Prerequisites

- If you would like to complete the packet sniffing demo, you'll need WireShark

Goals for this recitation

1. Understand how common security exploits are carried out on the web
2. Learn how to mitigate these attacks within Express

Security Concerns (7 minutes)

Introduction (2 minutes)

Prior to this project, security mostly just involved ensuring that proper authentication was required for a user to access resources located on protected routes. However, as covered in lecture last week, we'll now be looking at attacks where attackers have sophisticated tools at their disposal to attack your application not necessarily based on the code you've written, but instead finding weaknesses in the protocols and tools you've chosen to use.

In your documentation for the final project, you will be expected to complete the security concerns portion of the design. You'll have to explain three pieces for this portion: A summary of your security policy, how you'll mitigate standard attacks, and the threat model for potential attackers.

Threat Model (5 minutes)

When building a secure web application, you will need a security policy, or a set of goals for what behavior you intend to support. After defining valid behavior, you can generate threat models, or sets of assumptions about the capabilities of your attackers. Threat models can help you understand what steps you need to take to secure your application.

Here's a brief example for a note application:

Security Policy

- Authors of a note have read, write, delete to their own notes
- Only authors can share a note with users with read or edit privileges
- Other than the author, only shared users can read or edit notes

Threat Model
- An unauthenticated user can construct requests (curl, Postman) to eavesdrop or tamper with other users’ notes
- A user with basic credentials can construct requests or make requests via forms
- It’s unlikely much sensitive information will be stored on a note-taking application, so the likelihood of attracting interest from highly sophisticated attackers isn’t high

Exercise: Define a security policy and threat model for the Fritter application

Exploits, Demos, Mitigations (28 minutes)

Use the repo located at https://github.com/evanwang22/6170-r9 for all demos below. Note that the app must be running on localhost:3000, and you’ll need to run NPM install every time you check out a new branch.

SQL Injection (~7-8 minutes)

In a traditional SQL Injection, an attacker can modify your application’s database by including SQL commands in inputs. When your application make database queries with these inputs it interprets the inputs as SQL commands and executes them on your data.

Although this is traditionally an issue that affects SQL databases, it is very possible to perform a similar attack on a MongoDB database as well, as the following demo will show.

INSECURE DEMO: Check out branch injection

1. Create an account in the application via signup (form is located at ‘/users/new’) and then log in using that account (form is located at ‘/sessions/new’).
2. Verify that you are indeed logged in (a successful login is pretty obvious). Now, log back out.
3. Go to the login page, don’t enter any credentials, and click the red “inject” button. Wait for a second, and then navigate to the homepage (‘/’). Lo and behold, you’ll now be logged in!

The “inject” button sends an ajax request; look at the code in an editor or on the sources tab of the web console for the code. Notice that the username and password fields have been replaced with objects -- `{"$gt": ""}` to be specific. This is a valid MongoDB comparison operator, and will check to see if the username and password are greater than the empty string, which will always return true. Therefore, the MongoDB query will return the first object in the users collection.
In the above case we have bypassed a login check. The weakness in our application could also be exploited in other ways to cause extensive damage to the app and database -- a quick search will turn up several ways to exploit an injection weakness in MongoDB.

SECURE DEMO: Check out branch injection-fixed

Follow the above steps again, and you'll find that the injection no longer works. To solve this issue, we must sanitize our query inputs. In the example app, we've performed a check on the type of the inputs -- if they are objects, we'll cast them to strings, which protects us against the exploit. The validator.js module is a good choice for performing these types of sanitizations.

XSS (~7-8 minutes)

In an XSS attack, the attacker inserts malicious code into the web page. The browser trusts that the code came from the site to which it made the request. The attacker exploits this trust leading to the code running locally on victim's client. There are two main types of XSS: persistent and non-persistent.

Persistent XSS
Persistent XSS usually takes place via code injection. This is when an attacker provides code in an input to the web application, usually discretely, such that the code is saved within the application. Subsequent visits to the web application lead to this injected code being served along with the web page.
Non-Persistent XSS

Non-Persistent XSS involves the attacker inserting malicious code into a page without the web application saving it. One of the ways an attacker can do this is to construct a link to application with the attacking code in the query string parameters that are used when rendering the page. Another way an attacker can achieve this is by actively hijacking insecure requests made by the application and returning the attacking code in the response.

INSECURE DEMO: Checkout branch xss

1. Navigate to the notes page (located at ‘/notes’)
2. Hit the “new note” button and create a new note with a title and some text. It should just redirect you to the notes page on success.
3. Add another new note. This time, set the following as the text: 
   "<script>alert("hi!");</script>". Once you return to the notes page, you’ll get an alert!

This is a very simple example of persistent XSS in action – the script was saved to the database, so anytime anyone loads the page the script will run. There is obvious danger resulting from this scenario. An alert is pretty innocuous, but you can imagine an attacker performing something more malicious such as running a computation within your browser or stealing your cookies.

SECURE DEMO: Checkout branch xss-fixed

Follow the above steps again, and you’ll find the XSS attack no longer works. Both EJS and Handlebars HTML-escapes values using the syntax we’ve shown in class; “<%=” for EJS and “{{“ for Handlebars. This behavior ensures safety from XSS attacks. There are very few occasions on which you will need to use unescaped HTML, which can be accessed with “<%-” in EJS and “{{“ in Handlebars. These unescaped tags open your application up to XSS vulnerabilities. In other words, you’re already protected from this vulnerability in both persistent and non-persistent forms as long as you properly escape rendered content. Don't change what you've been doing.

CSRF (~7-8 minutes)

Cross Site Request Forgery (CSRF) is an attack where based on the request the server believes a legitimate client is making a request to the server, but it is actually an attacker is making the request on the user’s behalf. These attacks are generally executed by including a request to the target site on a separate website. When the browser makes the request, it may use the cookies for the target site. This means the attacker can use the victim user’s session to masquerade as the user and make requests on his behalf in an act known as session riding.
INSECURE DEMO: Checkout branch csrf

1. Log in to demo site and go to the notes page
2. In another tab, open “csrf_example.html”, located in the root directory of the sample app.
3. Wait a few seconds, then refresh the notes page. Looks like you’ve been saying naughty things about 6.170 …

View the source of “csrf_example.html”. You’ll note that there’s a bunch of hidden forms which are automatically submitted to the site at intervals, and the behavior of submitting those forms is changed so there’s no page reload or other affordance to indicate forms have been submitted. When you refresh the notes page though, it’s obvious that the other site was making requests to our r9 app.

SECURE DEMO: Checkout branch csrf-fixed

Follow the above steps again, and you’ll find this attack doesn’t work. Note that in fact, the only form you can interact with on the app itself is new note creation. Login and signup both no longer work. Using the csrf NPM module, we require that any form submitted contains a CSRF token, and the login and signup forms do not have them. It is very easy to add the tokens to a form (you’ll only need to change two lines of code). AJAX requests will also require CSRF tokens -- the links below show how you can store the tokens in the browser’s cookies so that AJAX requests may access and use the tokens.

Resources: http://maximilianschmitt.me/posts/tutorial-csrfs-express-4/, https://github.com/expressjs/csurf

Session Hijacking & Packet Sniffing (~3-4 minutes)

An attacker can listen to unencrypted network traffic and learn secret data like passwords or session keys. The attacker can then use the data to launch another attack.

PACKET SNIFFING DEMO: No need to check out a new branch; follow instructions in Wireshark document.

The demo for this attack may be fairly interesting, but is a much less relevant attack vector for the upcoming final projects. Attacks like these may be mitigated by either browsing on a secure network (such as MIT secure) or using the HTTPS protocol. In order to use HTTPS, you will need an SSL certificate which is beyond the scope of what’s expected of the final project.

Model Unit Tests (15 minutes)
Model + Controller separation (3 minutes)

Earlier in the term, we mentioned how Mongoose makes it easier to create a degree of Model + Controller separation. Moving into the final project, we expect all teams to incorporate this design pattern during the next phase of the class (even if you’re not using Mongoose or Express).

Here’s a quick refresher of how we can do this:

```javascript
var theaterSchema = mongoose.Schema({
  _id: Number,
  name: String,
  location: String,
});

theaterSchema.methods.getDescription = function() {
  return this.name + " - " + this.location;
};
```

As a rule of thumb, we want to have **skinny controllers whenever possible**. Therefore, if there is code that could either go in the controller or the model, it should be placed in the model. If there is code that changes the state of a resource in the database, it should go in the model.

Unit tests (12 minutes)

A big advantage of this style of Model-Controller separation is that it becomes much easier to write unit tests. Controller methods involve relatively complicated interactions with different modules -- an HTTP request is the input, it may interface with the database or session state, and it responds with data or a render call. On the other hand, model methods take well-defined inputs and conclude return well-defined outputs, so they are much easier to test.

We’ll be using the Mocha testing framework ([http://mochajs.org/](http://mochajs.org/)) for our examples today, but you can use whatever test frameworks you’d like for your project. Note that we don’t care how you test your code, just that it’s properly tested, but once you understand model tests, they are generally easier to write and more reliable than AJAX tests using QUnit on the frontend and give you the benefit of checking that you have proper controller+model separation

DEMO: [https://github.com/evanwang22/6170-mongoose-demo/tree/master](https://github.com/evanwang22/6170-mongoose-demo/tree/master), checkout branch unit-test. You will need to run the following commands once you do sol:

```bash
npm install -g mocha
npm install should
```
The above two modules give us the needed functionality for unit tests. Mocha is a framework which takes care of many of the gritty details such as allowing for asynchronous database accesses. Should is an assertion framework which reads very easily.

Now, running npm test will yield two passing tests for the theater model. Examine the code in ‘tests/data/theater.js’ -- in particular, take note of the beforeEach, afterEach, and done() methods.

beforeEach - run before every test in the file. In this case, we connect to the database and wipe it clean of any data from previous tests. If you have a large set of resources you’d like to insert into a collection before testing, this is a good place to run the insert commands.

afterEach - code run after every test in the file. In this case, all we do is close the database connection.

done() - this method, when present in your callback function, tells Mocha that you are writing an asynchronous test. This causes mocha to enter a timer when the function with the “done” parameter runs, waiting for the async function to finish -- which is facilitated by either calling the `done()` function or exceeding a 2 second timeout. The basic idea behind `done()` is that you call this after your async code has completed, and your test has modified everything it needs to modify, so you can check the results correctly.

**Exercise:** Add another test to theater.js, following the same format as the given tests. What functionality would be more interesting to test than what we have currently? Could we add more model methods to Theater in data.js so that we can test as much of the functionality of Theater as we can?