Protecting forms from CSRF

The web, unfortunately, is full of bad actors who consistently try to exploit any insecurities that a web application might have. This reading will talk about one common attack called Cross Site Request Forgery (CSRF).

In this reading, you will learn:

1. How to use the `csurf` middleware to embed a token value in forms to protect against CSRF exploits.

CSRF explained

Let's explain what CSRF is with an example. Imagine that you are a customer at a bank called "Bad Bank Inc.". To put it bluntly, this bank sucks, and their website is full of security issues.

In any case, you decide one day that you need to send your brother some money, so you go `http://badbank.com` and sign into your account. Once you have provided the correct credentials to log in, `http://badbank.com` sends back a cookie.
Brief overview of cookies: *At a super high level, when a user logs into a website, one way that the server can "log in the user" is by sending back a cookie to the client. For example, if you log to `facebook.com`, `facebook.com`'s server would send the browser back a cookie. Now, on every subsequent request to `facebook.com`, the browser would attach that cookie to the request. When the request arrives at the server, the server sees the cookie and sees that you're logged in and authorized to navigate around your account.*
Now that you're logged in, you navigate to `http://badbank.com/send-money`, which renders a a page that looks like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Bad Bank</title>
  </head>
  <body>
    <h1>Send Money</h1>
    <form action="/send-money" method="post">
      <label for="recipient">Recipient Email: </label>
      <input type="email" id="recipient" name="recipient" />
      <label for="amount">Amount: </label>
      <input type="number" id="amount" name="amount" />
      <input type="submit" value="Send Money" />
    </form>
  </body>
</html>
```

The page has a form where you can fill out a "recipient" field and an "amount" field. You fill out your brother's email `joe@gmail.com` in the recipient field and \$100 for the amount, and then hit the 'Send Money' button.

When you hit the 'Send Money' button, the following happens:

1. An HTTP POST request is made to `http://badbank.com/send-money`. When you logged in earlier, your browser received a cookie and stored it in association with `http://badbank.com`. Now, your browser sees this "send money" request going to the `http://badbank.com` domain, so it attaches that cookie to the HTTP request.
2. The request arrives at the server. The server sees that there is a cookie, and it checks the cookie.
3. Since the cookie is valid, the server knows that you are logged in, and the server processes the form data to see who you're sending money to and how much you are sending.
4. The server finishes processing the form data and sends \$100 to your brother Joe.

Things are going well so far!

Unfortunately, a devious hacker comes along. The hacker puts up another website that looks like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>See Cute Puppies</title>
    <style>
```

```
      form {
        visibility: hidden;
      }
      input[type="submit"] {
        visibility: visible;
      }
    </style>
  </head>
  <body>
    <form action="http://badbank.com/send-money" method="post">
      <label for="recipient">Recipient Email: </label>
      <input
        type="email"
        id="recipient"
        name="recipient"
        value="hacker@gmail.com"
      />
      <label for="amount">Amount: </label>
      <input type="number" id="amount" name="amount"
value="1000000" />
      <input type="submit" value="See the cutest puppies!" />
    </form>
  </body>
</html>
```

Let's break down what's going on in the hacker's website:

1. First, the hacker puts up the exact same "Send Money" form on his website: it hits the same endpoint (`http://badbank.com/send-money`) with the same method ("post"), and it has all the fields that the endpoint expects when parsing the form ("recipient" and "amount").
2. One difference here is that the hacker hid the form on the website with CSS by setting the form's visibility to hidden.
3. The other key difference is that the hacker went ahead and pre-filled the recipient field's value to his own email address, "hacker@gmail.com", and then also pre-filled the "amount" field's value to 1 million.
4. The only part of the form that the hacker decides to show is the submit button, and he changed the button text to an irresistible prompt: "See the cutest puppies!".
5. Naturally, you love puppies, so as you're browsing the web and land on this hacker's website, you click the button, thinking that you're about to see puppies.

6. Instead, a "post" request gets sent to `http://badbank.com/send-money` along with the pre-filled form data.

Here's the problem, because you had recently logged into `http://badbank.com`, your browser is currently storing the cookie to keep you logged in. When the browser sees that you are making another request to the `badbank.com` domain, it attaches the same cookie to the request that the hacker tricked you into making.

Now, when the hacker's request makes it to `badbank.com`'s server, it sees the cookie, sees that you're logged in and thinks that it's you making the request, so it sends \$1 million to "hacker@gmail.com".

Preventing CSRF

One foundational strategy to prevent a CSRF attack would be to have your server render a secret token as part of the form. Then, when the form gets submitted, it checks for the secret token to verify that it actually came from a form that the server itself had rendered, and not from some other malicious source.

Now, when a hacker tries to imitate the form on his own website, his form wouldn't have the secret token, and the server would know to reject any requests from that malicious form.

Let's pick up where we left off in our previous reading's "Guest List" example and walk through how to implement this CSRF token strategy.

Example setup

At this point, here's what the `index.js` file for the "Guest List" example project should look like:

```
const express = require("express");
```

```javascript
// Create the Express app.
const app = express();

// Set the pug view engine.
app.set("view engine", "pug");
app.use(express.urlencoded());

const guests = [];

// Define a route.
app.get("/", (req, res) => {
  res.render("index", { title: "Guest List", guests });
});

app.get("/guest-form", (req, res) => {
  res.render("guest-form", { title: "Guest Form" });
});

const validateGuest = (req, res, next) => {
  const { fullname, email, numGuests } = req.body;
  const numGuestsNum = parseInt(numGuests, 10);
  const errors = [];

  if (!fullname) {
    errors.push("Please fill out the full name field.");
  }

  if (!email) {
    errors.push("Please fill out the email field.");
  }

  if (!numGuests || numGuests < 1) {
    errors.push("Please fill out a valid number for number of
guests.");
  }

  req.errors = errors;
  next();
};

app.post("/guest", validateGuest, (req, res) => {
  const { fullname, email, numGuests } = req.body;
  if (req.errors.length > 0) {
    res.render("guest-form", {
      title: "Guest Form",
      errors: req.errors,
      email,
      fullname,
      numGuests
    });
    return;
  }

  const guest = {
    fullname,
    email,
    numGuests
  };
  guests.push(guest);
  res.redirect("/");
});

// Define a port and start listening for connections.
const port = 8081;

app.listen(port, () => console.log(`Listening on port
${port}...`));
```

Here's what the `index.pug` file looks like:

~~~
extends layout.pug

block content
  table
    thead
      tr
        th Full Name
        th Email
        th # Guests
    tbody
      each guest in guests
        tr
          td #{guest.fullname}
          td #{guest.email}
          td #{guest.numGuests}


Here's what the `guest-form.pug` file looks like:

```pug
extends layout.pug

block content
  if errors
    div
      ul
        each error in errors
          li #{error}
  h2 Add Guest
  form(method="post" action="/guest")
~~~

```
    label(for="fullname") Full Name:
    input(type="fullname" id="fullname" name="fullname"
value=fullname)
    label(for="email") Email:
    input(type="email" id="email" name="email" value=email)
    label(for="numGuests") Num Guests
    input(type="number" id="numGuests" name="numGuests"
value=numGuests)
    input(type="submit" value="Add Guest")
```

Using the `csurf` library

Let's use the `csurf` library to handle this whole process of creating a secret token for the form and then checking the secret token when forms are submitted.
The `csurf` library creates a middleware function that does the following:

1. It creates a secret value, which is sent to the client and stored as a cookie named `_csrf`.
2. On every request for a form, a CSRF token is generated from that secret value. That CSRF token is then sent back to the client as part of the form in a hidden input field.
3. Whenever the client submits the form, the server checks the CSRF token that's embedded in the form and verifies that it is a valid CSRF token by checking it against the secret `_csrf` value that was attached to the request as a cookie.

By taking the steps above, the hacker site would not have the CSRF token embedded as part of the form on his site, and therefore it would fail the CSRF token verification process.

Let's implement the above flow in the "Guest List" project. First, start by running `npm install csurf@^1.0.0`.
Then, go ahead and also install the `cookie-parser` middleware by running `npm install cookie-parser@^1.0.0`. Remember, the secret value is stored as a cookie named `_csrf`, so whenever the form is submitted, the server needs to be able to parse out the cookie in order to verify the CSRF token against the secret `_csrf` value.
At the top of `index.js`, require the `csurf` and `cookie-parser` dependencies. Add the `cookie-parser` middleware as an application-wide middleware function. For the `csurf` middleware,

go ahead and create the function now so that you can later use it in specific routes:

```
const express = require("express");
const cookieParser = require("cookie-parser");
const csrf = require("csurf");

// Create the Express app.
const app = express();

// Set the pug view engine.
app.set("view engine", "pug");
app.use(cookieParser()); // Adding cookieParser() as
application-wide middleware
app.use(express.urlencoded());
const csrfProtection = csrf({ cookie: true }); // creating
csrfProtection middleware to use in specific routes

// REST OF FILE NOT SHOWN
```

In the `app.get('/guest-form')` route, pass in the `csrfProtection` function as one of the middleware functions for that route. Then in the route's final callback middleware function, generate a CSRF token by calling `req.csrfToken()`. The `csrfToken()` function was added to the `req` object by the `csrfProtection` middleware. The middleware function also generated a secret `_csrf` value and stored it in the `res` object's headers so that the client can store it as a cookie. Finally, render the CSRF token under a `csrfToken` key so that the `guest-form.pug` template can use it:

```
app.get("/guest-form", csrfProtection, (req, res) => {
  res.render("guest-form", { title: "Guest Form", csrfToken:
req.csrfToken() });
});
```

In `guest-form.pug`, add a hidden input field with a `name` attribute of "_csrf" and the `value` attribute set to the `csrfToken` that the server renders:

```
extends layout.pug

block content
  if errors
    div
      ul
        each error in errors
          li #{error}
  h2 Add Guest
  form(method="post" action="/guest")
```

```
    input(type="hidden" name="_csrf" value=csrfToken)
    label(for="fullname") Full Name:
    input(type="fullname" id="fullname" name="fullname"
value=fullname)
    label(for="email") Email:
    input(type="email" id="email" name="email" value=email)
    label(for="numGuests") Num Guests
    input(type="number" id="numGuests" name="numGuests"
value=numGuests)
    input(type="submit" value="Add Guest")
```

Finally, back in `index.js`, pass in the `csrfProtection` function to `app.post('/guest')`:

```
app.post("/guest", csrfProtection, validateGuest, (req, res) => {
  // REST OF CODE NOT SHOWN
}
```

Now whenever the form is submitted, the `csrfProtection` middleware function verifies that the CSRF token that was set in the hidden input field is a valid token by checking it against the secret `_csrf` value that was sent as a cookie.

In summary, now, if a hacker tries to add a guest to your guest list, his form would be missing the CSRF token. Therefore, the endpoint throw an error and prevent a guest from being added.

There are other considerations to take into account to fully protect your web applications from CSRF attacks. For now, adding the CSRF token is a solid first line of defense.

What you learned

In this reading, you learned how to use the csurf middleware to embed a token value in forms to protect against CSRF exploits.

This reading concludes this lesson on HTML Forms, and you're now ready to build out your own Express application that uses forms!