Chapter 11. Cross-Site Request Forgery

Sometimes we already know an API endpoint exists that would allow us to perform an operation we wish to perform, but we do not have access to that endpoint because it requires privileged access (e.g., an admin account). In this chapter, we will be building Cross-Site Request Forgery (CSRF) exploits that result in an admin or privileged account performing an operation on our behalf rather than using a JavaScript code snippet.

CSRF attacks take advantage of the way browsers operate and the trust relationship between a website and the browser. By finding API calls that rely on this relationship to ensure security—but yield too much trust to the browser—we can craft links and forms that with a little bit of effort can cause a user to make requests on their own behalf—unknown to the user generating the request.

Oftentimes CSRF attacks will go unnoticed by the user that is being attacked because requests in the browser occur behind the scenes. This means that this type of attack can be used to take advantage of a privileged user and perform operations against a server without the user ever knowing. It is one of the most stealthy attacks and has caused havoc throughout the web since its inception in the early 2000s.

Query Parameter Tampering

Let's consider the most basic form of CSRF attack—parameter tampering via a hyperlink (see Figure 11-1). Most forms of hyperlink on the web correspond with HTTP GET requests. The most common hyperlink is simply an `<a href="https://my-site.com"></a>` embedded in an HTML snippet.

The anatomy of an HTTP GET request is simple and consistent regardless of where it is sent from, read from, or how it travels over the network. For an HTTP GET to be valid, it must follow a supported version of the HTTP specification—so we can rest assured that the structure of a GET request is the same across applications.

The anatomy of an HTTP GET request is as follows:

```
GET /resource-url?key=value HTTP/1.1
Host: www.mega-bank.com
```

Every HTTP GET request includes the HTTP method (`GET`), followed by a resource URL, and then followed by an optional set of query parameters. The start of the query params is denoted by `?` and continues until whitespace is found. After this comes the HTTP specification, and on the next line is the host at which the resource URL can be located.

When a web server gets this request it will be routed to the appropriate handler class, which will receive the query parameters alongside some additional information to identify the user that made the request, the type of browser they requested from, and what type of data format they expect in return.

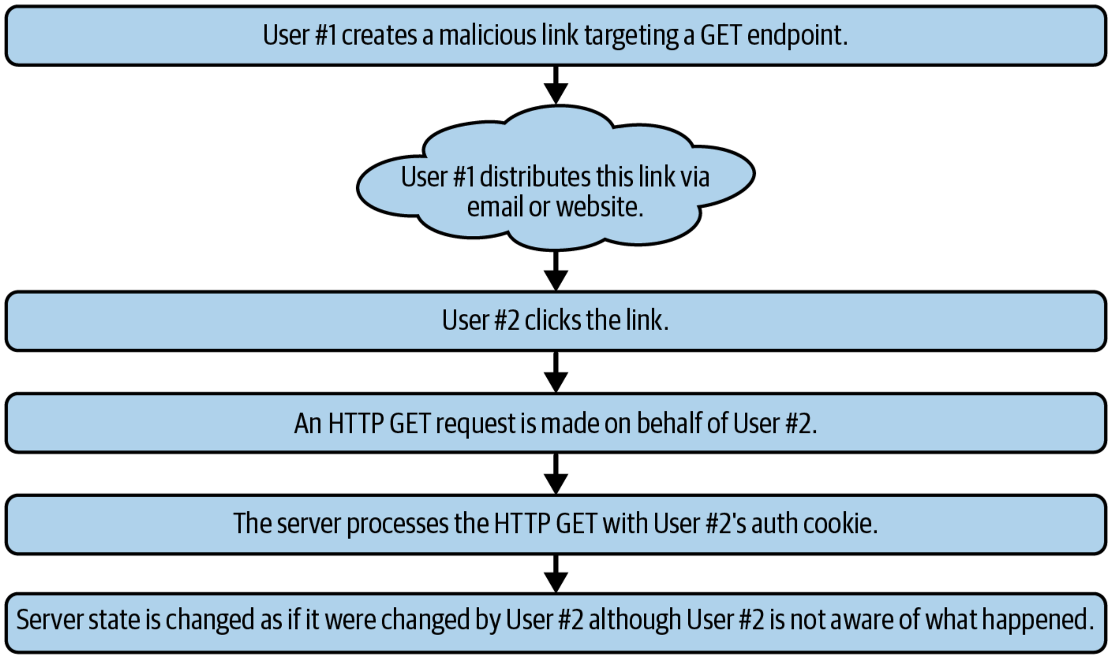
User #1 creates a malicious link targeting a GET endpoint.
User #1 distributes this link via email or website.
User #2 clicks the link.
An HTTP GET request is made on behalf of User #2.
The server processes the HTTP GET with User #2's auth cookie.
Server state is changed as if it were changed by User #2 although User #2 is not aware of what happened.

Figure 11-1. CSRF GET—a malicious link is spread that, when clicked, causes state-changing HTTP GET requests to be performed on behalf of the authenticated user

Let's look at an example in order to make this concept more concrete. The first example is a server-side routing class that is written on top of ExpressJS—the most popular Node.js-based web server software:

```
/*
 * An example route.
 *
 * Returns the query provided by the HTTP request back to the requester.
 * Returns an error if a query is not provided.
 */
app.get('/account', function(req, res) {
  if (!req.query) { return res.sendStatus(400); }
  return res.json(req.query);
});
```

This is an extremely simple route that will do only a few things:

- Accept only HTTP GET requests to `/account`
- Return an HTTP 400 error if no query params are provided
- Reflect query params to the sender in JSON format if they are provided

Let's make a request to this endpoint from a web browser:

```
/*
 * Generate a new HTTP GET request with no query attached.
 *
 * This will fail and an error will be returned.
 */
const xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  console.log(xhr.responseText);
}
xhr.open('GET', 'https://www.mega-bank.com/account', true);
xhr.send();
```

Here, from the browser we initiate an HTTP GET request to the server, which will return a 400 error because we did not provide any query parameters. We can add the query parameters to get a more interesting result:

```
/*
 * Generate a new HTTP GET request with a query attached.
 *
 * This will succeed and the query will be reflected in the response.
 */
const xhr = new XMLHttpRequest();
const params = 'id=12345';
xhr.onreadystatechange = function() {
  console.log(xhr.responseText);
}
xhr.open('GET', `https://www.mega-bank.com/account?${params}`, true);
xhr.send();
```

Shortly after making this request, a response will be returned with the content:

```
{
  id: 12345
}
```

It will also include an HTTP 200 status code if you check out the network request in your browser.

It is crucial to understand the flow of these requests in order to find and make use of CSRF vulnerabilities. Let’s backtrack a bit and talk about CSRF again.

The two main identifiers of a CSRF attack are:

- Privilege escalation
- The user account that initiates the request typically does not know it occurred (it is a stealthy attack)

Most create, read, update, and delete (CRUD) web applications that follow HTTP spec make use of many HTTP verbs, and GET is only one of them. Unfortunately, GET requests are the least secure of any request and one of the easiest ways to craft a CSRF attack.

The last GET endpoint we analyzed reflected data back, but the important part is it did read the query params we sent it. The URL bar in your browser initiates HTTP GET requests, so do `<a></a>` links in the browser or in a phone.

Furthermore, when we click on links throughout the internet, we rarely evaluate the source to see where the link is taking us. This link:

```
<a href="https://www.my-website.com?id=123">My Website</a>
```

would appear literally in the browser as "My Website." Most users would not know a parameter was attached to the link as an identifier. Any user that clicks that link will initiate a request from their browser that will send a query param to the associated server.

Let's imagine our fictional banking website, MegaBank, made use of GET requests with params. Look at this server-side route:

```
import session from '../authentication/session';
import transferFunds from '../banking/transfers';

/*
 * Transfers funds from the authenticated user's bank account
 * to a bank account chosen by the authenticated user.
 *
 * The authenticated user may choose the amount to be transferred.
 */
app.get('/transfer', function(req, res) {
  if (!session.isAuthenticated) { return res.sendStatus(401); }
  if (!req.query.to_user) { return res.sendStatus(400); }
  if (!req.query.amount) { return res.sendStatus(400); }

  transferFunds(session.currentUser, req.query.to_user, req.query.amount,
  (error) => {
            if (error) { return res.sendStatus(400); }
              return res.json({
                 operation: 'transfer',
                 amount: req.query.amount,
                 from: session.currentUser,
                 to: req.query.to_user,
                 status: 'complete'
```

```
        });
    });
  });
```

To the untrained eye, this route looks pretty simple. It checks that the user has the correct privileges, and checks that another user has been specified for the transfer. Because the user had the correct privileges, the amount specified should be accurate considering the user had to be authenticated to make this request (it assumes the request is made on behalf of the requesting user). Similarly, we assume that the transfer is being made to the right person. Unfortunately, because this was made using an HTTP GET request, a hyperlink pointing to this particular route could be easily crafted and sent to an authenticated user.

CSRF attacks involving HTTP GET param tampering usually proceed as follows:

1. A hacker figures out that a web server uses HTTP GET params to modify its flow of logic (in this case, determining the amount and target of a bank transfer).
2. The hacker crafts a URL string with those params: <*a href="https://www.mega-bank.com/transfer?to_user=<hacker's account>&amount=10000">click me</a*>.
3. The hacker develops a distribution strategy: usually either targeted (who has the highest chance of being logged in and having the correct amount of funds?) or bulk (how can I hit as many people with this in a short period of time before it is detected?).

Often these attacks are distributed via email or social media. Due to the ease of distribution, the effects can be devastating to a company. Hackers have even taken out web-advertising campaigns to seed their links in the hands of as many people as possible.

Alternate GET Payloads

Because the default HTTP request in the browser is a GET request, many HTML tags that accept a URL parameter will automatically make GET requests when interacted with or when loaded into the DOM. As a result of this, GET requests are the easiest to attack via CSRF.

In the prior examples, we used a hyperlink `<a></a>` tag in order to trick the user into executing a GET request in their own browser. Alternatively, we could have crafted an image to do the same thing:

```
<!--Unlike a link, an image performs an HTTP GET request right when it loads
 into the DOM. This means it requires no interaction from the user loading
 the web page.-->
<img src="https://www.mega-bank.com/transfer?
to_user=<hacker's account>&amount=10000" width="0" height="0" border="0">
```

When image tags are detected in the browser, the browser will initiate a GET request to the `src` endpoint included in the `<img>` tag (see Figure 11-2). This is how the image objects are loaded into the browser. As such, an image tag (in this case, an invisible 0 × 0 pixel image) can be used to initiate a CSRF without any user interaction required.

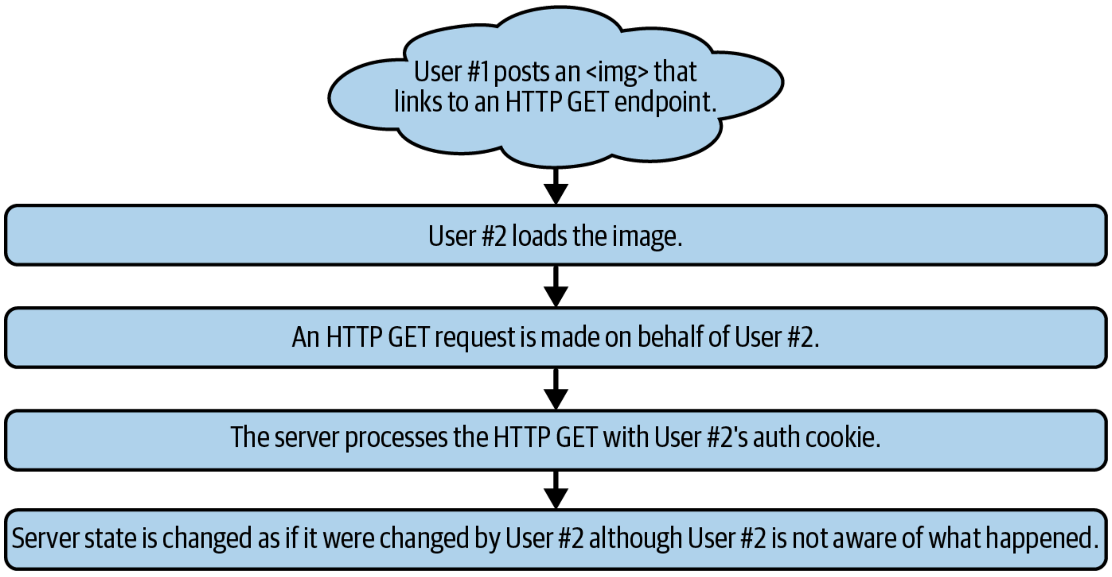
User #1 posts an <img> that links to an HTTP GET endpoint.
User #2 loads the image.
An HTTP GET request is made on behalf of User #2.
The server processes the HTTP GET with User #2's auth cookie.
Server state is changed as if it were changed by User #2 although User #2 is not aware of what happened.

Figure 11-2. CSRF IMG—inside of the target application, an `<img>` tag is posted that forces an HTTP GET when loaded

Likewise, most other HTML tags that allow a URL parameter can also be used to make malicious GET requests. Consider the HTML5 `<video></video>` tag:

```
<!-- Videos typically load into the DOM immediately, depending on the browser's
configuration. Some mobile browsers will not load until the element is interacte
```

```
with. -->
<video width="1280" height="720" controls>
  <source src="https://www.mega-bank.com/transfer?
  to_user=<hacker's account>&amount=10000" type="video/mp4">
</video>
```

The preceding video functions identically to the image tag used. As such, it’s important to be on the lookout for any type of tag that requests data from a server via an `src` attribute. Most of these can be used to launch a CSRF attack against an unsuspecting end user.

CSRF Against POST Endpoints

CSRF attacks typically take place against GET endpoints, as it is much easier to distribute a CSRF via a hyperlink, image, or other HTML tag that initiates an HTTP GET request automatically. However, it is still possible to deliver a CSRF payload that targets a POST, PUT, or DELETE endpoint. Delivery of a POST payload just requires a bit more work as well as some mandatory user interaction (see Figure 11-3).

Typically, CSRF attacks delivered by POST requests are created via browser forms, as the `<form></form>` object is one of the few HTML objects that can initiate a POST request without any script required:

```
<form action="https://www.mega-bank.com/transfer" method="POST">
  <input type="hidden" name="to_user" value="hacker">
  <input type="hidden" name="amount" value="10000">
  <input type="submit" value="Submit">
</form>
```

In the case of CSRF via POST form, we can make use of the “hidden” type attribute on form inputs in order to seed data that will not be rendered inside of the browser.

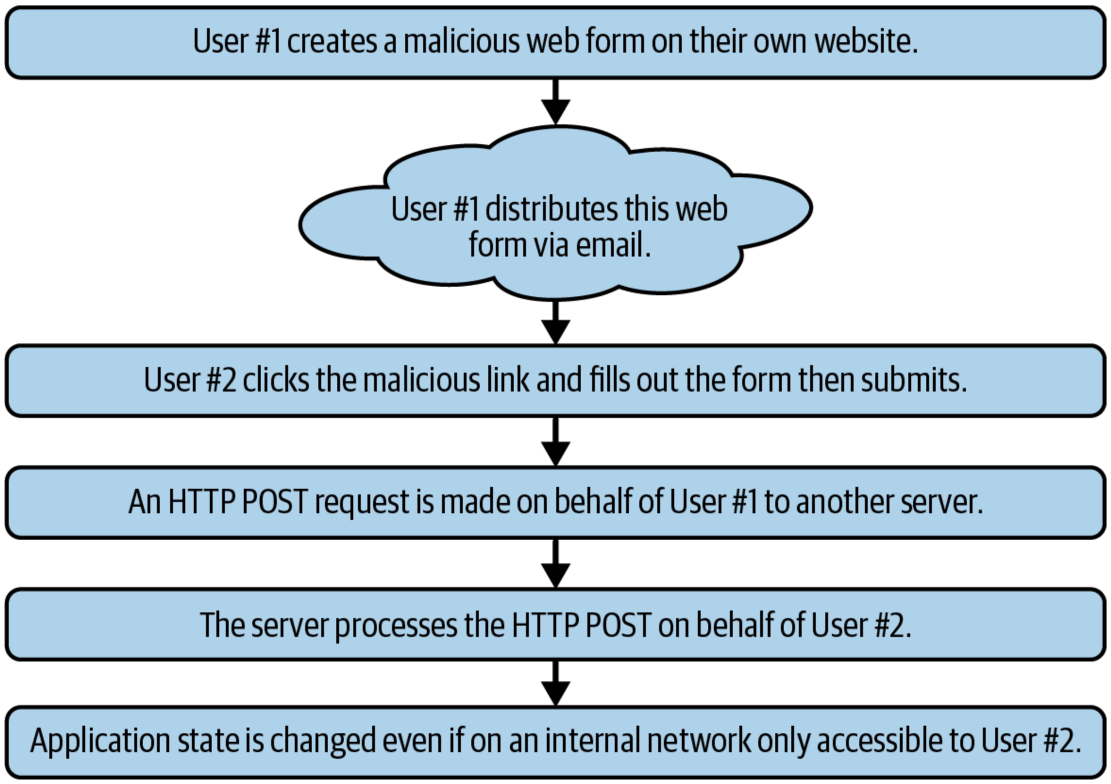
User #1 creates a malicious web form on their own website.
User #1 distributes this web form via email.
User #2 clicks the malicious link and fills out the form then submits.
An HTTP POST request is made on behalf of User #1 to another server.
The server processes the HTTP POST on behalf of User #2.
Application state is changed even if on an internal network only accessible to User #2.

Figure 11-3. CSRF POST—A form is submitted targeting another server that is not accessible to the creator of the form but is to the submitter of the form

We can further manipulate the user by offering legitimate form fields in addition to the hidden fields that are required to design the CSRF payload:

```
<form action="https://www.mega-bank.com/transfer" method="POST">
  <input type="hidden" name="to_user" value="hacker">
  <input type="hidden" name="amount" value="10000">
  <input type="text" name="username" value="username">
  <input type="password" name="password" value="password">
  <input type="submit" value="Submit">
</form>
```

In this example, the user will see a login form—perhaps to a legitimate website. But when the form is filled out, a request will actually be made against MegaBank—no login attempt to anything will be initiated.

This is an example of how legitimate-looking HTML components can be used to send requests taking advantage of the user's current application state in the browser. In this case, the user is signed into MegaBank, and although they are interacting with an entirely different website, we are able to take advantage of their current session in MegaBank to perform elevated operations on their behalf.

This technique can also be used to make requests on behalf of a user who has access to an internal network. The creator of a form cannot make requests to servers on an internal network, but if a user who is on the internal network fills out and submits the form, the request will be made against the internal server as a result of the target user's elevated network access.

Naturally, this type of CSRF (POST) is more complex than seeding a CSRF GET request via an `<a></a>` tag—but sometimes you must make an elevated request against a POST endpoint, in which case forms are the easiest way of successfully making an attack.

Bypassing CSRF Defenses

Depending on the application you are targeting, it is possible that anti-CSRF defenses can be bypassed with little effort on the attacker's part.

TIP

Remember, any request from the browser can be forged. Tools like Burp, Zap or even curl can be used to send requests with falsified headers, content type, or body.

Not all validators can be bypassed using the same techniques. Therefore, it's important to try a variety of options and document the testing progress in order to gain a deeper understanding of an application's validation workflow.

Header Validation

CSRF defenses that rely on the validation of headers such as `referrer` or `origin` may be bypassable via the elimination of those headers, leading the server to read a `null` or `undefined` during its validation step. In some cases this can be done by adding `rel=noreferrer` to `<a>` and `<form>` tags. In the case that `rel=noreferrer` can be added to one of these tags, the browser will automatically omit the `referrer` header. An example is shown here:

```
<a href="https://example.com/update_password?password=123"
rel="noreferrer">click me to update password</a>
```

As you progress through this book, you will note that I do not advocate using headers as a first-line security measure. HTTP headers are easily forged, bypassed, or altered, which is surprising given how often they are used in implementing security mechanisms.

Token Pools

Some legacy CSRF defense tools evaluate tokens within "pools" versus on a case-by-case basis. In these systems, it is possible to generate a legitimate CSRF token from within your own user account and attach it as part of a CSRF attack, hence validating the request successfully.

Legitimate tokens can often be taken directly from the browser developer tools by analyzing a specific HTTP request initiated from the web application on your own testing account. Tokens can be later attached to requests in a variety of ways, such as by a `curl` request from the Linux or macOS terminal:

```
curl https://website.com/auth
  -H "anti-csrf-token": "12345abc"
```

Some servers will attempt to reject requests that are not structured similarly enough to legitimate requests. So some effort may be required to emulate the structure and data of a valid request.

Weak Tokens

CSRF tokens that are generated in a predictable manner, such as the use of dates, times, usernames, or iterative integer values, can be emulated and forged. Upon evaluating an application for weaknesses that uses anti-CSRF tokens, look carefully at the structure of the CSRF token generated.

For example, consider the following weak CSRF tokens: `1691434927`, `1691434928`. Upon further evaluation you can determine that these CSRF tokens are actually Unix epoch timestamps (they represent the number of milliseconds that have passed since 00:00:00 UTC on January 1, 1970). By understanding the mechanism by which these tokens are generated, you can begin forging similar tokens that will be accepted by the server-side validator that generated them. In this particular example, a CSRF payload sent over the network on `08/07/2023` at `8:04:36 PM` would validate successfully with a forged CSRF token that contains the content `1691438676`.

Content Types

If the server expects a specific content type, such as `application/json`, it may act differently if you send a payload with `application/x-url-encoded` or `form-multipart`. These alternate content types may skip the validation step and allow your CSRF attack to succeed.

In rare cases, the following content types may bypass validation:

- `application/x-7z-compressed`
- `application/zip`
- `application/xml`
- `application/xhtml+xml`
- `application/rtf`
- `application/pdf`
- `application/ld+json`
- `application/gzip`
- `text/csv`
- `text/css`

Content type is declared as an HTTP header on all requests:

```
Content-Type: text/html; charset=utf-8
Content-Type: multipart/form-data;
```

HTTP forms include the content type as an attribute, `enctype`:

```html
<form action="/" method="post" enctype="multipart/form-data">
  <button type="submit">Submit</button>
</form>
```

In the HTTP form case, simply opening the browser developer tools and editing the `enctype` field will force the browser to modify the content type it sends alongside its HTTP request upon form submission.

Regex Filter Bypasses

If the server validation makes use of regex to detect CSRF payloads, alternate methods of writing the URL may bypass said regex. Some examples are as follows:

- Some servers accept semicolons (;) in addition to question marks (?) when defining where to start reading query params. For example, the query *https://example.com?test=123* could be written as *https://example.com;test=123*.
- Some servers will accept either forward or backward slashes when defining file locations, e.g., *https://example.com/test* or *https://example.com\test*.
- Improperly configured servers will accept *relative* location symbols (`..`). For example, *https://example.com/../test*.

It can be difficult to determine the specific regular expression a server is using for validation, so when in doubt, first attempt bypasses against the most common validator regular expression found on search engines. Many developers do not design their own regular expressions.

Iframe Payloads

Rather than making use of an image, script, or link, it is possible to produce CSRF via the iframe `src` attribute, which automatically loads and requires no user interaction. The method to do this is simple, as shown in the following:

```html
<iframe src="https://example.com/change_password?password=123"></iframe>
```

When loading into the DOM, the iframe will immediately make an HTTP request to the value within the `src` attribute. Note that iframe CSRF only works with HTTP GET endpoints.

AJAX Payloads

In the event that script execution is possible, asynchronous JavaScript and XML (AJAX) requests that are capable of operating as CSRF payloads can be made from within the JavaScript execution context. This looks like the following:

```
const url = "https://example.com/change_password?password=123";
const xhr = new XMLHTTPRequest();
xhr.open("GET", url);
xhr.setRequestHeader("Content-Type", "text/plain");
xhr.send();
```

This is often found on websites that allow user customization. Otherwise, it's rare to find it in the absence of an XSS vulnerability also existing on that website.

Zero Interaction Forms

If script execution is possible, CSRF can be performed within a form with no user interaction required. Instead, a JavaScript payload (typically obtained via XSS) will emulate the user interaction with the form via a DOM API:

```
<form id="pw_form" method="GET" action="https://example.com/change_password">
 <input id="pw" type="hidden" name="password" value="" />
 <input type="submit" value="submit"/>
</form>

<script>
  // obtain references to the form
  const el = document.querySelector("#el")
  const pw = document.querySelector("#pw")

  // change the password field
```

```
    pw.val = "new_password_123"

    // submit the form
    el.submit()
  </script>
```

For the most part, all DOM interactions can be invoked via JavaScript API calls where JavaScript script execution is possible.

Summary

CSRF attacks exploit the trust relationship that exists between a web browser, a user, and a web server/API. By default, the browser trusts that actions performed from the user's device are on behalf of that user.

In the case of CSRF, this is partially true because the user initiates the action but does not understand what the action is doing behind the scenes. When a user clicks on a link, the browser initiates an HTTP GET request on their behalf—regardless of where this link came from. Because the link is trusted, valuable authentication data can be sent alongside the GET request.

At its core, CSRF attacks work as a result of the trust model developed by browser standards committees like WHATWG. It's possible these standards will change in the future, making CSRF-style attacks much more difficult to pull off. But for the time being, these attacks are here to stay. They are common on the web and easy to exploit.