

Security Audit

Building safe applications

by Aaron Bedra

Security Audit

©2008 Aaron Bedra

Every effort was made to provide accurate information in this document. However, neither Aaron Bedra nor Topfunky Corporation shall have any liability for any errors in the code or descriptions presented in this book.

"Rails" and "Ruby on Rails" are trademarks of David Heinemeier Hansson.

This document is available for US\$9 at PeepCode.com (<http://peepcode.com>). Group discounts and site licenses can also be purchased by sending email to peepcode@topfunky.com.

OTHER PEEPCODE PRODUCTS

- RSpec (<http://peepcode.com/products/rspec-basics>) – A three part series on the popular behavior-driven development framework.
- Rails from Scratch (<http://peepcode.com>) – Learn Rails!
- RESTful Rails (<http://peepcode.com/products/restful-rails>) – Teaches the concepts of application design with REST.
- Subscription pack of 10 (<http://peepcode.com/products/subscription-pack-of-10>) – Save money! Buy 10 PeepCode credits.
- Javascript with Prototype (<http://peepcode.com/products/javascript-with-prototypejs>) – Code confidently with Javascript!
- Rails Code Review PDF (<http://peepcode.com/products/draft-rails-code-review-pdf>) – Common mistakes in Rails applications, and how to fix them.

CONTENTS

5 **So You Want to Audit your Rails App?**

- 6 You write tests don't you?
- 6 I don't know the first thing about security!
- 6 Keep Notes!!!
- 7 Be Brutal

8 **Securing Models**

- 8 SQL Injection
- 13 Unapproved Data Manipulation
- 15 Unauthorized Access Escalation

18 **Securing Views**

- 18 Cross Site Scripting (XSS)
- 22 Cross Site Request Forgery (CSRF)

24 **Crawling and Fuzz Testing**

- 26 Running the test
- 28 What does it all mean?
- 29 How do I fix it?
- 30 Disclaimer!

31 **Keeping your Host on Lockdown**

- 31 Platforms
- 31 Firewall
- 35 Sssshhhh, don't tell them we moved SSH
- 37 General Rules of Thumb

38 **What's the Risk?**

- 38 Why do I care about Risk Analysis?
- 45 Conclusions

47 **Final Thoughts**

- 47 Create Guidelines, not Rules!

So You Want to Audit your Rails App?

CHAPTER 1

Everyday, thousands of applications are hacked.

Bank account numbers are exposed, social security information is compromised, credit cards are revealed, and other types of personal information is accessed. Don't let your Rails application be one of them! Let's pick apart your code and see what places are vulnerable to attacks. We will cover SQL Injection in your models, sanitization logic in your controllers, Cross Site Scripting and Cross Site Request Forgery in your views, and improper setup of your host operating system. The information we cover could very well be the most important thing you learn as a developer aside from writing the code itself.

There are a ton of reasons to audit your application. You could work for a company that is regulated under Sarbanes-Oxley (SOX) or HIPAA. You might be launching a new startup that is going to be taking credit card transactions. Or, you might just want to keep your users' data safe. No matter why you decide to audit your application, the process is the same. Even if you don't have nuclear launch codes being passed over the Internet, you should be auditing your work to make sure you haven't introduced any security holes.

Audits take place every day for many different reasons. It's a great idea for you to learn how to perform an audit. Once you are comfortable with the process you can start scheduling regular audits. Along with increasing the security of your application, you will more than likely increase the overall quality of your code as well.

You write tests don't you?

Just think of this as another test. Some of it can be automated, but eventually there will be manual verification. This can be off-putting to some people, but I assure you it is worth it. During the process you will laugh a little, cry a little, and probably make a few Yaks really really cold (to be explained later). It is a process that will change the way you develop your next application.

I don't know the first thing about security!

That's OK, you don't have to be a security expert to audit your application. It does help to know a thing or two about haxoring, or to be able to think like a hacker, but it's not necessary. This book will get you well on your way to being an effective auditor and will probably inspire you to read more about information security.

Keep Notes!!!

One of the most important things you can do during an audit is keep a very good set of notes. Document everything you do. If you have even the slightest notion that something is awry, make a note of it, because you will need these notes later. I recommend using an electronic solution so you can easily search your notes for a keyword or phrase. Your notes will probably end up being a giant ball of nonsense that only you can make sense of, so having the ability to search through it will help when you go back to write a sane version in report form.

There are many applications that can help you with taking notes. Desktop programs like Voodoo Pad (<http://flyingmeat.com/voodooopad>), Yojimbo (<http://www.barebones.com/products/yojimbo>), or TaskPaper (<http://hogbaysoftware.com/products/taskpaper>) are great for this task. You can

even keep it simple and just use a text file or even the trusty emacs scratch buffer.

Be Brutal

It can be a hard thing to audit your own code. I actually recommend against it! If you audit your own application, you'll be thinking about how much work it will be to fix all the holes you find, which may cause you to treat it more gently than you should. So if you wrote it, let someone else audit it.

If you have no other choice, you need to be honest with yourself about things that are wrong. If it's not your code, it's much easier to be a jerk about what's wrong. You do however, need to be as brutal as you can be when it comes to auditing software. If something seems even the slightest bit wrong, document it for further scrutiny.

Securing Models

CHAPTER 2

The most important part of any application isn't your code or the graphics, it's the application's data. You can rewrite code, but it's very hard to recreate data that has been deleted or manipulated. And it's impossible to retrieve data once it has been leaked!

SQL Injection

Your application is running smoothly on the web. Everything is going nicely until one morning you get an email as you are reading your RSS reader. The site doesn't seem to be working.

As you sit down to troubleshoot, you quickly notice that the data you had yesterday is no longer there. In fact, all your data is missing. You restore from the backups you pulled last night and all is well. Then, a look at your logs reveals some nasty SQL as the culprit of your disaster.

SQL injection accounts for a significant number of web related application break-ins. In the past few years there have been huge improvements in the effort to stop SQL injection related exploits.

SQL injection is a technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when

- User input is incorrectly filtered.
- Escape characters embedded in SQL statements are not correctly sanitized.
- User input is not strongly typed and thereby unexpectedly exe-

SQL INJECTION RESOURCES

There are a lot of great SQL injection related resources on the web. OWASP has a great section (http://www.owasp.org/index.php/Testing_for_SQL_Injection) on testing for SQL injection if you want to learn more on the subject.

cuted.

SQL injection is in fact an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another.

How does it happen?

There are a lot of different ways to attack an application that is vulnerable to SQL injection attacks. Let's take a look at how to exploit insecure code using the aforementioned dangerous group of methods. We will use examples from a few and demonstrate what can happen if injection code is passed in unsanitized.

```
Asset.find(:all, :limit => "#{params}")
```

If we passed something similar to this:

```
10 procedure help()
```

We can expect an output and result similar to this:

```
Unknown procedure 'help' : SELECT * FROM 'assets'
```

This is not good! It looks like we are able to directly call stored procedures from the `limit` call. If we could enumerate what stored procedures are on this database, we could start executing those procedures.

Another thing that could happen would look like this:

```
Asset.find_by_sql("SELECT * FROM assets WHERE id=#{params}")
```

An attacker could pass in parameters similar to these:

```
' ; DELETE FROM assets WHERE 1'
```

The initial result of the call would just return an empty array. No errors would be thrown, and nothing would blow up in your log files, but your assets would all be gone! This is an example of a much more destructive attack that can be used on an application vulnerable to SQL injection.

A classic comic on this issue is at xkcd (<http://xkcd.com/327>).

The good news for you is that Rails gives you awesome protection right out of the box. The bad news is that you can still go out of your way to create holes in your application. Let's quickly highlight the good bits that Rails gives you for free.

```
Asset.find params[:id]
```

This `find` accepts a user argument passed in directly from the user. Normally this is bad practice to put user input directly into your database calls, however, Rails ensures that anything passed in via `params` will be safe to accept without any further intervention. That being said, let's move on to some more interesting things that can cause real problems in your application.

- `find_by_sql`
- `execute`
- `find` with conditions in a string (i.e. `:conditions => ["asset = #{little_bobby_tables}"]`)
- `limit`

LITTLE BOBBY DROPTABLES SAYS:

Make sure when you find bugs like these you write tests along with the bugfixes.

This will only take a few minutes more and will make sure that you or another developer doesn't accidentally reintroduce this bug later on in the development cycle.

- `offset`
- `group_by`
- `order`.

These methods are not automatically sanitized and should be used with caution. If you are auditing your source code and see any of these methods used, make sure that they are being properly sanitized.

How do I fix it?

SQL injection vulnerabilities are nasty little problems if left unfixed. Luckily, the fixes are rather simple and won't take too much of your time to complete.

Our first example of injectable code can be fixed by a regular expression as simple as:

```
params[:limit].gsub(/\D/, '')
```

This will strip out all non-numerical characters from the string and leave you with just the numerical `limit` for your search. You would need to call this method any time you accept these types of inputs.

You could make this into a utility method and duck-punch it into Ruby's `String` class.

Even though Ruby is a dynamic language and is not strictly typed, your application should still expect to see certain kinds of data. You may need to check for any of the following:

- Strings without whitespace

- Strings with whitespace trimmed from the front and end
- Numerical input
- Float or decimal values
- Special formats such as phone numbers, zip codes, tax ID numbers, email addresses, etc.

Some of these may be satisfied with a simple regular expression check. Others may need to be validated against a reference table with valid values. It's safest to check for a specific kind of input and reject all others.

Don't try to escape complicated SQL directly since the rules may be different depending on the database in use. To avoid costly mistakes, you can also take advantage of methods such as `ActiveRecord::Base::connection.quote` inside methods that expose raw SQL.

```
sql_injection/article.rb
class Article < ActiveRecord::Base

  def find_with_limit(limit)
    find(:all, :limit => connection.quote(limit))
  end

end
```

There are quite a few plugins out there that utilize some of the dangerous methods described above in unsanitized ways. If you're unsure, read the code and run the Tarantula plugin against your application, as will be shown later.

The `will_paginate` plugin is safe. It makes use of `limit` and `offset` but takes the potential problems into account.

Unapproved Data Manipulation

Rails makes it easy to update many fields at once by passing a Hash of values to ActiveRecord.

Unfortunately, this also makes it easy for a user to POST arbitrary values and manipulate data in unexpected ways.

How does it happen?

Here's a snippet from an idiomatic update action.

```
# Potentially unsafe update from a form
@order.update_attributes(params[:order])
```

But what if someone calls the update action with extra information, such as a new price for the order?

We might feel that we are safe since our HTML form only includes the fields that we want to update. Furthermore, how would anyone discover the names of our database fields? Aren't these stored safely and securely inside our code?

Unfortunately, the default Rails scaffolding exposes table field names via the XML format. Accessing <http://localhost:3000/orders/1.xml> shows them all:

```
<order>
<created-on type="datetime">2008-05-27</created-on>
<id type="integer">1</id>
<price type="integer">123</price>
<state>pending</state>
<updated-on type="datetime">2008-05-27</updated-on>
</order>
```

From here, it's a simple matter for someone to set their own price by sending a POST with a new price. They might even POST a state of *paid* and bypass the payment process altogether!

```
/orders/1?order%5Bprice%5D=1&order%5Bstate%5D=paid
```

This may seem like an obscure threat, but libraries such as `ActiveResource` make it possible to exploit vulnerabilities like this in only a few lines of code. Sites that use Rails scaffolding without modification are vulnerable to this kind of attack right out of the box.

How do I fix it?

`ActiveRecord` provides a simple directive to protect fields: `attr_protected`. List sensitive fields that should be protected and Rails will ignore them during bulk field updates via `new`, `create`, or `update_attributes`.

```
attr_protected/order.rb
class Order < ActiveRecord::Base

  attr_protected :price, :state

end
```

If you do need to modify these values, you can set them directly.

```
@order.price = params[:order][:price]
```

To do the reverse, use `attr_accessible`. This will expose only the fields you list and will lockdown all other fields.

Unauthorized Access Escalation

Every Rails tutorial shows you how to pull information from a table. It's a simple `find` with an `id`. It looks so innocent! What could be wrong with that?

The problem is that unscoped queries allow any user to access any piece of data, even if it does not belong to them.

How does it happen?

Let's assume that you have an application that stores and lists tasks for users. The `show` action for a single user's tasks might look like this:

```
# Potentially dangerous!  
@task = Task.find(params[:id])
```

The problem is that any user can look at any other user's tasks. If you use auto-incrementing integers as your `id` (which is the default), it's as easy as logging in as any user and typing `example.com/tasks/1`, `example.com/tasks/2`, `example.com/tasks/3`, into the browser's address bar.

Lather, rinse and repeat for `create`, `update`, and `destroy`, and you've just given every user the ability to manipulate any other user's data!

How do I fix it?

Fortunately, ActiveRecord provides an easy way to limit access to associated records. Instead of issuing queries against the model's class, you should issue a query through the model that owns the record.

For our user and task example, the restful-authentication plugin (<http://github.com/technoweenie/restful-authentication/tree/master>) provides a `current_user` method that represents the ActiveRecord row for the logged-in user. We can use this to generate subqueries on associated models.

Here is a basic User model:

```
models/user.rb
class User < ActiveRecord::Base

  has_many :tasks

end
```

Instead of querying the Task model, we can query the tasks associated with this user.

```
@task = current_user.tasks.find(params[:id])
```

This generates SQL roughly similar to the following:

```
SELECT * FROM tasks WHERE (tasks.user_id = 42)
```

Protecting data from unauthorized access is that easy! If a user tries to type in the ID of a record they don't own, an empty set will be returned.

Any query that would be run against the Task class can be run against the association.

```
# Dynamic conditions
current_user.tasks.find_all_by_status('Completed')
# Additional options
current_user.tasks.find(:all, :limit => sanitized_limit)
```


Securing Views

CHAPTER 3

Cross Site Scripting (XSS)

As you are running through the admin panel of your application checking out new user information for your awesome new startup, you come across a user's profile that does something very strange.

Every time you click to view the user's information, you are redirected to another page and your username and password hash are displayed right on the page.

What in the world is going on? There isn't any code in your application that points to this site, and for that matter would never display information in such a way.

Cross Site Scripting (XSS) is an issue for every web developer, no matter the framework. This problem is magnified by the general lack of understanding of the core problems that XSS introduces, as well as the simple fact that developers often overlook XSS issues when developing an application.

Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications which allow code injection by malicious web users into the web pages viewed by other users. Examples of such code include HTML code and client-side scripts. An exploited cross-site scripting vulnerability can be used by attackers to bypass access controls such as the same origin policy. Vulnerabilities of this kind have been exploited to craft powerful phishing attacks and browser exploits.

XSS RESOURCES

If you want to learn more about XSS you can find more information here. (http://www.owasp.org/index.php/Cross_Site_Scripting) If you would like to further explore CSRF you can look here. (http://www.owasp.org/index.php/Testing_for_CSRF)

Example

To be able to bookmark pages, search engines generally leave search variables in the URL address. In this case, the URL would look like:

```
http://test.example.com/search.php?q=XSS%20
```

Next we try to send the following query to the search engine:

```
<script type="text/javascript"> alert('This is an XSS  
Vulnerability') </script>
```

By submitting the query to `search.php`, it is encoded and the resulting URL would be something like:

```
http://test.example.com/search.php?q=%3Cscript%3Ealert%28%91  
This%20is%20an%20XSS%20Vulnerability%92%29%3C%2Fscript%3E
```

Upon loading the results page, the test search engine would probably display empty results for the search, but it will display a JavaScript alert which was injected into the page. This type of thing can lead to much more powerful and crafted attacks against your application.

An XSS attack takes advantage of unescaped output to add harmful Javascript code into the HTML rendered by your application. There is a potential vulnerability anywhere that users can enter content into your website (blog comments, profile pages, etc.). You can sanitize inputs (user-generated content) but sanitizing outputs is more bulletproof (rendered HTML).

How do I detect XSS?

Scan your source code looking for unescaped values where users could enter harmful data.

The guideline for this is simple. If you see an opening ERB tag without a corresponding `h()`, you have an item to add to your audit list. It doesn't necessarily mean you have a vulnerability, it just means you need to take a look at why you haven't escaped the data. This will provide the fastest turn around for hardening your application and is very easily scriptable.

How do I fix it?

By following the guidelines above, you will ultimately produce a huge amount of auditable code. This is OK, don't be overwhelmed. You will find that fixing these issues takes almost no time at all and is rather painless.

You will, however, find that there are a couple cases that you will have trouble with and you will have to make some decisions as to the applicable risk associated with the problem. We will cover how to assess risk later on.

For now here's what you need to do. When you see something like this:

```
<%= @asset.description %>
```

Make sure you that you consider it a potential for XSS code execution and simply fix it like so:

```
<%= h(@asset.description) %>
```

You can also omit the parentheses:

```
<%=h @asset.description %>
```

This simple fix will escape any potential problems that could arise from a user entering bad bits into your database.

Tainting

There comes a time in this process where you will have to make a decision as to how you want to globally handle XSS protection for your application. Ultimately you are faced with the issue of tainting versus not.

Simply put, *tainting* is a process where you mark any input that doesn't come directly from your application. This is done using ruby's `taint()` function. Once you have decided to do this you need some way to keep track of managing what has been tainted and make sure that you don't render anything that has been deemed as such.

Luckily for you there is a solution just for this. The SafeERB (<http://rubyforge.org/projects/safe-erb>) plugin originally written by Shinya Kasatani and further maintained and updated for Rails 2 by yours truly does just this.

There is, however, a drawback to using SafeERB. It raises an error anytime you try to render a tainted string. This can quickly become quite a pain, especially if you have a rather large application that you haven't already audited for XSS correctness.

The double whammy comes in the fashion of poor testing. If you don't have good tests, you may uncover errors in your production application without warning. If you have your heart set on using

SafeERB, you should consider it for new projects and probably nothing else.

If you don't wish to go the tainting route, there are other alternatives that end up being much more palatable. XSS Shield (<http://code.google.com/p/xss-shield>) is a plugin that will seamlessly protect your application via the `h()` method without any intervention from you. Per the documentation:

```
<%=# By default, the h() method must be used. %>
<h3><%= h(item.name) %></h3>
<p><%= link_to "#{h(item.first_name)}'s stuff",
              :action => :view,
              :id => item %>
</p>

<%=# Instead, the plugin does it for you. %>
<h3><%= item.name %></h3>
<p><%= link_to "#{item.first_name}'s stuff",
              :action => :view,
              :id => item %>
</p>
```

All your views will be automatically protected. This is a nice solution and requires very little effort to accomplish your goal. This pretty much makes a winning combination for solving the XSS problem on your application.

Cross Site Request Forgery (CSRF)

Cross-site request forgery, also known as *one click attack*, *sidejacking* or *session riding* is abbreviated as CSRF (Sea-Surf) or XSRF. CSRF is a type of malicious exploit of websites. Although this type of attack has similarities to cross-site scripting (XSS), cross-site scripting requires the attacker to inject unauthorized code into a website, while cross-site request forgery merely transmits unauthorized com-

mands from a user the website trusts.

Rails 2 gives you CSRF protection out of the box. It automatically provides a session token for all requests that it compares your request with so that hijacking can not take place. Unless the code you are auditing has done something to turn it off, you can rest assured that this shouldn't be an item of concern.

http://localhost:3000/orders/1/edit

☐ **1 cookie**

| | |
|----------------|--|
| NAME | _unsecureapp_session |
| VALUE | BAh7BllKZmxhc2hjQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNo%250ASGFzaHsABjoKQHVzZWR7AA%253D%25; |
| HOST | localhost |
| PATH | / |
| SECURE | No |
| EXPIRES | At End Of Session |

FIG. A RAILS 2 USES A SESSION KEY TO PROTECT AGAINST CSRF

Crawling and Fuzz Testing

CHAPTER 4

A Security fuzzer is a tool used by security professionals (and professional hackers) to test the parameters of an application. It feeds garbage data into various form and URL inputs in an attempt to cause an error.

Typical fuzzers test an application for buffer overflows, format string vulnerabilities, and error handling. More advanced fuzzers incorporate functionality to test for directory traversal attacks, command execution vulnerabilities, SQL Injection and Cross Site Scripting vulnerabilities. Web Vulnerability scanners typically perform all of this functionality, and can be considered an advanced fuzzer.

Your audit is not complete without a thorough examination of all the links and inputs of the target application. This can be a difficult process even for the most skilled of QA analysts. This kind of test really calls for automation. Writing a crawler and fuzzer from scratch can also be a daunting task if you are not sure what to look for. Luckily this task has been completed for you! It's as simple as installing the Tarantula plugin.

```
./script/plugin install \  
  git://github.com/relevance/tarantula.git
```

If your application isn't on Rails 2.1, you can clone it directly to the `vendor/plugins` directory.

```
git clone git://github.com/relevance/tarantula.git vendor/  
plugins/tarantula
```

You will have to install the `facets` and `htmlentities` gems along with

this plugin.

```
sudo gem install facets htmlentities
```

Once you have Tarantula installed, just run the `tarantula:setup` task to generate the default test.

```
rake tarantula:setup
```

You will end up with the following test in your application's `test/tarantula` folder.

Although this looks like a standard integration test, it will only work correctly if a single `test` method is defined. Future versions of the plugin may support fuzzing via multiple user sessions.

Here's the default test:

```
fuzzing/tarantula_test.rb
require "#{File.dirname(__FILE__)}/../test_helper"
require "relevance/tarantula"

class TarantulaTest < ActionController::IntegrationTest
  fixtures :all

  def test_tarantula
    post '/session', :login => 'quentin', :password =>
'password'
    follow_redirect!
    tarantula_crawl(self)
  end
end
```

Tarantula assumes that you have a set of fixtures with valid data inside. If you don't have fixtures for a table, you can use the `ar_fix-`

TARANTULA

Tarantula is an open source application written by Relevance (<http://opensource.thinkrelevance.com>) that crawls your application and fuzzes all available inputs. Feedback to this project is always welcome.

tures plugin (http://topfunky.net/svn/plugins/ar_fixtures) to dump development or production data to YAML fixtures.

RSpec doesn't currently have integration tests. You can use a `Test::Unit` integration test for fuzzing even if the rest of your specs are written with RSpec.

This is the quick way to get started. For more information and examples see the Wiki Page (<http://opensource.thinkrelevance.com/wiki/tarantula>).

Running the test

To run Tarantula, just grab a console and run:

```
rake tarantula:test
```

The Tarantula test may take a while to run depending on the size of your application and speed of your machine. One application we ran took 5 minutes on a slow laptop, while others ran in a few seconds on a fast desktop machine.

After it's finished, it will produce an HTML report detailing failures and successes. If you're on a Mac, it will automatically display the results in your default browser. If you are using any other system you will have to manually open the report in your web browser. The report is located in the `tmp/tarantula` directory of your application.

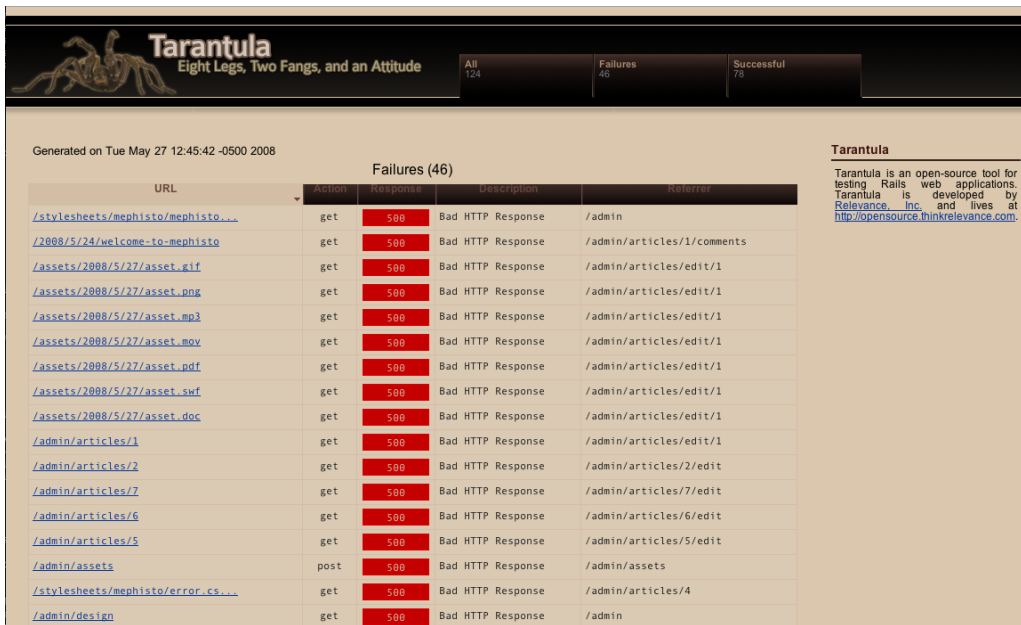
If no report is generated, try the following:

- Does your application have a `SessionController` that is used for login? If not, you may need to change the path that Tarantula

sends a post to.

- Does your login form have login and password fields? If not, you may need to change those in the test.
- Is the value of the login and password correct? The data in your fixtures will be used to verify the login.
- Do you have a full set of fixtures?
- If all else fails, run `rake tarantula:test -t` in order to see a more complete listing of the errors encountered.

Here is an example of what your report might look like.



The screenshot shows the Tarantula web interface. At the top, there is a header with the Tarantula logo and the tagline "Eight Legs, Two Fangs, and an Attitude". To the right of the logo, there are three tabs: "All 24", "Failures 46", and "Successful 76". Below the header, the report is generated on "Tue May 27 12:45:42 -0500 2008". The main content area is titled "Failures (46)" and contains a table with the following columns: "URL", "Action", "Response", "Description", and "Referer". The table lists 17 failed requests, all with a "500" response and a "Bad HTTP Response" description. The referers are either "/admin" or "/admin/articles/edit/1".

| URL | Action | Response | Description | Referer |
|---|--------|----------|-------------------|----------------------------|
| /stylesheets/mephisto/mephisto... | get | 500 | Bad HTTP Response | /admin |
| /2008/5/24/welcome-to-mephisto | get | 500 | Bad HTTP Response | /admin/articles/1/comments |
| /assets/2008/5/27/asset.gif | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /assets/2008/5/27/asset.png | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /assets/2008/5/27/asset.mp3 | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /assets/2008/5/27/asset.mov | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /assets/2008/5/27/asset.pdf | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /assets/2008/5/27/asset.swf | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /assets/2008/5/27/asset.doc | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /admin/articles/1 | get | 500 | Bad HTTP Response | /admin/articles/edit/1 |
| /admin/articles/2 | get | 500 | Bad HTTP Response | /admin/articles/2/edit |
| /admin/articles/7 | get | 500 | Bad HTTP Response | /admin/articles/7/edit |
| /admin/articles/6 | get | 500 | Bad HTTP Response | /admin/articles/6/edit |
| /admin/articles/5 | get | 500 | Bad HTTP Response | /admin/articles/5/edit |
| /admin/assets | post | 500 | Bad HTTP Response | /admin/assets |
| /stylesheets/mephisto/error.cs... | get | 500 | Bad HTTP Response | /admin/articles/4 |
| /admin/design | get | 500 | Bad HTTP Response | /admin |

FIG. B TARANTULA HTML REPORT.

You will come across several different types of results after running Tarantula.

The most obvious is the successful hits, marked by a green box. Tarantula keeps track of everything it does and will detail all links it crawls regardless of the response.

What does it all mean?

Tarantula will tell you *that* something was wrong, but it rarely tells you exactly *what* was wrong. That's your job!

Most of the application's failures are documented by the HTTP response code generated. You should aim to fix all the failures, but the ones you are immediately concerned with are the 500 errors. These are typically caused by the built in fuzzing that Tarantula offers.

Be aware that some of the errors you encounter can be caused by improperly formed HTML. These should still be fixed even if they don't reflect a problem with the operational code in your application.

If Tarantula finds a form, it will generate garbage data and feed it into the form. If you have not properly sanitized your inputs, your application will most likely generate a 500 error. This is the worst type of error to have in terms of security because it means that a user was able to do something that caused your application to crash. It also tells an attacker that there is improperly secured code that they can start using as an attack vector to hopefully break into your application. A good attacker can use enumeration techniques to inspect your application's HTTP response codes to isolate weak sections and act accordingly.

So the 500 errors are the first errors you should fix.

How do I fix it?

When analyzing the 500 errors, it is important to understand there are no innocuous errors. If your application generates an error, you may be able to analyze the code and prove to yourself that there is no real security vulnerability. This is usually a bad idea because it costs more time and effort than simply fixing the code, and you might be wrong.

Fixing the errors provided by Tarantula should be fairly straightforward. First, click on the URL link for the page in question. You'll see the Rails error message and backtrace of the error. If it was a POST, PUT, or DELETE, you'll see the parameters used to access the page. These will often be nonsensical values that were used to confuse the application.

```
"product[description]"=>4539, "product[title]"=>-3450,
```

The error detail output can sometimes be confusing since it's HTML-based and is meant to be viewed in a web browser. However, you can usually find the file and line number of the error.

Next, open the file in question and look at the code. Are certain types of data being expected but not found? Is there an error condition that isn't being caught properly?

For example, Tarantula identified a sloppy line of code in an application that was relying on the existence of specific form data.

```
item.type = @items.detect { |i| i.ext == item_ext }.name
```

The solution was to verify the result of the `detect` loop before assuming that we could extract the `name` from it.

You will more than likely stumble upon some head scratchers, but don't worry! With a little bit of time and some good tests, you will uncover the errors and be much better off than you were before you let this fuzzy spider attack your application.

Disclaimer!

Making the decision to use Tarantula means that you now have to write a solid test suite to back it up. If you do not have good tests, it's time to man up! You will not be very successful with Tarantula until you do.

PeepCode has several screencasts on RSpec (<http://peepcode.com/products/rspec-basics>) and one on Test-First Development (<http://peepcode.com/products/test-first-development>) if you're not familiar with the basics of testing.

When you create your Tarantula test, be sure to choose a user that has full access to everything in your application. You want to ensure that every possible link is crawled and every form fuzzed.

You can create separate tests for separate roles, but for a thorough examination, the user needs to be able to see everything.

Keeping your Host on Lockdown

CHAPTER 5

None of your newfound auditing-fu will do you any good unless the system you are deploying to is secure. If an attacker gains control of your server, they can do whatever they want to your application and your data. Since we obviously don't want that to happen, let's go over some helpful hints to check the security of your box.

Platforms

You can deploy a Rails application on just about any platform. Your options for deployment are Windows, Linux, Solaris, or the BSDs. Given this wide range of options, choosing a platform can be a touch choice. The most popular deployment platform for Rails applications is Linux, so we will focus our efforts on auditing a Linux machine.

There are hundreds of Linux distributions that are purposed for just about any task. We will be highlighting certain features of Linux distributions that aid in securing the machine.

Firewall

This is the cornerstone and basis for security on your server. Making sure you have a good firewall ruleset in place will get you farther than almost any other security tactic.

In order to test your server, let's use `nmap`. Nmap is a network mapping tool that shows which ports on a machine are open, along with version numbers of the running software being exposed to the Inter-

net. Nmap is easily installed via your package management system of choice. To get the latest version you can go to the nmap homepage (<http://insecure.org/nmap>) and compile the source yourself.

Now that we have nmap installed, let's get to checking our server. Open a terminal window and enter the following:

```
nmap -P0 [your host's ip address]
```

This will do a port scan of your host without pinging it. A lot of servers disable the echo ping protocol, so this will make sure that you get something useful back. You should get an output something similar to this:

```
Starting Nmap 4.20 ( http://insecure.org ) at 2008-04-13
09:55 EDT
Interesting ports on xxx-xxx-xxx-xxx.yourdomain.com (xxx.
xxx.xxx.xxx):
Not shown: 1691 closed ports
PORT      STATE SERVICE
21/tcp    open  ftp
22/tcp    open  ssh
80/tcp    open  http
554/tcp   open  rtsp
3333/tcp  open  dec-notes
7070/tcp  open  realserver
```

This is good but we can do better with the -A flag.

```
nmap -P0 -A [your host's ip address]
```

This will do a full signature scan of the entire host. It will determine what applications are running on the exposed ports and most of the time can even show what version of the software they are running. Now you get a more interesting output.


```
Starting Nmap 4.20 ( http://insecure.org ) at 2008-04-13
10:04 EDT
Interesting ports on xxx-xxx-xxx-xxx.yourdomain.com (xxx.
xxx.xxx.xxx):
Not shown: 1691 closed ports
PORT      STATE SERVICE      VERSION
21/tcp    open  tcpwrapped
22/tcp    open  ssh          OpenSSH 4.6p1 Debian 5ubuntu0.2
(protocol 2.0)
80/tcp    open  http-proxy  nginx http proxy 0.5.35
554/tcp   open  tcpwrapped
3333/tcp  open  dec-notes?
7070/tcp  open  tcpwrapped
Service Info: OS: Linux
```

This output shows you the version numbers of everything running and even takes a good guess as to what operating system you are running. An attacker will use any information they can get their hands on to attack your system, so you should do your best to restrict what they can see and access.

There are way too many open ports on this server. We really only want to be able to SSH into the host to manage it, and be able to serve traffic out of port 80, the HTTP port. Everything else must go!

There are several ways to accomplish this task. You can put a firewall up in front of your server that blocks everything but the desired traffic. You can also just take advantage of Linux's built in firewall software `iptables`. We will take the `iptables` approach since it comes standard on almost any Linux distribution and can be activated very easily. The only thing `iptables` requires is a script to tell it what to do.

Let's write a firewall script that only allows traffic into the server on port 80 and port 22 (some lines wrap).

```
*filter
-A INPUT -i lo -j ACCEPT
```

```

-A INPUT -d 127.0.0.0/255.0.0.0 -i ! lo -j REJECT --reject-
with icmp-port-unreachable
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p tcp -m tcp --dport 80 -j ACCEPT
-A INPUT -p tcp -m state --state NEW -m tcp --dport 22 -j
ACCEPT
-A INPUT -m limit --limit 5/min -j LOG --log-prefix "iptables
denied: " --log-level 7
-A INPUT -j REJECT --reject-with icmp-port-unreachable
-A FORWARD -j REJECT --reject-with icmp-port-unreachable
-A OUTPUT -j ACCEPT
COMMIT

```

Let's save this file as `iptables.rules` and store it in `/etc`. Now we just need to activate it and test it out. To load the ruleset into `iptables`, simply run the following:

```
sudo iptables-restore < /etc/iptables.rules
```

Once that's complete, we need to check `iptables` to make sure that our ruleset was applied properly.

```
sudo iptables -L
```

If things worked properly, you should see output similar to this:

```

Chain INPUT (policy ACCEPT)
target     prot opt source                destination
ACCEPT    0    --  anywhere              anywhere
REJECT    0    --  anywhere              127.0.0.0/8
reject-with icmp-port-unreachable
ACCEPT    0    --  anywhere              anywhere
state RELATED,ESTABLISHED
ACCEPT    tcp  --  anywhere              anywhere
tcp dpt:www
ACCEPT    tcp  --  anywhere              anywhere
state NEW tcp dpt:ssh
LOG       0    --  anywhere              anywhere

```

```

limit: avg 5/min burst 5 LOG level debug prefix `iptables
denied: '
REJECT    0    --  anywhere          anywhere
reject-with icmp-port-unreachable

Chain FORWARD (policy ACCEPT)
target    prot opt source          destination
REJECT    0    --  anywhere          anywhere
reject-with icmp-port-unreachable

Chain OUTPUT (policy ACCEPT)
target    prot opt source          destination
ACCEPT    0    --  anywhere          anywhere

```

The last thing we need to do is to test our host again with nmap. Let's use the first nmap method as shown above. Now we get an output similar to this:

```

Starting Nmap 4.20 ( http://insecure.org ) at 2008-04-13
09:55 EDT
Interesting ports on xxx-xxx-xxx-xxx.yourdomain.com (xxx.
xxx.xxx.xxx):
Not shown: 1691 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http

```

This looks much better! Let's move onto some other tips and tricks to help you keep the hackers at bay

Sssshhhh, don't tell them we moved SSH

There are thousands of brute force bots on the Internet launching attacks against port 22. The solution? Simply move the port your SSH server runs on. You can put an end to this quickly by modifying

your sshd config located in `/etc/ssh/sshd_config` from:

Port 22

to:

Port 31337

In order for this change to take effect, you'll need to restart SSH. This may differ depending on your distro, but many servers have an sshd script in `/etc/init.d`:

```
sudo /etc/init.d/sshd reload
```

Make sure that all SSH-enabled users in your system have strong passwords. A general rule of thumb for password strength is eight characters in length with a combination of letters, numbers, and special characters. If you are still concerned about attackers having the ability to brute force your logins (rapidly guess passwords), you can utilize SSH keys (<http://www.sshkeychain.org/mirrors/SSH-with-Keys-HOWTO/SSH-with-Keys-HOWTO-4.html>).

Mac OS X Leopard comes with an agent that will automatically cache your local SSH key password, making it much easier to interact with many servers via SSH keys.

This method along with completely disabling password authentication can take your level of security one level higher. You can take your SSH config a lot further by disabling password authentication entirely and only allowing certain users, but for our purposes, we will stop here.

General Rules of Thumb

Configuring a strong firewall and securing SSH offer the biggest bang for the buck when it comes to securing your server. There are hundreds of little things you can do to enhance security, but a lot of them are specific to your distribution of Linux. Read up on your distribution and see what it has to offer.

- If your distribution offers SELinux (http://en.wikipedia.org/wiki/Security-Enhanced_Linux), try to take advantage of it. It's a set of security policies developed by the National Security Agency and works together with many different Linux distros.
- Turn off any services that you aren't using. Not only will it help with security, it will also free up valuable system resources you can use for things that actually need them.

If you are still struggling with server-side security, try reading up on your distribution of choice. If at the end of the day you still aren't satisfied, hire someone to help. Your time as a developer should be spent on the things you are good at. If security is a major concern, leave it to the pros to help you out and give you the right tools to succeed!

What's the Risk?

CHAPTER 6

There is a fine art to information security that rears its head in the form of risk analysis. I guarantee at some point you will audit an application and find something in the form of a security hole. Your job as an auditor is to present the vulnerability, period. You might decide to offer suggestions for fixing that vulnerability, or you and your team might be the ones who end up fixing the problems later. The step between the auditor handing off a report and the developers hunkering down to fix the problems should include some kind of risk analysis.

Why do I care about Risk Analysis?

It all comes down to money. If you have a vulnerability that is in an extremely isolated place in your application, you may not need to fix that issue immediately (especially if that issue involved rewriting a component or set of components to solve the problem). Suppose you discovered an issue that would only happen if the user was logged in as an administrator at the end of February on a leap year. Let's do a high level risk analysis of that issue.

| Issue | Description |
|-------------------|---|
| Threat Capability | The attacker's ability to successfully compromise a system. |
| Control Strength | The systems you have in place to protect against attacks. |

| Issue | Description |
|-------------------------|---|
| Threat Event Frequency | The number of times annually that a threat comes in contact with your system. |
| Vulnerability | The chance that a contact is able to successfully take action on a system. |
| Loss Event Frequency | The annual expectation of the number of successful attacks on a system. |
| Probable Loss Magnitude | The monetary value that a single loss event represents. |
| Risk | The number of loss events in proportion to the probable loss magnitude. |

FAIR

The FAIR (Factor Analysis of Information Risk) framework was developed by Jack Jones. He has written a great whitepaper on the subject that can be freely downloaded here (http://risk-managementinsight.com/media/docs/FAIR_introduction.pdf). The risk analysis shown here is a very high level interpretation of a risk study using FAIR.

THREAT CAPABILITY

The attacker has to already have administrative privileges, so pending any other vulnerabilities they would have to find a way to get administrative access via social engineering or man in the middle (MITM) attacks. This tells us that the attacker would have to be very capable and experienced.

| Rating | Description |
|-----------|---|
| Very High | The attacker is either very skilled or possesses an unknown exploit. |
| High | The attacker is proficient with scripting utilities and can create their own exploits. |
| Moderate | The attacker has knowledge of script utilities and some information on how to use them. |
| Low | The attacker has just started hacking and really doesn't possess much knowledge. |
| Very Low | The attacker just learned that hacking was possible and decided to give it a shot. |

Conclusion: Threat Capability = Very High

CONTROL STRENGTH

If your application has authentication and authorization (users, rights, and roles) you have a start. If you have a role of administrator that has top level access and restricts all other users from performing administrative functions then you have a good foundation. In this case let's assume there are no vulnerabilities associated with these controls.

Conclusion: Control Strength = High

THREAT EVENT FREQUENCY

It is possible that this attack can occur every leap year on February 29th. For argument sake we will call this every four years. Given our per annum is yearly that makes the threat event frequency once every four years.

| Rating | Description |
|----------------|---|
| Very High (VH) | More than 100 times per year |
| High (H) | Between 10 and 100 times per year |
| Moderate (M) | Between 1 and 100 times per year |
| Low (L) | Between .1 and 1 times per year |
| Very Low (VL) | Less than .1 times per year (once every 10 years) |

Conclusion: Threat Event Frequency = Low

VULNERABILITY

In order to determine our vulnerability we simply need to cross reference our Control Strength and Threat Capability. We have a Control Strength of High and a Threat Capability of Very High, which gives us a Vulnerability of High.

Here is a reference pulled from the FAIR Wiki (<http://fairwiki.riskmanagementinsight.com>)

Vulnerability

| | | | | | | |
|-------------------|------------------|----|----|----|----|----|
| Threat Capability | VH | VH | VH | VH | H | M |
| | H | VH | VH | H | M | L |
| | M | VH | H | M | L | VL |
| | L | H | M | L | VL | VL |
| | VL | M | L | VL | VL | VL |
| | VL | L | M | H | VH | |
| | Control Strength | | | | | |

Conclusion: Vulnerability = High

LOSS EVENT FREQUENCY

This vulnerability only has the chance of arising every four years (yes there are a couple of edge cases in leap years but let's ignore them for simplicity purposes). Let's say our per annum in this case is yearly. This would make the possibility of a threat occurring 0.25%.

Now let's move that into the likelihood that that threat actually could occur. Given a possibility of 0.25%, a high control strength and the need for a very highly skilled attacker, the probability of a successful attack is almost negligible.

A Vulnerability of High and a Threat Event Frequency of Low gives us a Loss Event Frequency of Low.

Loss Event Frequency

| | | | | | | |
|------------------------|----|---------------|----|----|----|----|
| Threat Event Frequency | VH | M | H | VH | VH | VH |
| | H | L | M | H | H | H |
| | M | VL | L | M | M | M |
| | L | VL | VL | L | L | L |
| | VL | VL | VL | VL | VL | VL |
| | | VL | L | M | H | VH |
| | | Vulnerability | | | | |

Conclusion: Loss Event Frequency = Low

PROBABLE LOSS MAGNITUDE

This is where we define how much we could actually lose monetarily if a loss event were to occur. This is categorized into sections that we allocate dollar amounts to. Unless you have access to company financials, you will have to defer this section to a company CFO. I would actually recommend doing this even if you *do* have access to financial information because CFOs or their equivalent are much more likely to come up with numbers that more accurately represent reality. These categories are:

- Productivity
- Response
- Replacement
- Fines and Judgements
- Competitive Advantage
- Reputation

Once you assign dollar amounts to these categories, you can come up with an actual analysis of the risk involved with your vulnerability. Let's say the total in our case amounts to US\$8,000.

| Magnitude | Range Low End | Range High End |
|------------------|---------------|----------------|
| Severe (SV) | \$10,000,000 | - |
| High (H) | \$1,000,000 | \$9,999,999 |
| Significant (Sg) | \$100,000 | \$999,999 |
| Moderate (M) | \$10,000 | \$99,999 |
| Low (L) | \$1,000 | \$9,999 |
| Very Low (VL) | \$0 | \$999 |

Conclusion: Probable Loss Magnitude = Low

RISK

Now that we have all of our answers, the culmination of risk is as simple as our Loss Event Frequency in correspondence with our Probable Loss Magnitude. What is our company's tolerance level for realistically losing \$60,000 once every (let's say) 12 years? What's the likelihood that your application will still be in production in 12 years in the form that it is now? How much would it cost to fix this vulnerability up front?

We have a Loss Event Frequency of Low and a Probable Loss Magnitude of Low, resulting in an overall Low risk.

Risk

| | | | | | | |
|-------------------------|-------------|----------------------|---|---|---|----|
| Probable Loss Magnitude | Severe | H | H | C | C | C |
| | High | M | H | H | C | C |
| | Significant | M | M | H | H | C |
| | Moderate | L | M | M | H | H |
| | Low | L | L | M | M | M |
| | Very Low | L | L | M | M | M |
| | | VL | L | M | H | VH |
| | | Loss Event Frequency | | | | |

Conclusion: Risk = Low

Conclusions

In our test scenario, we proved that fixing our vulnerability was just not worth it. Some will be much more significant and require immediate attention, but I wanted to demonstrate that not all vulnerabilities merit attention.

If you constantly strive for perfection and must have it right, then risk analysis won't even be on your radar. In the real world however, costs sometimes outweigh the need to have perfect security in all areas.

Final Thoughts

CHAPTER 7

Throughout the auditing process, you will probably go down several rabbit holes that go down more rabbit holes. Normal development practices tell you that this is a bad idea (commonly referred to as *yak shaving*), but when you are auditing, you should just let it happen. You will probably do a lot of refactoring in the process. These things strengthen you as a developer.

The critical element here is to keep notes. You need to meticulously detail everything you are doing and hope to do later. If you are using a Mac, Taskpaper (<http://hogbaysoftware.com/products/taskpaper>) is a wonderful application for keeping track of what you have completed and what still needs to be done. If you are using `emacs`, create a notes buffer and use it to keep track of things. The key is to document everything! I promise you will need it later.

Create Guidelines, not Rules!

Applying rules to security auditing will drive you insane. It's great to start with a nice set of rules, especially if this is your first audit, but in the end they just get in the way. Instead, replace these rules with guidelines and use good judgement when dealing with things that surface as problems. If in doubt, ask someone. If your problem has NDA-related material, then make up a pseudo problem similar to your own and ask people on the Internet.

Take your now secure application and deploy it with more confidence than you ever had before! When your pointy haired boss (http://en.wikipedia.org/wiki/Pointy_Haired_Boss) brings up the security question, you can respond in confidence and tell him or her it's all good in the hood.