Prerequisites
Have ROR 4.0 setup. For packet sniffing demo, install WireShark.
Demo App: https://github.com/6170-fa13/r9

Goals
- Understand how common security exploits are carried out
- Learn tools built into Rails to help mitigate these attacks

Overview

Security
When trying to make your web application secure, you need to come up with 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 will help you figure out what steps you need to take to secure your application.

Example: 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 application
- A user with basic credentials can construct requests or make requests via forms
- A third party website can try to load data from application

**Basic Attacks**

**SQL Injection**
The attacker can attempt to 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. Injected SQL commands such as `DROP TABLE users` can wreck havoc on your database and lead to one of the worst possible scenarios for a developer, data loss.

**XSS**
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.
DEMO
Include script in a note. For example put `<script>alert("hi!");</script>` in the content of the note. Refresh the show page for the note and see the script execute.

Note: this will not typically work for reasons covered later

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.

CSRF
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 on separate website a request to the target site. When the browser makes the request, it uses 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.

Packet Sniffing
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.

Rails Security Features
As a web application framework, Ruby on Rails has several security measures built-in to make development easier.

Strong Parameters/Mass Assignment (XSS)
Rails provides several useful functions to set several attributes in a model by passing along the params hash from the controller. Unfortunately, this can be a dangerous practice if we are not careful about which values we allow to be set in the model. For example, suppose you have a User model with the functionality to support admin users. The way you implemented admin users is by having a boolean attribute in your User model that reflects whether a User instance is an admin. Although you don’t include a field on your edit or new pages for the admin attribute of user, an attacker can construct a post request that includes `admin=true` in the user parameters. Rails takes a whitelist approach to passing parameters from the controller through to the model for mass assignment. You can use Rails Strong Parameters to whitelist a specific set of parameters for use in mass assignment.
You’ve probably seen something similar to

```ruby
params.require(:model_name).permit(:attr1, :attr2)
```

`require` requires that the params hash contains a key-value pair with key `:model_name`

`permit` allows attributes to be mass assigned

```ruby
params.permit(:attr1, :attr2)
```

**Escaped HTML (XSS)**

Since Rails 3, strings rendered in your views are automatically HTML escaped by default. You can use the `raw` method to render the unescaped string.

**Escaped**

```html
<% = "<script> alert('hello there!'); </script>" %>
```

**Unescaped**

```html
<% = raw "<script> alert('hello there!'); </script>" %>
```

Revisit `app/views/notes/show.html.erb` to see this in the demo app.

**Automatically Sanitized Active Record Queries (SQL Injection)**

In order to make database queries safe and easy, Rails has several query functions that automatically sanitize inputs depending on how they are provided. For example,

```ruby
Model.where("attr1 = ? AND attr2 = ?", val1, val2)
```

escapes the inputs `val1` and `val2` when constructing the query. However, if you construct the query string directly and do not escape the inputs as in

```ruby
Model.where("attr1 = '#{val1}' AND attr2 = '#{val2}'")
```

then there is risk of SQL injection. **When possible, use ActiveRecord methods that format strings and sanitize inputs for database queries.**

**CSRF Security Token (CSRF)**

Adding one line to your application controller automatically calculates a security token to prevent CSRF. Rails even adds the line to your application controller automatically.

```ruby
# app/controllers/application_controller
```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgey with: :exception
end

COORs (Optional material)
Allows requests to be made from other origins.

HTTPS and TLS/SSL (Packet Sniffing and XSS)
HTTP Secure (HTTPS) uses Transport Layer Security (TLS) to encrypt requests and responses between the server and client. This prevents attackers from reading the data or performing various Man-in-the-Middle attacks.

Rails makes implementing HTTPS very easy

# config/application.rb
module MyApp
  class Application < Rails::Application
    config.force_ssl = true
  end
end

Additional Resources
For a great book on security in web applications, read The Tangled Web.

If you find these topics interesting, check out 6.858 Computer Systems Security.
Packet Sniffing Demo

1. Start WireShark
2. Select Wifi as interface and press ‘Start’

3. Create filter for packets
   a. Look up ip address of your machine (ifconfig)
   b. Create a filter in WireShark ‘http & ip.dst == XXX.XXX.XXX.XXX && (tcp.dstport == 3000 || udp.dstport == 3000)’

4. Look for incoming packets
5. Hijack session (need to finish)
   a. Get session hash from cookie header
b. Make request as other user `curl -b cookie_key=value XXX.XXX.XXX.XXX:3000`