Secure User State

Signed Cookies & Sessions

The Problems With Cookies

- Cookies live on the client's computer, so they're insecure
  - Anyone can add, remove, or alter cookies on their computer
- This means that if we store sensitive data (passwords, dev-only info) that anyone could read it in plain-text
- And if we use the cookie server side, anyone could set the data to anything, so we can't trust *who* sent it
- So how do we trust the client with secure data?

Solution #1: Signed Cookies

- On the server side alone, we can set special cookies known as **signed cookies**
- These cookies are still readable by the user, but can't be written
- We need to provide `cookieParser` with an argument that is a secret code that "signs" the cookie using encryption
- Our server can later read this cookie by comparing the signature and making sure it's valid
- These cookies are then available under `req.signedCookies`, instead of `req.cookies`
- **Make sure you check `req.signedCookies`**, because a `req.cookies` key of the same name can still be set client-side

Signed Cookie Example (Setup)

```
// Setup cookieParser with a secret code to sign our cookies.
// Provide a default in case the developer forgets to add a
// secret code to .env
const cookieParser = require("cookie-parser");
app.use(cookieParser(process.env.COOKIE_SECRET || "dev"));
```

Signed Cookies Examples (Setting)

```
// Login form POST submit
app.post("/login", function(req, res) {
    // A promise that checks that the username and password
    // are correct. Resolves if correct, rejects if incorrect.
    User.login(req.body.username, req.body.password)
        // Save the cookie using `signed: true` if correct
        .then(function() {
            res.cookie("username", req.body.username, {
                signed: true,
                maxAge: 7 * 24 * 60 * 60 * 1000,
            });
            res.redirect("/home");
        })
        // Re-render the form with an error message if incorrect
        .catch(function() {
            res.status(403).render("login", {
                error: "Bad username / password",
            });
        });
});
```

Signed Cookies Example (Reading)

```
// If user has signed cookie username, let them through.
// Otherwise, redirect them to login form.
function loggedInMW(req, res, next) {
    if (req.signedCookies.username) {
        next();
    } else {
        res.redirect("/login");
    }
}

// Require user to have logged before accessing this page.
app.get("/home", loggedInMW, function(req, res) {
    res.render("home", {
        username: req.signedCookies.username,
    });
});
```

Quick Aside: Cryptography & Hashes

- So how does our server know our signed cookie is "legit"?
- The cookie is "signed" with a hash of its name and value, and our secret string
    - A hash is a way of generating a unique key to identify a large amount of data
- Because the client doesn't know our secret string, it can't make a valid hash of any name / value it wants
- This is why it's important to keep that string SECRET!

Signed Cookies Exercise: Save Login

- Remember the `pwform` from a few classes back we used to secure `/traffic`?
- Don't worry if you lost it, go ahead and download the project here:

https://github.com/wbobeirne/nycda-express-middleware

- We'll want this middleware to use a signed cookie instead of checking the query:
  - Set up the `cookie-parser` middleware with a `COOKIE_SECRET` env var
  - Upon successfully typing in the password, you should save a signed cookie under the key `authenticated` to something truthy (`true` is fine)
  - In addition to checking `req.query.password`, if that cookie is set, you should also `next()` in the middleware
- Test to make sure it only requests the password once per session. You can use Chrome's incognito window to more easily test this.

Solution #2: Sessions

- With signed cookies, we're still making the data readable on the user's side
- We can also still only store simple strings
- Instead of that, we can use **sessions**
- Sessions use signed cookies to save one of those hashes
- That hash is a "key" that "unlocks" a Javascript object on our server
- That object can store all the secret data it wants, and have it stay on the server, the client needs to only hold on to the hash key

Using the `express-session` Module

```
# Terminal
npm install --save express-session
```

```
// app.js
const session = require("express-session");
app.use(session({
    // Must be the same secret as cookie-parser
    secret: PROCESS.ENV.COOKIE_SECRET || "dev",
}));
```

- Instead of implementing this all ourselves, we'll use the `express-session` middleware to handle most of this for us
- Setting up the middleware is very similar to `cookie-parser`, only this **requires** a secret key since it uses signed cookies
  - This key should be the exact same as `cookie-parser`
- Once it's setup, you'll have access to `req.session`

Sessions Example (Setting)

```
app.post("/login", function(req, res) {
    User.login(req.body.username, req.body.password)
        // Save the session variable "username" on success
        .then(function() {
            req.session.username = req.body.username;
            res.redirect("/home");
        })
        // Render login form with error on failure
        .catch(function() {
            res.status(403).render("login", {
                error: "Bad username / password",
            });
        });
});
```

- Unlike `res.cookie`, `req.session` doesn't use functions, you simply set it
- Everyone gets a session
- It will be saved between requests automatically

Sessions Example (Getting)

```
// If user has session username, let them through.
// Otherwise, redirect them to login form.
function loggedInMW(req, res, next) {
    if (req.session.username) {
        next();
    } else {
        res.redirect("/login");
    }
}

// Require user to have logged before accessing this page.
app.get("/home", loggedInMW, function(req, res) {
    res.render("home", {
        username: req.session.username,
    });
});
```

- As simply as it was set, we can get things from `req.session` directly

Session Exercise: Traffic Session

- In addition to altering the `pwform.js` middleware, we'll also want to upgrade the `traffic.js` middleware to use sessions
- Let's track traffic on a *per user basis*, in addition to the all traffic tracking
  - Set up the `express-session` middleware with the `COOKIE_SECRET` env var
  - In addition to the variables, you should also initialize / increment `req.session.totalTraffic` and `req.session.pathTraffic`
  - You should also update the `/traffic` page render to show the session traffic for the current user as well
- Try browsing around as a few users using Chrome's incognito mode to make sure that all users increment the `req.*Traffic` values, but only increment their own `req.session.*Traffic` values.

So Which Should We Use?

- You were taught both because sessions are a superset of signed cookies
  - Anything you can do with signed cookies, you can do with sessions
  - But it's important to understand how sessions work using signed cookies
- However, sessions allow us to do more since we're not constrained by cookies:
  - Storing larger, complex data, not just small strings
  - Storing private data that shouldn't live on the client
  - Sharing sessions between multiple browsers by using the same key
  - Storing the session in something persistent
  - We'll discuss these more in the future
- So for our projects, **use sessions instead of signed cookies**
- But you should continue to use regular cookies that are set on the client side

Additional Reading

- Session Management in Express - A very simple example of sessions in Express
- Encryption 101: Understanding the Basics - Not required, but explains how cookies are signed, why we need a "secret", and why users wouldn't be able to modify them.
- Express's `req.signedCookies` docs
- `express-session`'s Docs