NCC Group Whitepaper

Going AUTH the Rails on a Crazy Train

A Dive into Rails Authentication and Authorization

November 13, 2015 – Version 1.1

Prepared by

Tomek Rabczak — Senior Security Consultant at NCC Group

Jeff Jarmoc — Lead Product Security Engineer at Salesforce

Abstract

Ruby on Rails is one of the most popular web application frameworks in use today. It's especially popular among startups, but also powers large sites such as Github, Airbnb, Hulu, Square, Indiegogo, Kickstarter, and Basecamp.[1] Penetration Testers and Application Security Engineers are very likely to encounter Rails applications, so a solid familiarity with common Rails issues is warranted.

In this paper, we explore Ruby on Rails Authentication and Authorization patterns and pitfalls. Authentication and Authorization are particularly interesting because while most applications will need these features, the Ruby on Rails framework provides little native support for them. Our experience as Software Security Consultants has shown us first hand how this often leads developers to hastily implement poorly thought out mechanisms. We've often seen similar authentication and authorization implementation issues time and time again in a wide variety of applications. We hope that through this paper, the reader will gain an understanding of common authentication and authorization problems in Rails applications, and be better prepared to exploit or improve such applications, as their role may dictate.

We also introduce a new tool, Boilerman, developed by one of this paper's authors to ease the process of auditing Rails applications for common authorization flaws.

Table of Contents

1 Introduction

1.1 The Rails Way

When learning about Rails development, it won't take long to come across the expression 'The Rails Way.' It's been the title of a series of books[2] about[3] Rails[4] and is a term frequently used in other books[5] and blogposts[6] about Rails. But what is this 'Rails Way?'

David Heinemeier Hansson,[*] the creator of Rails, described his design philosophy of Rails in a blogpost titled Rails is omakase.[7] In the post, he describes Rails as 'Omakase,' a term which is borrowed from Japanese cooking, and translates to "I'll leave it up to you." To order a meal omakase is to delegate your menu choices to the chef, trusting that they are better equipped to make a decision than you yourself are. This too, is generally true of the Ruby on Rails framework. Rails provides a common set of libraries for common tasks, and there is generally one way, 'The Rails Way', to accomplish a given task.

This stands in contrast to other application frameworks which DHH describes as 'à la carte software environments.' In these frameworks, the developer must choose how to implement various functions such as database queries, web page templating, and request routing. This is also the case when it comes to security controls such as output encoding, Cross-Site Request Forgery prevention, and session management.

> " Rails is not that. Rails is omakase. A team of chefs picked out the ingredients, designed the APIs, and arranged the order of consumption on your behalf according to their idea of what would make for a tasty full-stack "
>
> *David Heinemeier Hansson*

Some criticize the Rails approach as limiting, or of favoring certain use cases at the expense of others. Be that as it may, the common format of Rails applications means different applications share many common patterns, easing the process of auditing security posture and enforcing consistent controls.

1.2 Structure of a Rails Application

Rails follows the Model View Controller (MVC) architecture.[8] MVC divides an application into three components:

Models represent objects used by the application. In Rails, models usually inherit from ActiveRecord[9] which provides an Object-relational mapping (ORM) layer, transparently backing them with database representations. ActiveRecord takes measures to limit the likelihood of SQL injection attacks through it's most common finder methods.

Views are responsible for what the end user sees. In Rails, views are handled by ActionView[10] which applies ERB (HTML with embedded Ruby) templating, and provides a number of helper methods to build and format responses. ActionView also performs HTML output encoding, reducing the opportunities for Cross-Site Scripting attacks significantly.

Controllers provide most of the application logic. Requests are processed by Controllers, which take actions on models and respond with views. In Rails, Controllers are implemented by ActionController.[11] ActionController's security features include default enforcement of Cross-Site Requst Forgery tokens, and protection from Mass Assignment by restricting what model attributes can be updated.

[*]Often simply 'DHH'.

1.2.1 The Rails Filesystem

The filesystem of a Rails application reflects its MVC architecture. After creating a new Rails application by running the command `rails new sample_app`, a directory called `sample_app` is created which contains all files related to the new application. Within this tree we see several directories and files, a few of the more notable ones are discussed below.

```
sample_app/
    app/
      models/
      views/
      controllers/
    ...
      config/
          routes.rb
    ...
      Gemfile
      Gemfile.lock
```

Rails `sample_app` Directory Tree

The `app` directory contains further subdirectories which hold files defining the application's model, view, and controller classes. As noted earlier, these are typically implemented through ActiveRecord, ActionView, and ActionController respectively.

The `config` directory stores many different files defining configuration variables used by the application and its libraries. Perhaps most interesting when auditing an application is the `routes.rb` file. This file defines all the application's routes, tying controller actions to user accessible URLs. It's therefore key in understanding the exposed surface of the application.

The `Gemfile` defines a list of Ruby Gems upon which the application depends, and the versions which satisfy its requirement. In contrast, the `Gemfile.lock` provides a record of the exact version of a given gem which were found by Bundler[12] when the application was installed. This allows for creating an exact copy of the application environment by copying `Gemfile.lock` along with the rest of the application code. This is useful for cloning an application between deployment environments, or ensuring that various instances share common versions of gems.

Many other files are created but a full description is beyond the scope of this document. For our purposes, familiarity with the basic structure outlined above is sufficient to begin exploring application functionality.

2 Authentication and Authorization

The remainder of this paper discusses authentication and authorization patterns and pitfalls in Rails. We'll discuss each in its own section. It's important to note that while these are related concepts, authentication and authorization are two distinctly different concepts.

Authentication is solely the process of identifying the user. If you find yourself working with code that seeks to answer the question 'Who is the user?' you're dealing with authentication. Authorization is the process of determining what that user is allowed to do in the context of the application. Authorization logic implicitly requires that we've already authenticated the user to establish their identity, but is otherwise an entirely separate process.

Thinking of authentication and authorization as two discreet processes is the first step to designing reasonably secure systems. In the authors' experience, when developers mix the two concepts, it's a huge red flag and often an indicator of an application which will exhibit security flaws.

3 Authentication

Rails provides little native support for authentication. HTTP Basic and Digest authentication are both supported, but neither is sufficient for the majority of use cases. These methods of authentication require transmission of credentials with every request, prevent the application from storing password digests securely, and for many more reasons. We won't discuss them in detail, but their shortcomings are well known. Realistically, we'll need something stronger which in Rails leaves us with two choices; rolling our own authentication system, or using an off-the-shelf gem.

3.1 Roll Your Own

The first option is to 'roll your own' authentication system. Often, developers believe their requirements are so unusual and application specific, that writing their own system is faster and easier. This is the view espoused by the Ruby on Rails Tutorial which in Chapter 6, 'Modeling Users' states:

> "For one, practical experience shows that authentication on most sites requires extensive customization, and modifying a third-party product is often more work than writing the system from scratch. In addition, off-the-shelf systems can be "black boxes", with potentially mysterious innards; when you write your own system, you are far more likely to understand it. Moreover, recent additions to Rails (Section 6.3) make it easy to write a custom authentication system. Finally, if you do end up using a third-party system later on, you'll be in a much better position to understand and modify it if you've first built one yourself."
>
> *Ruby on Rails Tutorial*

However, this approach has some downsides which we believe outweigh the benefits of this approach.

- When writing our own authentication system, we're essentially re-inventing the wheel. Sure, there's a chance we can do better than what's come before, but given the complexities it's far more likely that we'll introduce flaws that have previously been identified and corrected in other systems.
- There's a lot more to authentication systems than simply storing and checking credentials. In later sections, we'll discuss some of the other needed features in greater detail.

Our experience shows that developers who write authentication systems, much like cryptosystems, may not be well versed in the nuance and the history of attack. Therefore, we recommend integrating an existing authentication system rather than writing your own. While it's true that some integration effort is required, authentication is a fairly routine action, so configuration of strong authentication systems will be minimal.

Writing an authentication system from scratch is a solid learning exercise, and will certainly help the developer to understand some of the nuances of authentication flows. We do not recommend writing your own authentication system for production use without careful consideration and review. However, as an exercise we'll next look at what would be required to write an authentication system from scratch in Rails.

3.1.1 Writing it

We said before that Rails provides little support for authentication. However, since Rails 3.1 the ActiveModel::SecurePassword class and associated has_secure_password helper method are available. These features do not fully implement a complete authentication system, but provide a basic facility for adding secure password storage to a model.

For example, assume a class called `User` with a schema that includes two strings called `name` and `password_digest`. With a class that includes a `password_digest` attribute, we can easily create a secure password storage mechanism by invoking the `has_secure_password` helper:

```
class User < ActiveRecord::Base
  has_secure_password
end
```

A Basic User Model

That small bit of code adds virtual attributes `password` and `password_confirmation` which we can use to set the user's password:

```
user = User.new(:name => "Jeff", :password => "hunter2", :password_confirmation => "hunter2")
user.save # => true
```

The `password_confirmation` attribute is optional, but if included must match the `password` value. This makes the function suitable for use in user-facing views allowing for account creation or password changes.

Notice that `password` and `password_confirmation` attributes don't map directly to the database schema. Instead, they're consumed by 'has_secure_password' which utilizes the 'BCrypt' gem to create a digest of the password, which is then stored in the `password_digest` column.

The application must also include `gem 'bcrypt'` in its `Gemfile` and should configure an appropriate workfactor by setting `BCrypt::Engine::DEFAULT_COST` in the environment config file. If a workfactor isn't configured, the current default is 10 which is a reasonable work factor in many environments, though could be improved on some systems. For more about secure password storage and choosing an appropriate workfactor, you may wish to reference our recent blog post on the subject.

We're also provided with an `authenticate` method on our `User` model. When called with a password's plain text value as a parameter, BCrypt is invoked to process the password and compare its digest to the `User.password_digest` using a secure comparison function. If the values match, the `User` object is returned otherwise, `false` is returned. For example:

```
user.authenticate("hunter2") # => user
user.authenticate("CorrectHorseBatteryStaple") # => false
```

Since the `User` class is backed by ActiveRecord, we can also use any of the available ActiveRecord finders to locate a given instance. This allows us to search for a user by name and attempt to authenticate them in a single line of code.

```
User.find_by_name("jeff").authenticate("hunter2") # => user
```

In many applications, we'll see a sessions controller implement a `create` method, which processes the user's authentication request and performs a similar query using user-supplied values. If successful, we update the session with the `user.id`, thus allowing it to be referenced by authorization checks elsewhere in the application.

```
def create
  user = User.find_by_name(params[:name]).authenticate(params[:password])

  if user
    session[:user_id] = user.id
    redirect_to '/'
  else
    redirect_to '/login'
  end
end
```

app/controllers/sessions_controller.rb

3.1.2 # TODO

In the last section, we saw how simple it can be to write a small authentication function in Rails. But this authentication function is far from a complete authentication system. Just a few of the necessary features that remain to be developed include:

1. Creating / Registering new user accounts
2. Allowing users to change their own passwords
3. Enforcing account name/email uniqueness
4. Enforcing password complexity
5. Session Management
    - Allowing users to log off
    - Timing out inactive sessions
6. Allowing the user to recover from a lost/forgotten password.
7. Possibly other authentication schemes
    - API Tokens
    - Multifactor Authentication
    - OAUTH
    - SAML
    - etc.
8. Email confirmation

As the complexity of such an authentication system grows, so too does the chance for error. Developers tend to be primarily concerned about the business use cases of their application. Ancillary features like authentication, which are required but aren't the main purpose of the application, often don't get the attention they require to implement in a reasonably secure fashion.

3.2 Use a Gem

The alternative is to use an off the shelf RubyGem that provides a ready made authentication system. This provides some immediate benefits:

- Core code is generally well vetted by a larger community of users.
- Past community experience and security events spur ongoing improvements.
- Subtle weaknesses may be entirely avoided, even without being aware of their specifics.

However, there are some downsides:

- Vulnerabilities affect far more applications, which may increase motivation for malicious actors to research weaknesses.
- Ongoing updating and maintainence of the installed code is required, as with any dependency.
- Integration effort is still required and can introduce vulnerabilities.

On balance, it is our opinion that the benefits of deploying a well known authentication system and configuring it for the environment offset the possible risks.

3.2.1 Common Authentication Gems

While there are a wide number of Gems available to handle different authentication cases, few have seen much adoption in the larger Ruby on Rails community. Some of the most notable are:

- Devise - Far and away the most popular authentication system for Rails. Its modular design and wealth of plugins make it easy to adapt to nearly any application's requirements.
- Omniauth - A multi-provider authentication gem, primarily geared toward OAuth authentication with 3rd party authenticators. It can be integrated into Devise, allowing for side-by-side password and OAuth authentication with a common interface.
- DoorKeeper - Turns your application into an OAuth provider. This is most frequently used to provide API access, or to authorize 3rd party applications.
- Authlogic - A simpler authentication solution, far less common than Devise, which relies on a somewhat unique session model.

Since Devise is by far the most popular gem, it's also the solution which we're most likely to encounter when reviewing Rails application security. For that reason, we'll discuss it in further detail.

3.2.2 Installing Devise

The installation process for Devise is simple and well documented. A brief overview will be sufficient for the common case.

First, add Devise to the application's `Gemfile`:

```
gem 'devise'
```

From the command line we install our bundle to load the gem, then run Devise's installation process to include it in the application.

```
bundle install
rails generate devise:install
```

At this point, we should review the Devise initializer, found in `config/initializers/devise.rb`. This file contains most configuration settings for Devise, including module settings and configuration options. If using or auditing an application with Devise, you should at minimum familiarize yourself with this file and the options it contains.

Next, we add devise to a model in our application by running its generator. If the model exists Devise will extend it, otherwise it creates a new model. Let's create a new `User` model for authentication:

```
rails generate devise User
```

Now, let's look at the `User` model in the application to see what Devise has created.

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

app/models/user.rb

Here we see a default Devise generated `User` model, backed by ActiveRecord, which includes the `devise` helper method being called. The symbols passed to `devise` indicate what modules are used. Most of the names are fairly descriptive, and details of each of the ten default modules can be found linked at the very top of the Devise homepage. It may be necessary to enable or disable modules to suit our needs.

Once we've configured Devise settings and adjusted the module configuration for our model, we can update our database with the required fields by invoking `rake` from a shell.

```
rake db:migrate
```

Assuming our module configuration includes flows that rely on email, we'll need to configure ActionMailer so Rails can send emails with appropriately formatted links. This is necessary for modules such as `:recoverable` for forgotten password recovery, or `:confirmable` to confirm email account ownership.

```
config.action_mailer.default_url_options = { host: 'localhost', port: 3000 }
```

config/environments/<ENVIRONMENT>.rb

Restart the application and Devise should be functional.

3.2.3 Devise Routes

Devise will add a number of routes in order to handle requests to its various modules. If we look in the application's `config/routes.rb` we'll see only a minimal change.

```
Rails.application.routes.draw do
  devise_for :users
end
```

To get a more complete picture of our routes, we can run `rake routes`. A default installation will expose the following:

```
 (in /tmp/devise_sample)
                  Prefix    Verb      URI Pattern                        Controller#Action
        new_user_session    GET       /users/sign_in(.:format)           devise/sessions#new
            user_session    POST      /users/sign_in(.:format)           devise/sessions#create
    destroy_user_session    DELETE    /users/sign_out(.:format)          devise/sessions#destroy
           user_password    POST      /users/password(.:format)          devise/passwords#create
       new_user_password    GET       /users/password/new(.:format)      devise/passwords#new
      edit_user_password    GET       /users/password/edit(.:format)     devise/passwords#edit
                            PATCH     /users/password(.:format)          devise/passwords#update
                            PUT       /users/password(.:format)          devise/passwords#update
cancel_user_registration    GET       /users/cancel(.:format)            devise/registrations#cancel
       user_registration    POST      /users(.:format)                   devise/registrations#create
   new_user_registration    GET       /users/sign_up(.:format)           devise/registrations#new
  edit_user_registration    GET       /users/edit(.:format)              devise/registrations#edit
                            PATCH     /users(.:format)                   devise/registrations#update
                            PUT       /users(.:format)                   devise/registrations#update
                            DELETE    /users(.:format)                   devise/registrations#destroy
```

We see that Devise has created a number of routes mapped to its own controllers to handle various actions per our module configuration. The configuration of routes can be modified if needed by modifying the `devise_for` parameters in `config/routes.rb`. Since Devise is implemented as a Rails Engine, the view templates are contained within the gem. To modify these, we'll need to tell Devise to generate views in our application tree where we can edit them. Similarly, if we need to modify controller logic, we can generate controllers, though this is less frequently needed.

There's much more that can be configured in Devise to suit nearly any use case, but we now have a reasonable overview of the common cases.

3.2.4 Using Devise

Once Devise is installed and configured, using it in an application is simple. To ensure a given controller requires authentication, we include a `before_action` callback in the controller. The name includes the model to which Devise was installed, so if we have multiple models (ie. `User` and `Admin`) we can also require different levels of authentication. For example, to require authentication against the `User` model we created earlier, a controller need only include:

```
before_action :authenticate_user!
```

Our recommendation is to include this callback in the `ApplicationController` found in `controllers/application_controller.rb`. This is the primary controller in the applications from which other controllers should then inherit. Configuring our callback here will cause authentication to be required across all controllers throughout the application. To handle actions which are anonymously accessible by design, we can explicitly configure the apppropriate controller to `skip_before_filter :authenticate_user!` on actions to which anonymous access is allowed. For example:

```
skip_before_filter :authenticate_user!, :only => [:public_action1, :public_action2]
```

Allowing Anonymous Access to Public Actions

This whitelist approach greatly reduces the liklihood of authorization bypass vulnerabilities by uniformly enforcing authentication except where explicitly configured otherwise. All pages are authenticated by default.

Devise also provides some helper methods to use in our application. Again, the names are dependant on the name of the model for which Devise is enabled. For our `User` model example they will be:

- `user_signed_in?` returns a boolean - `true` if a user is authenticated, `false` otherwise.
- `current_user` returns the `User` object of the currently authenticated user.
- `user_session` returns the session object associated with the currently authenticated user.

These helpers and callbacks provide the foundations upon which a strong authorization scheme can be built.

3.3 Session Management

Whether an application implements its own authentication system or uses a 3rd party gem it'll need to interact with the application's session. Like most web applications, Ruby on Rails uses a sesssion to persist data betwen each HTTP request. This allows us to build a stateful application atop the stateless HTTP protocol. As they relate to authentication flow, session management has the following steps:

1. Exchange user provided credentials for a token (HTTP Cookie)
2. On subsequent requests, identify the user's session by their token.
3. Invalidate the token when needed.
    - When the user logs out.
    - When the session reaches its maximum lifetime or idle time.

In all cases, the user's browser receives cookie which is presented to the server to identify their session. However, there are a few options for where session state is stored in Rails. Which type an application uses is configured in `config/initializers/session_store.rb` by setting the value of `Rails.application.config.session_store`.

3.3.1 Cookie Store

The default session store scheme starting with Rails 3 is the `cookie_store`. Using this session store mechanism, session data is serialized, encoded with base64, and signed. This value is passed to the user as their session cookie. This provides a stateful interface, without server overhead caused by needing to maintain state server-side.

The key used to generate and validate signatures is stores in `config/secrets.yml` with the name `secret_token` in the appropriate environment heading. By validating the signature, Rails can ensure the cookie has not been tampered with.

However, there are some caveats to be aware of:

- Cookies can be a maximum of 4K. However, sessions should only contain identifiers, such as a `user_id`, rather than full objects, such as an object of class `User`. As such, this poses little practical concern.
- Exposure of the `secret_token` will allow malicious users to craft their own session objects. If the session object stores the `user_id` of the current user, which is very common, an attacker can impersonate any user they wish by crafting their own session object.
- Replay attacks are possible if values in the session are trusted without validation. For example, storing an `account.balance` which is decremented when a user makes a purchase could give rise to an attack allowing replay of an older token with a higher balance generated by the server prior to the transaction. This makes consideration for the type of data stored in a session important.
- Unless accounted for through other means, users can have multiple active sessions concurrently. Depending on the application, this can be a security vulnerability.
- Invalidating sessions is problematic. When a user explicitly logs out of the application, we can ask their browser to clear the cookie, but if it's been captured and stored elsewhere it will still be seen as valid. Embedding an expiry time within the session and updating it per-request can limit this lifetime, but to expire sessions which are kept active also requires validation of its create time.
- Signed and serialized objects can be read from the cookie without knowledge of the signing key. If the application stores sensitive data in the session object, this may present an information disclosure vulnerability or facilitate further attacks.

To address the exposure of data in a session object to the user, versions of Rails greater than 4.0 introduce a `secret_key_base` variable in `config/secrets.yml`. When this variable is set, its value is used to encrypt the serialized session object in addition to signing. This eliminates the possibility of reading session data without the key, but otherwise does not change the overall `cookie_store` security posture.

It's important to ensure that applications which use the `cookie_store` include a `secret_key_base` instead of or in addition to a `secret_token`.

3.3.1.1 Session Serialization When a session object is converted to a cookie, it must be serialized. This is performed through a serializer, configurable in `config/initializer/session_store.rb`. On Rails 4.1 or later, the relevant setting will default to using the `:json` serializer.

```
Rails.application.config.action_dispatch.cookies_serializer = :json
```

config/initializer/session_store.rb

Prior to version 4.1, this setting defaults to using the `:marshal` serializer. When processing a session cookie to deserialize the session object, Rails will vaidate the signature, decrypt if applicable, and call 'Marshal.load()' on the user-supplied cookie value. A 'security considerations' section in Marshal's documentation explains the problem:

> "By design, `::load` can deserialize almost any class loaded into the Ruby process. In many cases this can lead to remote code execution if the Marshal data is loaded from an untrusted source."
>
> *Marshal Documentation*

`Marshal.load()` was the ultimate function invoked through Rails' XML and JSON parsers in older versions. This led to the well known deserialization vulnerabilities CVE-2013-0156 and CVE-2013-0333. Since signing limits an attacker's abilty to craft a malicious session cookie which will reach the deserializer, the situation isn't as dire. Still, `:json` is preferable. Again, Marshal's 'security considerations' is descriptive:

> "If you need to deserialize untrusted data, use JSON or another serialization format that is only able to load simple, 'primitive' types such as String, Array, Hash, etc. Never allow user input to specify arbitrary types to deserialize into."
>
> *Marshal Documentation*

Exposure of the secret keys used to serialize `:marshal` sessions will allow arbitrary remote code execution

A third serializer setting, `:hybrid`, is intended for backwards compatability. With `:hybrid` serialization configured, the application will issue session cookies serialized using `:json` but will accept either `:json` or `:marshal` serialized cookies. This means that applications configured with `:hybrid` sessions remain vulnerable, as `:marshal` is still reachable.

3.3.2 ActiveRecord Store

Another session storage option is known as ActiveRecord store. Again, the relevant configuration is in `config/initializers/session_store.rb` which is set to `:active_record_store`. While included in older versions of Rails, ActiveRecord store is no longer shipped as part of the core framework in version 4.0 and later. A gem which enables ActiveRecord store is still available.

ActiveRecord store creates a sessions table in the server's database. In this system, the user's browser is provided only a cryptographically random token which serves as their session cookie. When the application receives a request, it queries the database and loads the session identified by that object. From a security standpoint, this has many benefits as compared to cookie sessions:

1. Sessions can be easily revoked by deleting the relevant entry from the database.
2. ActiveRecord provides 'created_at' and 'updated_at' timestamps, making session timeouts easier to manage.

3. Enforcement of session concurrency limits is significant simpler.

4. Attack surface exposed to users is significantly reduced.

   - No risk of session information disclosure.
   - No possibility of deserialization vulnerabilities.
   - No possibility of cryptographic attacks.

The downside is that in an ActiveRecord session configuration, the application must perform a database query for each request. This is the rationale behind its removal from Rails. This can pose a significant performance burden for large applications, an issue which is exacerbated by distributed and load balanced applications. Recent performance improvements to ActiveRecord may reduce the impacts, and in many applications the performance overhead may not pose a significant concern.

Other storage mechanisms which function similar to ActiveRecord store but use higher performance memory-caching and key-value storage systems are available. Rails includes a native MemCacheStore. Gems such as Dalli and redis-session-store can be used to back Rails sessions with Memcached and Redis, respectively.

3.3.3 Session timeout

Regardless of the session type, applications must be cautious to expire sessions periodically. Sessions which do not expire greatly increase the impact of session hijacking and fixation attacks. Rails allows us to configure the lifetime of the session cookie.

`config/initializiers/session_store.rb` should include an `:expire_after` symbol set to a reasonable lifetime for the session cookie such as `2.hours`.

If a cookie is obtained maliciously, the attacker can ignore this lifetime. Applications should also ensure that the session object itself contains an expiry time, and the server enforces that restriction. This will reduce exposure of data and application access resulting from a compromised session cookie.

Applications must take steps to protect session cookies from compromise. In recent versions of Rails, session cookies can easily be set `Secure` and `HTTPOnly`. By including `config.force_ssl = true` in `config/environments/<ENVIRONMENT>.rb` ' the application will require SSL, and will also set these cookie security flags. `:httponly => true` and `:secure => true` can also be passed in the `session_store` initializer.

3.4 Lost/Forgotten Password Recovery

It's unavoidable that for any authentication system that relies on secret knowledge, these secrets will be forgotten. Therefore, any authentication system which relies on passwords, needs some provision for restoring access to the legitimate user should they be unable to login. For some applications, this may be a manual process of proving your identity to an administrator outside the application flow, but such a process won't scale. Most applications will have some automated process allowing users to restore access to their account. These systems can often be exploited as a means of gaining illicit access to an application, and so they require careful thought.

3.4.1 Poor Recovery Mechanisms

There are no shortage of poor mechanisms for recovering accounts. Recovery mechanisms become a de-facto secondary means of authentication, and must be treated as such. Too often they are significantly weakened when compared to the primary mechanism. Technical problems, including failure to protect sensitive data at rest as password-equivilants, abound. The problems with many systems are less technical, and more ingrained in the mechanism by which they operate.

Perhaps the most popular is the 'secret question' system. When the account is created the user provides some piece of information known to them, which they can use to 'confirm' their identity at a later date.

Such information is only a reliable authenticator to the degree the information is private. Questions like 'What is your Mother's Maiden Name?' or 'What city were you born in?' can often be answered by researching a user's social media profile. Other questions like 'What is your favorite sports team?' or 'What is your favorite color?' can be easily guessed from a relatively small list of candidates; there are only so many sports teams, after all. The situation only worsens the more sites share the same 'secrets', exposing that information to a broader circle of administrators, customer service representatives, etc. In an effort to improve the variety of questions, some applications will allow the user to generate their own question and answer. This often results in significantly weaker questions as users become frustrated when asked to provide more information.

Another commonly seen and equally poor practice is to provide a 'password hint' which can be displayed to the user to give them a clue what their password is. Unfortunately, users will sometimes enter their password itself as the hint, completely compromising the security of their own account. If we prohibit that practice, we may end up with riddles describing their password or other phrases that rhyme with it, etc. Such hints will make it easier for the user to recall their password, but so too they ease the process of a third party guessing it. In extreme cases, password hints can even compromise the benefits of password digests making guessing their plain-text values a simple game.[13]

3.4.2 Strong Recovery Mechanisms

While there are many weak recovery mechanisms, there are far fewer that stand up to scrutiny. The solution we favor takes the following form:

1. User visits site, clicks 'forgot password' link.
2. User provides their username.
3. The application looks up the user's account to verify existence and validity.
4. The application generates a random token using a cryptographically secure random number generator and a long enough value to prevent brute force.
5. The token is saved to the database, along with an association to the user's account, and a timestamp indicating when it was generated.
6. The token is sent to the user via some out-of-band mechanism; Email, SMS, etc.
7. The application responds and inform the user that a message has be sent to the account they requested.
    - ****NOTE**** The system should take care not to disclose whether or not the given username was valid, to avoid a usrname enumeration vulnerability.
8. Upon receiving the token, the user visits the site and provides it, often by following a link in an email.

9. The application looks up the token to confirm its validity, verifies its timestamp is within an expiry threshold, and allows the user to reset their password.
10. Finally, the token is deleted from the database, preventing reuse.
11. If the user logs in at any point using their normal credentials, outstanding tokens associated with their account should immediately be revoked.

Such a system provides no information to the unverified user, while leveraging their access to an already known means of contact to provide them a time-limited means of changing their password. Tokens cannot be used more than once, and only within a short interval.

The security of this system depends upon the security of that out-of-band communication mechanism. While imperfect, this is far better than most alternatives. Even manual contact with a customer service agent may allow for social engineering ploys and human error while posing additional expense and descreasing customer satisfaction. The problem of forgotten passwords is just one reason we should continue to seek improved solutions to authentication.

3.5 Devise Password Recovery

In Devise, password recovery is accomplished through use of the `:recoverable` module. Recall that when we discussed Devise earlier, we saw it enabled by defaul in our Devise-created `User` model.

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

app/models/user.rb

There are several controllers which relate to the `:recoverable` module.

```
(in /tmp/devise_sample)
            Prefix     Verb     URI Pattern                         Controller#Action
     user_password     POST     /users/password(.:format)           devise/passwords#create
 new_user_password     GET      /users/password/new(.:format)       devise/passwords#new
edit_user_password     GET      /users/password/edit(.:format)      devise/passwords#edit
                       PATCH    /users/password(.:format)           devise/passwords#update
                       PUT      /users/password(.:format)           devise/passwords#update
```

We see that these routes all call various actions in the `PasswordsController`. The Devise source code[14] for which is freely available and can be found within the gem itself. Quickly reviewing the code shows us the purpose of each action. Each of the controllers performs some actions on the class to which the module is applied. The source for the model mixins associated with recoverable can be found in lib/devise/models/recoverable.rb[15] In our case, these actions will be performed on our `User` class, since that's the model we've enabled Devise on.

- `passwords#new` - Presents the user with a form asking for the User ID of their account. The form posts this address to the route associated with `passwords#create`
- `passwords#create` - Sends instructions to the user to reset their password by calling `User.send_reset_password_instructions`

- `passwords#edit` - This is the action the email a user receives directs them to. It accepts a parameter in the GET request called `reset_password_token` and confirms it matches a valid token in the database. If it does, it provides a form asking the user to enter and confirm the new password value they would like. When the user submits the form, the resulting information is sent to `passwords#update` as parameters in the HTTP body.
- `passwords#update` - The is where the password update actually occurs. The primary action taken is a call to `User.reset_password_by_token()`.

If we look in `recoverable.rb` we see what occurs there, along with some helpful comments.

```
 1 # Attempt to find a user by its reset_password_token to reset its
 2 # password. If a user is found and token is still valid, reset its password and automatically
 3 # try saving the record. If not user is found, returns a new user
 4 # containing an error in reset_password_token attribute.
 5 # Attributes must contain reset_password_token, password and confirmation
 6 def reset_password_by_token(attributes={})
 7   original_token       = attributes[:reset_password_token]
 8   reset_password_token = Devise.token_generator.digest(self, :reset_password_token, original_token)
 9 
10   recoverable = find_or_initialize_with_error_by(:reset_password_token, reset_password_token)
11 
12   if recoverable.persisted?
13     if recoverable.reset_password_period_valid?
14       recoverable.reset_password(attributes[:password], attributes[:password_confirmation])
15     else
16       recoverable.errors.add(:reset_password_token, :expired)
17     end
18   end
```

devise/lib/devise/models/recoverable.rb

The actual comparison of of the provided token and the actual token occurs with the comparison to the value returned from `Devise.token_generator.digest` on line 8. The remaining code validates that the object is recoverable, and that the token was issued within the maximum lifetime.

```
def digest(klass, column, value)
  value.present? && OpenSSL::HMAC.hexdigest(@digest, key_for(column), value.to_s)
end
```

devise/lib/devise/token_generator.rb

This code looks up a key associated with the column in which the recovery token digests are stored. It then uses this same key to create an HMAC digest of the reset token passed to it, and returns the result. This is noteworthy, because it means that in this version of Devise, the reset tokens themselves are note stored in the server, which keeps only an HMAC digest of them and calculates the HMAC of the submitted value. This means that even a compromise of the server's database would not provide valid reset tokens. Put another way, they're protecting reset tokens in much the same way as passwords; certainly a good approach!

Some other recent changes show care given to this process that may not be taken among in-house developed solutions. For example, reset tokens are cleared when the associated object's password changes,[16] or its email changes.[17]

However, things haven't always been this well thought out, as we'll see in our next section.

3.5.1 Devise Security History

We noted earlier that Devise stores password reset tokens only as HMAC digests. However, this wasn't always the case as we see from a commit[18] dated August of 2013. In fact, Devise has a strong history of improving security, sometimes in response to vulnerabilities discovered by researchers. From newest to oldest here's some of the highlights from the Devise change log.[19]

Unreleased/HEAD Adds an option to send an email to users when their password changes, potentially alerting them to a compromised account.

3.5.1 Removes active password reset tokens upon email address or password change. Improves handling of 'remember me' tokens when integrators improperly extend the base functionality.

3.1.2 Addresses an email enumeration bug.

3.1.0 Stores password reset tokens as HMACs instead of plain-text tokens, eliminating the possibility they can be stolen from the database.

3.0.1 Addresses a CSRF token fixation flaw.

2.2.3 Fixes a type confusion vulnerability in password reset functionality that allows compromise of accounts.

Despite these issues, our view is not that Devise has a poor security security track record. In fact, it's quite the opposite. Most of these issues are subtle, and only impact security in specific scenarios. The fact that the Devise team has a demonstrable history of addressing such issues and improving security speaks to their commitment to security of the product. If there's any message to take away from this, it's that authentication systems are complex, the bugs subtle, and avoiding problems is challenging. Developers who integrate Devise into their applications benefit from an increasingly strong security posture through ongoing updates and improvements. The key take away is to track the state of dependencies, and follow a regular process for updating on an ongoing basis.

3.5.2 Devise Type Confusion

The last vulnerability in the above list, relating to password reset, is likely the most significant vulnerablity Devise has faced in its history. It's also not unique to Devise, and thus makes an interesting case study. In this section, we'll discuss the details of this vulnerability, including means of exploitation and how the core bug may manifest itself in other applications.

This bug was first disclosed by Joernchen of Phenoelit.[20] His blogpost from February of 2013 explains the issue well, but we'll briefly review.

If we simplify the Devise password reset mechanism we discussed earlier, and account for some of the changes that have taken place since, we can construct some pseudo-code that simplifies it. This code is similar to mechanisms the authors have seen developed independently by others who have written their own password reset systems.

```
def reset
  user = User.find_by_token(params[:user][:reset_password_token])
  if user
    user.change_password(params[:user][:password], \
        params[:user][:confirm_password])
  end
end
```

Simplified Password Reset Pseudocode

From our earlier discussions of password reset mechanisms, the code should be pretty clear. This controller action takes three parameters from the user's request; `:reset_password_token`, `:password`, and `:confirm_password`. The first is the token that was sent to the user out of band, while the others contain the new password, for verification that there weren't any typos.

In line 2, the controller fetches the `User` object for which the `:reset_password_token` applies. Assuming a user is returned, lines 3-6 perform the reset of their password to the desired value. At a glance, this all looks proper assuming that tokens are generated securely, and can't be guessed. The problem, as Joern discovered, relates to a behavior of the underlying database.

When the application is backed by MySQL, there's some unusual equality logic that makes this seemingly benign code extremely dangerous. Consider the query below:

```
mysql> select "foo" from dual where 1="1string";
+------+
| foo  |
+------+
| foo  |
+------+
1 row in set, 1 warning (0.00 sec)
```

MySQL Equality Typecasting

In this query, we see the integer '1' compared to the string '1string'. Shockingly, MySQL typecasts the string to an integer before performing the comparison. Since '1' equals '1', the comparison evaluates as `true` and in our sample query the select statement returns "foo". This behavior applies to any string that begins with an integer, when compared to that same integer.

```
mysql> select "foo" from dual where 0="string";
+------+
| foo  |
+------+
| foo  |
+------+
1 row in set, 1 warning (0.00 sec)
```

Even More Unexpected MySQL Typecasting

Even more unusual is the case where the integer '0' is compared to a string that does not begin with an integer. Again, MySQL typecasts the string, but this time it evalutes true against '0'. Let's put this in the context of queries generated by ActiveRecord when performing the `User.find_by_token(params[:user][:reset_password_token])` query seen in our pseudocode above.

```
 SELECT  "users".* FROM "users" WHERE "users"."token" = ? ORDER BY \
    "users"."id" ASC LIMIT 1 [["token", "Q2ixrCpSS72SB2tvNrB2"]]
```

Example MySQL Query

In the above snippet,[†] we see a query for a user where the token equals the user-supplied string. But what if instead of a string, this were the integer '0'? If we could manipulate the parameters, such that we passed the integer '0', the query would return the first user who had an outstanding token that did not begin with an integer. But can that be done?

[†]This is a real query, from a log of a test instance run locally.

```
<foo>bar</foo>
<fizz type“=”integer>1</fizz>
```

Example of XML Typecasting

3.5.2.1 Rails Magic Luckily for us,[‡] There's some Rails Magic that can help. Often, developers expect that the `params` hash which provides user input will only contain `String`s. This is the most common case, but there are execeptions. Since the CVE-2013-0156 XML YAML deserialization vulnerablity,[21] Rails typecasting has been fairly well known. By specifying the XML type of an attribute included in an HTTP body, we can influence Rails to interpret the input of that type.

In this example, we end up with a params hash that looks like

```
params => {"foo"=>"bar", “fizz"=>1}
```

Notice that the value of 'fizz' contains an integer. This is what we need to exploit the type confusion. Since this vulnerability was discovered, Rails has disabled XML parsing by default. On Rails 4.0 and later, the actionpack-xml_parser gem[22] is needed to provide support for XML. So we instead turn to JSON.

```
{"user":{\
"password":"GAMEOVER",\
"password_confirmation":"GAMEOVER"“,reset_password_token":0}\
}
```

The effect is similar. The JSON format allows us to easily encode integers and is supported natively by all versions of Rails.

3.5.2.2 Metasploit Module To ease testing of the vulnerability, one of the authors of this paper produced a Metasploit Module[23] targetting the issue. Despite the fact that the vulnerability affects the token without identifying a specific account, the exploit can be fairly well targetted. To exploit a specific account, some further steps are necessary.

- Choose an integer representing the largest integer-prefix on a token which will be targetted.
- Loop through all integers until reaching that maximum value, and send them as reset tokens.
- If the response indicates a password was successfully reset, repeat this value.
- Increment the integer and repeat.
- Once all tokens up to the maximum integer are cleared, send a password reset request for a known user account. User enumeration vulnerabilities may help in exposing accounts to target.
- Now that the targetted user has an active token, repeat the earlier process to reset their password.

The Metasploit module makes this relatively easy to accomplish. The maximum integer value, choice of clearing outstanding tokens or not, and URLs associated with affected controllers are all configurable. The module is also useful, with slight modifications, for exploiting similar patterns in applications that do not use a

[‡]... or unluckily, depending on your view.

vulnerable version of Devise. The authors of this paper have seen similar issues arrive in systems developed by internal teams, unaware of this issue.

```
msf auxiliary(rails_devise_pass_reset) > exploit
[*] Clearing existing tokens...
[*] Generating reset token for
    admin@example.com...
[+] Reset token generated successfully
[*] Resetting password to "w00tw00t"...
[+] Password reset worked successfully
[*] Auxiliary module execution completed
```

3.5.2.3 Patch Status This vulnerability was assigned CVE-2013-0233[24] and was patched in Devise some time ago.[§] This was accomplished through casting the supplied value to a string prior to calling the query. In our pseudocode example, the fix is essentially `User.find_by_token(params[:user][:reset_password_token].to_s)` with `.to_s` being the extent of the change.

For Rails itself, the situation is more complex. The issue was initially patched in 3.2.12[25] by a change to ActiveRecord which cast the query parameter type such that it matched the database column type. However, this caused problems with some functionality, and was reverted in 3.2.13.[26] The issue is finally fixed in ActiveRecord versions 4.2.0 and later.[27]

This Affects More Than Just Devise

ActiveRecord will still build queries that can allow for exploitation of MySQL's equality type confusion on all versions prior to 4.2.0.

In 4.2.0 and later, developers can still introduce the issue if instead of using ActiveRecord's finder methods, they build their own through queries like: `User.where("token=?", params[token])`

MySQL shows no signs of changing their behavior.[a]

It's likely this affects other platforms beyond Rails.

[a]Likely due to compatability concerns.

[§]Specifically, in versions 2.2.3, 2.1.3, 2.0.5, 1.5.4, and later.

4 Authorization

Earlier, we noted that the authentication process is reasonable for establishing user identity. Authorization is instead focused on establishing and enforcing permissions granted to the use according to that identity. Authorization within web applications today is often tied to the concept of roles. Let's use the following role heirarchy as an example of a typical authorization scheme:

- **Super Administrator:** Full administrative access to the entire application. This role has no limits to the data and routes it can access.
- **Organization Administrator:** Administrative access to the subset of data belonging to their organization. The organization admin of 'FooCorp' should not be able to access any data belonging to 'BarCorp' and vise versa.
- **Team Manager:** This role may have read/write access to only the data belonging to their team within an organization. They should not be able to access other team's data within their organization or other organizations.
- **Employee:** This role may have only read access to their team but no other teams within their organization or other organizations

In this scenario, we see a combination of both vertical and horizontal authorization models.

The term 'Vertical Authorization' is used to describe different levels of access to data within a single organization. For example, an Organization Administrator should be able to access all data within their organiztion, while an employee should only be able to access data associated with their own team.

In contrast, 'Horizontal Authorization' describes differences in permission between various users with the same role. In our example, an Organization Admin belonging to an organization called 'FooCorp' should be able to access data of all FooCorp employees, but have no access to any data owned by 'BarCorp.'

The OWASP project summarizes these two types of authorization.[28]

> "Different clients/users should not see other clients' data (Horizontal authorization). Authorization can also be used to restrict functionality to a subset of users. "Super users" would have extra admin functionality that a "regular user" would not have access to (Vertical authorization)."
>
> *OWASP Project Wiki*

The 'Super Administrator' role in our example crosses this horizontal authorization boundary and allows access to all data, regardless of owning organization or employee. Often, such a role is provided for use by systems administrators and support staff, and should not be generally available to the public. In some circumstances, it may be reasonable to control access to these highly privileged accounts by IP restrictions, or by exposing a seperate application instance for their use which is segmented from public networks.

4.1 Vertical Authorization in Rails

Vertical authorization within Rails is typically implemented through the use of a `before_filter` or `before_action`. The purpose of these is common, but the nomenclature has changed within recent versions of rails. A controller uses a `before_action`[¶] as a callback which is executed prior to calling the controller action. In our case, it's used to enforce authentication and verify authorization, but they're also sometimes used to set up variables, check the state of the session, etc.

[¶]Or before filter, the terms are more or less interchangeable for now.

```
class SuperAdminController < ApplicationController
  before_action :require_super_admin

  # An action available for the AdminController
  # This might be reached via the /admin/all_users path
  def all_users
    respond_with User.all.to_json
  end

  private

  # Redirect to the home page if the user is not an admin
  def require_super_admin
    unless current_user.super_admin?
        redirect_to root_url
    end
  end
end
```

Example 'before_action' Enforcing Authorization.

In the example code above, we've created a SuperAdminController with a single action called `all_users` that renders all of the users within the application as JSON. At line two we see `before_action :require_super_admin`. This tells Rack to run the `require_super_admin` method before processing any SuperAdminController action. The `require_super_admin` method will then check to see if the user is a super admin. If they are not, they are redirected to the root of the application. If they are, the controller continues processing the `all_users` action.

If a filter redirects or renders, the controller action will not be run. In addition, further controllers which inherit from AdminController will also inherit this callback.

NOTE `before_filter` and `before_action` are the same thing as can be seen here.[30] The `before_filter` method includes a deprecation warning that `before_filter` will be removed in Rails 5.1.

Rack[29] is a middleware layer for Rails applications.

4.2 Horizontal Authorization in Rails

Horizontal authorization tends to be implemented through the use of ActiveRecord associations.[31] These associations are established through the model. In the following example, we establish an association where a User has many Customer Accounts and Customer Accounts belong to a User.

```
class User < ActiveRecord::Base
  has_many :customer_accounts, dependent: :destroy
end

class CustomerAccount < ActiveRecord::Base
  belongs_to :user
end
```

Then when Customer Accounts are created, they are assigned to a User using syntax similar to the following:

```
class CustomerAccountsController < ApplicationController
  def create
    @customer_account = current_user.customer_accounts.build(customer_account_params)
    @customer_account.save
  end
```

Through the association, we can query only Customer Accounts that belong to the currently logged in user.

```
class CustomerAccountsController < ApplicationController
  def index
    @customer_accounts = current_user.customer_accounts
  end
```

This process requires that the `current_user` object is set in our session. This is typically handled by the authentication process.

4.3 Routing

Rails establishes all of its routes via the `config/routes.rb` file. Here all URL paths are mapped to some kind of controller and action. Let's take a look at the following example routes file:

```
1  # The root/home page of the site
2  root to: 'visitors#index'
3
4  # resources sets up default set of CRUD routes (e.g. index, create, update,
5  # destroy, etc.)
6  resources :customer_accounts
7  resources :users
8
9  # custom routes
10 post '/customer_accounts/:id/place_on_hold', to: 'customer_accounts#place_on_hold'
11 get  '/customer_accounts/accounts_on_hold',  to: 'customer_accounts#accounts_on_hold'
```

Example config/routes.rb File

Line 2 shows where the root of our application is. This is the controller and action that gets run when you visit http://www.myrailsapp.com/. At lines 6 and 7, the route definition utilizes the resources helper. This helper sets up a number of common CRUD routes for a given controller and model pair. See CRUD, Verbs, and Actions[32] for more on this. The last set of routes on lines 10 and 11 specify custom path strings with mappings to the controller and action that should be called when that path is visited. For more information on Rails routing, see the official Rails guide, Rails Routing from the Outside In.[33] For a technical dive into the code for how routing works in Rails, see Andrew Berl's post on Routing.[34]

4.4 Controller Hierarchy

Every new Rails application contains a single controller by default.[**] This is the ApplicationController.

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

app/controllers/application_controller.rb

If we generate a Posts controller with an index action:

```
rails generate controller Posts index
```

We can now to see the Rails controller hierarchy start to build. Looking at the `PostsController`, we see that it inherits from the `ApplicationController`.

```
class PostsController < ApplicationController
  def index
  end
end
```

app/controllers/posts_controller.rb

Next, the `ApplicationController` itself inherits from the internal Rails controller `ActionController::Base`.

And `ActionController::Base` itself inherits from a few other classes seen in Figure 1.

The important concept to note here is that by default all new controllers inherit from `ApplicationController`. This means that any actions or filters declared within the `ApplicationController` will automatically be available in new controllers unless otherwise specified.

See the Rails guide for ActionController and Andrew Berl's technical dive into the code behind ActionController.

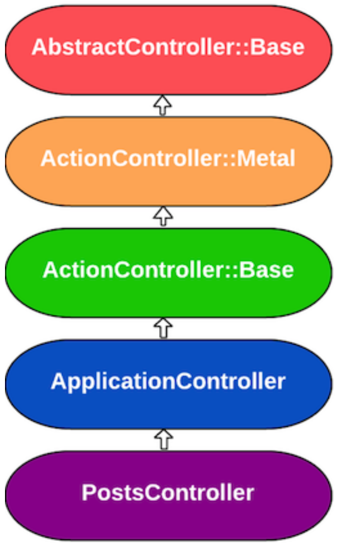


Figure 1: Controller hierarchy[36]

[**]Technically there is also Rails::InfoController, Rails::WelcomeController, and Rails::MailersController but these are hidden.

4.5 The Different Flavors of Rails Filters

Rails filters can also include several modifiers to control which actions they apply on.

```
before_action :authorize_user, only: [:action1, :action2, …]
before_action :authorize_user, except: [:action1, :action2, …]
before_action :authorize_user, if: method_call
before_action :authorize_user, unless: method_call
skip_before_action :authorize_user, only: [:action1, :action2, …
skip_before_action :authorize_user, except: [:action1, :action2, …]
```

The options `only` and `accept` accept any array of actions to which they be applied or conversely not applied.

Similarly, `if` and `unless` accept a method argument and apply when the result is `true` or `false`, respectively.

4.5.1 Less common flavors

There are a few other filter modifiers which are far less commonly seen.

```
class ApplicationController < ActionController::Base
  before_action do |controller|
    unless controller.send(:logged_in?)
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url
    end
  end
end
```

Specifying a Block Inline

This example specifies a block inline with the filter. The effect is similar to an `if:` modifier, but the block is anonymous and does not need to be declared as a method.

```
# Specifying a class
class ApplicationController < ActionController::Base
  before_action LoginFilter
end

class LoginFilter
  def self.before(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in to access this section"
      controller.redirect_to controller.new_login_url
    end
  end
end
```

Specifying a Class

Here we specify a class as the modifier. The `before` method of that class is run as our filter.

4.6 Authorization Gems

Unlike authentication, authorization tends to require a good deal of customized logic which causes development teams to roll their own authorization more often. However, there are a good number[37] of authorization gems available for Rails. The two most popular that we've come across during our Rails assessments are Pundit[38] and CanCanCan,[39] formerly CanCan,[40] which was forked into CanCanCan to provide support for Rails 4. Both of these gems make testing a bit easier by centralizing the role definitions into a single location.

4.6.1 CanCan(Can)

> CanCan is an authorization library for Ruby on Rails which restricts what resources a given user is allowed to access. All permissions are defined in a single location (the Ability class) and not duplicated across controllers, views, and database queries.

Let's take a look at the example Ability file from the CanCanCan wiki:

```
class Ability
  include CanCan::Ability

  def initialize(user)
    user ||= User.new # guest user (not logged in)
    if user.admin?
      can :manage, :all
    else
      can :read, :all
    end
  end
end
```

This is a basic `Ability` class that contains an initialization method that expects a user object. Often times this is the `current_user` of the request (usually provided by Devise). The main workhorse of this gem is the `can` method which defines what the user can or can not do. This method needs two arguments: usually this is one of the 7 RESTful methods[41] (e.g. create, read, update, destroy, etc.) and a model. Therefore, a permission of `can [:update, :destroy], [Article, Comment]` will allow a user to update and destroy Article and Comment objects. In Rails, these objects pertain to the rows of the Articles and Comments database tables.

Although the basic CRUD methods are common enough that they are built in as actions, CanCanCan does allow you to define custom actions in the `ability.rb` file. The CanCanCan documentation gives the following example:

```
# in models/ability.rb
can :assign_roles, User if user.admin?
```

Note that CanCanCan lets you add a `:manage` action as well. `:manage` gives the user permisions for all defined actions, including any custom actions in the `ability.rb` file.

```
can :assign_roles, User if user.admin?
can :manage, User
```

The `can :manage, User` would override the admin check used for `:assign_roles` and give all users the ability to assign roles.

Once abilities are defined, the `can?` method is used to test and check abilities. The following is an example from the CanCanCan wiki that would be seen in a view:

```
<% if can? :create, Project %>
  <%= link_to "New Project", new_project_path %>
<% end %>
```

More often, abilities will be checked in the controllers through the use of code such as:

```
# Checking against instances...
@project = Project.find(params[:id])
can? :destroy, @project

# ... or checking against the class name of the model itself
can? :destroy, Project
```

From here, it's a matter of checking the controller actions for the existence of these methods and assessing the context to ensure the ability checks are correct. For more information on CanCanCan, refer to the project's Wiki page.[42]

4.6.2 Pundit

> Pundit provides a set of helpers which guide you in leveraging regular Ruby classes and object oriented design patterns to build a simple, robust and scaleable authorization system.
>
> *Pundit Github Page*[43]

Similar to CanCanCan, Pundit attempts to centralize the definitions of permissions in a single location. For Pundit, this is through policy files located in the `app/policies` directory. Let's take a look at a policy file for the User model.

```
class UserPolicy
  attr_reader :current_user, :model

  def initialize(current_user, model)
    @current_user = current_user
    @user = model
  end

  def index?
    @current_user.admin?
  end

  def show?
    @current_user.admin? or @current_user == @user
  end

  def update?
    @current_user.admin?
  end

  def destroy?
    return false if @current_user == @user
    @current_user.admin?
  end
end
```

app/policies/user_policy.rb

Pundit has quite a few assumptions about the structure and name of this policy file.

> - The class has the same name as some kind of model class, only suffixed with the word "Policy".
> - The first argument is a user. In your controller, Pundit will call the current_user method to retrieve what to send into this argument
> - The second argument is some kind of model object, whose authorization you want to check. This does not need to be an ActiveRecord or even an ActiveModel object, it can be anything really.
> - The class implements some kind of query method, in this case update?. Usually, this will map to the name of a particular controller action.
>
> *Pundit Documentation*[44]

The example policy file above allows admins to view, update, and destroy all users. While allowing users to destroy and view their own accounts. Checking these defined permissions in controllers is done through the use of the `authorize` method.

Pundit gives you the ability to ensure that all of the actions inside of a controller run the `authorize` method using the :verify_authorized[45] filter.

```
class UsersController < ApplicationController
  before_action :authenticate_user!
  after_action :verify_authorized

  def index
    @users = User.all
  end
end
```

In this code, when rendering the `index` action the application throws a `Pundit::AuthorizationNotPerformedError in UsersController#index` exception. This will force the developer to add proper authorization checks to the `index` action.

As nice as the `:verify_authorized` method is, this should only be relied on during code development or non state changing actions since this filter only works in an `after_action` callback. This means that in a state changing action, the changes will first be made to the database and then the exception will be thrown. In the following `UsersController` example we'd expect the controller to throw an exception when attempting the `destroy` action because the `authorize` method was never called.

```
class UsersController < ApplicationController
  before_action :authenticate_user!
  after_action :verify_authorized

  def index
    @users = User.all
    authorize User
  end

  def destroy
    user = User.find(params[:id])
    user.destroy
    redirect_to users_path, :notice => "User deleted."
  end
end
```

However, because the exception happens after the action completes, the destroy action completes and destroys the user in the database. This can be seen in the following server logs:

```
Started DELETE "/users/2" for ::1 at 2015-11-04 23:00:45 -0600
Processing by UsersController#destroy as HTML
  User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ?
       ORDER BY "users"."id" ASC LIMIT 1  [["id", 1]]
  User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ?
      LIMIT 1  [["id", 2]]
   (0.1ms)  begin transaction
  SQL (0.4ms)  DELETE FROM "users" WHERE "users"."id" = ?  [["id", 2]]
   (0.7ms)  commit transaction
Redirected to http://localhost:3000/users
Completed 500 Internal Server Error in 8ms (ActiveRecord: 1.5ms)

Pundit::AuthorizationNotPerformedError
    (Pundit::AuthorizationNotPerformedError):
  pundit (1.0.1) lib/pundit.rb:103:in `verify_authorized'
  activesupport (4.2.4) lib/active_support/callbacks.rb:432:in `block in
      make_lambda'
  activesupport (4.2.4) lib/active_support/callbacks.rb:239:in `call'
```

Although there are some potential pitfalls with Pundit, it's still quite easy to use and seems to be on the verge of overtaking CanCan(Can) as the leading authorization gem. For more information on Pundit, refer to the project's README.[46]

4.7 Patterns to watch out for

The following sections are meant to show code patterns that we've seen result in authorization vulnerabilities. However, seeing these patterns does not guarantee a vulnerability. Instead it should constitute some extra effort in auditing the code around these patterns.

4.7.1 Finder methods called directly on the model

Instead of using model associations, applications will sometimes use ActiveRecord finder methods (e.g. `find`, `find_by`, `find_all_by`) directly on the model. An example of this might be passing in a user-supplied `:id` parameter to the CustomerAccount model:

```
class CustomerAccountsController < ApplicationController
  def show
    @customer_account = CustomerAccount.find(params[:id])
    render json: @customer_account
  end
end
```

Although this code will work perfectly fine for retrieving and rendering a single Customer Account, there is nothing stopping a user from specifying an arbitrary `:id` for a Customer Account that does not belong to the user.

A better pattern would be calling finder methods on associations. Going back to the `show` action from the example above:

```
class CustomerAccountsController < ApplicationController
  def show
    @customer_account = current_user.customer_accounts.find(params[:id])
  end
end
```

This limits the search to the subset of Customer Accounts that belong to the currently logged in user, regardless of whether the `:id` is valid for an account outside this scope.

4.7.2 Action whitelisting `before_action only: [:action1, action2]`

Let's take and expand our SuperAdminController example from earlier. Imagine you have the following SuperAdminController:

```
class SuperAdminController < ApplicationController
  before_action :require_super_admin, only: [all_users, all_customer_accounts]

  def all_users
    render json: User.all
  end

  def all_customer_accounts
    render json: CustomerAccount.all
  end

  private

  # Redirect to the home page if the user is not an admin
  def require_super_admin
    unless current_user.super_admin?
      redirect_to root_url
    end
  end
end
```

From here, a developer wants to add a new feature that would allow admins to see all open Customer Accounts. This would require adding a new action called `all_open_customer_accounts` like so:

```
class SuperAdminController < ApplicationController
  # ...
  def all_open_customer_accounts
    render json: CustomerAccount.where(status: "open")
  end
  # ...
end
```

The developer tests the action with their admin account and everything works fine. However, they forgot to add `:open_customer_accounts` to the before_action's `:only` option, resulting in an authorization vulnerability, since this action can be invoked without satisfying the `require_super_admin` callback as intended.

Instead, the use of the `:except` option to specify only the actions you explicitly do not want to run callbacks on is preferred.

```
class SuperAdminController < ApplicationController
  before_action :require_super_admin, except: [:health_check]

  def all_users
    render json: User.all
  end

  def all_customer_accounts
    render json: CustomerAccount.all
  end

  # Used by monitoring service to check if server is up
  def health_check
    render text: "OK"
  end

  private

  # Redirect to the home page if the user is not an admin
  def require_super_admin
    unless current_user.super_admin?
      redirect_to root_url
    end
  end
end
```

With this pattern, developers can add new actions to the SuperAdminController or its descendants, which will pick up the `:require_super_admin` callback by default.

4.7.3 Lightweight Controllers

As we discussed in the Controller Hierarchy section, any controller that inherits from `ApplicationController` inherits all of its filters as well. The main filter that every new application gets by default is `:verify_authenticity_token`. This filter is added by the `protect_from_forgery` line that is seen at the top of the ApplicationController by default.

Sometimes you'll see controllers that don't inherit from `ApplicationController` and instead inherit from the class above `ApplicationController`, `ActionController::Base`.

```
class BackupsController < ActionController::Base
  def create_backup
    # Backup database
  end

  def remove_backups
    # Remove specified backup
  end
end
```

In the above example, the developers chose to inherit from ActionController::Base to create a simple backup controller. In this instance, any authentication/authorization filters from the ApplicationController as well as the `protect_from_forgery` filters are lost.

There may be legitimate use cases for this, such as in API controllers, where the developer may wish to enforce fewer filters. Unusual inheritence patterns can easily give rise to security vulnerabilities, by opting out of default protections afforded by Ruby on Rails.

For a real world example, see Egor Homakov's post, "CSRF in Doorkeeper" which details a significant Cross-Site Request Forgery bug in Doorkeeper that was originally found by Sergey Belove.

4.7.4 Authorization Logic in Views

At times, an application will need to render parts of a page based on the resources the currently logged in user is authorized to view. A common example of this is to show/hide administrative functionality in the view. This logic would potentially be found in a view in the following form:

```
<% if current_user.super_admin? %>
  <a href="#admin-users-panel">
    # Render administrative actions
  </a>
<% end %>
```

Here, we need to make sure that the controller corresponding to this view is also properly checking authorization. Although the view is only displayed for super administrators, there is nothing stopping an attacker from bypassing the client side view and making requests directly to the server.

4.7.5 Skipping of filters

When auditing the use of before filters, special attention should be paid to any uses of `skip_before_filter`/`skip_before_action`. The occurrences of this method should be evaluated to make sure the filters were indeed meant to be skipped.

An example scenario for this may be a developer adding a new state changing action to a controller that utilizes a POST method. To easily test this action using a cURL command, the developer adds `skip_before_filter :verify_authenticity_token` to the controller. After finishing their feature, they forget to remove this exception from their code and push the code up to production.

```
class SuperAdminController < ApplicationController
  # CSRF check is making testing annoying with cURL...
  skip_before_action :verify_authenticity_token

  # ...

  def new_post_action
    # ...
  end
end
```

The developer has just introduced a Cross-Site Request Forgery vulnerability across the entire controller. The use of `skip_before_action` is implicitly circumventing some kind of logic, and it's beneficial to ensure that the developer intentions were correct.

To make assessing these instances easier, the following grep commands can be used to help find instances of these methods within the application:

```
grep -rni --include='*.rb' 'skip_before' .
grep -rniE --include='*.rb' 'before_(action|filter).*(only|except)' .
```

5 Boilerman

Boilerman is a dynamic analysis tool developed by one of this paper's authors to help in auditing the authentication and authorization filters within a given Rails application. Boilerman itself is an isolated Rails engine that plugs into existing Rails applications. The source code can be found on the Boilerman Github page.[47]

In this section we will be going through an example using OWASP's RailsGoat[48] application.

> "RailsGoat is a vulnerable version of the Ruby on Rails Framework both versions 3 and 4. It includes vulnerabilities from the OWASP Top 10, as well as some "extras" that the initial project contributors felt worthwhile to share. This project is designed to educate both developers, as well as security professionals."

To follow along, first setup the RailsGoat application. See the project's Getting Started[49] page to set the application up.[††]

Once the RailsGoat application is up and running, we can now plug Boilerman into the application.

5.1 Installation

To plug Boilerman into the RailsGoat application, all we need to do is add the boilerman gem into the Gemfile of the RailsGoat application and generate the Boilerman routes. To do this, go into the RailsGoat root and open the `Gemfile`. Add `gem 'boilerman'` to this file:

```
# ...
gem 'crack', '0.3.1'

# Pry for Rails, not in dev group in case running via prod/staging @ a training
gem 'pry-rails'

gem 'boilerman'

group :development, :mysql do

# ...
```

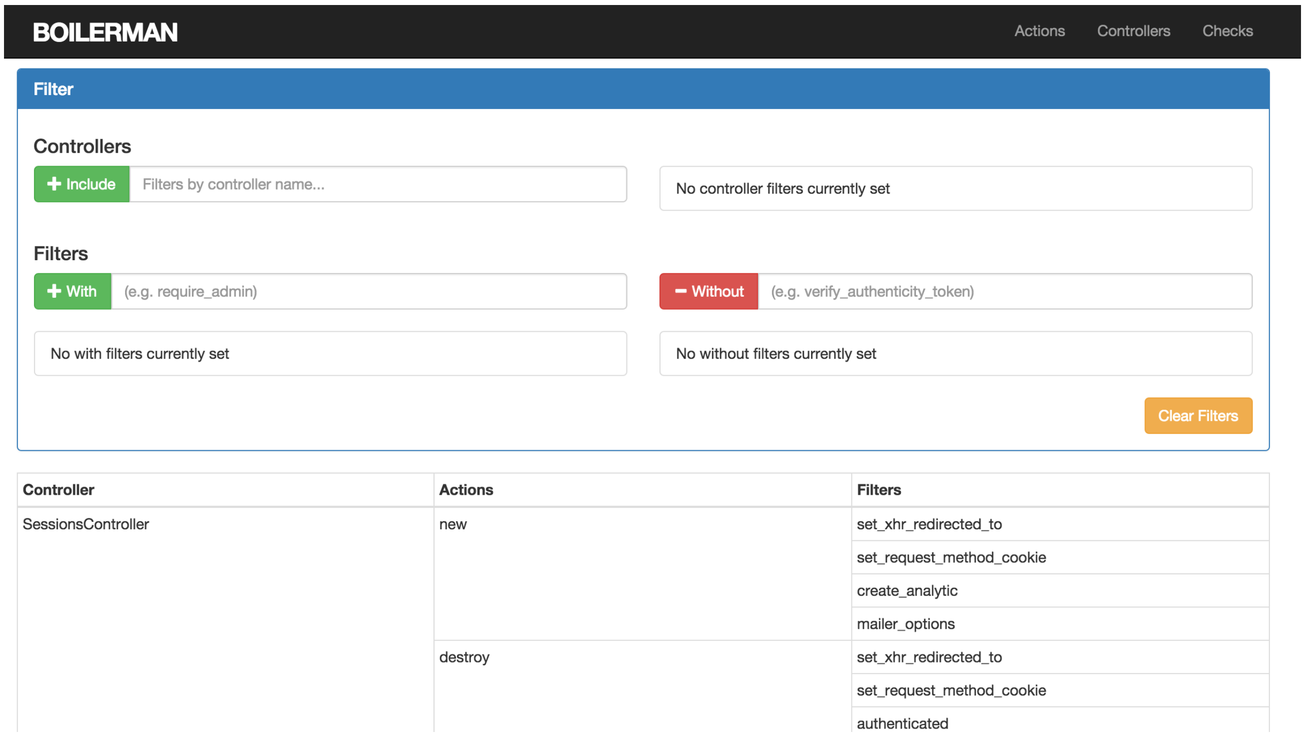
railsgoat/Gemfile

Now run the `bundle install` command after which there is a generator available to add the relevant boilerman routes to the application.

```
railsgoat git:(master): rails generate boilerman:install
       route  mount Boilerman::Engine, at: 'boilerman'
# You can access the Boilerman URL at '/boilerman
```

You can now start the RailsGoat application with the `rails server` and navigate to http://localhost:3000/boilerman. You should now see the following breakdown of controllers, actions, and filters:

[††]If you have issues with libv8, 'brew install v8' and try installing libv8 with: 'gem install libv8 -v '3.16.14.11' – –with-system-v8'


BOILERMAN

Actions Controllers Checks

Filter

Controllers

+ Include | Filters by controller name...

No controller filters currently set

Filters

+ With | (e.g. require_admin)

− Without | (e.g. verify_authenticity_token)

No with filters currently set

No without filters currently set

Clear Filters

Controller	Actions	Filters
SessionsController	new	set_xhr_redirected_to
		set_request_method_cookie
		create_analytic
		mailer_options
	destroy	set_xhr_redirected_to
		set_request_method_cookie
		authenticated

Figure 2: Boilerman view after initial install

5.2 How to use Boilerman

Now that we have Boilerman up and running lets take a look at the filters being used across the controllers of the RailsGoat application. The astute Rails pentester might notice that there is a specific filter missing from each one of the controller's actions. This would be the `:verify_authenticity_token` before_action provided by the `protect_from_forgery` method in the ApplicationController. Sure enough, if we check RailsGoat's ApplicationController, we see that the `protect_from_forgery` method has been commented out.

```
class ApplicationController < ActionController::Base
  before_action :authenticated, :has_info, :create_analytic, :mailer_options
  helper_method :current_user, :is_admin?, :sanitize_font

  # Our security guy keep talking about sea-surfing, cool story bro.
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  #protect_from_forgery with: :exception

  # ...
```

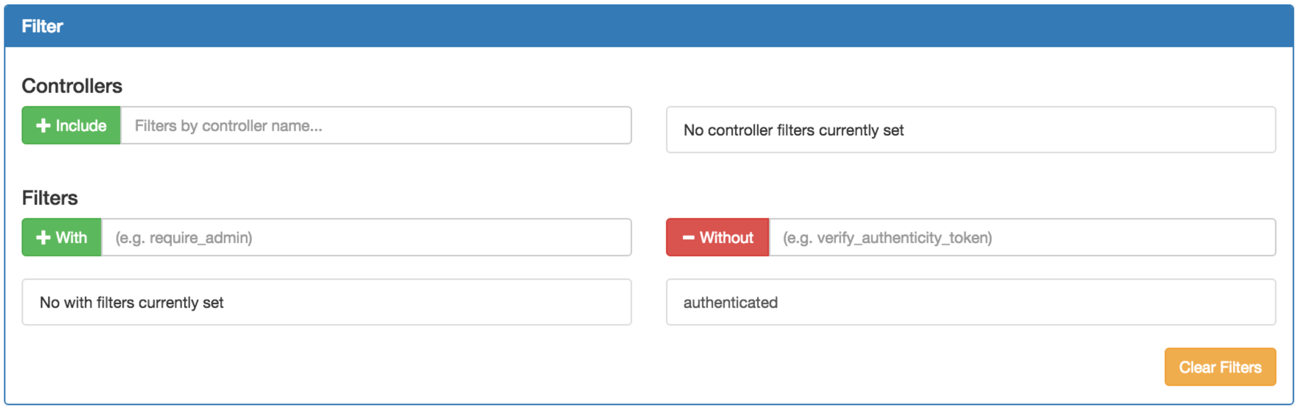
Now lets take a look at what kind of auth based filters are being run on the various controller actions. A good place to begin looking for auth-based controller filters is in the Application-Controller. Looking back at the ApplicationController above, we see the listed before_actions, `before_action :authenticated, :has_info, :create_analytic, :mailer_options`. Of interest is the `:authenticated` before_action.

```
def authenticated
   path = request.fullpath.present? ? root_url(:url =>  request.fullpath) : root_url
   redirect_to path and reset_session if not current_user
end
```

Here we see that the filter redirects to path and resets the session if current_user is nil.

```
def current_user
  @current_user ||= (
    User.find_by_auth_token(cookies[:auth_token].to_s) ||
    User.find_by_user_id(session[:user_id].to_s)
  )
end
```

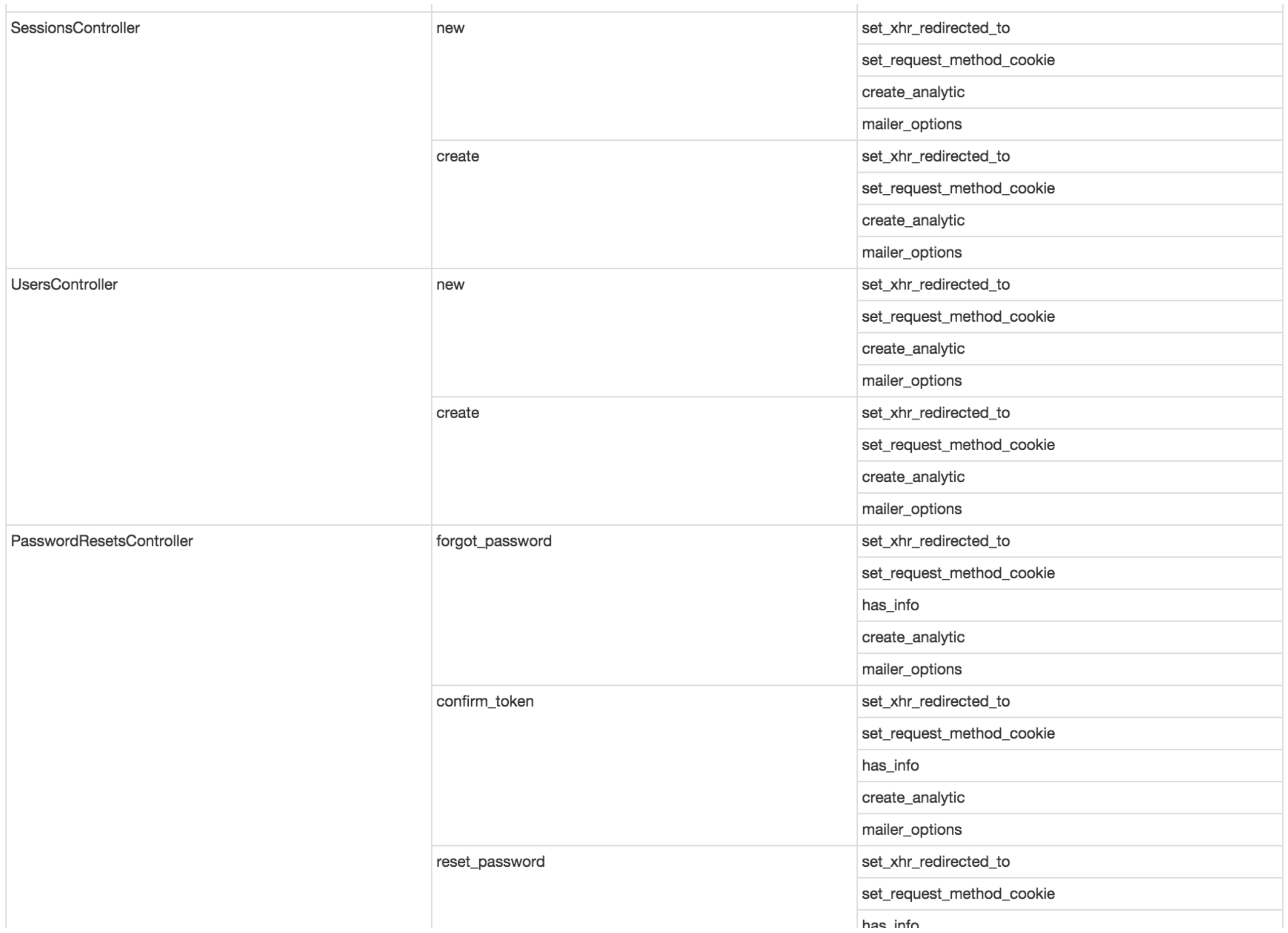
Going back to Boilerman, if we add "authenticated" to the Without filter list:



Filter
Controllers
+ Include
Filters by controller name...
No controller filters currently set
Filters
+ With
(e.g. require_admin)
− Without
(e.g. verify_authenticity_token)
No with filters currently set
authenticated
Clear Filters

Figure 3: "authenticated" added to the Without filter list

We see the controller actions which do not have the `:authenticated` filter applied to it. Initially we see some controller actions that you wouldn't except to require authentication:



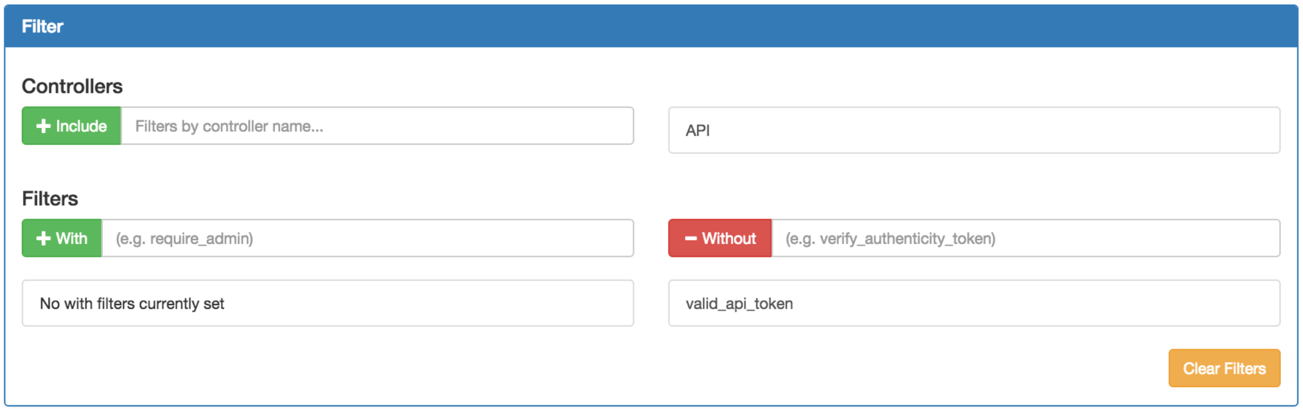
SessionsController	new	set_xhr_redirected_to
		set_request_method_cookie
		create_analytic
		mailer_options
	create	set_xhr_redirected_to
		set_request_method_cookie
		create_analytic
		mailer_options
UsersController	new	set_xhr_redirected_to
		set_request_method_cookie
		create_analytic
		mailer_options
	create	set_xhr_redirected_to
		set_request_method_cookie
		create_analytic
		mailer_options
PasswordResetsController	forgot_password	set_xhr_redirected_to
		set_request_method_cookie
		has_info
		create_analytic
		mailer_options
	confirm_token	set_xhr_redirected_to
		set_request_method_cookie
		has_info
		create_analytic
		mailer_options
	reset_password	set_xhr_redirected_to
		set_request_method_cookie
		has_info

Figure 4: Filtered controller actions

However, if we scroll down through the relevant controller actions, we notice that the `Api::V1::UsersController` does not make use of the `:authenticated` filter and instead uses what looks to be another authentication filter, `:valid_api_token`.

```
def valid_api_token
  authenticate_or_request_with_http_token do |token, options|
    # TODO :add some functionality to check if the HTTP Header is valid
    identify_user(token)
  end
end
```

Now if we add "API" to the controllers list and "valid_api_token" to the without list, we start to see that none of the controller actions on the `Api::V1::MobileController` are using this filter.



Filter

Controllers

+ Include | Filters by controller name...

API

Filters

+ With | (e.g. require_admin)

− Without | (e.g. verify_authenticity_token)

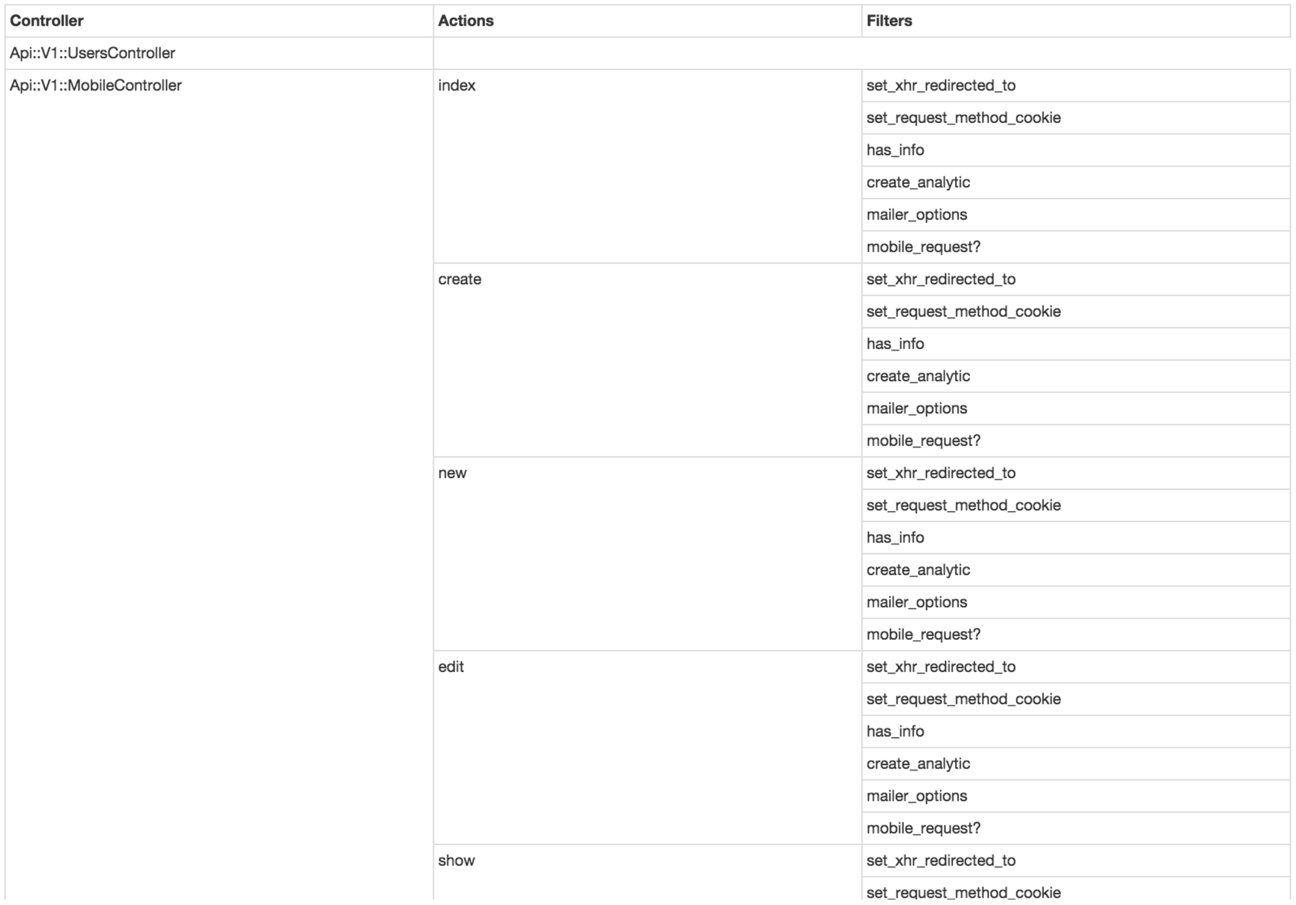
No with filters currently set

valid_api_token

Clear Filters

Figure 5: Searching for "API" controllers without a :valid_api_token filter

Skimming the existing list of filters on the actions, it doesn't look like there are any other auth based filters.



Controller	Actions	Filters
Api::V1::UsersController		
Api::V1::MobileController	index	set_xhr_redirected_to
		set_request_method_cookie
		has_info
		create_analytic
		mailer_options
		mobile_request?
	create	set_xhr_redirected_to
		set_request_method_cookie
		has_info
		create_analytic
		mailer_options
		mobile_request?
	new	set_xhr_redirected_to
		set_request_method_cookie
		has_info
		create_analytic
		mailer_options
		mobile_request?
	edit	set_xhr_redirected_to
		set_request_method_cookie
		has_info
		create_analytic
		mailer_options
		mobile_request?
	show	set_xhr_redirected_to
		set_request_method_cookie

Figure 6: Api::V1::MobileController filter results

If we take a look at `Api::V1::MobileController`, we have two available controller actions, `show` and `index` that seem to contain an authentication bypass.

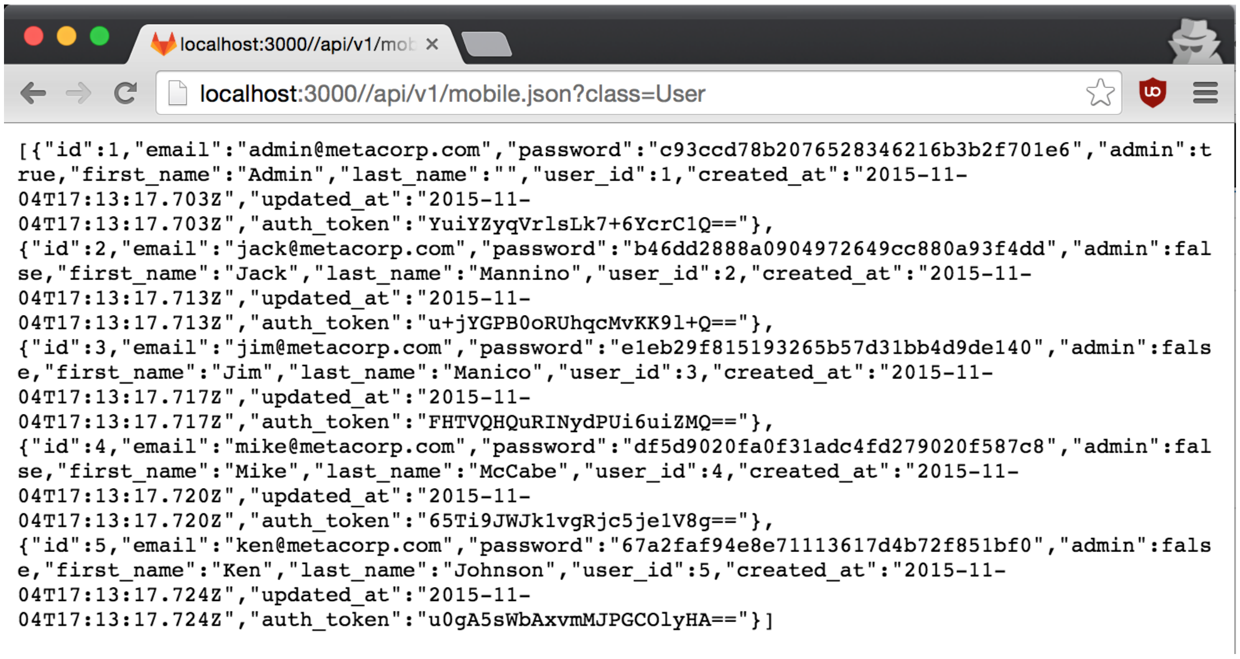
```ruby
class Api::V1::MobileController < ApplicationController
  skip_before_filter :authenticated
  before_filter :mobile_request?

  respond_to :json

  def show
    if params[:class]
      model = params[:class].classify.constantize
      respond_with model.find(params[:id]).to_json
    end
  end

  def index
    if params[:class]
      model = params[:class].classify.constantize
      respond_with model.all.to_json
    else
      respond_with nil.to_json
    end
  end
end
```

Sure enough, if we visit the index route (http://localhost:3000//api/v1/mobile.json?class=User in an unauthenticated browser, we can see that the server responds with a dump of all of the Users if we pass in the `?class=User` query string.


localhost:3000//api/v1/mob
localhost:3000//api/v1/mobile.json?class=User

```
[{"id":1,"email":"admin@metacorp.com","password":"c93ccd78b2076528346216b3b2f701e6","admin":t
rue,"first_name":"Admin","last_name":"","user_id":1,"created_at":"2015-11-
04T17:13:17.703Z","updated_at":"2015-11-
04T17:13:17.703Z","auth_token":"YuiYZyqVrlsLk7+6YcrC1Q=="},
{"id":2,"email":"jack@metacorp.com","password":"b46dd2888a0904972649cc880a93f4dd","admin":fal
se,"first_name":"Jack","last_name":"Mannino","user_id":2,"created_at":"2015-11-
04T17:13:17.713Z","updated_at":"2015-11-
04T17:13:17.713Z","auth_token":"u+jYGPB0oRUhqcMvKK9l+Q=="},
{"id":3,"email":"jim@metacorp.com","password":"e1eb29f815193265b57d31bb4d9de140","admin":fals
e,"first_name":"Jim","last_name":"Manico","user_id":3,"created_at":"2015-11-
04T17:13:17.717Z","updated_at":"2015-11-
04T17:13:17.717Z","auth_token":"FHTVQHQuRINydPUi6uiZMQ=="},
{"id":4,"email":"mike@metacorp.com","password":"df5d9020fa0f31adc4fd279020f587c8","admin":fal
se,"first_name":"Mike","last_name":"McCabe","user_id":4,"created_at":"2015-11-
04T17:13:17.720Z","updated_at":"2015-11-
04T17:13:17.720Z","auth_token":"65Ti9JWJk1vgRjc5je1V8g=="},
{"id":5,"email":"ken@metacorp.com","password":"67a2faf94e8e71113617d4b72f851bf0","admin":fals
e,"first_name":"Ken","last_name":"Johnson","user_id":5,"created_at":"2015-11-
04T17:13:17.724Z","updated_at":"2015-11-
04T17:13:17.724Z","auth_token":"u0gA5sWbAxvmMJPGCOlyHA=="}]
```

Figure 7: RailsGoat returns a dump of all of the User objects

5.3 Conclusion

Before Boilerman, auditing these filters would be a laborious task in which each and every controller in the application would have to be assessed. Boilerman puts all of this information in one place for you in a queryable interface. Unlike Brakeman,[50] Boilerman does require access to the running instance of the application with the ability to modify the source code and relaunch the application. [‡‡] Note that Boilerman isn't meant to replace Brakeman, which is a fantastic static analysis tool that should be used alongside Boilerman (hence the naming convention for this tool). However, Brakeman can't help in assessing filter based authentication and authorization and this is the goal of Boilerman.

[‡‡]In certain restricted environments where access to the host is possible but modification of the application code is not possible, it may be useful to sideload Boilerman through Rails console. See "Force loading Boilerman into a Rails console" https://github.com/tomekr/boilerman#force-loading-boilerman-into-a-rails-console

References

[1] http://skillcrush.com/2015/02/02/37-rails-sites/

[2] http://www.amazon.com/Rails-Way-Obie-Fernandez/dp/0321445619

[3] http://www.amazon.com/Rails-Way-Addison-Wesley-Professional-Ruby/dp/0321601661

[4] http://www.amazon.com/Rails-Way-Addison-Wesley-Professional-Ruby/dp/0321944275

[5] https://www.railstutorial.org/book

[6] http://andrzejonsoftware.blogspot.com/2014/04/be-careful-with-rails-way.html

[7] http://david.heinemeierhansson.com/2012/rails-is-omakase.html

[8] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[9] http://api.rubyonrails.org/classes/ActiveRecord.html

[10] http://api.rubyonrails.org/classes/ActionView.html

[11] http://api.rubyonrails.org/classes/ActionController.html

[12] http://bundler.io/

[13] http://zed0.co.uk/crossword/

[14] https://github.com/plataformatec/devise/blob/7df57d5081f9884849ca15e4fde179ef164a575f/app/controllers/devise/passwords_controller.rb

[15] https://github.com/plataformatec/devise/blob/18a8260535e5469d05ace375b3db3bcace6755c1/lib/devise/models/recoverable.rb

[16] https://github.com/plataformatec/devise/commit/31901bc862db60878130fcd9cbf9c4895d41b2d2

[17] https://github.com/plataformatec/devise/commit/e641b4b7b97159054b7d92fb14df557ac18ae6f4

[18] https://github.com/plataformatec/devise/commit/143794d701bcd7b8c900c5bb8a216026c3c68afc

[19] https://github.com/plataformatec/devise/blob/6ed6e09bf3f8b4e32f16dfe253c89ea6bc0bf525/CHANGELOG.md

[20] http://www.phenoelit.org/blog/archives/2013/02/05/mysql_madness_and_rails/index.html

[21] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0156

[22] https://github.com/rails/actionpack-xml_parser

[23] https://github.com/rapid7/metasploit-framework/blob/master/modules/auxiliary/admin/http/rails_devise_pass_reset.rb

[24] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0233

[25] https://github.com/rails/rails/pull/9208

[26] https://github.com/rails/rails/issues/9292

[27] https://github.com/rails/rails/pull/16069

[28] https://www.owasp.org/index.php/Codereview-Authorization

[29] http://guides.rubyonrails.org/rails_on_rack.html

[30] https://github.com/rails/rails/blob/master/actionpack/lib/abstract_controller/callbacks.rb#L189-L191

[31] http://guides.rubyonrails.org/association_basics.html

[32] http://guides.rubyonrails.org/routing.html#crud-verbs-and-actions

[33] http://guides.rubyonrails.org/routing.html

[34] http://andrewberls.com/blog/post/rails-from-request-to-response-part-2--routing

[36] http://andrewberls.com/images/posts/controller_hierarchy_large.png

[37] https://www.ruby-toolbox.com/categories/rails_authorization

[38] https://github.com/elabs/pundit

[39] https://github.com/CanCanCommunity/cancancan

[40] https://github.com/ryanb/cancan

[41] http://guides.rubyonrails.org/routing.html#crud-verbs-and-actions

[42] https://github.com/CanCanCommunity/cancancan/wiki

[43] https://github.com/elabs/pundit

[44] https://github.com/elabs/pundit#policies

[45] https://github.com/elabs/pundit#ensuring-policies-are-used

[46] https://github.com/elabs/pundit#pundi

[47] https://github.com/tomekr/boilerman

[48] https://github.com/OWASP/railsgoat

[49] https://github.com/OWASP/railsgoat#getting-started

[50] http://brakemanscanner.org/