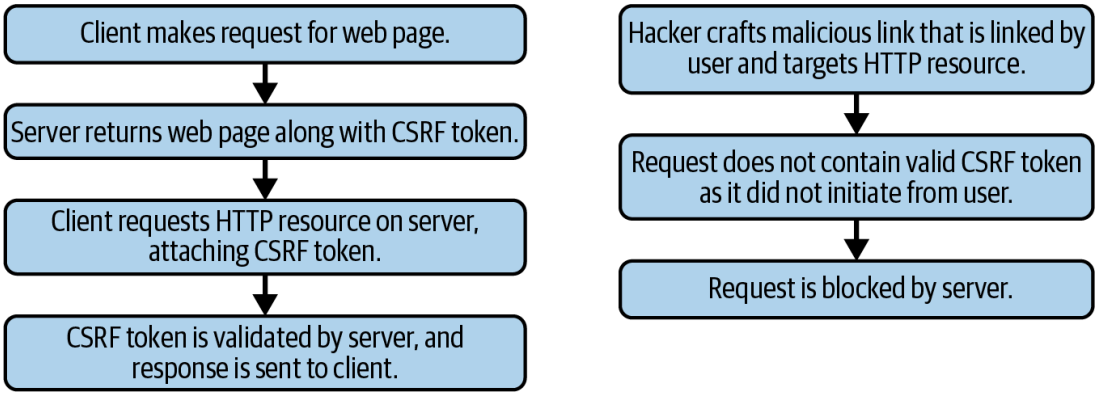

Figure 29-1. CSRF tokens, the most effective and reliable method of eliminating CSRF attacks

At its core, CSRF token defense works like this:

1. Your web server sends a special token to the client. This token is generated cryptographically with a very low collision algorithm, which means that the odds of getting two identical tokens are exceedingly rare. The token can be regenerated as often as per request, but generally is generated per session.
2. Each request from your web application now sends the token back with it; this should be sent back in forms as well as AJAX requests. When the request gets to the server, the token is verified to make sure it is live (not expired), authentic, and has not been manipulated. If verification fails, the request is logged and fails as well.
3. As a result of requests requiring a valid CSRF token, which is unique per session and unique to each user, CSRF attacks originating from other origins become extremely difficult to pull off. Not only would the attacker need a live and up-to-date CSRF token, but they would also now need to target a specific user versus a large number of users. Furthermore, with token expiration compromised, CSRF tokens can be dead by the time a user clicks a malicious link—a beneficial side effect of CSRF tokens as a defensive strategy.

In the past, especially prior to the rise of REST architecture for APIs, many servers would keep a record of the clients connected. Because of this, it was feasible for servers to manage the CSRF tokens for the clients.

In modern web applications, statelessness is often a prerequisite to API design. The benefits carried by a stateless design cannot be understated. It would not be wise to change a stateless design to a stateful one just for the sake of adding CSRF tokens. CSRF tokens can be easily added to stateless APIs, but encryption must be involved.

Much like stateless authentication tokens, a stateless CSRF token should consist of the following:

- A unique identifier of the user the token belongs to
- A timestamp (which can be used for expiration)
- A cryptographic nonce whose key only exists on the server

Combining these elements nets you a CSRF token that is not only practical but also consumes fewer server resources than the stateful alternative, as managing sessions does not scale well compared to a sessionless alternative.

Anti-CRSF Coding Best Practices

There are many methods of eliminating or mitigating CRSF risk in your web application that start at the code or design phase. Several of the most effective methods are:

- Refactoring to stateless GET requests
- Implementation of application-wide CSRF defenses
- Introduction of request-checking middleware

Implementing these simple defenses in your web application will dramatically reduce the risk of falling prey to CSRF-targeting hackers.

Stateless GET Requests

Because the most common and easily distributable CSRF attacks come via HTTP GET requests, it is important to correctly structure our API calls to mitigate this risk. HTTP GET requests should not store or modify any server-side state. Doing so leaves future GET requests or modifications to GET requests open to potential CSRF vulnerabilities.

Consider the following APIs:

```
// GET
const user = function(req, res) {
 getUserById(req.query.id).then((user) => {
   if (req.query.updates) { user.update(req.updates); }
   return res.json(user);
 });
};
```

```
// GET
const getUser = function(req, res) {
```

```
  getUserById(req.query.id).then((user) => {
    return res.json(user);
  });
};

// POST
const updateUser = function(req, res) {
  getUserById(req.query.id).then((user) => {
    user.update(req.updates).then((updated) => {
      if (!updated) { return res.sendStatus(400); }
      return res.sendStatus(200);
    });
  });
};
```

The first API combines the two operations into a single request, with an optional update. The second API splits retrieving and updating users into a GET and POST request, respectively.

The first API can be taken advantage of by CSRF in any HTTP GET (e.g., a link or image: *https://<url>/user?user=123&updates=email:hacker*). The second API, while still an HTTP POST and potentially vulnerable to more advanced CSRF, cannot be taken advantage of by links, images, or other HTTP GET-style CSRF attacks.

This seems like a simple architecture flaw (modifying state in HTTP GET requests), and in all honesty, it is. But the key point here applies to any and all GET requests that could potentially modify server-side application state—don't do it. HTTP GET requests are at risk by default; the nature of the web makes them much more vulnerable to CSRF attacks, and you should avoid them for stateful operations.

Application-Wide CSRF Mitigation

The techniques in this chapter for defending against CSRF attacks are useful but only when implemented application wide. As with many attacks, the weakest link breaks the chain. With careful forethought you can build an application architected specifically to protect against such attacks. Let's consider how to build such an application.

Anti-CSRF middleware

Most modern web server stacks allow for the creation of *middleware*, or scripts that run on every request, prior to any logic being performed by a route. Such middleware can be developed to implement these techniques on all of your server-side routes. Let's take a look at some middleware that accomplishes just this:

```
const crypto = require('../util/crypto');
const dateTime = require('../util/dateTime');
const session = require('../util/session');
const logger = require('../util/logger');

const validLocations = [
 'https://www.mega-bank.com',
 'https://api.mega-bank.com',
 'https://portal.mega-bank.com'
 ];

const validateHeaders = function(headers, method) {
  const origin = headers.origin;
  const referer = headers.referer;
  let isValid = false;

  if (method === 'POST') {
    isValid = validLocations.includes(referer) && validLocations.includes(origin)
  } else {
    isValid = validLocations.includes(referer);
  }

  return isValid;
};

const validateCSRFToken = function(token, user) {
  // get data from CSRF token
  const text_token = crypto.decrypt(token);
  const user_id = text_token.split(':')[0];
  const date = text_token.split(':')[1];
  const nonce = text_token.split(':')[2];

  // check validity of data
  let validUser = false;
  let validDate = false;
```

```
    let validNonce = false;

    if (user_id === user.id) { validUser = true; }
    if (dateTime.lessThan(1, 'week', date)) { validDate = true; }
    if (crypto.validateNonce(user_id, date, nonce)) { validNonce = true; }

    return validUser && validDate && validNonce;
  };

  const CSRFShield = function(req, res, next) {
   if (!validateHeaders(req.headers, req.method) ||
       !validateCSRFToken(req.csrf, session.currentUser) {
       logger.log(req);
       return res.sendStatus(401);
    }

   return next();
  };
```

This middleware can be invoked on all requests made to the server or individually defined to run on specific requests. The middleware simply verifies that the origin and/or referrer headers are correct, and then it ensures that the CSRF token is valid. It returns an error before any other logic is called if either fail; otherwise, it moves on to the next middleware and allows the application to continue execution unaltered.

Because this middleware relies on a client consistently passing a CSRF token to the server on each request, it would be optimal to replicate such automation on the client as well. This can be done with a number of techniques. For example, you could use the proxy pattern to overwrite the `XMLHTTPRequest` default behavior to always include the token. Alternatively, you could use a more simple approach that would rely on building a library for generating requests that would simply wrap the `XMLHTTPRequest` and inject the correct token, depending on the HTTP verb.

Summary

CSRF attacks can be mitigated for the most part by ensuring that HTTP GET requests never alter any application state. Further, CSRF mitigations

should consider validating headers and adding CSRF tokens to each of your requests. With these mitigations in place, your users will be able to feel more comfortable entering your web application from other origins, and they'll face a lower risk of compromising their account permissions by a hacker with malicious intent.