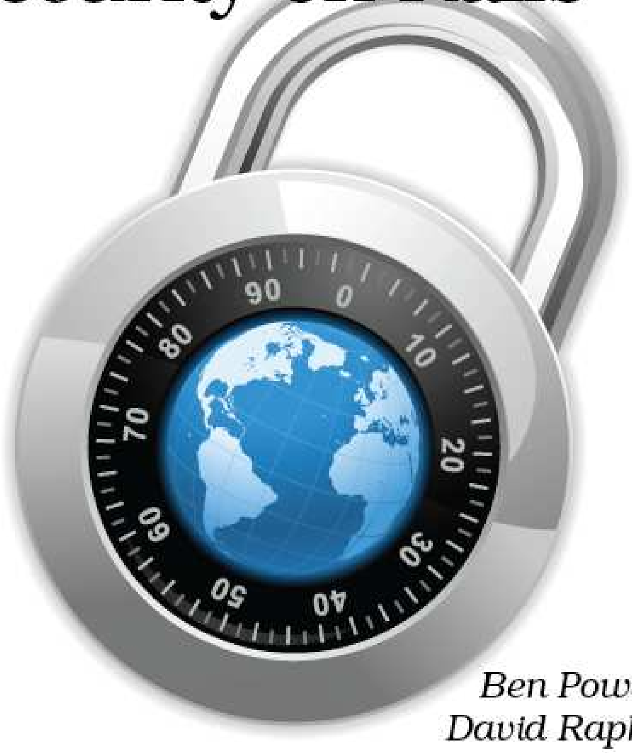
Extracted from:

Security on Rails

The Pragmatic Programmers

Security on Rails



Ben Poweski
David Raphael

Edited by Colleen Toporek

ISBN-10: 1-934356-48-4

ISBN-13: 978-1-934356-48-7

Printed on acid-free paper.

P1.0 printing, December 2009

Version: 2010-1-16

Chapter 1

Security in Ruby on Rails

Security is a major concern in all software development, and the creators of Ruby on Rails have built many security features into the framework. In order to build secure applications, we must use the built-in security features correctly. Additionally, most problem domains have their own set of security needs beyond the scope of any framework.

Large companies often utilize a software-development process that overlays security on each phase of a project. For example, at McAfee, we review code for security vulnerabilities before releasing it to the world. Many development organizations require that architecture be reviewed by someone security-savvy. This process works well for projects that have long life cycles and massive budgets.

But let's face it, heavyweight processes don't fit for the majority of Ruby on Rails developers. One of the strengths of Ruby on Rails is its agility—security should be applied with the same spirit.

1.1 Who's This Book For?

This book is for any Rails developer looking to:

- Improve general security knowledge
- Write more secure software
- Defend applications from common threats
- Encrypt data
- Control access to information
- Authenticate users from within applications
- Integrate with external user management systems
- And more...

We do not cover many of the community plug-ins—not out of dislike, but because building components from scratch is the best way to learn about writing secure code. Please make copious use of the community code available: the more eyes that have looked at a particular code base, the more secure that code base is likely to be. You can apply the principles covered in this book to help you choose which community code base to use for your projects.

1.2 What Does This Book Cover?

When we decided to write a book about Ruby on Rails security, we began by outlining an exhaustive list of security topics. We realized quickly that there simply wasn't enough time to give every topic the appropriate amount of focus, so we had to pick and choose to narrow things down.

Here's what the chapters cover:

- Chapter 2, *Hacking the Example*, on page 25, takes you through the process of hacking an application. We walk you through each exploit and, by the end of the chapter, you'll learn many common techniques used by hackers.
- Chapter 3, *Fixing the Example*, on page 51, takes you through fixing the application. You will learn how to fix the vulnerabilities discovered in the previous chapter.
- Chapter 4, *Testing for Security*, on page 70, takes you through testing the application. You will learn how to test the application from the standpoint of security.
- Chapter 5, *Validation*, on page 86, covers techniques for accepting, rejecting, or sanitizing system input and output. Rails provides excellent facilities for validation, and this chapter provides some background and general techniques for ensuring that we leverage Rails' facilities to the fullest extent possible.
- In Chapter 6, *Authentication: Decentralized Authentication*, beginning on page 97, we cover authentication techniques that are tightly coupled with your application. This chapter is ideal for anyone who wants to control user accounts directly in the application database or in an application-controlled directory server.

- Chapter 7, *Authorization*, on page 137, covers the process of controlling access to application resources. In this chapter, we will modify the way our controllers determine if a user is allowed to invoke a particular action based on the user's role.
- Chapter 8, *Data Protection Using Cryptography*, on page 157, covers the basics of how to use cryptography in applications. This is an important issue if you need to store sensitive information like Social Security numbers or credit card data.
- Chapter 9, *Digital Signatures and Email*, on page 174, builds on the cryptography concepts covered in the preceding chapter. You will learn to implement code that creates digitally signed emails.
- Chapter 10, *SSO: Centralized Authentication*, on page 193, covers authenticating users against multiple external systems. We demonstrate CAS, OpenID, and Kerberos (Active Directory).

Feel free to skip ahead to any chapter. We will spend the rest of this chapter focusing on security concepts and principles.

1.3 Entering a Security Mind-Set

How did things like door locks, car alarms, PINs, and passwords come to be designed in the first place? How many are a result of human nature versus the nature of technology?

At my son's elementary school, there is a sign that says, "You can do anything you want—as long as it doesn't bother anyone else." This sign is intended to help the kids develop into good, law-abiding adults. Unfortunately, many people don't live by this sign's rule. On the Web, theft, harassment, and other malicious activities occur every day. Taking these risks into account is a necessary first step toward entering a security mind-set when developing code.

Our applications are like little islands in an ocean of merchants, pirates, and passers-by. We have to protect our population from the dangers that lurk around it. And while there are endless analogies comparing software security and real-world security, in this book we'll associate these concepts with established security principles.

1.4 Defense in Depth

The concept of *Defense in Depth* originates with military teachings on fortress construction. Multiple layers of defense can stop or slow down attackers, and each layer typically deals with a different type of attack.

Our homes have doors with locks. To enhance the security of our homes further, we can add alarm systems to our homes. In many communities, gates and security guards at the entrance further limit unauthorized access.

The way that we deploy web applications implements a form of Defense in Depth:

1. Network: *firewalls, network intrusion detection systems*

   Detecting attacks at a network level enables us to add an additional defense against an attack that might not be detected at any of the other layers.

2. Operating system: *host-level intrusion detection, stack protection*

   By detecting malware, stack overflows, and other OS-level behaviors, we decrease the likelihood that an attacker can gain access to our systems.

3. Web server: *HTTP protocol enforcement*

   If our web server enforces strict adherence to HTTP specifications, then our application is less likely to break unexpectedly because of a protocol error.

4. Web application: *application-specific, user login*

   Our application can control access prior to any database or web service connectivity.

5. Database: *host-specific access control, DB user access control*

   The database can ensure that only authorized systems can connect through host names.

Defense in Depth applies to the architecture of our web applications as well.

Let's explore a real-world example of Defense in Depth. A banking customer is browsing her bank account balances and activities. In the banking application's menu, there is an option for external account

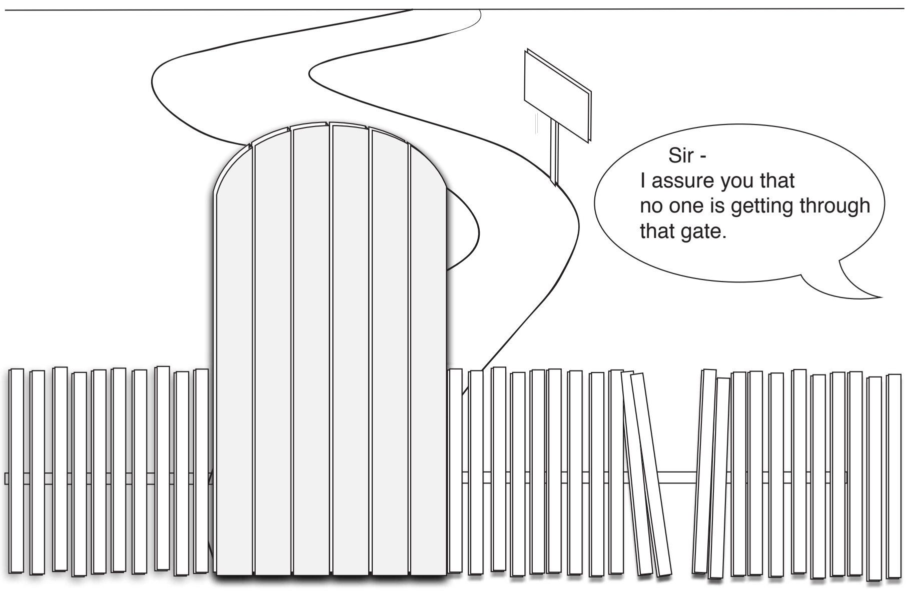


Figure 1.1: Look familiar?

transfers. This allows the customer to transfer money from her account to an external account based on an account and routing number.

What if our user gets up from her computer and forgets to logoff? Could someone else walk up and make a nice little deposit into an account of his choice? How would the banking customer repudiate the transfer?

What if we also require the user to answer some secret questions before transferring the funds? Or maybe we can mail a separate PIN in the mail for transfers.

1.5 Only Secure as the Weakest Link

The weakest link of an application within the context of its security also happens to be the highest level of security we can hope to achieve for that same application. What this means is that we can assume an attacker is going to go after the easiest target. Figure 1.1 is a tongue-in-cheek illustration of the way many security systems work. We can see that the gate is strong and sturdy, but the fence is not.

Often, developers will go to great lengths to enforce password complexity, but the application may still have a SQL Injection vulnerability. We explore this in Chapter 2, *Hacking the Example*, on page 25. A hacker can bypass the password altogether.

Remember we're not in control of every component in our applications. There are databases, third-party libraries, legacy applications, and other participants in our application stack. Knowing the weaknesses of each component is crucial. Perhaps there's a known exploit against a particular database or parsing library. We must proceed with caution when incorporating external systems and create well-defined boundaries that allow applications to isolate dangerous transitions.

1.6 Fail Close

Fail close is the opposite of fail open. For example, if a building catches on fire, all electronically locked doors will open to allow the fire department in and the people who were inside to get out. This is probably the desired functionality of a security system for a building with people inside it. However, in most software applications, we want our systems to fail in such a way that they are secure post-failure.

Consider the following code:

```ruby
def show_invoice
  @invoice = Invoice.find(params['id'])
  unless @user.validate_secret_code(@invoice.secret_code)
    redirect_to :action => 'not_authorized'
  end
end
```

This code checks to see if the invoice record contains the secret code that exists in the user object. If it doesn't exist, the code redirects the user to another page. Otherwise, the code returns the invoice. While this code might or might not work, it shows a flaw (not a bug) in the design. The code should redirect to the authorized page on a positive condition and continue with a fail condition otherwise:

```ruby
def show_invoice
  @invoice = Invoice.find(params['id'])
  if @user.validate_secret_code(@invoice.secret_code)
    redirect_to :action => 'authorized'
  else
    # this is where we tell the user that they are
    # not authorized to view the invoice.
    redirect_to :action => 'not_authorized'
  end
end
```

If the condition in the first code example fails for some reason, the application will continue as if the user is authorized. In the second example, the opposite is true. The application will continue on the unauthorized path.

1.7 Whitelisting

Consider the following word list:

```
bad_words = %w{< > ' " script}
```

This is a very limited list of words and characters that we want to restrict our users from inputting. The problem is that we are guessing what characters or words are potentially malicious. What happens if our user Base64-encodes the input or HTML-encodes the input? There are many exploits possible in these cases. Let's look at another example:

```
restricted_users %w{alice bob kevin}
```

We've created a list of users who have abused their privileges. While this list might be useful from a reporting standpoint, we shouldn't use this list for a security control.

Fundamentally, any list that tries to assert what is *not* allowed is known as a *blacklist*. Blacklists should be avoided whenever possible in designing security controls because they tend to be flexible in nature. Flexibility and security don't mix well.

At this point, it is probably a good idea to break up lists in the context of security controls into two categories:

- Access control
- Filters

Access Control Lists

Consider the following list:

```
admins = %w{dave ben}
```

The admins list includes Dave and Ben. We now have a situation where our control can implement the following logic:

```
if admins.include? user
  redirect_to :action => "admin_console"
else
  redirect_to :action => "user_console"
end
```

This is a *whitelist*, a list of acceptable items. Whitelists work quite well with our principle of failing closed. Anytime we are making a security decision based on some criteria, that criteria should come in the form of a whitelist. Additionally, we must write our code to fail close. Here is an example of what *not* to do:

```
# Don't do this.
if !admins.include? user
  redirect_to :action => "user_console"
else
  redirect_to :action => "admin_console"
end
```

In the previous example, we've succeeded in creating a whitelist, but we've managed to violate our principle of failing closed.

Input Filters

Consider the following filter:

```
# First and Last Name Tokens
acceptable_name = /[a-zA-Z]+(([']{1}[a-zA-Z]+)|([\-]{1}[a-zA-Z]+))*/
```

We've constructed a whitelist of acceptable characters for a person's name. It whitelists any normal name like David or O'Reilly, or even O'Reilly-Smith. It is very difficult to write a blacklist that effectively detects an invalid name. Here's our attempt at a list:

```
unacceptable_name = /''|--|[\-]{2}-+|[\d];/
```

This is hardly a complete blacklist. And that is really the point of this example. It is nontrivial to create a complete blacklist, and really unnecessary for most situations. You can probably already see a number of characters that we should include in this blacklist. But it's not an effective technique if you want to cover your bases completely.

1.8 Least Privilege

Least privilege is the notion that users or subsystems should only be able to execute code that they need in order to function per business requirements, and nothing else. We can think of least privilege as a whitelist of necessary functionality.

Here's a dialogue that illustrates a typical scenario in the corporate world:

Business Analyst: *Why can't I add a new user to the document management system?*

Administrator: *Because you are not authorized to make those kind of changes.*

Business Analyst: *Why not?*

Administrator: *Because you are not a member of the administrator group.*

Business Analyst: *OK. Make me a member of the administrator group.*

Administrator: *No.*

Business Analyst: *Why not?*

Administrator: *Because business analysts are not allowed to be administrators. You would grant permissions to the wrong people. Feel free to write a business justification document explaining why you need to add users to the system. Perhaps we could add a group called "business analyst admins." This group would be allowed to add only other business analysts. You have only read access to the system, for the purpose of reading documents for business analysts; this is consistent with our policy of enforcing least privilege.*

Business Analyst: *Gee. Thanks.*

As we engineer more complex types of software, we must understand our usage scenarios to implement security controls across *security boundaries*. Security boundaries are areas of our software that execute some code that might require different levels of access. We cover specifics of this in Chapter 7.

Regardless of the mechanism that supports our transition across a security boundary, we should implement our security controls to enforce or allow the expression of least privilege.

An example of the misuse of this principle is when developers deploy database-connected applications. Often, if an attacker is able to break the application-layer defenses, that attacker will then have administrator access to a database system. This is not a good idea. The software should be accessing the database with the least amount of privilege to perform the mandated functionality. This can also be looked at within the context of Defense in Depth. At every level, part of the defense should be least privilege.

1.9 Do Not Repeat Yourself

DRY, or Do Not Repeat Yourself, is a principle first coined by *The Pragmatic Programmer* [HT00]. If we focus on not cutting and pasting, our code will be more secure—we won't repeat security mechanisms in different ways if we share common code across our security controls. This allows us to also correct vulnerabilities as they are discovered, all in one place as opposed to many.

1.10 Avoid Complexity

Complexity is bad. Complexity is an enemy to development cost, maintainability, and security. As software complexity increases, so does the risk of vulnerability. Throughout the brief history of modern, distributed application development, developers, business drivers, and excessive feature lists in general have caused application code bases to become unmanageable. There is no question that complexity has created numerous security flaws. It begins during the design phase, when long lists of desired features pile up like dirty laundry.

It is almost impossible to describe a secure system by specification. We can have software requirements descriptions (SRDs) that enumerate "secure requirements." They might look something like this:

- All cryptographic keys must be stored in separate files outside of the application directory.
- Key strength for symmetric cryptographic routines must be a minimum of 256 bits.
- Key strength for asymmetric cryptographic routines must be a minimum of 2,048 bits.
- A system should filter SQL keywords and SQL escape sequences.
- A system should filter all JavaScript.

This is a very basic list. In reality, a security consulting firm will provide you with a book of security requirements to code against for the sum of several thousand dollars. And while these can be useful, they also tend to be fairly context-insensitive.

Ultimately, security requirements are much like usability requirements. X UI component should be *usable*. And likewise, X component should be *secure*. It's a bit vague, isn't it?

If we apply some of the previously described principles to our development, we can reduce the amount of complexity involved in making more secure applications. Here's a summary of the principles that drive the techniques we will explore in this book:

- Defense in Depth pushes us to look at a layered-application model. Layers break up our security boundaries naturally in our software. This leads to simpler software that delegates duties to each appropriate layer. You need to focus on creating silos of defense, which makes your components both more reusable *and* more secure!
- Whitelisting makes life easier. If we were to try to figure out all the things users shouldn't be doing before we ship our code, well, we'd never ship any code. However, if we focus on what our users can do, we have a finite list to concentrate on, which requires much less of our time. Additionally, we can be explicit about what our users can do, which fits nicely into the next principle.
- Least privilege pushes our security logic further down the application stack. This inspires us to write security-aware components that play nicely in our system's ecosystem. Instead of creating infinitely configurable security controls, our components can push for convention-driven security rather than configuration-driven security. Business analysts prefer to have infinitely configurable security subsystems. Don't listen to them. Have you ever encountered an administration console allowing you to configure every conceivable permission in the world? That is complexity at its worst.
- Secure by default allows more convention-driven security in our application. Building software in such a way that it is secure out-of-the-box drastically reduces the dependency on users and administrators to harden the security of a particular application.
- Writing code that fails close, incidentally, is easier for most developers to read. Easier-to-read code is easier to maintain, and maintenance is an important aspect of application complexity.
- DRYing your code up is one of Rails' guiding principles. It keeps your code base much more manageable, which in turn leads to easier maintenance. We all make mistakes, but we can make sure those mistakes are easy to fix.