

The
Pragmatic
Programmers

Security on Rails



*Ben Poweski
David Raphael*

Edited by Colleen Toporek

What Readers Are Saying About *Security on Rails*

A thorough and comprehensive guide to the most common security issues faced by Rails developers, this book provides real code examples of security concerns and exploits and covers how to mitigate them in a Rails application and make it more secure.

► **Tore Darell**

Ruby on Rails developer

Security on Rails is not a mere tutorial on how to beef up your application, but a rich opportunity to learn what vulnerabilities exist and how they're exploited. The reader is taught not only the solutions, but how they were derived. This is a must for anyone who hosts a Rails application.

► **Kevin W. Gisi**

Ruby on Rails developer

Do not deploy your next Rails application until you've read this book. This book details both common and uncommon security issues that you need to be aware of, whether you're working for yourself or for a large enterprise. It's a "must have" for the beginning or intermediate Rails developer.

► **Brian Hogan**

Rails consultant and trainer

Security on Rails is a good book to take security in Rails applications a step further. It includes detailed guides for how to implement advanced security technologies in Rails. Once again, it illustrates how secure Rails apps can be.

► **Heiko Webers**

CEO of bauland42, Ruby on Rails Security Project

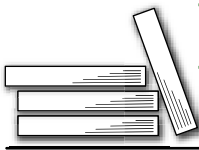
This book is the life preserver that Ruby on Rails developers need to keep them from drowning in a sea of security problems.

► **John Viega**

CTO, Software-as-a-Service, McAfee, Inc.

Security on Rails

Ben Poweski
David Raphael



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragprog.com>

Copyright © 2009 The Pragmatic Programmers LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-48-4

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

Printed on acid-free paper.

P1.0 printing, December 2009

Version: 2009-12-14

Contents

1	Security in Ruby on Rails	10
1.1	Who's This Book For?	10
1.2	What Does This Book Cover?	11
1.3	Entering a Security Mind-Set	12
1.4	Defense in Depth	13
1.5	Only Secure as the Weakest Link	14
1.6	Fail Close	15
1.7	Whitelisting	16
1.8	Least Privilege	17
1.9	Do Not Repeat Yourself	19
1.10	Avoid Complexity	19
1.11	Acknowledgments	21
I	Getting Started	22
2	Hacking the Example	23
2.1	Getting Started with LunchedIn	23
2.2	The Tools of the Trade	24
2.3	Exploit: Parameter Manipulation	26
2.4	Exploit: Broken Authorization	31
2.5	Exploit: SQL Injection	36
2.6	Exploit: Cross-Site Scripting	40
2.7	Exploit: Cross-Site Request Forgery	45

3	Fixing the Example	49
3.1	Protecting Against Parameter Manipulation	49
3.2	Adding Authorization	52
3.3	Preventing SQL Injection	57
3.4	Cross-Site Scripting Countermeasures	60
3.5	Protecting Against a Cross-Site Request Forgery	64
II	Building on the Basics	67
4	Testing for Security	68
4.1	Things You Should Test	68
4.2	Testing in Layers	70
4.3	Authentication Tests	70
4.4	Authorization Tests	73
4.5	Data Protection Tests	74
4.6	Input Validation and Sanitization Tests	78
4.7	Measuring Test Coverage	81
4.8	Other Testing Frameworks	82
5	Validation	84
5.1	Command Injection	85
5.2	SQL Injection	88
5.3	Mitigating SQL Injection	89
5.4	Built-in SQL Injection Mitigation in Rails	92
6	Authentication: Decentralized Authentication	95
6.1	Authentication Concepts	95
6.2	User Registration	99
6.3	LDAP/Directory-Based Authentication	107
6.4	Flexible Authentication	122
7	Authorization	135
7.1	What Is Authorization?	135
7.2	Mandatory Access Control	136
7.3	Discretionary Access Control	138
7.4	Role-Based Access Control	140
7.5	Implementing RBAC	142
7.6	A Simple Approach to Model Security	151

8	Data Protection Using Cryptography	155
8.1	Message Digests	156
8.2	Encrypting Data	158
8.3	Using Cryptography with ActiveRecord	165
9	Digital Signatures and Email	172
9.1	Digital Signatures	172
9.2	Storing Certificates	178
9.3	Secure Email Exchange	178
9.4	Sending Signed Messages	182
9.5	Verifying Received Messages	187
10	SSO: Centralized Authentication	191
10.1	Public Versus Private	191
10.2	OpenID	193
10.3	Implementing an OpenID Provider	204
10.4	Kerberos, GSSAPI, and SPNEGO	219
10.5	CAS	251
III	Reference	260
11	Web Application Proxies	261
11.1	Web Application Proxies	261
12	Authentication Appendix	263
12.1	OpenLDAP Setup	263
12.2	GSSAPI	267
12.3	SPNEGO	280
13	Bibliography	281
	Index	282

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 23, 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 49, 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 68, takes you through testing the application. You will learn how to test the application from the standpoint of security.
- Chapter 5, *Validation*, on page 84, 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 95, 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 135, 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 155, 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 172, 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 191, 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

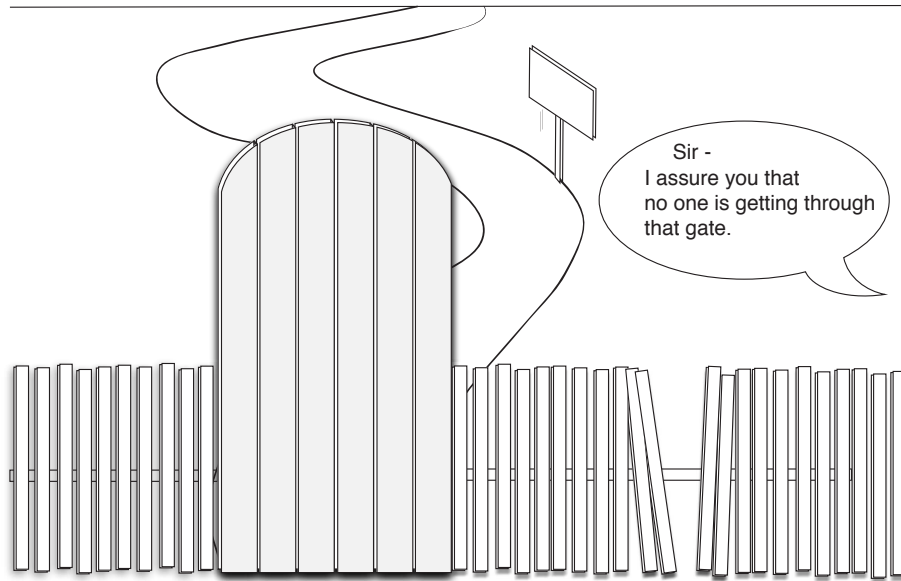


Figure 1.1: Look familiar?

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 23. 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:

```
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:

```
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* [HTOO]. 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.

1.11 Acknowledgments

The authors wish to thank the tech reviewers for their valuable insights and assistance, including Greg Vaughn, Jeffrey Turner, Heiko Webers, Frederick Cheung, Brian Hogan, Paul Butcher, Christophe Porteneuve, Kevin Gisi, and Tore Darell. Thank you also to Randall Munroe, for allowing us to reprint one of his exceptional comic strips (you can find more at <http://xkcd.com>).

Ben Poweski

I'd like to thank my mother, father, and sister for supporting me in this endeavor. Also, I'd like to thank David, for pushing me to become better at my craft, and Glenn Vanderburg, because without him this book would not be possible. And finally, I'd like to thank everyone that has put up with my incessant cancellation of social outings for sake of the "book."

Dave Raphael

Dave Thomas is a total joy to work with as an editor. This book would never have been realized if it weren't for his patience and guidance. Glenn Vanderburg was right when he said, "There are not too many guys like Dave Thomas... I don't know how he does all of the things he does and still has time to sleep." Thanks to Glenn for introducing us to Dave.

I'd like to thank my son, Pierce, for being so patient on the weekends when "daddy" needed to work on his book. This was also made possible with the support of my dear family and friends—all of you provided me inspiration when I was ready to give up. Thank you.

Part I

Getting Started

Hacking the Example

Before we can protect software, we need to understand how to break it. The purpose of this chapter is to learn techniques of *web-application hacking*. We will exploit a sample application called LunchedIn, which we will discuss in greater detail later in this chapter (and throughout the book). We will patch the major security holes in Chapter 3, *Fixing the Example*, on page 49. In this chapter, we cover the following attacks:

- Parameter manipulation
- Broken authorization
- SQL injection
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF; pronounced “sea-surf”)

Rails developers will likely encounter these vulnerabilities in their quest for web application greatness. Later in the book, we discuss more advanced attacks, and mitigation techniques to prevent them.

There is no better way to learn how to improve security than to experience firsthand how to break it. In other words, we hack an application to help us become better developers because we increase our understanding of what we need to do to protect applications.

2.1 Getting Started with LunchedIn

All the examples in this chapter use a sample application that contains vulnerabilities. This application, LunchedIn, is a collaborative tool for scheduling, rating, and managing lunch dates.

Danger, Will Robinson!

The code in this chapter is not meant to be used for anything other than learning how to attack software!

You can download the code for the LunchedIn application, as well as the rest of the code from this book, by visiting the book's home page at http://pragprog.com/titles/fr_secure and following the link to the code. Once you download and unpack the code archive, you'll be able to find the file corresponding to a particular code listing in the book by following the path shown in the gray lozenge at the top of the printed extract. If you're reading the PDF version of the book, try clicking the gray lozenge.

To access the code for the vulnerable LunchedIn application, you can navigate to this path in the book's code archive: `code/getting_started/lunchedin_broken`.

You can load the sample data used in exercises throughout this book with this command:

```
lunchedin_broken> rake db:create:all
lunchedin_broken> rake db:migrate
lunchedin_broken> rake db:fixtures:load
```

If you feel the need to start over with clean data, simply execute the commands again, and new data will appear.

2.2 The Tools of the Trade

So how do we break into a web application? Well, we probably have to go beyond the standard web interface provided by the application's developers. In fact, we want to peel back the layers and see what's happening at the protocol level. This will let us see just what the application is sending to the browser and what the browser is sending to the server. We'll also need to manipulate traffic, injecting information that the application developer wasn't expecting. If the developer wasn't expecting it, it's possible she didn't bother to deal with it, and that gives us (the bad guy) an edge. The tool of choice for doing this is an application proxy.

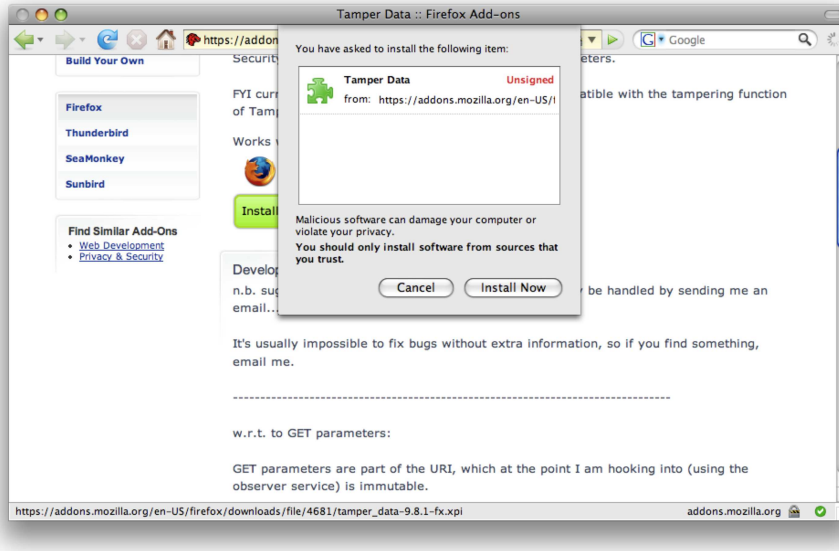


Figure 2.1: Installing Tamper Data

Application proxies are simply web servers that act as mediators between the browser and the destination server. They allow us to manipulate the *request* before the browser sends it to the server. We can also use them to manipulate the response before it is received by the browser.

You can find a list of HTTP proxies useful for security testing and analysis in Chapter 11, *Web Application Proxies*, on page 261. The proxy we will use for hacking is the Firefox extension, Tamper Data.

Using Firefox, navigate to Tamper Data’s home page.¹ Click the install link and agree to the installation dialog (see Figure 2.1).

We have our extension installed; now let’s use it to start hacking into the LunchedIn application.

1. <https://addons.mozilla.org/en-US/firefox/addon/966>

2.3 Exploit: Parameter Manipulation

Parameter manipulation is simply the act of modifying the information sent back and forth between the client (the web browser) and the server in a way the application never would. Anything sent by the client is fair game to be modified. Any form field, HTTP header, cookie, or query parameter can be changed and even removed from the outgoing request.

Enabling Administrator Access

Like many applications, LunchedIn provides an administrative console exposed to the Web. We will hack the administration console using an ordinary account.

Navigate your browser to <http://localhost:3000/users/new> of LunchedIn, and view the HTML source of the page. We should see something similar to the following:

Download `getting_started/create_user.html`

```
<form action="/users" method="post">
  <div style="margin:0;padding:0">
    <input name="authenticity_token"
      type="hidden" value="3b9f8f1e0915d96ba3d9e804bf46703d53261173"/>
  </div>
  <div class="highlighted-form">
    <table class="form">
      <tr>
        <td class="label">First Name:</td>
        <td>
          <input id="user_first_name" name="user[first_name]"
            size="16" type="text"/>
        </td>
      </tr>
      <tr>
        <td class="label">Last Name:</td>
        <td>
          <input id="user_last_name" name="user[last_name]"
            size="16" type="text"/>
        </td>
      </tr>
      ...
    </table>
  </div>
  <div style="padding: 10px;">
    <input name="commit" type="submit" value="Create"/>
  </div>
</form>
```

The Art of Pen-Testing

How do we know to do this? In short, we *don't*. For the sake of brevity, we will use the omniscient knowledge of the author to skip the part where the attacker modifies the request and painstakingly looks at the output. Security professionals refer to this as penetration, or *pen-testing*.^{*} Pen-testers are the professional hackers of the security world, and much of their work involves painstaking trial-and-error processes, such as the one described in Figure 2.2, on the following page.

*. You can find more information about pen-testing at <http://www.securityfocus.com/pen-test>.

A simple inspection of the HTML source using our browser reveals a typical Rails HTML form. Assuming this application follows Rails conventions, we can infer the following:

- The application invokes a POST request to <http://localhost:3000/users> after the form is submitted. This URI maps to a ActionController::Base subclass called UsersController and the create() method.²
- A User object is instantiated; then the save() or create() method is invoked.
- The values inside of the left and right brackets map to object attribute names and database columns, so we can infer that the User class and corresponding table have the attributes first_name, last_name, email, and probably several more.

Unless *countermeasures* are present to prevent the automatic binding of fields within the form, all attributes passed to the new() method will be set in the User object. This means if the developer skipped implementing protection mechanisms, such as use of the attr_protected() class method, we can insert our own form fields and the application will gladly set values if a column exists.

If a setter method does not exist on the object being instantiated, we get an undefined method error, and the request stops processing. This usually shows up as a nasty error in our web browser, but the Rails designers thought of this and changed the verbosity of error messages running

2. This pattern is inferred from the Rails REST architecture.

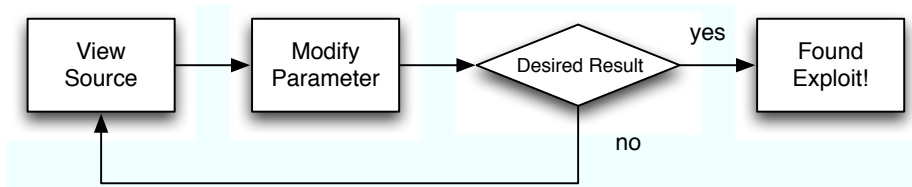


Figure 2.2: Testing process for parameter manipulation

in different configuration modes. This protection helps prevent leaks of internal information that might be used against our applications.

How can we enable administrator access by using parameter manipulation? The attack is actually quite simple, but it requires us to understand how the application determines a user is an administrator.

A common approach to designing user management is to create a User model, a corresponding set of Role models, and relationships via a `belongs_to()`, `has_many()`, or `has_and_belongs_to_many()` method. We will discuss more about authentication and authorization in later chapters, but let's expand our assumptions along this line of thinking.

We have a pretty good idea that the application contains a User model and a `UsersController` file, based on the information leaked from the URI (<http://localhost:3000/users/new>). So let's take a guess that LunchedIn's authorization model or method for determining access control is built on a *user belongs to a role* relationship. If this guess holds true, then the User model should contain the attribute `role_id` or something similar.

From a black hat perspective, setting this field involves a little guesswork. First, a hacker must work out what we called the "this user is an administrator" column, and then he must guess what value means "yes." We'll skip that guessing step (but, as a hint, common column names that control user access include `role_id`, `role`, `is_admin`, `group`, `permission_id`, `static_permission_id`, `admin`, `group_id`, and `grp_id`).

The field is called `role_id` in the LunchedIn application. To add administrator access to a new User object, we need to add the following form field to the request:

```
<input type="text" name="user[role_id]" value="1"/>
```

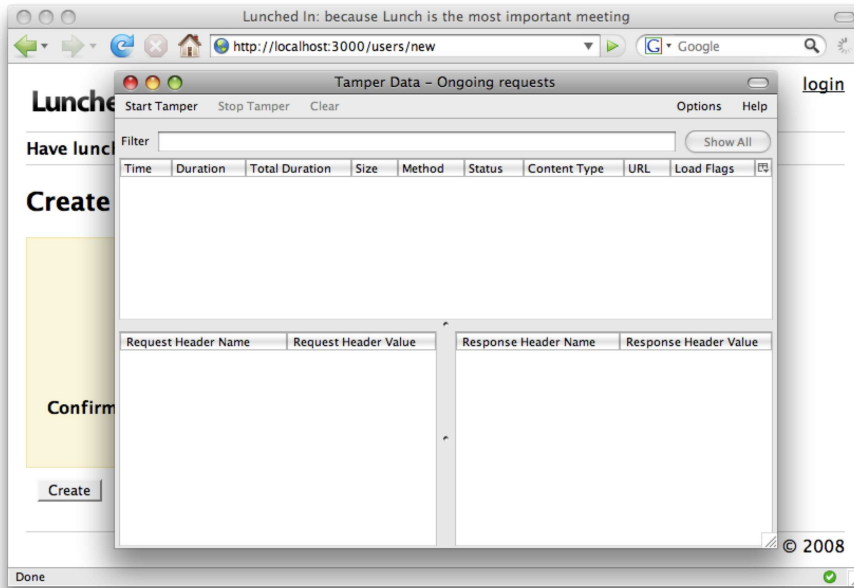


Figure 2.3: Starting Tamper Data

We will use the Firefox extension installed earlier to add the `role_id` parameter to the outgoing request. To start Tamper Data, select the Tools menu and then the Tamper Data menu item. This will bring up the Tamper Data dialog console, as shown in Figure 2.3.

Navigate to the <http://localhost:3000/profile/new> and enable Tamper Data. Next, enter data for a new user account. Now clicking Create causes Tamper Data to intercept the outgoing request and bring up a dialog box, as shown in Figure 2.4, on the following page. Click the Tamper button in the dialog box, as shown in Figure 2.6, on page 32.

At this point, we have successfully intercepted the outgoing create request to the LunchedIn application. We are able to modify this request in any way we want. To add administrator functionality to the User object, right-click or `[Control]`-click the column in the dialog box, then select Add Element.

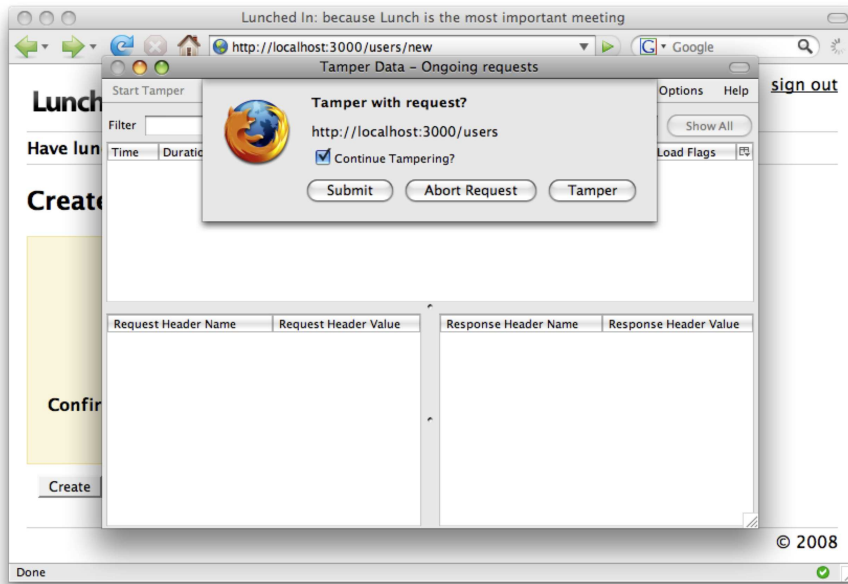


Figure 2.4: Creating a user

This brings up another dialog box that tells the user to enter post data; the dialog should look something like Figure 2.5, on the following page. Enter the following into the text field, and click .

```
user%5Brole_id%5D=1
```

We add this value as a new form element to accommodate the way Rails maps form parameters to newly instantiated objects. Specifically, HTTP parameters in the form of `foo[bar]` are mapped to hashes in the form `params[:foo][:bar]`. The values `%5B` and `%5D` are the URL-encoded values of `[` and `]`. The values are usually passed directly into the `create()` or `update_attributes()` methods of Active Records.

Now click on the “Outgoing requests” dialog; if all goes well, the User object will be added successfully to the application with administrator rights.

When browsing through the application with our new administrative privileges, the comments on a venue should look something like Figure 2.7, on page 33. We have changed the role of the user, so the

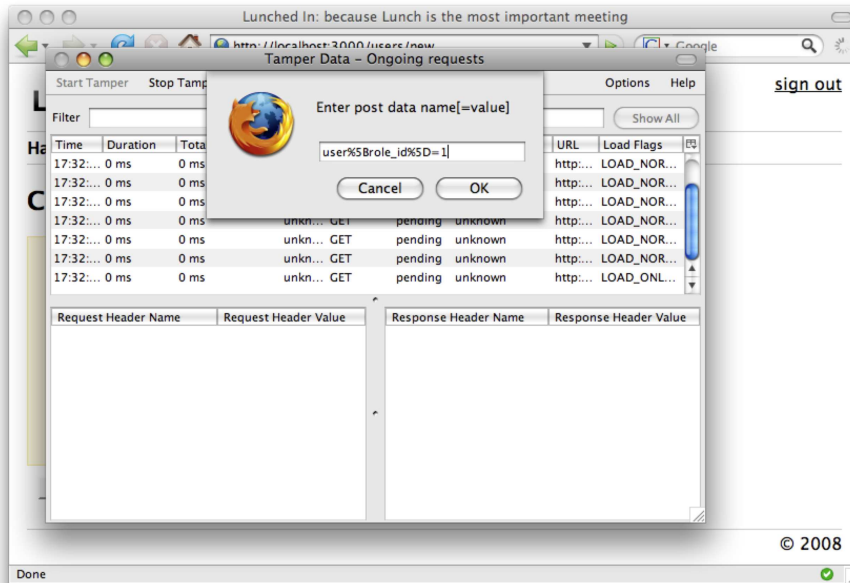


Figure 2.5: Adding a form element with Tamper Data

application now gives us the ability to mark comments as spam, thus allowing us to remove that nasty comment about our brother-in-law's restaurant—or eliminate all the good ones.

So far we've learned how to manipulate the request and response using the Tamper Data proxy. This technique is very useful for testing possible security vulnerabilities, as well as for learning about what is going on between the browser and the server.

2.4 Exploit: Broken Authorization

As application developers, we often limit access to various parts of our applications. To gain access to these areas, a user must be authorized or allowed. Sometimes the logic behind limiting access is flawed; we refer to this as an *authorization vulnerability*. To illustrate authorization vulnerabilities, we will exploit the voting mechanism in LunchedIn and show how to access the events of other users.

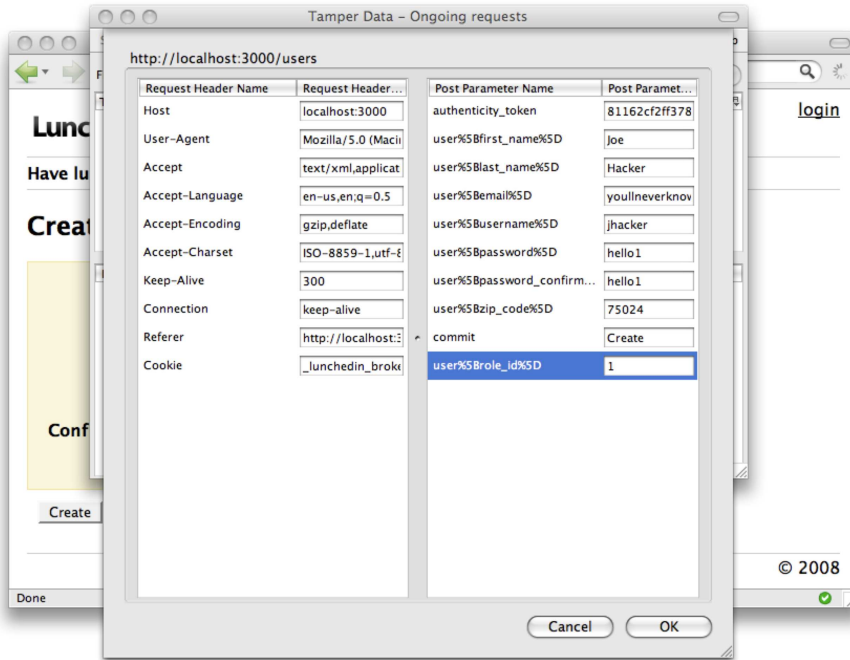


Figure 2.6: Enabling an administrator role

Failure to Authorize Ajax Actions

In Rails, Ajax actions are the natural way of expressing usable solutions to complex UI interactions. Unfortunately, when we implement such functionality, it is easy to lose sight of where our code is placing trust. As developers, we must think about traversing server to client and all of the changes in between. These interactions can be difficult to understand, which can make identifying vulnerabilities that much more difficult. In the next example, we will look at a flaw around a voting mechanism in our application.

Our application allows only a single vote—at least, that was the intention. When we look at the source of <http://localhost:3000/venues/show/1>, we can see there are two links that control voting via the `onclick()` JavaScript event.

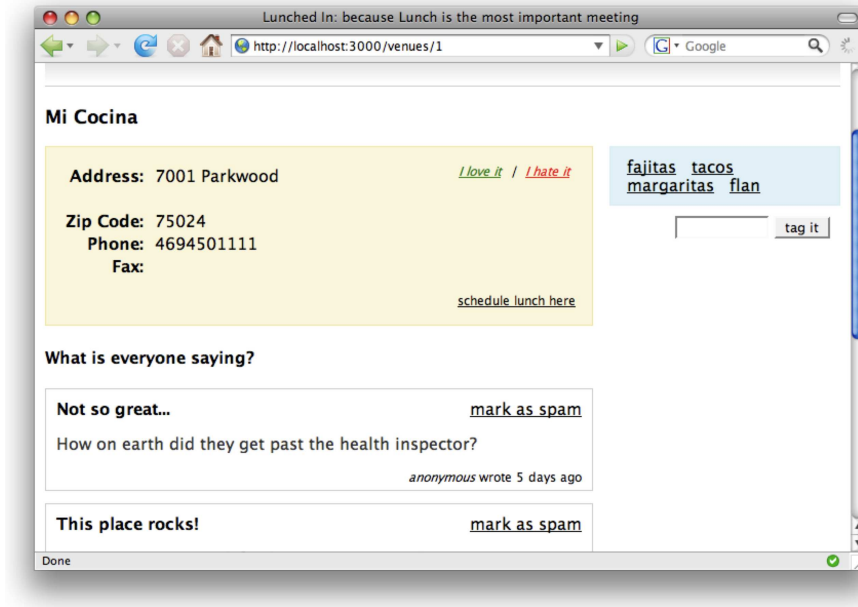


Figure 2.7: Administrator privileges

Download [getting_started/venue_vote.html](#)

```

...
<div id="venue-rating">
<a href="#" onclick="new Ajax.Updater('venue-rating',
  '/ratings/rate?score=1&venue_id=1',
  {asynchronous:true, evalScripts:true, parameters:'authenticity_token=' +
  encodeURIComponent('43986a596e750503311132c669057d3e089f')}); return false;"
  style="color:green">I love it</a>
|
<a href="#" onclick="new Ajax.Updater('venue-rating',
  '/ratings/rate?score=0&venue_id=1',
  {asynchronous:true, evalScripts:true, parameters:'authenticity_token=' +
  encodeURIComponent('43986a596e750503311132c669057d3e089f')}); return false;"
  style="color:red">I hate it</a>
</div>
...

```

Rails favors convention over configuration, so we have to keep in mind that attackers can draw assumptions about our application, just as we can. Note that these assumptions are not the cause of security vulnerabilities; quite the contrary, they help us produce correct code

by reducing the number of steps it takes to solve a particular problem. Adhering to conventions is an example of reducing complexity, the sworn enemy of security.

Now let's look at the markup generated by the application and see what we can learn. We can use our general knowledge of Rails to derive the following:

- There is a controller called `RatingsController` that includes a `create()` method.
- The `rate()` method accepts a parameter, `score`. We assume this because the rendered markup contains references to `http://localhost:3000/ratings/rate/1?score=1`.
- The `rate()` method returns a JavaScript fragment. The integration of the Rails and Ajax toolkits thrive on lightweight marshaling between the client and server. A common approach is to send JavaScript to the client to execute after a remote action is invoked.

We are able to draw these assumptions without ever looking the source code on the server, deriving them *entirely* from the visible HTML our browser uses to render the view. Of course, this is unavoidable because the browser *must* access the markup.

Hacking the Voting Links

The ability to vote is removed in LunchedIn after a user votes on a particular venue—or so the designers thought. Let's hack this functionality using Tamper Data.

When a user clicks the voting link, the browser submits an Ajax request using `Ajax.Request` from the Prototype JavaScript library. After the vote is submitted, the server responds with a bundle of JavaScript, telling the browser what parts of the page need to be updated. We are going to intercept this request and replay it.

We need to log into LunchedIn, navigate to `http://localhost:3000/venues/1`, and enable Tamper Data. We do this by selecting Tools > Tamper Data > Start Tamper. Choose and click a voting link; then click the Submit button when you get to Figure 2.8, on the following page.

At this point, we have the outgoing request logged in our proxy. The first vote has been processed, and we now can replay it by selecting "Replay in browser," as shown in Figure 2.9, on page 36.

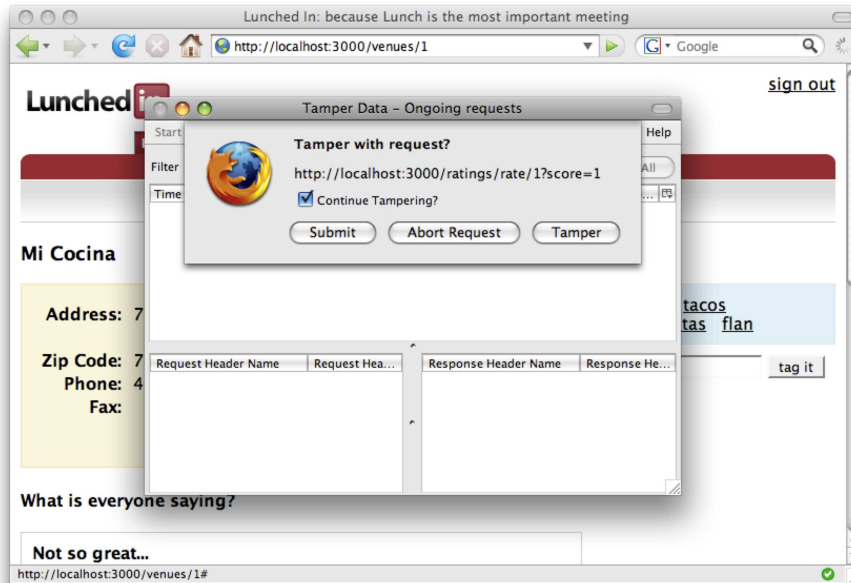


Figure 2.8: Intercepting the Ajax vote request

Selecting “Replay in browser” brings up the outgoing request dialog, as shown in Figure 2.10, on page 37.

If all goes well, we will be able to resubmit the same outgoing request using our web proxy. This demonstrates that, while the view was removed after the Ajax request, we are still able to send it again. This means the server isn’t properly limiting what we see versus what the server will process.

Unprotected ID Access

One exploit that is easy to overlook concerns allowing direct access to objects not belonging to a user. It is all too easy to forget to add authorization checks, not least because we have to look for such behavior. Let’s test LunchedIn for such a flaw:

1. We need to log in to LunchedIn using the bob account and the hello password.

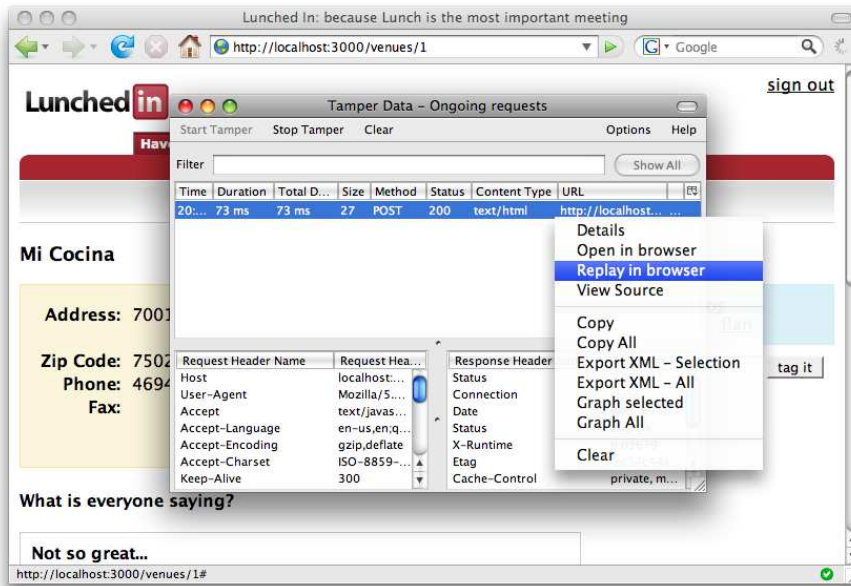


Figure 2.9: Replaying a request in Tamper Data

- Next, we navigate to the URI, `http://localhost:3000/events/1`. This brings up an event by Wally Webcoder.

If all is well, you should be able to access the event created by Wally, as shown in Figure 2.11, on page 38. You can infer from this that LunchedIn probably suffers from an authorization flaw allowing direct access to its resources.

So far we've learned that broken authorization allows malicious users to execute functionality the designer never planned for. Using a proxy like Tamper Data, we can insert arbitrary markup and exploit these flaws, as we did when we created the "improved" voting button.

2.5 Exploit: SQL Injection

SQL injection vulnerabilities can have potentially disastrous consequences. The concept behind the attack is quite simple: inject SQL into the input points the application uses to build a query. This input will be sent to the database and interpreted. Our example will use MySQL, but

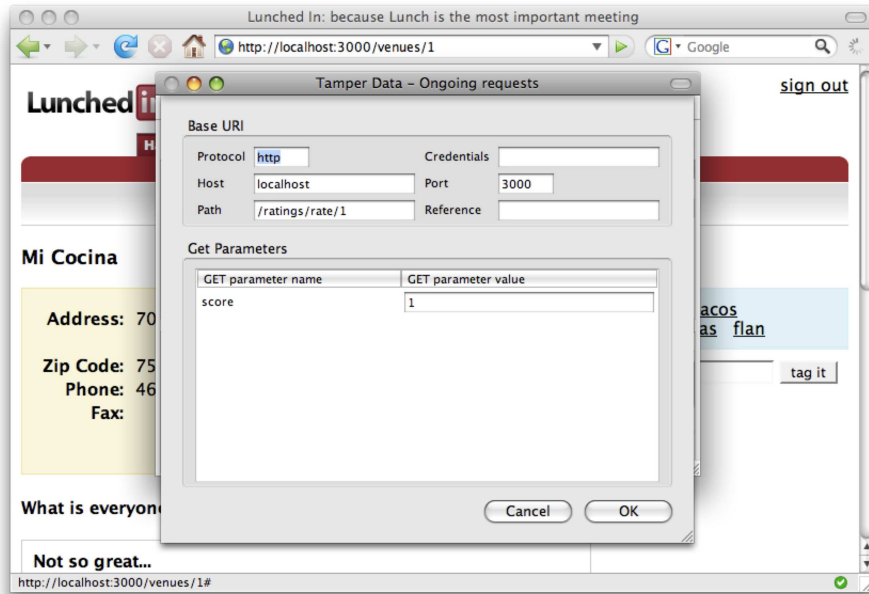


Figure 2.10: An outgoing request in Tamper Data

the code we create should be portable across a variety of databases. Some SQL injection attacks are not so portable because they use features specific to the RDBMS. For example, SQL Server uses the `xp_cmdshell` stored procedure to execute shell commands; some hackers target this specific implementation.

Testing the Water

The first task is to poke at the application and see if it responds to input that would signal a vulnerability. One of the easiest ways to do this is to send single quotation marks in the input field of an application. If an unhandled error occurs, this is a good indication that the application might be vulnerable to SQL injection exploits.

In LunchedIn's login screen, enter the following into the username field and submit the form:

```
hello'
```

When you insert a single quote in the username field, there is a surprising effect: an unhandled `ActiveRecordError` occurs. If we are running the

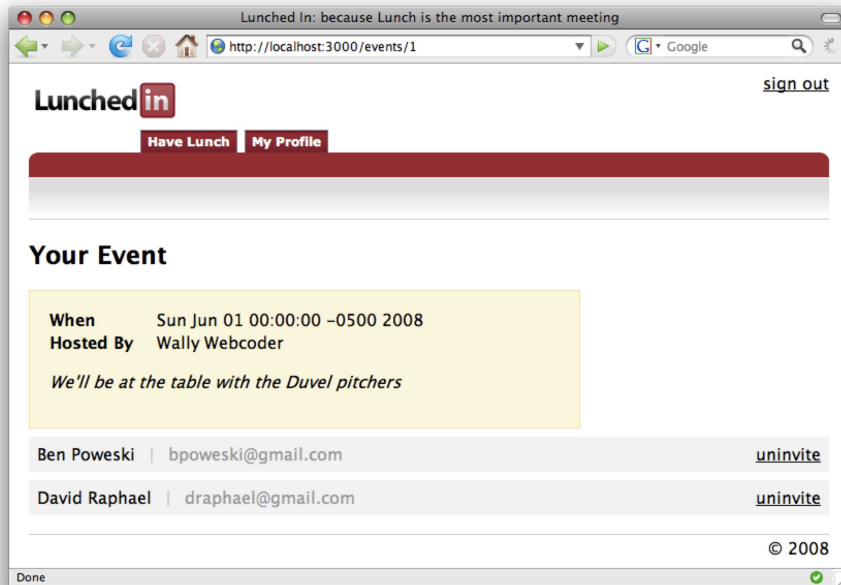


Figure 2.11: Direct ID access

sample application with `RAILS_ENV` set to development, then we should see the details of the error, as shown in Figure 2.12, on the following page. If the `RAILS_ENV` variable is set to production, a limited view of the error will be displayed.

Attacking the Login

Now that we have a good idea our application might not be handling input properly, it is time to attempt another exploit: circumventing the login process. The presence of a quote character has triggered an error outside of the expected norm, a rather nasty error message, so we can guess it is affecting the validity of the underlying database query.

A common approach for authenticating users in web applications is to delegate the logic of the login process to the database. While this isn't always the most sound design, it *can* be done securely. Our example does not. We cover authentication in detail in Chapter 6, *Authentication: Decentralized Authentication*, on page 95.

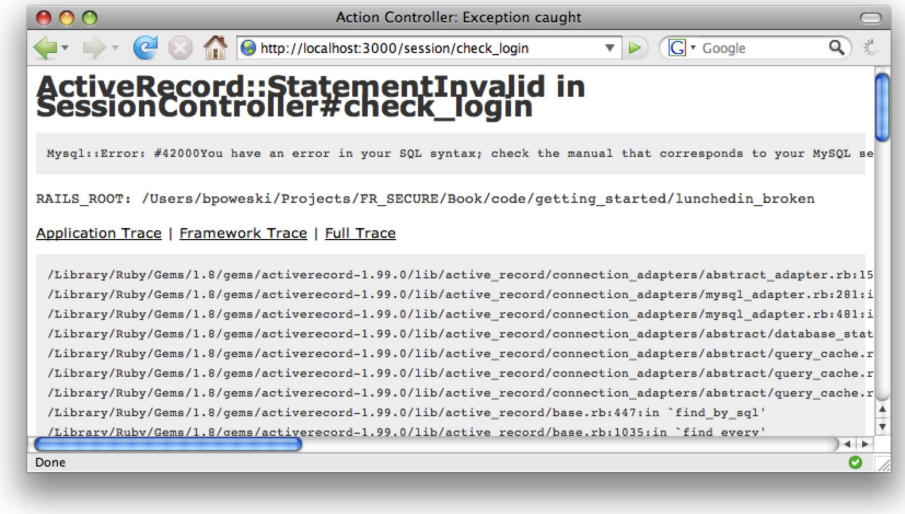


Figure 2.12: SQL interpretation error

For now, we are going to break the authentication scheme of the LunchedIn application. Enter the following values into the username and password form fields:

Field	Value
Username	wally
Password	fakepass' OR 'a' = 'a

When we input the preceding values into the form field, we are whisked away into the protected areas of the application. Congratulations, we've defeated the *authentication* process!

How did this happen? In the password field, we entered the values in the form of `value' or 'a' = 'a`. This allowed us to alter the boolean logic of the select statement to always return true.

Think of this attack as a crossword puzzle. We need a SQL fragment that starts with `'` that yields a true result when mixed with an existing query. Choose a real username (wally); then use `'a' = 'a` as the terminating, *always true* condition. The values sent will produce the following query, which returns all rows and then the first value:

```
SELECT * FROM users WHERE (username = 'wally'
AND password = 'fakepass' or 'a' = 'a') LIMIT 1
```

In the wild, hackers don't *usually* have access to your source, but don't let that set you at ease! Security professionals refer to the belief that you are secure because attackers don't know about your vulnerabilities as *security through obscurity*. Security through obscurity is an anti-pattern, or a *pattern of what not to do* in your application.

The consequences of SQL injection vulnerabilities can be devastating. The extensive security controls built around your database server—the private network, IP tables, and encrypted transport—become entirely useless as an attacker takes an express lane to the center of your database.

These examples have taught us the following:

- Characters that have meaning within the SQL language for your database can be used to identify possible vulnerabilities.
- SQL statements that always return true can be used to alter delegated authentication or authorization controls that utilize results from the database.

2.6 Exploit: Cross-Site Scripting

For the purposes of this book, we will focus on identifying the unique attributes of a cross-site scripting exploit, and leave the naming and semantics to someone else. As with many technology concepts, various groups have embarked on an effort to categorize the subcategories of this exploit. This leads to the unfortunate side-effect that if someone mentions *cross-site scripting*, there isn't a precise definition of what it means. However, we can all agree that cross-site scripting vulnerabilities are bad, regardless of what subcategory anyone wants to put them under.

HTML Injection

Let's begin by showing an example of *HTML injection* that uses the login screen we attacked in the previous section. HTML injection is just what it sounds like: inserting markup so that the server will echo it back into a page.

When entering various values into the username form field, you might have noticed that the application echos back the supplied value. Given this, let's see if the developer has missed some important input-vali-

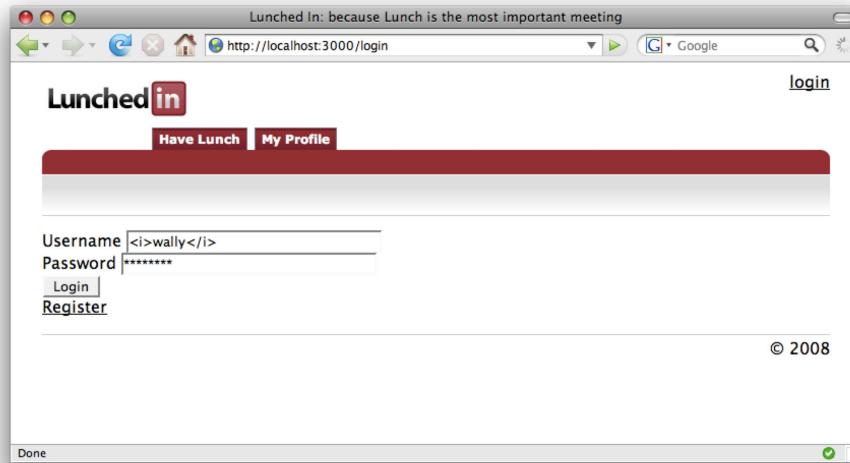


Figure 2.13: Simple XSS example

validation techniques, such as those discussed in Chapter 5, *Validation*, on page 84.

We can test our hypothesis by entering the following value into the Username field of the login screen, along with an incorrect password so the application will trigger the failure notification:

```
<i>wally</i>
```

You should see the username in a slightly different form, as shown in Figure 2.14, on the next page. Italics aren't especially menacing, but this action proves the application is in fact vulnerable to *HTML injection* exploits. However this doesn't mean too much until we look at more advanced attacks. It might be helpful to think of HTML injection as the basic building block of all other XSS vulnerabilities.

Reflected XSS

Getting users to enter JavaScript into a text field on the behalf of an attacker is not *entirely* out the question, but it isn't the most practical approach, either. A far more common approach is to embed the payload in a hyperlink. This is done with a technique called *reflected cross-site scripting*.

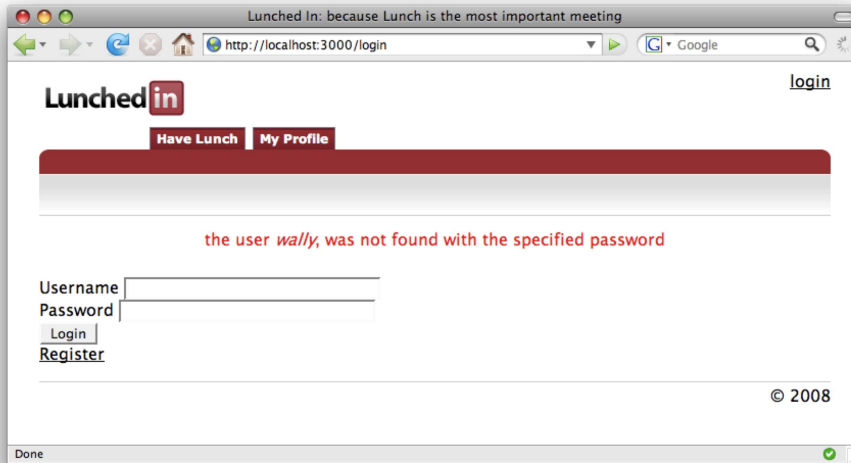


Figure 2.14: Simple XSS result

Enter the following into your browser:

http://localhost:3000/session/check_login?user%5Busername%5D=%3C%3Ewally%3C/i%3E

The result should be similar to what we saw in Figure 2.14. Note how the login screen of the application echos the user name “wally” in the error message. In Rails, the special characters [and] are used to group sets of like parameters together. When sending special and reserved characters, we must URL-encode them. We can accomplish this in Ruby by calling the `escape(string)` method of the CGI class:

```
irb(main):001:0> require 'cgi'
=> true
irb(main):002:0> CGI.escape('user[user_name]')
=> "user%5Buser_name%5D"
```

Ruby’s CGI class is part of the standard library. The `escape()` method is used to URL-encode the input. Keep in mind that the similar `escape-HTML()` method lets you escape special HTML characters from interpretation by a browser:

- All HTTP requests are vulnerable, but GET requests require the least effort to exploit.

Reserved Characters	Hex	Unsafe Characters	Hex
\$	%24	Space	%20
&	%26	'	%22
+	%2B	<	%3C
,	%2C	>	%3E
/	%2F	#	%23
:	%3A	%	%25
;	%3B	{	%7B
=	%3D	}	%7D
?	%3F		%7C
@	%40	/	%5C
		^	%5E
		~	%7E
		[%5B
]	%5D
		'	%60

Figure 2.15: URL-encoded values

- Be wary of links with HEX encoding because hackers might use them to smuggle HTML payloads.

Cookie Theft Using Persistent XSS

Persistent XSS is an attack where *JavaScript* is submitted to a site, saved, and then repeated back to the world long after the attacker has gone. Enter the following value into a venue's comment:

```
<script>
  alert("I'm vulnerable. Please hold me.");
</script>
```

If all goes as expected, we should see some something like Figure 2.16, on the next page.

This alert will be shown every time the page is loaded. Apart from being really annoying, the markup serves an important purpose: it is a good test for other kinds of attacks.

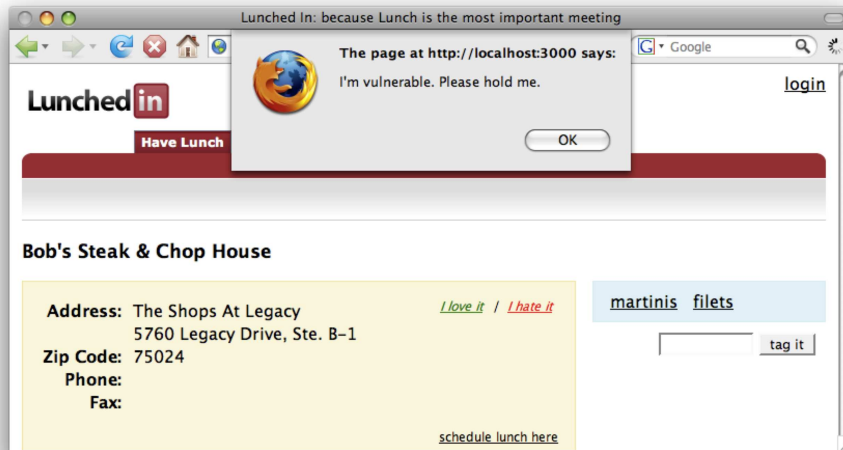


Figure 2.16: Persistent cross-site scripting

If we can inject JavaScript into a comment, then we can probably harvest session identifiers,³ invoke fancy Ajax methods, or leak information accessible from the page.

Navigate to a venue, such as the one at <http://localhost:3000/venues/1>, and enter the following as a comment:

```
<script>
  new Image().src = "http://localhost:3000/?cookie=" + escape(document.cookie);
</script>
```

If you check the console log of your Ruby server, you'll see a request containing the contents of your cookie. Note how we escaped the value of the `document.cookie` property using the JavaScript `escape(contents)` method. This is important because cookies typically contain values not legal within URLs.

In our example, we sent the session tokens to the same web server that we lifted the session information from. In the wild, an attacker would send your session token to an entirely different site. This practice is known as *session harvesting* or cookie theft. At this point, the

3. Rails 2.0+ contains session fixation attack protections that mitigate the impact of these kinds of attacks.

attackers could use the session identifier as their own, enabling them to do anything that the account is allowed to do. Rails gives us the *protect_from_forgery* mechanism to make this type of attack more difficult; we'll cover this in Chapter 3, *Fixing the Example*, on page 49.

It should also be noted that this attack is valid only for the duration of the active session. Limiting the length of active sessions through the use of session timeouts enables us to reduce the window of time that this kind of attack is viable:

```
Completed in 0.05237 (19 reqs/sec) | Rendering: 0.02995 (57%) | DB: 0.016 (31%)
200 OK [http://localhost/?cookie=_lunchedin_session_id%3D2379e020624035e45d7]
```

In our access log, we can see what session it is: 2379e020624035e45d762d-880527f41d. The prefix, *_lunchedin_session_id*, is used by Rails to make sure each application's session ID does not collide with another. The value in between, *%3D*, is a hex-encoded equals sign.

So far we've learned the following about cross-site scripting:

- If an application will accept input from a user that includes HTML markup and the markup is echoed back on the screen, then the application is likely vulnerable to XSS exploits.
- An XSS flaw exists if we can embed properly encoded markup into parameters that are echoed by the application.
- XSS vulnerabilities can be used to send sensitive data to remote sites or to automate unwanted actions.

2.7 Exploit: Cross-Site Request Forgery

In a *cross-site request forgery* (CSRF, pronounced “see-surf”), a user is tricked into executing malicious code on a site using his own credentials.

Computers are not very good at determining the intent of requests sent by users, which makes it difficult for applications to protect sites against CSRF attacks.

Attack Clients

To initiate a CSRF attack, we need a mechanism to initiate the network call from the victim's browser. Fortunately for us, HTML and JavaScript provide this functionality! The following is a brief list of various HTML

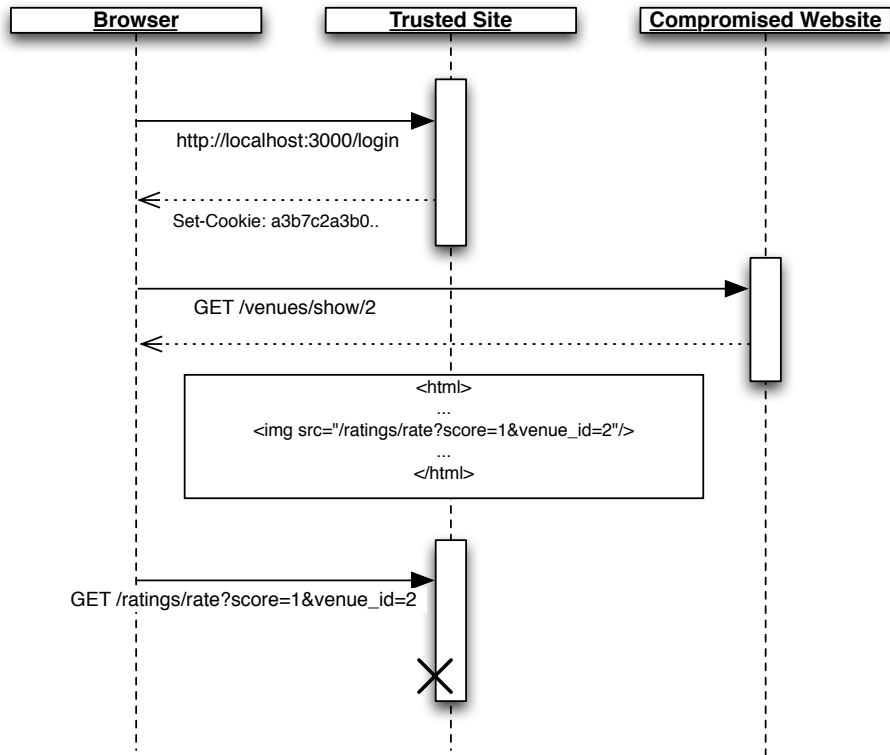


Figure 2.17: Cross-site request forgery

elements and JavaScript classes that permit remote access of network resources using HTTP GET requests:

- script
- img
- link
- form
- object
- applet
- embed



Joe Asks...

How Does CSRF Differ from XSS?

How does CSRF differ from XSS? People disagree about the exact differences; the best explanation we can give is that XSS exploits the trust a client has for a site, while CSRF exploits the trust a site has for a user.

For example, a browser interpreting malicious markup saved into a comment on a forum is an example of XSS because the user trusts the site. If that markup caused the client's browser to do something against another site, such as execute a stock trade via an existing authenticated session, it would be a CSRF attack against the remote site. In many cases, the techniques are combined to create a more devastating attack than could be managed with either technique on its own.

Similarly, the following JavaScript classes can be instantiated and used to invoke remote HTTP GET requests:

- XMLHttpRequest⁴
- Image

Now let's add the following markup as a comment to one of the restaurants listed under <http://localhost:3000/venues>:

```
<script>
  new Image().src = "http://localhost:3000/ratings/rate/2?score=1"
</script>
```

After we add this HTML to the comment, we should see something like the following in the server log files:

```
Processing RatingsController#rate (for 127.0.0.1 at 2007-07-01 13:53:10) [GET]
  Session ID: b7b9c3c9a1dc311f08fdb3112e4d1022
  Parameters: {"score"=>"1", "action"=>"rate", "venue_id"=>"2",
    "controller"=>"ratings"}
```

This is a sign the CSRF attack worked! We initiated a vote by simply viewing the comment. What makes this attack so powerful is that the

4. The XMLHttpRequest object is subject to the same-origin policy.

JavaScript located on this site could be used to attack an *entirely* different one. For example, a comment viewed on LunchedIn could trigger an action to a site that you visited previously. The worst part is, this traffic would be nearly indistinguishable from normal traffic patterns.

So far we've learned the following about cross-site request forgery:

- Compromised sites can be used to attack sites that do *not* have cross-site scripting vulnerabilities.
- Cross-site vulnerabilities typically occur without the user's knowledge and typically appear as normal traffic.

The purpose of this chapter was to show the thought process an attacker uses to exploit a web application's vulnerabilities. We hope that highlighting and understanding some common exploits will enable us to protect our own software from such exploits.

Fixing the Example

In the previous chapter, we saw how vulnerabilities in our sample application could be exploited. Let's spend this chapter patching up these holes. The goal isn't to show the most elegant solutions possible, but something that works. In Part II of this book, we'll go a step further and strengthen the security of the application by adding additional defenses.

Keep in mind that removing every possible security flaw from an application is impossible. It only takes one vulnerability to compromise an application, so it's easy to see how this might be a discouraging topic. Don't get frustrated! Security isn't a black art, contrary to what many would have you believe. Through an iterative process, we can build up defenses that *will* reduce the risk of your application being compromised. Reducing the risk is what securing your application is all about.

Topics that we'll explore in this chapter include the following:

- How to whitelist and blacklist the binding of parameters by using ActiveRecord class methods; we'll also cover when it is appropriate to use each method.
- How to use authorization techniques that rely on *roles*, including when it is appropriate to use them.
- How to implement *input validation* and output sanitization techniques to prevent SQL injection and XSS exploits.
- How to implement basic techniques to prevent CSRF attacks.

3.1 Protecting Against Parameter Manipulation

The first exploit we looked at in the previous chapter was a case of parameter manipulation. Within Rails, HTTP parameters are mapped

to object properties. In typical Rails fashion, this *convention* saves us time because we can instantiate a model object by passing a Hash of HTTP parameters to our object.

Let's look at an example. The following is the vulnerable code that we attacked during the creation of a User accessible at the <http://localhost:3000/users/new> URI:

Download [getting_started/lunchedin_broken/app/controllers/users_controller.rb](#)

```
Line 1 def create
-   @hide_tabs = true
-   @user = User.new(params[:user])
-   if @user.save
5     flash[:notice] = 'your account was successfully created.'
-     session[:user_id] = @user.id
-     redirect_to user_path(@user)
-   else
-     render :action => 'new'
10  end
- end
```

Note how the value `params[:user]` is passed into the `new(params)` method on line 3 of the `User` class. At this point, all parameters of the HTTP request that have the form of `user[property_name]` are bound to their respective properties. The `role_id` attribute determines the privilege level of a user. This is why the application gladly assigned the unintended privilege level when we added the `user[role_id]` parameter.

The question is this: how can we prevent automatic binding of parameters to ActiveRecord properties? One approach would be to remove the sensitive parameters from the `params` hash before we pass it to our ActiveRecord object. The Rails designers took this technique a step further and created the `attr_protected()` class method that does exactly this. Let's protect our application from the exploit by updating the `User` class to look like this:

Download [getting_started/lunchedin/app/models/user.rb](#)

```
Line 1 class User < ActiveRecord::Base
-   attr_protected :role_id
-   validates_presence_of :email, :first_name, :last_name
-   validates_presence_of :username, :password_hash
5   validates_uniqueness_of :username
-   validates_length_of :zip_code, :is => 5
-   validates_length_of :password, :in => 7..32
-   validates_numericality_of :zip_code
-   validates_confirmation_of :password
10  has_many :events
-   has_many :comments, :as => :commentable
-   belongs_to :role
-
```

```

-   def password=(password)
15   @password = password
-   write_attribute(:password_hash,
-     OpenSSL::Digest::SHA1.new(password).hexdigest)
-   end
-
20  def password
-   @password
-   end
-
-   def authentic?(p)
25   password_hash == OpenSSL::Digest::SHA1.new(p).hexdigest
-   end
-
-   def self.find_and_authenticate(username, password)
-     user = User.find_by_username(username)
30     return false if user.nil?
-     if user.authentic?(password)
-       user
-     else
-       false
35     end
-   end
- end
end

```

Note how line 2 of `user.rb` invokes the `attr_protected()` class method, passing the `:role_id` symbol. The `role_id` attribute will not be populated via the `new(params)` or `update_attributes(params)` methods.

Let's update our test case to verify that this parameter is *not* set when the `create()` action is executed:

[Download](#) `getting_started/lunchedin/test/functional/users_controller_test.rb`

```

Line 1 test "should not create user with mass assigned role id" do
-   post :create, :user => {
-     :username => 'bsmith',
-     :password => 'hello11',
5     :password_confirmation => 'hello11',
-     :email => 'bob@yahoo.com',
-     :zip_code => '75024',
-     :first_name => 'Bob',
-     :last_name => 'Smith',
10    :role_id => 1
-   }
-   user = User.find_by_username('bsmith')
-   assert_not_nil user, assigns(:user).errors.inspect
-   assert_equal 2, user.role_id
15  assert_response :redirect
-   assert_redirected_to user_path(user)
-   assert_equal @request.session[:user_id], user.id
- end
end

```

The `:role_id` parameter is added to the HTTP POST request with a value of 1 on line 10. As you might recall from the previous chapter, 1 is the primary key of the administrator role for LunchedIn. Next, we assert that the persisted User has a `role_id` of 2 on line 14. It is the `attr_protected(properties)` class method that makes all this possible.

Listing sensitive parameters works well when there are only a handful of items. You will likely run into situations where you want the *default* behavior to be secure. We can accomplish this by using the `attr_accessible(properties)` class method. This method switches off automatic binding for all properties *except* those passed as arguments. This means we need to use the `new()` method to list *every* parameter we want set during instantiation.

When using ActiveRecord, it's important that you remember several key points:

- Use `attr_protected()` to *blacklist* sensitive parameters. This method is ideal when there are few sensitive parameters because it can reduce the lines of code required in the model class definition.
- Use `attr_accessible()` to *whitelist* nonsensitive parameters. This method is our best option when *most* of the application parameters are sensitive.
- *The default behavior in Rails is to bind all HTTP parameters to ActiveRecord model objects.* This approach makes things easy on the developer, but it is not always secure. We must exercise care when building our applications, and we must understand the limitations of using ActiveRecord without calling `attr_accessible()`.

3.2 Adding Authorization

Authorization is the act of enforcing what a user can or cannot do. The algorithm behind allowing or disallowing behavior is simple:

```
if user X is permitted to do Y
  permit user X to do Y
else
  disallow action and show error
end
```

Unfortunately, the designs required to implement the abstraction of such functionality are not always as simple. Before diving into the topic of *authorization*, there are a few terms we need to become acquainted

with. Security people often refer to *users* as *principals*. Rather than declaring all permitted actions of a *principal*, it is often easier to group them into something called a *role*. We can now revise our pseudo-code to say this:

```
if principal X is in role Admin
  show admin console
else
  redirect to main page and show error
end
```

This snippet illustrates *role-based access control*, which is a fancy name for an *authorization model* that maps *principals* to groups of allowable actions or *permissions*. We'll focus on authorization in more detail in Chapter 7, *Authorization*, on page 135. For now, let's concentrate on fixing our code.

Fixing the Voting Link

In the previous chapter, we exploited the voting functionality of LunchedIn by replaying the Ajax request using Tamper Data. Let's investigate where the vulnerability might be:

Download `getting_started/lunchedin_broken/app/views/venues/show.html.erb`

```
<div id="venue-rating">
  <%= unless @venue.has_user_rated?(session[:user_id]) %>
    <%= render :partial => 'ratings/new' %>
  <%= else %>
    <%= render :partial => 'ratings/show' %>
  <%= end %>
</div>
```

The logic to determine whether the user should receive voting links or a summary is determined by the result of the `has_user_rated?(user_id)` method. This method takes a single argument, the `user_id` of the currently logged-in user. So far, so good. Now let's look at `RatingsController`:

Download `getting_started/lunchedin_broken/app/controllers/ratings_controller.rb`

```
Line 1 def rate
2   @venue = Venue.find(params[:id])
3   @venue.ratings << Rating.new do |r|
4     r.score = params[:score]
5     r.user_id = session[:user_id]
6   end
7   render :partial => 'show'
8 end
```

Eureka! Now we see the source of the vulnerability, which is that the `rate()` action of `RatingsController` is not applying the same protection afforded by the view. We can see on line 3 that a new `Rating` object is instantiated and added to the `Venue` object. Nothing stops this code from creating a new `Rating` object.

To fix the vulnerability, we need to add the same authorization check that we see in the view of `show.html.erb`. Update `RatingsController` to look like the following example:

Download `getting_started/lunchedin/app/controllers/ratings_controller.rb`

```
Line 1 def rate
2   @venue = Venue.find(params[:id])
3   unless @venue.has_user_rated?(session[:user_id])
4     @venue.ratings << Rating.new do |r|
5       r.score = params[:score]
6       r.user_id = session[:user_id]
7     end
8   end
9   render :partial => 'show'
10 end
```

On line 3, we add the `unless` statement, which causes the code to fail safely into the `render()` method. Now subsequent calls to vote on a venue will be blocked, regardless of which rendered view is displayed to the user.

The following test cases try out the fix applied to `RatingsController`:

Download `getting_started/lunchedin/test/functional/ratings_controller_test.rb`

```
test "should rate positively using post" do
  assert_difference 'Rating.count', 1 do
    @request.session[:user_id] = users(:bob).id
    2.times do
      xhr :post, :rate, :id => venues(:micos).id, :score => 1
      assert_response :success
    end
  end
end
```

Download `getting_started/lunchedin/test/functional/ratings_controller_test.rb`

```
test "should rate negatively using post" do
  assert_difference 'Rating.count', 1 do
    @request.session[:user_id] = users(:bob).id
    2.times do
      xhr :post, :rate, :id => venues(:micos).id, :score => 0
      assert_response :success
    end
  end
end
```

Our updated functional test case mimics the behavior of multiple votes for a Venue with the same user. When developing authorization controls, it is important to do the following:

- Enforce the same level of authorization throughout your application. Changing the output of a view can give a false sense of protection from an attack. This behavior needs to be enforced during the execution of the corresponding action.
- Add authorization test methods in your controller test cases. Good test cases will trigger failures if the behavior changes; adding these methods will help you discover vulnerabilities sooner than you would if you were to employ only manual testing procedures.

Protecting ID Parameters

Direct ID access is easy to overlook. Most software processes focus entirely on the domain, rather than aspects such as security. In other words, security is usually an afterthought when we write software. Implementing proper authorization is imperative for creating a secure application.

In LunchedIn, we exploited direct access of Event objects. This means that any user could manipulate the model associated with the primary key as if it were her own. Let's look at the EventController class and see if we can find the source of the flaw.

Several actions in this controller operate on event data. The data for any particular event object is owned by a specific user, so we need to be careful not to reveal it to other users:

[Download](#) `getting_started/lunchedin_broken/app/controllers/events_controller.rb`

```
def show
  @event = Event.find(params[:id])
end
```

We must determine which methods need protection from prying eyes. First, the show() method should be limited to those events created by the current user only. Let's update this method with the following code to authorize access properly:

[Download](#) `getting_started/lunchedin/app/controllers/events_controller.rb`

```
Line 1 def show
      2   begin
      3     @event = @authenticated_user.events.find(params[:id])
      4   rescue
      5     render :action => 'no_access'
      6   end
      7 end
```

On line 3, you can find the query from `User.find(params[:id])` to `@authenticated_user.events.find(params[:id])`. This technique takes advantage of the `has_many :events` association to scope the `find` method of the event to the `user_id` of the currently logged-in user. We can accomplish this by using the `@authenticated_user` instance variable that is set in a filter defined in `application.rb`. The `show()` action is taken care of, so let's look at the issues that remain with the `update()` method. The following code shows the broken `update()` action:

Download `getting_started/lunchedin_broken/app/controllers/events_controller.rb`

```
def update
  @event = Event.find(params[:id])
  if @event.update_attributes(params[:event])
    redirect_to event_path(@event)
  else
    render :action => 'show'
  end
end
```

You might have noticed that this method also lacks appropriate access controls. To fix this, we need to update the following methods with authorization logic:

Download `getting_started/lunchedin/app/controllers/events_controller.rb`

```
def update
  begin
    @event = @authenticated_user.events.find(params[:id])
    if @event.update_attributes(params[:event])
      redirect_to event_path(@event)
    else
      render :action => 'show'
    end
  rescue
    render :action => 'no_access'
  end
end
```

Download `getting_started/lunchedin/app/controllers/attendees_controller.rb`

```
def create
  begin
    @event = @authenticated_user.events.find(params[:event_id])
    if @attendee = @event.attendees.create(params[:attendee])
      respond_to do |format|
        format.js
      end
    end
  rescue
    render :nothing => true
  end
end
```


Download `getting_started/lunchedin/app/controllers/attendees_controller.rb`

```
def destroy
  begin
    @attendee = @authenticated_user.events.find(params[:event_id]).
      attendees.find(params[:id])
    if @attendee.destroy
      respond_to do |format|
        format.js do
          render :update do |page|
            page.visual_effect :fade, "attendee_#{@attendee.id}"
          end
        end
      end
    end
  rescue
    render :nothing => true
  end
end
```

The updated code isn't pretty, but it does the job of protecting actions from users with valid sessions that didn't create the Event. We'll look at additional techniques to provide more concise declarations of authorization logic in Chapter 7, *Authorization*, on page 135. For now, let's move on to the issues with our login functionality.

3.3 Preventing SQL Injection

We can prevent SQL injection vulnerabilities by filtering properly or sanitizing the input. SQL injection occurs because the application trusts the data received will not contain interpretable code. The correct approach is to code defensively and assume that the input *will* likely contain these dangerous characters and then take the proper actions to prevent the application from interpreting such code.

What Is Wrong with the Login Controller?

Recall that we discussed how to break the login functionality of our application in Chapter 2, *Hacking the Example*, on page 23. The following code in SessionController enables SQL injection:

Download `getting_started/lunchedin_broken/app/controllers/session_controller.rb`

```
Line 1 def check_login
-   if @user = User.find(:first, :conditions =>
-     "username = '#{params[:user][:username]}' " +
-     "AND password = '#{params[:user][:password]}'")
5     session[:user_id] = @user.id
-     redirect_to user_path(@user)
```

```

-   else
-     flash[:notice] = "the user #{params[:user][:username]}, "
-     flash[:notice] << "was not found with the specified password"
10    redirect_to '/login'
-   end
- end

```

Let's drill down on what makes controller vulnerable. Take a look at the code in line 3 and line 4. Note how the query is constructed by interpolating or evaluating variables within a string using the `#{variable_name}` operators. Using variable interpolation for query construction is dangerous! This is because SQL fragments can be inserted into the constructed query, altering the outcome. In the previous chapter, we attacked LunchedIn by injecting SQL fragments in such a way that the login would never fail.

Getting Our Fix On

Now that we've identified the source of a vulnerability, let's patch the security hole. To fix this flaw, we need to add protection mechanisms before the query is sent to the database and evaluated. There are several techniques within Rails to mitigate the risk of such an attack. For example, we could try the following:

1. Add custom sanitization or filter methods using the `ActionController::Base` class method, `before_filter()`.
2. Add sanitization or filter logic to model objects using custom finder methods.
3. Rewrite the query to use the bind variables or generated finder methods. Doing so would delegate the responsibility for filtering dangerous characters to the ActiveRecord framework.

Each of these techniques will accomplish the goal, albeit in varying degrees of elegance.

Building sanitization logic into ActionController, as described in technique #1, is generally not advisable because web applications typically have more complex entry points at the controller layer than in the model layer. This added complexity increases the likelihood of introducing additional vulnerabilities

The model is *usually* where we should put filtering or sanitization logic. This is because the model, not the controller, is the abstraction layer to the persistent store where the attack is directed. If more complex

filtering is required, it makes sense that the *model* should contain the logic because we will probably want to reuse these protections in other entry points.

Technique #3 presents a different solution altogether: delegating the responsibility to the framework. This is the ideal technique for queries with a small number of input parameters. For cases where a flat list of substituted variables becomes unwieldy, we can use *named bind variables*. We have only two fields in this case, so let's use the generated finder methods.

The following code shows our `find_and_authenticate(username, password)` class method, which returns a `User` object if the password is valid against the hash stored in the database:

Download `getting_started/lunchedin/app/models/user.rb`

```
def self.find_and_authenticate(username, password)
  user = User.find_by_username(username)
  return false if user.nil?
  if user.authentic?(password)
    user
  else
    false
  end
end
```

Next we need to address the `authentic?(password)` instance method that hashes the password and compares the result:

Download `getting_started/lunchedin/app/models/user.rb`

```
def authentic?(p)
  password_hash == OpenSSL::Digest::SHA1.new(p).hexdigest
end
```

Download `getting_started/lunchedin/app/controllers/session_controller.rb`

```
Line 1 def check_login
-   username, password = params[:user][:username], params[:user][:password]
-   if @authenticated_user = User.find_and_authenticate(username, password)
-     session[:user_id] = @authenticated_user.id
5     redirect_to user_path(@authenticated_user)
-   else
-     flash[:notice] = "the user #{params[:user][:username]}, "
-     flash[:notice] << "was not found with the specified password"
-     redirect_to login_path
10  end
-  end
```

On line 3, we altered `check_login()` so it uses a class method on our `User` model. We also let `ActiveRecord` do the substitution because it *knows* how to sanitize the data that it uses to construct SQL statements.

That fixes our SQL injection flaw; now we need to update our unit test. On line 7, we assert the controller redirects to the login view. On line 8, we now assert that the `:user_id`—the value used to determine whether the user is logged in—is not set.

Download `getting_started/lunchedin/test/functional/session_controller_test.rb`

```
Line 1 test "should not validate login using sql injection" do
  2   post :check_login, :user => {
  3     :username => "bob" OR 'a' = 'a',
  4     :password => "fakepass" OR 'a' = 'a'
  5   }
  6   assert_redirected_to login_path
  7   assert_nil @request.session[:user_id]
  8 end
```

You might have noticed that our flash still suffers from an XSS flaw. Hold your horses, eagle-eyed readers; we'll get to that in the next section.

So far we've looked at techniques for securing the `SessionController` class. We will cover defenses that extend these concepts more thoroughly throughout the book; for now, we can use the following guidelines to list some dos and don'ts that detail our motivation for the design we chose:

- *Do* put sanitization and filter methods inside of methods on model objects. It protects your application at a low level, and if alternate interfaces such as Web Services use the same methods, these filters and methods will inherit the protection mechanisms.
- *Do* use Rails finder methods when possible. These methods are well-tested and will do the heavy lifting of preventing injection vulnerabilities for you.
- *Do not* build your own `ActiveRecord` query strings unless you protect the input sufficiently. Using variable interpolation operators such as `#{variable_name}` can be a source of injection flaws.

3.4 Cross-Site Scripting Countermeasures

As you did for SQL injection vulnerabilities, you need to take steps to mitigate XSS vulnerabilities through a combination of input validation

and output sanitization. XSS presents a more challenging input to filter, as safe and unsafe inputs look very similar.

What exactly is the difference between safe and unsafe input? Well, it depends on whom we ask and what the application is doing. Any input that is vulnerable to XSS attacks is unsafe. Unfortunately, these input patterns are deceptively difficult to blacklist. This is why blacklisting input patterns is considered a bad security practice.

Validating the Input

Getting input validation correct isn't easy. Fortunately, Rails provides excellent facilities to control input that enters the perimeter of a web application.

You might remember from the previous chapter that we inserted HTML markup into several areas of our web application. One area we inserted HTML was the `user[user_name]` form field at the `/session/login` URL. We fixed the SQL injection vulnerability in Section 3.3, *Preventing SQL Injection*, on page 57, but an XSS vulnerability still exists!

Update the `comment.rb` file with the following code:

Download `getting_started/lunchedin/app/models/comment.rb`

```
Line 1 class Comment < ActiveRecord::Base
2     belongs_to :user
3     belongs_to :commentable, :polymorphic => true
4     validates_presence_of :subject, :body, :commentable_id
5     validates_format_of :subject, :with => /\A[a-zA-Z0-9\n\t\s,.!?\-]*\Z/
6     validates_format_of :body, :with => /\A[a-zA-Z0-9\n\t\s,.!?\-]*\Z/
7 end
```

We add two calls to the `validates_format_of()` class method on line 5 and line 6. Next, we pass a regular expression that defines our allowable input for a `Comment` object. For good measure, let's create a unit test that describes a case where the `save()` method should not succeed because of forbidden input:

Download `getting_started/lunchedin/test/unit/comment_test.rb`

```
def test_save_fails
  comment = Comment.new do |c|
    c.subject = "Comment with markup <br/>"
    c.body = "hacked! <script>alert('hello');</script>"
    c.commentable_id = 1
  end
  assert_equal false, comment.save
  assert_equal 2, comment.errors.length
end
```

We will cover input validation in Chapter 5, *Validation*, on page 84. For now, you need to know that Rails provides excellent validation functionality with its validation class methods in the ActiveRecord library.

Sanitizing the Output

If attacks manage to slip through our application's force field of input validation, there is still hope. We can also sanitize the output of our application to prevent XSS flaws that have entered our system.

The following code comprises the `_comment.html.erb` partial that is used to display comments in the broken version of LunchedIn. In this partial, we render the output of each variable to produce HTML. The problem with this is that if *any* attributes contain dangerous characters, they are rendered blindly:

Download `getting_started/lunchedin_broken/app/views/comments/_comment.html.erb`

```
<div class="comment" id="comment_<%= comment.id %>">
  <% if @user && @user.role.name == 'admin' %>
    <div class="comment-delete-link">
      <%= link_to_remote 'mark as spam',
        :url => {:controller => 'comments', :action => 'destroy', :id => comment},
        :method => :delete %>
    </div>
  <% end %>
  <h4><%= comment.subject || "Comment" %></h4>
  <p><%= comment.body %></p>
  <div class="comment-meta">
    <i><%= comment.user ? comment.user.username : "anonymous" %></i> wrote
    <%= time_ago_in_words(comment.created_at) if comment.created_at %> ago
  </div>
</div>
```

We can fix this by changing how each attribute is rendered in the view. Rails includes the `h()` view helper method that is designed for just such a purpose. This method sanitizes the output of dangerous HTML markup.

You should always use `h(content)` or `html_escape(content)` methods when writing ActiveRecord properties! It's an easy thing to add, and it could end up saving your application from XSS vulnerabilities.

The following code shows the updated `_comment.html.erb` partial that contains the sanitized ActiveRecord properties.

Download `getting_started/lunchedin/app/views/comments/_comment.html.erb`

```
<div class="comment" id="comment_<%= comment.id %>">
  <% if @user && @user.role.name == 'admin' %>
    <div class="comment-delete-link">
      <%= link_to_remote 'mark as spam',
        :url => {:controller => 'comments', :action => 'destroy', :id => comment},
        :method => :delete %>
    </div>
  <% end %>
  <h4><%= h(comment.subject) || "Comment" %></h4>
  <p><%= h(comment.body) %></p>
  <div class="comment-meta">
    <i><%= comment.user ? h(comment.user.username) : "anonymous" %></i> wrote
    <%= time_ago_in_words(comment.created_at) if comment.created_at %> ago
  </div>
</div>
```

Congratulations; the dangerous markup will be escaped before it is ever rendered to the view!

Let's run the reflected XSS exploit from Chapter 2, *Hacking the Example*, on page 23. Enter the following into the patched example application:

```
http://localhost:3000/login/check_login?user%5Buser_name%5D=%3Ci%3Ewally%3C/i%3E
```

Note that the exploit is still present in the application:

Download `getting_started/lunchedin/app/controllers/session_controller.rb`

```
flash[:notice] = "the user #{params[:user][:username]}, "
flash[:notice] << "was not found with the specified password"
redirect_to login_path
```

Our goal is to sanitize the output so that the application will not blindly repeat dangerous input. Your first thought might be to use the `h()` method directly in `SessionController`, but the easiest approach is to call it from the view and pass the contents of the flash object:

Download `getting_started/lunchedin/app/views/session/login.html.erb`

```
<% if flash[:notice] %>
<div class="notice">
  <%= h(flash[:notice]) %>
</div>
<% end %>
```

To avoid repeating ourselves, we put the flash contents inside the `login.html.erb` view. To sanitize the output, we pass the results of the `flash[:notice]` method call to `h()`.

While we've undertaken a significant effort to sanitize, we have also created new problems for our application. Our solution is unfortunately hostile to internationalization, and it doesn't allow for common types of punctuation. Never mind this for now; we'll cover input validation in depth in Chapter 5, *Validation*, on page 84.

Here are a handful of points to keep in mind; following them will help us minimize our exposure to XSS exploits:

- The safest approach to preventing XSS exploits is to filter or escape all potentially dangerous characters. Use the `escapeHTML()` method of the `CGI` class to provide this functionality.
- *Do* filter or sanitize input before records are persisted. `ActiveRecord` callbacks such as `before_validation()` and `before_save()` can provide such functionality hooks, regardless of which method initiated the action.
- *Do* filter or sanitize model attributes displayed in a view. Actively sanitizing the markup generated from a persisted object will mitigate against input that infiltrated our input validation defenses. We can accomplish this using the `h()` method in the `.html.erb` files.

3.5 Protecting Against a Cross-Site Request Forgery

CSRF vulnerabilities can be difficult to protect against because they blend in with normal, user-generated traffic. The nature of these attacks is such that an unsuspecting user loads markup from a remote site and then initiates an attack on an entirely different site using his authenticated session.

The distributed nature of these attacks make them especially dangerous. For example, CSRF attacks occur in the context of the network location the victim's browser is running in, which means the target system could be any system visible from that user's location. Such systems, which can include trusted networks such as routers, firewalls, or intranet sites, can now be accessed by the hacker.

To protect our applications, we have to add countermeasures that distinguish this traffic and reduce the likelihood that non-user-generated traffic passes through. To do this, we combine several different techniques.

We can use the following techniques to mitigate against *cross-site request forgery* attacks:

- Enforce the use of proper HTTP request methods. This alone will not keep you safe, but it can thwart simple attacks using `` and `<link>` tags. Enforcing request methods also has the side benefit of reducing the chance accidental page loads will cause irreversible damage, as well as preventing proxy accelerators from precaching every link on a page.
- Ensure that session timeout values are enforced—the shorter the better. This reduces the window of exposure to a given attack.
- Use `protect_from_forgery()` to mitigate CSRF attacks through the use of request tokens.
- Implement CAPTCHA technologies, which ferret out computer-based bots from live humans. CAPTCHA is a bit of a two-edged sword; on the one hand, CAPTCHA technologies provides the best protection for mitigating CSRF attacks; on the other, the technologies themselves can be quite annoying, and they often introduce accessibility problems.

This chapter will focus on the first technique. We will cover techniques for leveraging CAPTCHA in Chapter 6, *Authentication: Decentralized Authentication*, on page 95.

Enforce Proper HTTP Methods

The HTTP specification¹ states that GET requests should be used only for *safe interactions*.² The authors of the HTTP protocol define *safe interactions* as those that take no action other than retrieval. This means that we should use *only* POST, PUT, or DELETE requests for any action that modifies data on the server. The methods GET and HEAD should be reserved for *idempotent* activities—those that produce the same result across multiple invocations.

We exploited an *unsafe interaction* in our venue voting attack. This was accomplished by embedding the `http://localhost:3000/venues/rate` URL with the proper parameters inside of the `src` attribute of an `` tag.

1. You can find more information on the HTTP specification at <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.

2. See Section 9.1 of the HTTP/1.1 specification for more details on safe interactions.

The Rails designers are aware of such situations, and they have given us the `verify()` class method to combat this kind of exploit. Add the following code to the `VenuesController`:

[Download](#) `getting_started/lunchedin/app/controllers/venues_controller.rb`

```
Line 1 class VenuesController < ApplicationController
  2
  3   verify :method => :post,
  4         :only => [ :create, :add_tag, :comment, :rate ],
  5         :redirect_to => { :action => :list }
```

On line 3, we invoke the `verify()` class method. This method allows `create()`, `add_tag()`, `comment()`, and `rate()` actions only if they are invoked using an HTTP POST request. If another method is used, a redirect to the action `list()` is returned.

The first line of defense in protecting against CSRF vulnerabilities is to enforce proper usage of HTTP methods. Unfortunately, this will not protect us fully from every attack. Attacks that are initiated with ECMAScript can forge the HTTP request method, thus rendering your application vulnerable.

Note that verification is built in to RESTful routes if you declare them using `map.resource` in `routes.rb`.

It's probably obvious by now that protecting our application from a multitude of attacks can be quite involved. Part of becoming a proficient developer of secure code is knowing when enough is enough. There are still issues surrounding our application, but the purpose of this chapter wasn't to remove every last possible vulnerability (that is impossible), but instead to introduce techniques we can build on in later chapters.

Enforcing HTTP POST requests has the side benefit of reducing the trail of form data in access logs.

Part II

Building on the Basics

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

► Edsger W. Dijkstra, 1972

Download at WoweBook.com

Chapter 4

Testing for Security

If you want to sleep at night knowing your software is secure, you must test it, and test it well. But testing software is not enough to guarantee that it *is* secure—you have to test the right things. The *right things* vary for each application, but there is a large amount of commonality. Our programs use authorization, authentication, validation, and data protection as defenses against attack. We can test these defenses programmatically. In the same way that we describe how our application should respond to valid inputs, we can test it to make sure it responds appropriately to common attack scenarios.

4.1 Things You Should Test

The following sections include lists of questions to think about when deciding which test cases to write for your applications. These aren't comprehensive lists, but the questions will get us thinking about how to approach testing our security concerns. In later chapters, we'll continue to focus on creating sound test cases for the techniques described.

Keeping in mind four basic defense points can help us write good test cases for our application: authentication, authorization, data protection, and input validation and sanitization.

Authentication

We need to consider several issues when deciding which authentication test cases to apply to passwords and other aspects of verifying a user's identity:

- How robustly does the application handle the authentication of users? Can malicious users bypass the authentication process altogether?
- Are we enforcing an acceptable password strength for users? Can the user register with no password? Can the user update her password to a blank value?

Authorization

Once we authenticate a user, we need to ensure that he can access the material intended for him—and only the material intended for him. Test cases should take into account several authentication issues:

- Does the application verify that only authenticated users with appropriate privileges can access sensitive areas of the application?
- Do we have authorization checks for all `show()`, `edit()`, `update()`, `create()`, and `delete()` actions? Do we have authorization checks for custom action methods?

Data Protection

There is nothing more important to our application than the data that drives it, so protecting its contents is critical. We should keep the following questions in mind when designing test cases for data protection:

- Are sensitive parameters masked properly in log files? Are such parameters within the application's view?
- Are sensitive fields encrypted or hashed during persistence?
- Are we using ciphers that are currently regarded as insecure, such as DES or MD5?
- Are we enforcing the use of TLS during the transmission of sensitive information such as passwords, credit card numbers, and Social Security numbers?

Input Validation and Sanitization

Proper validation and sanitization are essential for providing a line of defense against would be attackers of our web application; questions to consider when creating test cases to measure our validation and sanitization efficacy include the following:

- Are we allowing dangerous input to enter through our application's interfaces?
- When displaying information, are we sanitizing the output?
- Is all input validated on the server, regardless of what browser checks might be in place?
- Are we following secure coding practices when we construct SQL queries? Have we designed these queries so they cannot accept unsafe input? Have we given enough attention to XPath, LDAP, system commands, or filenames?

4.2 Testing in Layers

As Rails developers, we know that we already have access to a wide range of internal application testing tools. When Rails creates a new application, it also creates unit, functional, and integration test directories. When we add new models, the corresponding stub unit tests are added. Similarly, each new controller gets a stub functional test.

These unit and functional tests provide a vital foundation when testing our application's security. We'll look at each shortly. In Figure 4.1, on the next page, we can see what types of tests are suited for various vulnerability categories.

But we will need more than the basic Rails testing infrastructure to address all the security questions in Section 4.1, *Things You Should Test*, on page 68. We'll also need external tools—tools that let us prod and poke at the application from the outside. Later in this chapter, we'll explore one of these applications, Selenium, in greater depth.

4.3 Authentication Tests

Authentication is the act of verifying a user's identity. To test our authentication measures, we must go through all the possible permutations of how our application enacts authentication. We start by testing

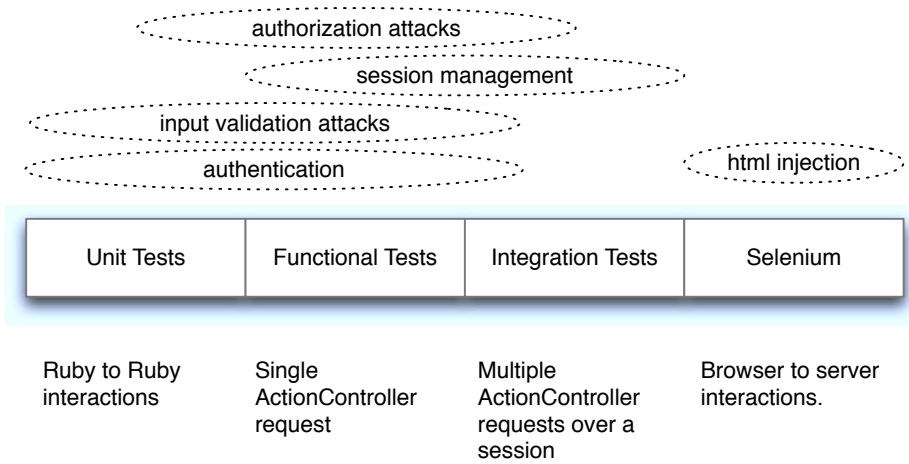


Figure 4.1: The Rails testing stack

a happy case of a user successfully logging in. The authentication process is really tied to the business logic of verifying a user's password, so let's create a unit test that reflects this:

[Download](#) testing/lunchedin/test/unit/user_test.rb

```
test "authentication is successful with valid password" do
  @user = users(:bob)
  assert @user.authentic?('hello')
end
```

The user model handles the business logic behind the authentication process, but the *act of authentication* or the *login process* occurs in a different layer. In our application, it occurs in the `SessionController` class. Let's add a functional test case that verifies this part works as expected:

[Download](#) testing/lunchedin/test/functional/session_controller_test.rb

```
def test_login_success
  @request.env['HTTPS'] = 'on'
  post :check_login, :user => { :username => 'bob', :password => 'hello' }
  assert_response :redirect
  assert_not_nil @request.session[:user_id]
  assert_equal users(:bob).id, @request.session[:user_id]
end
```

Logging out a user properly is equally important, so let's add a test case for this functionality, as well:

[Download](#) `testing/lunchedin/test/functional/session_controller_test.rb`

```
def test_logout_success
  @request.session[:user_id] = users(:bob).id
  post :logout
  assert_response :success
  assert_nil @request.session[:user_id]
end
```

At this point, we know our application is able to authenticate the user. What we don't know—and this is far more important—are the circumstances that surround the process of rejecting authentication. Let's look at some code that tests an authentication failure:

[Download](#) `testing/lunchedin/test/unit/user_test.rb`

```
test "authentication fails with invalid password" do
  @user = users(:bob)
  assert !@user.authentic?('hello1')
end
```

The preceding bit of code shows a test case that asserts how our authentication process *shouldn't* behave. When testing the security of our application, it is important that we focus on how it *shouldn't* work, rather than how it should. Security vulnerabilities exist because developers didn't anticipate properly how their application would respond to a particular input.

So we should also make certain that our authentication filter is not bypassed. We're operating at the controller layer, so we use a functional test:

[Download](#) `testing/lunchedin/test/functional/venues_controller_test.rb`

```
def test_logged_out_redirect
  get :index
  assert_response :redirect
  assert_redirected_to :controller => 'session'
end
```

The next step is to create test cases to verify the strength of our authentication factors; in our example, authentication consists of a simple password. The effectiveness of our authentication process is dependent upon the strength of our user's password, so it would behoove us to write test cases that exercise these policies!

Let's start by writing a test case that asserts a user cannot enter an empty password. Our user is a model, so a unit test is ideal for this scenario:

Download `testing/lunchedin/test/unit/user_test.rb`

```
test "user creation should fail with weak password" do
  @user = User.new do |u|
    u.first_name = 'Bob'
    u.last_name = 'Smith'
    u.password = 'short'
    u.password_confirmation = 'short'
    u.email = 'bob@somedomain.com'
    u.username = 'bsmith'
    u.zip_code = '75024'
  end
  assert !@user.save
end
```

So far we're confident that our code is checking whether authentication occurs where it is supposed to. We also know that we're protecting our users from themselves by enforcing a password-length requirement. We'll cover user registration in more detail in Chapter 6, *Authentication: Decentralized Authentication*, on page 95.

Once we have a user authenticated, we now need to make sure our application restricts what is allowed.

4.4 Authorization Tests

Testing for authorization is about making sure users are able to access only the data or operations they're supposed to. To test this, we need to mimic illegal access and make sure our application is responding accordingly. Let's look at the following functional test case:

Download `testing/lunchedin/test/functional/events_controller_test.rb`

```
test "not_authorized_update_status" do
  @request.session[:user_id] = users(:bob).id
  post :update, :id => events(:rubycon), :event => { :status => 'organized' }
  assert_template 'no_access'
end
```

Here we mimic an authenticated user who is attempting to modify another user's event. We use the `users(:bob)` user fixture, as defined in `users.yml`, to attempt to modify the `events(:rubycon)` event owned by Wally. Our controller utilizes the `events` association of the `User` object to query for an `Event`, so the test case won't find any event with Wally's ID. The point of this test case is twofold: first, we want to make sure that

we can't associate any event with Wally's ID; second, we want to make sure that the `no_access.html.erb` template is rendered.

But what if our authorization is dependent upon previous requests against a URI? We encountered such a scenario when we attacked the voting links of our application in Chapter 2, *Hacking the Example*, on page 23. To test for this, we can use an integration test:

```
Download testing/luncheon/test/integration/attack_voting_link_test.rb

def test_double_vote
  get '/venues'
  assert_response :redirect
  assert_redirected_to :controller => 'session'
  https!
  post_via_redirect '/session/check_login',
    :user => { :username => 'bob', :password => 'hello' }
  assert_response :success
  assert_template 'show'
  get_via_redirect venue_path(2)

  assert_response :success
  assert_template 'show'

  assert_difference 'Venue.find(2).ratings.count' do
    xhr :post, '/ratings/rate/2?score=1'
  end

  # try to post again
  assert_no_difference 'Venue.find(2).ratings.count' do
    xhr :post, '/ratings/rate/2?score=1'
  end
end
```

Our application now uses the `ssl_requirement` plug-in, so the following call was added before the post to the `check_login()` action:

```
https!
```

At a minimum, we should add tests that exercise the authorization checks of our domain objects. For RESTful controllers, this means testing the following methods: `show()`, `update()`, `create()`, and `delete()`. This is especially true if we rely on code created by `script/generate` because the default code created has no built-in restrictions.

4.5 Data Protection Tests

The question confronting us is this: how can we test to ensure *sensitive parameters are masked properly in log files*? To do this, we must invoke the `filter_parameter_logging(*params)` method for all *sensitive* parameters

in our application. The purpose of this test is to make certain that we prevent sensitive information from leaking into log files that might not be secured to the same degree as other parts of our application.

Let's begin by adding this call for the password and password_confirmation fields:

```
Download testing/lunchedin/app/controllers/application.rb
```

```
filter_parameter_logging :password
```

Next, let's look at how we can ensure that *sensitive fields are encrypted or hashed during persistence* using a unit test:

```
Download testing/lunchedin/test/unit/user_test.rb
```

```
test "password is hashed" do
  @user = users(:bob)
  assert_equal OpenSSL::Digest::SHA1.new('hello').hexdigest, @user.password_hash
end
```

Enforcing cipher selection policies is a bit trickier. If we want to write a test case around our application without using a particular cipher, we need to think outside the box.

One way we to discover the use of a forbidden cipher in our application is to replace the offending Ruby library with our own version, one that raises a RuntimeError informing us that a given library is using code we don't want it to use:

```
Download testing/illegal_cipher_test.rb
```

```
require 'openssl'

module OpenSSL
  module Cipher
    class Cipher
      alias :initialize_old :initialize

      def initialize(cipher)
        if cipher.upcase.match(/^DES/)
          raise "Cipher: #{cipher} is not acceptable!"
        else
          initialize_old(cipher)
        end
      end
    end
  end
end
```

Let's write some tests so we can see this principle in action:

Download `testing/illegal_cipher_test.rb`

```
def test_des_with_named_cipher
  assert_raises RuntimeError do
    cipher = OpenSSL::Cipher::Cipher.new("DES")
  end
end
```

Download `testing/illegal_cipher_test.rb`

```
def test_des_with_cipher_class
  assert_raises RuntimeError do
    cipher = OpenSSL::Cipher::DES.new
  end
end
```

Now let's add a test for a different cipher, asserting that it is allowed:

Download `testing/illegal_cipher_test.rb`

```
def test_aes_successful
  assert_nothing_raised do
    cipher = OpenSSL::Cipher::Cipher.new("AES128")
  end
end
```

Testing for the proper use of Transport Layer Security (TLS) seems odd, not least because our application isn't even operating in the layer where this would happen.

Secure Sockets Layer (SSL) is the predecessor of Transport Layer Security.

We can't really test for this in a traditional unit-testing manner, but we can leverage the `ssl_requirement` plug-in to prevent access. We'll look at this in more detail in a bit. For now, let's install the plug-in and lock down our application:

```
$ ruby script/plugin install ssl_requirement
```

After we install the plug-in, we need to include the `SslRequirement` module in our base application controller and then mark the methods that need confidentiality:

Download `testing/lunchedin/app/controllers/application.rb`

```
class ApplicationController < ActionController::Base
  include SslRequirement
  helper :all # include all helpers, all the time
end
```

Let's mark `SessionController` and `UsersController`, indicating they require a confidential conversation.

Download `testing/lunchedin/app/controllers/session_controller.rb`

```
class SessionController < ApplicationController
  ssl_required :check_login unless RAILS_ENV['development']
  ssl_allowed :index, :logout
```

Download `testing/lunchedin/app/controllers/users_controller.rb`

```
class UsersController < ApplicationController
  ssl_required :create unless RAILS_ENV['development']
  ssl_allowed :new, :show
```

Now when we run our functional tests, they fail! This is because they do not appear to occur in a confidential manner; more precisely, an environment variable sent with the request is not set. To fix this, we need to set the HTTPS environment variable in the same way our web server sets our Rails processes:

Download `testing/lunchedin/test/functional/session_controller_test.rb`

```
def test_login_success
  @request.env['HTTPS'] = 'on'
  post :check_login, :user => { :username => 'bob', :password => 'hello' }
  assert_response :redirect
  assert_not_nil @request.session[:user_id]
  assert_equal users(:bob).id, @request.session[:user_id]
end
```

Download `testing/lunchedin/test/functional/session_controller_test.rb`

```
Line 1 test "should not validate login using sql injection" do
  2   @request.env['HTTPS'] = 'on'
  3   post :check_login, :user => {
  4     :username => "bob' OR 'a' = 'a",
  5     :password => "fakepass' OR 'a' = 'a"
  6   }
  7   assert_redirected_to login_path
  8   assert_nil @request.session[:user_id]
  9 end
```

We also need to use the same technique to update our tests in `user_controller_test.rb`. We can accomplish this by adding the following line to each of our test methods; this line tells the `ssl_required` plug-in that our request is originating from a confidential transport:

```
@request.env['HTTPS'] = 'on'
```

Even with a confidential connection, it is critical that we prevent malicious input within our application.

4.6 Input Validation and Sanitization Tests

Building test cases for input validation and sanitization is usually fairly involved because such tests typically span multiple layers of our application. To test in the client layer, we need to solicit the help of the tool, Selenium, which we'll examine soon.

We saw that HTML injection and XSS vulnerabilities derive from the way the *browser* interprets content in Chapter 2, *Hacking the Example*, on page 23. However, it is our server-side code that allowed exploits in the first place. How should we approach testing for this?

First, let's consider the following test case:

`Download` `testing/lunchedin/test/unit/comment_test.rb`

```
def test_save_fails
  comment = Comment.new do |c|
    c.subject = "Comment with markup <br/>"
    c.body = "hacked! <script>alert('hello');</script>"
    c.commentable_id = 1
  end
  assert_equal false, comment.save
  assert_equal 2, comment.errors.length
end
```

Here we have a unit test that expects our model to reject persistence because the body and subject attributes contain potentially dangerous characters. As you might recall, our `Comment` model uses a regular expression in conjunction with the `validates_format_of()` class method:

```
validates_format_of :subject, :with => /\A[a-zA-Z0-9\n\t\s,!\?-\]*\Z/
```

How do we know that this is a safe validation method? Let's start by testing against a dictionary of known attacks.¹ This site provides a database of known XSS attacks you can use to test against. Let's create a functional test to execute our XSS attacks database against our `CommentsController`.

1. You can find an XML version of RSnake's cheat sheet for XSS attacks at <http://hackers.org/xssAttacks.xml>.

Installing Selenium for Rails

Selenium is installed using the *Selenium* gem and a corresponding Rails plug-in. Execute the following to install the Selenium gem and the Selenium rails plug-in:

```
testing> gem install selenium
testing> ruby script/plugin source \
  http://svn.openqa.org/svn/selenium-on-rails/stable
testing> ruby script/plugin install selenium-on-rails
```

If you're a Windows user, you will also need to make sure you have installed the win32-open3 gem. You can install it using this command:

```
C:\> gem install win32-open3
```

Now we're ready to build our in-browser test cases.

Download [testing/lunchedin/test/functional/comments_controller_test.rb](#)

```
def test_attack_comment_with_dictionary
  @request.session[:user_id] = users(:bob).id
  attacks = File.dirname(__FILE__) + '/../files/xssAttacks.xml'
  document = REXML::Document.new(File::read(attacks))
  document.elements.each("xss/attack") do |e|
    info = "#{e.elements['label'].text}: #{e.elements['code'].text}"
    assert_no_difference 'Comment.count', info do
      xhr :post, :create, :venue_id => 1, :comment => {
        :subject => "XSS Attack", :body => e.elements["code"].text }
    end
  end
end
```

The first time it is run, our test case passes the signature database. Let's see what happens when we modify the validation routine on our comment model and remove the sanitization method call to `h(input)` in the `_comment.html.erb` partial:

```
validates_format_of :body,
  :with => /\A[a-zA-Z0-9\n\t\s,.\!?\%\#\(\)\|<>]*\Z/
```

Here we've included a few extra characters that would not be out of place in the body of a comment in a public-facing application. Let's use Selenium to see how our browser reacts to such input.

At this point, we are ready to use Selenium to test in the browser for XSS vulnerabilities. Let's begin by adding a Selenium test case; we can accomplish by executing the following script:

```
testing> ruby script/generate selenium xss_test
```

Next, we add the following code to the generated `add_xss_comment.rsel` script under `test/selenium`:

```
Download testing/lunchedin/test/selenium/add_xss_comment.rsel
```

```
setup :clear_tables => [:comments], :fixtures => :all
wait_for_page_to_load(1000)
open '/login'
wait_for_page_to_load(1000)
type 'user_username', 'bob'
type 'user_password', 'hello'
click 'commit'
wait_for_page_to_load(1000)

open '/venues/show/3'
type 'comment_subject', 'XSS'
type 'comment_body',
  '<SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>'
click 'document.forms[1].commit'
pause '1000'
verify_failure_on_next 'There were no alerts'
verify_not_alert 'XSS'
```

We need to sync up Ajax with Selenium when we combine them, which we can accomplish with a `pause(time)` command. Otherwise, our next command might execute before the asynchronous call is complete.

We're using the `ssl_required` plug-in; however, the default configuration of Selenium operates only under the test Rails environment, so we need to update it so that Selenium works under the `RAILS_ENV['development']`, too. Create the `config.yml` file under `vendor/plugins/selenium-on-rails`. Update the location of the Firefox executable, if necessary:

```
Download testing/lunchedin/vendor/plugins/selenium-on-rails/config.yml
```

```
environments:
  - development
```

```
browsers:
  firefox: '/Applications/Firefox.app/Contents/MacOS/firefox-bin'
```

We can start the LunchedIn application with the following command:

```
testing> rake db:fixtures:load
testing> ruby script/server
```


Now let's execute our Selenium tests:

```
testing> rake test:acceptance
Starting firefox
"/Applications/Firefox.app/Contents/MacOS/firefox" "http://localhost:3000/se1...
..
Stopping browser (pid=1444) ...
Finished in 1 seconds.

0 tests passed, 1 tests failed
rake aborted!
Test failures
```

We now have a failing test case that evaluates the input in the context of a browser! Let's add the sanitization method call back into the `app/views/comments/_comment.html.erb` partial:

```
Download testing/lunchedin/app/views/comments/_comment.html.erb
```

```
<p><%= h(comment.body) %></p>
```

Let's run our Selenium test cases again:

```
testing> rake test:acceptance
Starting firefox
"/Applications/Firefox.app/Contents/MacOS/firefox" "http://localhost:3000/se1...
..
Stopping browser (pid=1555) ...
Finished in 1 seconds.

1 tests passed, 0 tests failed
```

Our input validation allowed the XSS markup through, but we have a test case asserting that the browser won't evaluate it because we're sanitizing the output using the `h(input)` method! We should assume that our application is only as secure as what we've tested for. The next logical step is to provide insight into what we've tested.

4.7 Measuring Test Coverage

Benchmarking the coverage of our testing effort can provide us with clues to potential security vulnerabilities. To do this, we'll need tools beyond those included with a standard Rails application.

Rcov and its corresponding plug-in `rails_rcov` are extremely powerful tools we can use to measure code coverage. The standard Rails application gives us the *quantity* of test code using the `stats` Rake task. This metric is interesting, but it tells us only how many lines of code we've written for our test cases. It says nothing about the *quality* of those test

cases. Rcov gives us insight into which lines of code are never executed during our test process. If we don't have a test case for a particular part of our application, we can assume only that it *might* be a potential security vulnerability!

Installing Rcov

Execute the following commands to install the Rcov gem and corresponding Rails plug-in:

```
testing> gem install rcov
testing> ruby script/plugin install http://svn.codahale.com/rails_rcov
```

Let's report the code coverage of LunchedIn using Rcov. Execute the following command to build the coverage reports:

```
testing> rake test:units:rcov SHOW_ONLY=m
testing> rake test:functionals:rcov SHOW_ONLY=c
```

The Rcov tasks execute the test suite and then generate an HTML code coverage report under the directory `coverage`.² To view the report, open the `index.html` file under the directory `coverage/test`. We can use the `SHOW_ONLY=m` and `SHOW_ONLY=c` parameters to filter test coverage only by *models* and *controllers*.

The `index.html` file contains a summary of each file analyzed. The report shows the source and highlights lines that were not executed. We can see this at work in Figure 4.2, on the following page. Rcov tells us our test suite has not exercised the failure condition of the `check_login()` of the `SessionController` class.

Rcov helps us find untested areas of our code. It doesn't prove that our tests are good, but it does give us a baseline of what has been executed in a set of test cases.

4.8 Other Testing Frameworks

The Rails community has blessed us with a multitude of testing frameworks to help us achieve our goal of building well-tested software. One such framework, Tarantula, provides an implementation of a technique called *fuzz testing*.

Fuzz testing generates random and invalid data in attempt to find flaws within an application. Tarantula incorporates this technique, as well as

2. To prevent the generated report from being committed alongside your project in Subversion, execute `svn propedit svn:ignore coverage` from the root of your project directory.

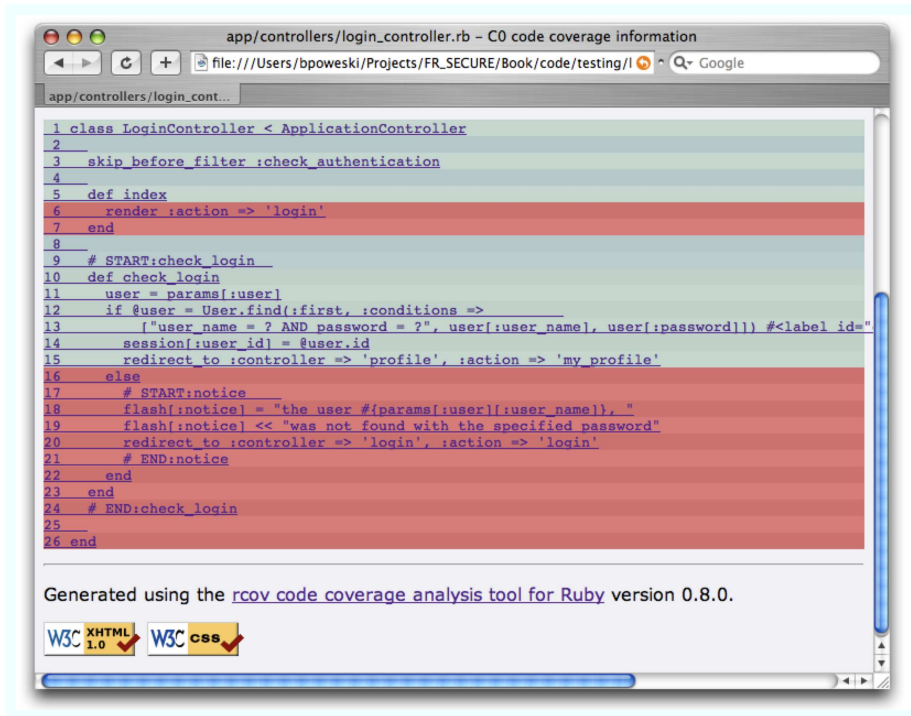


Figure 4.2: Rcov code coverage report

tests for common security exploits such as XSS and SQL injection. In essence, Tarantula is a spider that integrates into your Rails application and explores everything, including the paths less traveled. You can acquire Tarantula at <http://github.com/relevance/tarantula>.

Behavior-driven development (BDD) is another testing strategy you can apply to useful for testing security concerns. RSpec is the original BDD testing framework and has a vibrant community. RSpec can be found at <http://rspec.info/>.

Given the choice of improving our code or writing documents about our code, we'd choose improving our code every time. Rails provides world-class frameworks and tools useful for building and executing test cases, and it's a good idea to use them. Keep in mind that your application testing is only as good as your test cases!

Mastering the art of building quality test suites can reduce the defect rate of your software tremendously. This applies to defects related to both the domain and the security aspects of your application.

Chapter 5

Validation

Validation is the process of accepting, rejecting, or sanitizing user input or server output. Sometimes people refer to validation separately as input validation, data validation, or other variations based on the specificity of the validation. In this chapter, we will focus on validation as an activity that helps us prevent harm from input or output data.

When software developers initially started creating websites that contained more than static HTML content, little consideration was given to the potential security implications of exposing databases and OS filesystems to the public Internet. The first web applications were nothing more than Perl or C programs that listened for inbound traffic and responded based on the parameters provided by the user. Very quickly, security folks became aware of the dangers inherent from trusting input from the outside world. It seemed that the developers of the world were more concerned with getting the shopping cart interface perfect for those Beanie Babies that were selling like hot cakes. Even today, many developers focus on cool features and deadlines, and things like validation are secondary.

And while we should deliver the coolest software we can, we must never forget about these critical rules for input and output:

- Don't trust data from your users.
- Don't trust data from other systems.
- Validate all data passing to/from a trust boundary.
- Return safe data when possible.

These rules haven't changed since the very first line of code ever written, and they won't change any time soon. Fundamentally, validation is the first line of defense for all software, whether it's a C/C++ function that

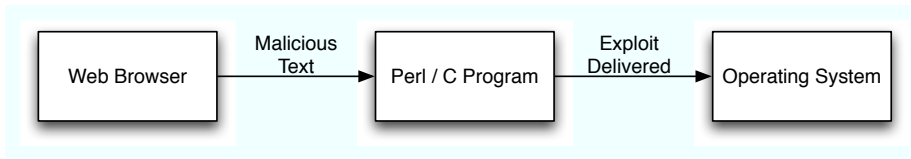


Figure 5.1: Simple, old-school attack

handles string data, or it's the latest Ajax control that sorts your friends by how much money they spent on you for your birthday.

Let's take a look at the various parts of our Rails applications where we need to validate data.

5.1 Command Injection

Injection is the underlying principle behind these attacks. Injection flaws exist in parts of a web application where input from a user or system can be passed to an underlying API or system call, as demonstrated in Figure 5.1. Let's take a look at an example of an injection flaw:

[Download](#) input_validation/lunched_in/app/controllers/root_controller.rb

```

def system_command_run
  index
  if params[:command]
    system("ls #{params[:command]} > __tmp.file")
    file = File.open("__tmp.file")
    @text = file.read
    File.delete("__tmp.file")
  end
end
end

```

end

[Download](#) input_validation/lunched_in/app/views/root/system_command_run.rhtml

```

<div class="welcome-tags">
  <% for tag in @tags %>
    <%= link_to(tag.name, :controller => 'venues', :action => 'tag',
      :id => tag.name) %>
  <% end %>
</div>
<br/><br/><div id="bad_ways"><pre><%= @text %></pre></div>

```

W/ Joe Asks...

Why Would You Call a System Command Based on User Input?

The code in our methods with injection flaws might appear a little bit out of place, but it is not uncommon for an application to support interaction with the system. At the same time, this sort of design is one of the most heavily exploited types of web application vulnerabilities. For a sense of how heavily exploited this is, visit Google and enter this search: `cmd.exe + web application vulnerabilities`. Often, we need to access the underlying filesystem or call a legacy script to support our web application. However, take extra care when dealing with such calls to the system.

In this example, we accept a parameter from the user called `:command`. This parameter is used as the directory path for an `ls` command.

Once we execute this action with a system command—a parameter, for example—the results of the command are placed in `__tmp.file` and inserted into `@text`. The results are then displayed by the corresponding page template. These results should look something like Figure 5.2, on the following page, where we executed `ps aux`. This is a trivial example, but we could easily execute any known vulnerability for a particular platform because we have full command-shell access to the system.

While we might not be accessing system calls in most of our web applications, this example is important because we need to appreciate that the possibility for injection vulnerabilities exist wherever we accept input from a user or system. This example also demonstrates an additional reason why the principle of Defense in Depth is so important, as discussed in Chapter 1, *Security in Ruby on Rails*, on page 10. If we force our web application to run under a user account that has no command-shell or limited command-shell access, this particular type of vulnerability becomes harder to exploit.

Have Lunch with Lunched In

Find somewhere to eat

Know somewhere great to eat?

fajitas tacos margaritas flan
martinis steak filet lobster

Awesome!!!

Make sure you try the cre

wally wrote about [Bob's](#)!

Fantastic!

The steaks are fantastic a

wally wrote about [Bob's](#)!

USER	PID	%CPU	%MEM	VSZ	RSS	TT	STAT	STARTED	TIME (
draphael	389	23.4	0.9	435544	19040	??	S	Sat05PM	0:37.42
draphael	1503	20.9	5.0	575144	104520	??	R	Sun09PM	31:54.93
draphael	3855	12.8	8.2	629080	172000	??	U	7:54PM	18:22.12
draphael	5606	8.5	1.6	635520	32972	s000	S+	10:59PM	0:03.63
draphael	3903	5.0	2.0	477016	41628	??	S	9:01PM	8:02.86
_windowsserver	259	3.6	6.1	1063224	128420	??	Ss	Sat05PM	36:4
root	267	1.0	0.4	85184	7420	??	Ss	Sat05PM	1:08.82
draphael	3805	0.6	1.8	518644	36920	??	U	6:19PM	3:37.86
root	65	0.2	4.4	813508	92704	??	Ss	Sat05PM	6:53.84
root	11	0.1	0.3	603436	5628	??	Ss	Sat05PM	0:45.54
_mysql	234	0.1	0.2	634632	4324	??	S	Sat05PM	2:18.01
draphael	3809	0.0	0.6	624120	12764	??	Ss	6:20PM	0:02.57
draphael	3804	0.0	3.6	496252	74592	??	S	6:19PM	1:24.94
draphael	1451	0.0	0.6	403228	11876	??	S	Sun09PM	0:33.88
draphael	605	0.0	0.0	600680	880	??	S	Sat06PM	0:00.01
draphael	582	0.0	0.0	600252	980	s004	S	Sat06PM	0:00.06

Figure 5.2: Command injection

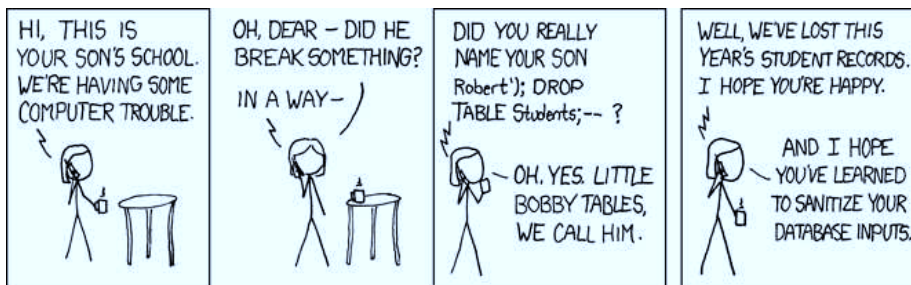


Figure 5.3: Exploits of a mom, xkcd.com

5.2 SQL Injection

It's time to start the music from *Jaws*. The phrase SQL injection is probably the most pervasive security term in the web application security world. No matter how many times this vulnerability is explained, SQL injection consistently makes it to the top of the list of web application vulnerabilities. In Figure 5.3, on the preceding page, we see a possible scenario depicted in Randall Munroe's comic, *xkcd*.

SQL injection is the process of manipulating data passed to a web application in a way that changes the result of a SQL command. Let's start with a simple SQL statement that queries a user database using a username and password:

```
select username, email, first_name, last_name,\
  role from users where username = "dave" and \
  password = "let_me_in"
```

The natural tendency is to do something like the following:

```
sql_query = 'select username, email, first_name,\
  last_name, role from users where username = \
  "#{username}" and password = "#{password}"'
```

The problem lies in the fact that we are passing input from an untrusted source, unabated to the database server. Consider the following scenario:

```
▶ username = params[:username] # => '" or 1=1; --'
password = password[:password] # => 'hacking you!'
sql_query =
  'select username, email, first_name,\
    last_name, role from users where \
    username = "#{username}" and \
    password = "#{password}"' # => \
  'select username, email, first_name,\
    last_name, role from users where \
  ▶ username = "" or 1=1; -- and \
    password = "hacking you!"'
```

Our malicious user wrote some nasty SQL code that terminated our username clause, appended an `or 1=1`, and added `--` to terminate our SQL statement prematurely. The result: this returns every user row in our database! We demonstrated an exploit like this early in Chapter 2, *Hacking the Example*, on page 23.

We can derive literally dozens of variations on this attack. Fundamentally, the flaw is improper input-string handling. The fundamental

Joe Asks...

What About Unicode Character Sets?

In Rails, there is support for internationalized characters from the methods added to String by ActiveSupport::Multibyte. This means that regular expressions and other techniques for detecting malicious characters should work as expected. However, if you are using a database that isn't well supported in Rails, you will need to ensure that any multibyte characters are handled correctly by the connection.

structure of a SQL injection attack is very simple, but preventing SQL-injection attacks can be quite challenging.

So how do we mitigate this vulnerability once and for all?

5.3 Mitigating SQL Injection

Let's take a look at what characters are dangerous for SQL query construction:

Character	Typical Meaning
'	Single quote: Typically, this initiates or terminates a string parameter for a SQL query.
"	Double quote: Such quotes are often interchangeable with a single quote. You should treat them similarly from the perspective of SQL injection.
—	Double dash: This introduces a SQL comment.
;	Semicolon: This character terminates a SQL statement.
\	Backslash: This character typically escapes the next character.

At first glance, mitigating attacks based on the exploitation of these special characters seems like a fairly straightforward task. We could just write the following regular expression that replaces evil characters with asterisks:

[Download](#) `input_validation/lunched_in/test/unit/user_test.rb`

```
# This is an example of what not to do.
input = '" or 1=1; --\''
input.gsub(/;|--'|"/"\|\\/, "*") # => "* or 1=1* ****"
```

In this code, we are substituting a `*` for any of the characters we don't like. We could just as easily have removed them from the string altogether. This is a blacklist technique. While this might seem like a reasonable approach at first glance, it is overkill for most situations. In fact, it is both overkill and fragile. A better approach would be to refactor this into a whitelist:

Download `input_validation/lunched_in/test/unit/user_test.rb`

```
# This is an example of what not to do.
input = "' or 1=1; --'"
input.gsub(/[^a-zA-Z0-9= ]/, "*") # => "* or 1=1* *****"
```

With this whitelist, we substitute any unsafe characters with a `*` as well. The difference here is that we are following our principle of whitelisting. And while we should be proud of our principled development, we've also essentially blocked anyone with a name that could be dangerous, such as O'Reilly:

Download `input_validation/lunched_in/test/unit/user_test.rb`

```
# This is an example of what not to do.
input = "O'Reilly"
input.gsub(/[^a-zA-Z0-9= ]/, "*") # => "O*Reilly"
```

And this code illustrates, O'Reilly is not a valid name! That's not very fair to the Irish, and it also doesn't account for hyphenated names. What if a Bush married a Clinton and took the name Clinton-Bush? That wouldn't work with this code, either. Let's take a look at another method:

Download `input_validation/lunched_in/test/unit/user_test.rb`

```
# This is a better example, but not perfect.
input = "O'Reilly"
input.gsub(/\'/, "'") # => "O'Reilly"
```

Now we are getting somewhere. In this example, we are not really whitelisting or blacklisting. Instead, we are sanitizing the data so it is safe for SQL. These characters are not intrinsically evil; rather, they simply require that we escape them. Now it is a simple matter of using the rules of SQL for escaping our quotes. But what about other quotes? For example, assume that I want to use the following signature: *O'Reilly, the "Daredevil."* Let's take a look at how to handle this:

Download `input_validation/lunched_in/test/unit/user_test.rb`

```
# This is an example of what to do.
input = "O'Reilly, the \"Daredevil.\""
input.gsub(/\\/, '\&\&').gsub(/'/, "'") # => "O'Reilly, the \"Daredevil.\""

```

We've made some significant progress; next, let's put this new method to a more malicious test:

```
Download input_validation/lunched_in/test/unit/user_test.rb
```

```
# This is an example of what to do.
input = '" or 1=1; --\''
input.gsub(/\\/, '\&\&').gsub(/'/, "'") # => "\" or 1=1; --\\\""
```

This looks pretty good. In fact, this is the basic code that Rails uses to protect its ActiveRecord users from string-based attacks. However, this doesn't protect us from non-string-based parameters. In fact, many SQL injection vulnerabilities are a result of SQL statements that don't use strings. In many cases, we construct a query based on a numeric value, as in this example:

```
Download input_validation/lunched_in/test/unit/user_test.rb
```

```
# This is an example of what not to do.
input = "6"
sql_statement = "select * from foo where id = #{input}"
puts sql_statement # >> "select * from foo where id = 6"
```

Well, now we have a problem: a SQL injection attack can occur without any *malicious* characters. Let's look at such an attack:

```
Download input_validation/lunched_in/test/unit/user_test.rb
```

```
# This is an example of what not to do.
input = "6 or 1=1"
sql_statement = "select * from foo where id = #{input}"
puts sql_statement # >> "select * from foo where id = 6 or 1=1"
```

This is a new problem. Here, the attacker is exploiting the fact that the input is passing from the user interface as a string—as all HTML form data does. Let's take a look at a possible fix:

```
Download input_validation/lunched_in/test/unit/user_test.rb
```

```
# This is a better example, but not perfect.
input = "6 or 1=1"
sql_statement = "select * from foo where id = #{input.to_i}"
puts sql_statement # >> select * from foo where id = 6
```

By invoking `to_i()` on the input, we are forcing the type of integer. We know what kind of data we are dealing with in our SQL database, so we can force the cast. In Ruby, this does a nice conversion, but there are cases where this won't have a desirable behavior. Ruby will default to 0 for all non-numeric strings. However, this is a reasonable behavior that counters this form of injection attack.

The little bag of goodies we've built up already exists in Rails. In fact, we learned about a quick fix using bind parameters in Chapter 3, *Fixing the Example*, on page 49. However, let's dive a little deeper and look at how we can leverage the various types of SQL injection prevention in Rails.

5.4 Built-in SQL Injection Mitigation in Rails

When we leverage ActiveRecord for our database access, we get several useful features:

- Character escaping
- Bind parameters for SQL statement construction
- Named bind parameters for SQL statement construction

In our earlier examples, we came up with a reasonable solution for preventing string and numeric SQL injection attacks. Rails has already implemented this in the ActiveRecord::ConnectionAdapters::Quoting module. In this module, Rails implements the `quote_string(s)` method, which performs the regular expression we built in Section 5.3, *Mitigating SQL Injection*, on page 89. Let's take a look at the default `quote_string(s)` method in the Rails source:

[Download](#) `input_validation/lunched_in/doc/quoting.rb`

```
# Quotes a string, escaping any ' (single quote) and \ (backslash)
# characters.
def quote_string(s)
  s.gsub(/\\/, '\\&\\').gsub(/'/, "'") # ' (for ruby-mode)
end
```

Next, let's look at the DB2 implementation:

[Download](#) `input_validation/lunched_in/doc/quoting.rb`

```
def quote_string(string)
  string.gsub(/'/, "'") # ' (for ruby-mode)
end
```

Not all databases treat data the same way, and the different implementations out there can undermine the methods you use to handle input sanitization.

We can leverage the ActiveRecord API directly by invoking this code:

```
Download input_validation/lunched_in/test/unit/user_test.rb
```

```
username = 'dave " or 1=1; --\''
q_username = User.quote_value(username)
assert(q_username == "'dave \\'" or 1=1; --\''\''\''\''\''\''\''')
```

The result is a safe String that we can use with our database query.

This solution covers only basic string handling. What about integers? Floats? DateTime? Well, we could write more code to handle these cases, as well. Or we could leverage the API in a more thorough manner by using bind variables.

Many security articles refer to the use of stored procedures as the best way to mitigate SQL injection. This is simply not true. Stored procedures are vulnerable to string interpolation. If it often appears that they are not, it is probably because the APIs are being used correctly within the stored procedure. In this case, protection is provided by bind variables. We should follow the same technique here that we see in ActiveRecord and its use of bind variables. Let's look at a simple example:

```
Download input_validation/lunched_in/test/unit/user_test.rb
```

```
username = 'dave " or 1=1; --\''
password = 'letmein'
User.find(:all, :conditions => ["user_name = ?", username])
```

Using the `?` constant in our `:condition` variable enables us to create a *placeholder* for ActiveRecord. This placeholder acts as an anchor for the variables that are passed as the second part of the condition array. We can also use named bind variables for better readability:

```
Download input_validation/lunched_in/test/unit/user_test.rb
```

```
username = 'dave " or 1=1; --\''
password = 'letmein'
bind_vars = {:username => username, :password => 'letmein'}
User.find(:all, :conditions =>
  ["user_name = :username and password = :password", bind_vars])
```

Both strategies leverage the underlying mechanism:

[Download](#) input_validation/lunched_in/doc/quoting.rb

```
# From Rails source...
def quote(value, column = nil)
  # records are quoted as their primary key
  return value.quoted_id if value.respond_to?(:quoted_id)

  case value
  when String, ActiveSupport::Multibyte::Chars
    value = value.to_s
    if column && column.type ==
      :binary && column.class.respond_to?(:string_to_binary)
      "'#{quote_string(column.class.string_to_binary(value))}'"
      # ' (for ruby-mode)
    elsif column && [:integer, :float].include?(column.type)
      value = column.type == :integer ? value.to_i : value.to_f
      value.to_s
    else
      "'#{quote_string(value)}'" # ' (for ruby-mode)
    end
  when NilClass          then "NULL"
  when TrueClass         then (column && column.type == :integer ? '1' : quoted_true)
  when FalseClass        then (column && column.type == :integer ? '0' : quoted_false)
  when Float, Fixnum, Bignum then value.to_s
  # BigDecimals need to be output in a non-normalized form and quoted.
  when BigDecimal        then value.to_s('F')
  when Date               then "'#{value.to_s}'"
  when Time, DateTime     then "'#{quoted_date(value)}'"
  else                    "'#{quote_string(value.to_yaml)}'"
  end
end
```

These values might differ, depending on the database we are leveraging, so it's always a good idea to look at the Rails source if you are unsure how a particular database query works.

*Let me in, let me in, or I'll huff and I'll puff and I'll blow
your house down!*

► Big Bad Wolf

Chapter 6

Authentication: Decentralized Authentication

In our examples so far, we've gathered the username and password directly from the user. And in doing so, we've implemented *decentralized authentication*. It is considered *decentralized* because we don't rely on any additional infrastructure to authenticate our users. In other words, we are our own little island.

We'll learn how to delegate the responsibility of gathering the username and password to other systems in Chapter 10, *SSO: Centralized Authentication*, on page 191. These systems, which include OpenID, Kerberos, or CAS, will all rely on *centralized authentication* mechanisms.

For now, let's focus on decentralized authentication. Along the way, we'll improve the robustness of our decentralized authentication by improving the robustness of our existing user database, adding support for LDAP, and improving our architecture to support LDAP or our application's database. Let's get started by reviewing some key concepts.

6.1 Authentication Concepts

We need to make sure to keep the difference between authentication and authorization clear. *Authentication* is the process of identifying and verifying a principal. We'll see what a principal is in just a few paragraphs. For now, just think of principal as a *user*.

Authorization is the process of allowing or denying a principal access to a target resource or action. Authorization is also known as *access control*. Chapter 7, *Authorization*, on page 135, covers authorization in depth.

We need to keep our authentication code separate from our authorization code, as a general rule. We will see some of the benefits of this separation when we explore supporting multiple authentication mechanisms in Section 6.4, *Flexible Authentication*, on page 122.

Users, Services, and Principals

Users are not the only entities that must prove their identities. In fact, our Rails application connects to a database using a username and password. However, our Rails application itself isn't really a user. The Rails application is a *service*. Nonhuman accounts are typically referred to as *service accounts*.

In `database.yml`, the username for the Rails application grants access to an account configured specifically for that application. This is a service account for the database.

Our Rails application runs with a service account, such as `apache` or some other account designated for running our website. `apache` is a service account for the operating system.

In the Windows family of operating systems, service accounts are different than user accounts. In the Unix family, they are accounts that are set up without the ability to be used by a human interactively, such as in a terminal session. At least, we hope that nobody logs in as `apache`.

You should try interchanging the word *user* or *service* with the word *principal* when you are thinking about authentication. It proves an especially good habit to form when dealing with more complex authentication frameworks and libraries that refer to principals instead of users or services.

Identification, Credentials, and Factors

Imagine we want to purchase something with our credit card at a store. When we present our credit card to the clerk, the clerk asks for proof of our identity.

So we hand our photo ID to the clerk, and the clerk does the following:

1. She compares the name on the ID to the name on the card.

At this point, she believes that the ID credential is correct for the credit card.

Let's call this the first *factor*.

2. She compares your face with the photo on the ID.

And finally, because you've aged gracefully since your last ID photo was taken, the clerk believes that you are the person in the picture.

Let's call this the second *factor*.

If the credit card also had a photo, that would have been one more factor to strengthen the clerk's belief in the authenticity of the principal.

You probably noticed that we introduced the term *factor*. A factor is simply a piece of information provided by a principal to prove identity. Credentials are one or more factors provided by a principal.

In this example, the clerk didn't question the credentials too much because high-quality factors were used. To build strong authentication systems, we must look at the quality of factors that our principal provides:

- Any factor that a principal *is* or *does*—such as DNA or voice samples—is extremely unlikely to collide with someone else's factor of the same kind. That makes it very difficult or impossible to impersonate this kind of factor. Factors of this type are very high-quality, but they aren't practical for a lot of systems. These are known as *biometric* factors.
- Any factor that a principal *has*, such as a security token device or email account, provides out-of-band proof. Such proofs *must* be coupled with another factor. These are medium-quality factors. Let's call these *out-of-band* factors.
- Typically, the user sets any factor that a principal *knows*, such as a PIN or password. This is a weak factor, but it is the most usable and pragmatic type of factor. We'll call these *secret* factors.

We need to choose our factors based on the sensitivity of our data. Sometimes, it's only necessary to gather a username and password from a user. Other times, we might send a text message to the user's phone with a verification code or physically verify some information with a third party.

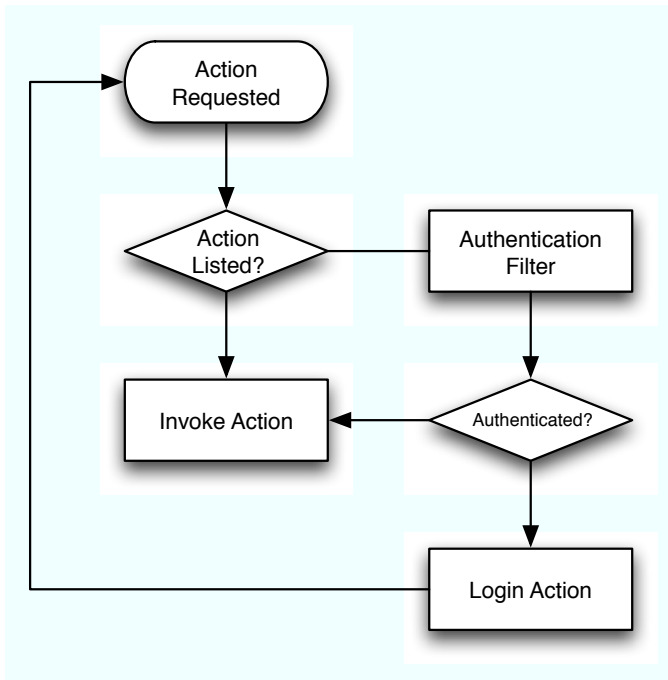


Figure 6.1: Basic authentication flow in Rails

Sessions and Contexts

Of course, all of this effort is meaningless if we can't secure our dialog with our authenticated principal. Such dialogs are referred to as *sessions* or *contexts*. Just as we use names to identify principals, we need to be able to refer to dialogs through some sort of handle.

SSL primarily provides the ability to protect sessions on the Web. It provides a suite of protection mechanisms that drastically reduce the likelihood of an attacker gaining access to the session. But let's not get too comfortable just yet.

SSL provides protection against eavesdroppers, but it doesn't really help us with authentication. SSL will gladly transport an attacker who has guessed a password correctly.

6.2 User Registration

Before we can authenticate users in our application, the users must exist in our database! LunchedIn allows anyone with a pulse to sign up for the service. After all, it's all about user acquisition! Maybe one day we'll be able to charge for our LunchedIn service, but for now it's available to anyone who wants to try it.

LunchedIn already supports user registrations. Let's see how we can mitigate some of the typical vulnerabilities associated with a new user registration.

The only requirement of a user to sign up for LunchedIn is a *pulse*, so how do we verify that they are human?

CAPTCHA

CAPTCHA stands for *Completely Automated Public Turing test to tell Computers and Humans Apart*. I prefer this definition: *those weird and distorted pictures of words that I have to type in for Google and Yahoo sites*.

Wow! That sounds like a perfect match for our “user needs a pulse” requirement. Implementing a CAPTCHA from scratch is not something you should try, unless you are just doing it for fun. Let's go ahead and install a *simple_captcha*¹ Rails plug-in for our application.

Simple Captcha Installation

First, we'll need RMagick.² This tool is a Ruby interface to the ImageMagick image-processing library. Simple Captcha uses RMagick to generate the CAPTCHA images. RMagick is available as a gem:

```
> gem install rmagick > ruby script/plugin install \
  svn://rubyforge.org/var/svn/expressica/plugins/simple_captcha
```

Next, we set up our application for Simple Captcha:

```
> rake simple_captcha:setup
> rake db:migrate
> rake db:test:clone
```

1. Sur Max created and maintains the popular *simple_captcha* plug-in at <http://expressica.com>.

2. You can download RMagick at <http://rmagick.rubyforge.org>.

RMagick Is a Pain to Install!

The installation challenges of a particular dependency aren't usually worth commenting on, but RMagick can be a challenging library to install. Be sure to search the Web for the latest information on building RMagick for your particular platform.

At this point, we can add the necessary route in routes.rb:

Download authentication/lunchedin_captcha/config/routes.rb

```
map.simple_captcha '/simple_captcha/:action', :controller => 'simple_captcha'
map.connect '/login', :controller => 'session', :action => 'login'
map.connect '/logout', :controller => 'session', :action => 'logout'
```

Finally, we need to include the SimpleCaptcha::ControllerHelpers in our ApplicationController:

Download authentication/lunchedin_captcha/app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  include SimpleCaptcha::ControllerHelpers
  helper :all # include all helpers, all the time
  protect_from_forgery # :secret => 'cf9d327f2d16793b606c6689f76e90c3'

  before_filter :check_authentication, :except => [ :logout ]

  def check_authentication
    if session[:user_id]
      begin
        @authenticated_user = User.find(session[:user_id])
        return
      rescue
        reset_session
        redirect_to :controller => "session" and return
      end
    end
    redirect_to :controller => "session"
  end
end
```

Adding CAPTCHA to LunchedIn's Registration

The Simple Captcha plug-in's integrated helpers support a controller or model-oriented approach. The model-oriented approach keeps the domain logic out of our controllers.

Let's begin by adding the `apply_simple_captcha()` method call to the User model.

Download authentication/lunchedin_captcha/app/models/user.rb

```

▶ apply_simple_captcha
def password=(password)
  @password = password
  write_attribute(:password_hash,
    OpenSSL::Digest::SHA1.new(password).hexdigest)
end

```

The `apply_simple_captcha()` method call adds the following instance methods to our User model:

- `save_with_captcha()`
- `valid_with_captcha?()`

Let's use the `save_with_captcha()` method in order to save the model to `UsersController`:

Download authentication/lunchedin_captcha/app/controllers/users_controller.rb

```

def create
  @user = User.new(params[:user])
  respond_to do |format|
    ▶ if @user.save_with_captcha
      flash[:notice] = 'your account was successfully created.'
      session[:user_id] = @user.id
      format.html { redirect_to(@user) }
    else
      format.html { render :action => 'new' }
    end
  end
end
end

```

The `save_with_captcha()` method ensures that a correctly matching CAPTCHA is validated before saving the user. Next, let's add the `show_simple_captcha` view helper to the User form:

Download authentication/lunchedin_captcha/app/views/users/new.html.erb

```

<tr>
  <td class="label">Zip Code:</td>
  <td><%= text_field('user', 'zip_code', :size => 6) %></td>
</tr>
<tr>
  <td class="label">Are you human?</td>
  <td><%= show_simple_captcha(:object => "user",
                             :distortion => 'high',
                             :image_style => 'random') %>
  </td>
</tr>
</table>
</div>

```

The screenshot shows a web browser window titled "Lunched In: because Lunch is the most important meeting" with the URL "http://localhost:3000/users/new". The page features the LunchedIn logo and navigation links for "Have Lunch" and "Create A Profile". Below these is a "Have lunch with us" section and a "Create A Profile" form. The form includes input fields for First Name, Last Name, Email, User Name, Password, Confirm Password, and Zip Code. A CAPTCHA image displays the code "BATCYI" with the instruction "(type the code from the image)". A "Create" button is located at the bottom of the form. The footer of the page contains the copyright notice "© 2008".

Figure 6.2: New user form with CAPTCHA

```
<div style="padding: 10px;">
<%= submit_tag 'Create' %>
</div>
<% end %>
```

At this point, we have a CAPTCHA-enabled LunchedIn. If we navigate to our registration page, we should see the CAPTCHA. The result is shown in Figure 6.2.

Adding Salts to the Passwords

Keeping a password secret is the top priority of any password storage mechanism. As a security precaution, we don't even store the password in LunchedIn's database; instead, we store a *digest* of the password. We



Joe Asks...

Why Do We Need So Much Password Security?

You might be asking yourself, “Why does it matter if the users’ passwords are so hard to break? A hacker has to figure out only how to get in the database. Once he is in, he can read whatever data he pleases.” This is true for data that is stored in plaintext in the database; however, we will learn how to use password-based encryption to protect user privacy in Chapter 8, *Data Protection Using Cryptography*, on page 155. Even if the attacker has access to the database in this scenario, he would then have to break each individual user’s encryption to gain access to private data.

only need to verify that the user has provided a correct password. The digest acts like a fingerprint of the password.

We don’t store the password directly, but there is still a threat: dictionary-based attacks can discover the value of a password from its digest. Let’s take a look at how this attack works. For our example, the passwords have a maximum length of 1 and must be an English vowel: *a*, *e*, *i*, *o*, or *u*.

The following table is a dictionary containing all of the single-letter vowels and their digests.³

Password Digest

a	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
e	58e6b3a414a1e090dfc6029add0f3555ccba127f
i	042dc4512fa3d391c5170cf3aa61e6a638f84342
o	7a81af3e591ac713f81ea1efe93dcf36157d8376
u	51e69892ab49df85c6230ccc57f8e1d1606cacc

We can determine the password by looking up the password’s digest as the key. So how do we make this harder for attackers? We *salt* the password. A salt is a small amount of random data that is included in the digest calculation. Let’s look at what happens to that same dictionary

3. If this were a real dictionary, there would be hundreds of thousands of entries.

if we also include the precalculated values with salts. For this example, we will limit our possible salt values to 0, 1, 2, and 3:

Password	Salt	Digest
a	0	40b25eac438260c9ad4e3142adc38a8d0885e5f3
a	1	f29bc91bbdab169fc0c0a326965953d11c7dff83
a	2	b9f85daa6f83cf02ce5c31913d1f64d3f5c8fade
a	3	252bc06763afb3b6c2a0802f7346700ab55f46f5
e	0	8b5eaccb28f2182b656fc1a2589dd64891df3a08
e	1	84896d3e067884621c0f54334b8d840949665844
e	2	dc1885915a8902bc29de6d834477a26d748b60b8
e	3	0e647f28da2af6b6f4a51e217990dd1cd4f66124
i	0	3553fbf24132495ad14b3c090956d321c8c68490
i	1	3795b54c5ba62df52f7f5132a3c17a2191fc7f74
i	2	ce4e91a86af472f2ee56d8ba8a3a2530ae3c9dd4
i	3	c5f511a6b910604bf76b6450f64e404cfa24c440
o	0	3c72690938e0e42ad8dc15f563c27bf1e0625d64
o	1	a473a8c1ab29e55e40e14f2c8d0fe078ad9920ad
o	2	0f927dce74ec3475c7a39299e5bffb222ca665d
o	3	d099916d553b31fdbfa04885af9355bce9757722
u	0	a165438a9e9fdbbb6b0354143903f5830e77e316
u	1	c5ea71554c774daf7fab320fc3476afc0617eb00
u	2	b9c69e5488ff88b92096f1137382e67fe1172fa6
u	3	e2e5f0fd9054a323948762af20580a9403523e36

As you can see, the size of our table grew to four times its original size. If we apply that same logic to a dictionary of 50,000 words with 8-byte salts, we get a dictionary with 893,905,167,390,644,531,250,000 entries. This makes for a much more difficult dictionary attack.

Now that we have reviewed the theory, let's put salts to use in our LunchedIn app.

Adding Salts to LunchedIn

Adding salt to our passwords in LunchedIn requires that we accomplish several tasks:

- Create and store the salts
- Modify the authentication code to support the salts
- Deal with existing unsalted user passwords
- Rerun unit tests to ensure that we haven't broken anything

In our initial example, we will store the salt in a separate column of our User model. We will take a different approach, integrating with a system that stores the salt as part of the password digest string, in Section 6.3, *LDAP/Directory-Based Authentication*, on page 107. But for now, we will enjoy the readability of separate data.

Now let's add the salt column:

Download authentication/lunchedin/db/migrate/011_add_salt_to_user.rb

```
def self.up
  add_column :users, :salt, :text
end
```

We should probably make this migration irreversible, as well:

Download authentication/lunchedin/db/migrate/011_add_salt_to_user.rb

```
def self.down
  raise IrreversibleMigration "Salts are a one way road."
end
```

Next, let's add some code to the User class:

Download authentication/lunchedin/app/models/user.rb

```
def hashed_and_salted_password(p, s)
  OpenSSL::Digest::SHA1.new([s,p].to_s).hexdigest
end
```

The `hashed_and_salted_password()` method calculates a SHA1 digest of the password with the salt data prepended. Does it matter where the salt is placed in the calculation? Nope—we just need to be consistent in how we do this across our applications.

Now we need to make sure our password persistence works smoothly with the rest of our application. Add the following code to the `password=()` method:

Download authentication/lunchedin/app/models/user.rb

```
Line 1 def password=(password)
      2   @password = password
      3   write_attribute(:salt,
      4     OpenSSL::Digest::SHA1.new(
      5       OpenSSL::Random.random_bytes(256)).hexdigest)
      6   write_attribute(:password_hash,
      7     hashed_and_salted_password(password, salt))
      8 end
```

On line 2, this code stores the plaintext password in the `@password` attribute. This means we do not break our `validates_confirmation_of`

validation, which expects a plaintext password to be present for confirmation. Next, we generate a salt to be used in the password digest on line 7. Finally, we generate a digest using a SHA1 hash function computed from the salt+password on line 7.

Don't worry, we're getting there. Next, we modify our `authentic?()` method so it supports the salt:

[Download](#) authentication/lunchedin/app/models/user.rb

```
def authentic?(p)
  password_hash ==
    hashed_and_salted_password(p,salt)
end
```

Now let's add a fixture to `Users.yml` that uses a salt. We will leave our existing fixtures to ensure that we test passwords without salts, as well:

[Download](#) authentication/lunchedin/test/fixtures/users.yml

```
# With salt:
salty:
  id: 5
  first_name: Wally
  last_name: Salty
  username: salty
  salt: salt
  password_hash: <%= OpenSSL::Digest::SHA1.new(
    ["salt", "password"].to_s).hexdigest %>
  email: bpoweski@yahoo.com
  role_id: 1
  zip_code: 75024
  created_at: 2007-11-27 14:18:26
  updated_at: 2007-11-27 14:18:26

# Without salt:
admin:
  id: 4
  first_name: The
  last_name: Admin
  username: admin
  password_hash: <%= OpenSSL::Digest::SHA1.new("admin1").hexdigest %>
  email: bpoweski@yahoo.com
  role_id: 1
  zip_code: 75024
  created_at: 2007-11-27 14:18:26
  updated_at: 2007-11-27 14:18:26
```

The next step is to add a positive test for our new salt functionality.

Download authentication/lunchedin/test/unit/user_test.rb

```
def test_user_with_salt
  @user = users(:salty)
  assert @user.authentic?('password')
end
```

We also want to add a negative test:

Download authentication/lunchedin/test/unit/user_test.rb

```
def test_user_fail_with_salt
  @user = users(:salty)
  assert !@user.authentic?('wrong_password')
end
```

To be thorough, let's also add a find and authenticate test method:

Download authentication/lunchedin/test/unit/user_test.rb

```
def test_find_and_authenticate_with_salt_success
  assert User.find_and_authenticate('salty', 'password')
end
```

At this point, we need to rerun our unit tests and make sure we haven't broken anything.

This code should not create any failures. We have preserved our test logic, and our changes did nothing more than rework the password-handling functionality of LunchedIn. Additionally, the rest of the application's functionality should be intact. For example, user registrations will continue working without a hitch.

6.3 LDAP/Directory-Based Authentication

Let's imagine for a moment that we've been asked to integrate LunchedIn into a company that has a sales force of 5,000. LunchedIn will be the internal lunch-planning tool for this company's salespeople. We'll call our fictional company *SellIt During Lunch*. Let's begin by imagining a fictional sales dialogue with the SellIt During Lunch management rep, Joe, and our LunchedIn sales rep, Dave:

Dave: *We are very happy that you're considering the purchase of a corporate edition of LunchedIn. Do you have any questions?*

Joe: *How do you control access to the software?*

Dave: *We have a very user-friendly registration page. Your users will create an account with a username and password. . .*

Joe: Wait! Are you saying that users will have to remember yet another password? That is not going to work. Our salespeople don't remember their own names very well. If we're going to buy LunchedIn, it will have to integrate with our existing LDAP-compliant directory.⁴

Dave: I will have to go talk to our developers about this. Let me get back to you.

Joe: Sounds good!

The dialogue between Joe and Dave touches on the fact that most companies don't want their users to have to remember multiple passwords. In fact, most users don't want to remember multiple passwords. Dialogues like this drive many products to support various alternative authentication mechanisms.

Most companies deploy *Lightweight Directory Access Protocol*, or LDAP, a protocol for communicating with directory services. There are dozens of LDAP-compliant directories available on the market, including Microsoft Active Directory, OpenLDAP, and ApacheDS, among others.

This section will use a technique known called *LDAP binding* to authenticate the user. Depending on your specific deployment, you might have to implement some additional steps. Let's begin by setting up a basic ApacheDS instance.

Setting up ApacheDS

Download and install the following:

- *Apache Directory Studio 1.5*. This is the version that was tested with these instructions. However, any LDAP-compliant directory should work fine for this chapter's examples.
- *Apache Studio*. You should download this regardless of your existing LDAP technology. Apache Studio is a superb tool for dealing with any directory.

Let's begin by firing up Apache Studio. This is where we will do most of our LDAP-oriented work. Trust me, this is a lot less painful than using `vi` and `ldif` files. You should see something very close to Figure 6.3, on the following page.

4. It is highly unlikely that any company would actually refer to its directory as "LDAP-compliant."

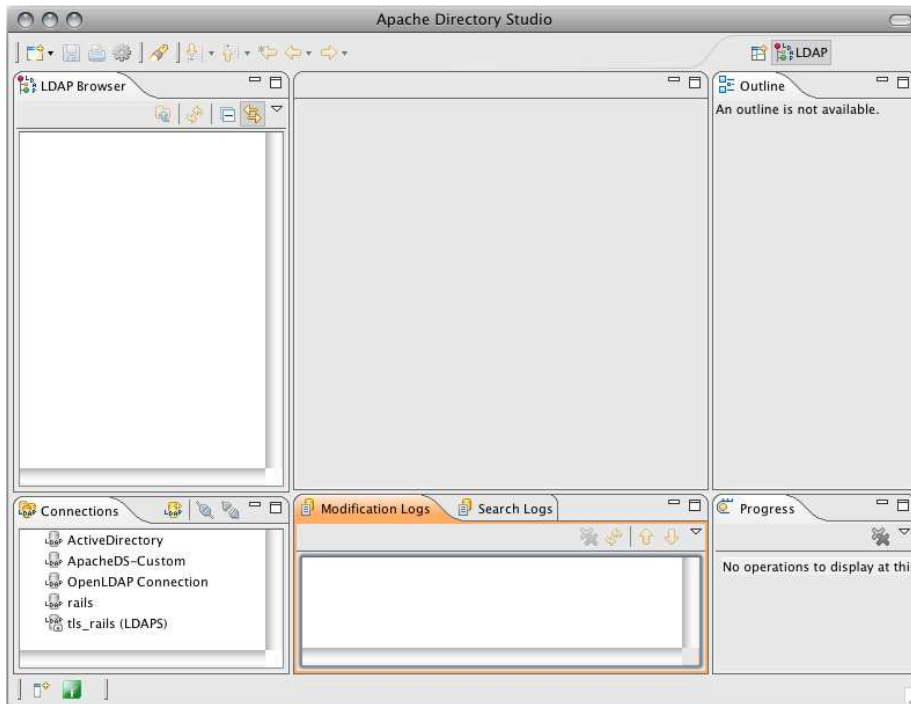


Figure 6.3: Apache Directory Studio

We'll start by creating a new directory instance for our example:

1. Navigate to the Servers View by selecting `Window > Show View > Other`.
2. Create a New Server. Call it something like *Rails Example*. Right-click the Servers Panel and select `New > New Server`.
3. Double-click the newly created server.
4. Click the Partitions tab at the bottom of view and rename the sample partition to anything you want. Most of the defaults are fine for this example.
5. Create a new connection to the server. Right-click the server you want and go to `LDAP Browser > Create a Connection`.
6. Finally, start the server. At this point, you can right-click the server and click `Run` to launch your new directory instance.



Figure 6.4: Apache Directory Studio: new LDAP entry

Joe Asks...

What's inetOrgPerson?

Years ago, CCITT, now known as the ITU-T, came up with a series of standards known as the X.500 Directory Standards. These standards define the items of information common across organizations. Since the rise of the Internet, things like JPEG images and cryptographic certificates can be associated with people. The RFC describes inetOrgPerson like this: “The inetOrgPerson object class is a general-purpose object class that holds attributes about people.” Cool, eh?

Create a New LDAP User

Assuming everything with the ApacheDS setup went smoothly, we can create our test user.

Let's create Wally Webcoder in our newly created LDAP Instance. Let's use *wally* for the username—as we did in our User test fixture:

1. Right-click the little earth icon, and click New Entry.

This should bring up a dialog similar to the one in Figure 6.4.



Figure 6.5: Apache Directory Studio: selecting object class

2. Now let's select `inetOrgPerson` as our object class, as shown in Figure 6.5. Do this by typing in the first few letters of the desired class and double-clicking when it is highlighted.
3. Next, we will use UID in the drop-down list to the left of the = sign. To the right of the = sign, we enter the username. In this case, we will use *wally*. For help on how to do this, take a look at Figure 6.6, on the next page.
4. Before we click the Finish button, we need to click the Add Attribute button. This button is hard to find, but you can see it highlighted in Figure 6.7, on page 113.
5. Next, enter `userPassword` into the "Attribute type" input field, as shown in Figure 6.8, on page 113.
6. Now we get to employ a couple clever tricks we learned previously in the book. First, we get to take advantage of salting passwords



Figure 6.6: Apache Directory Studio: selecting distinguished name

from Section 6.2, *Adding Salts to LunchedIn*, on page 104. Second, we can leverage the hashing example from Chapter 3, *Fixing the Example*, on page 49. This might sound exciting, but all it entails here is marking a checkbox. We don't need to implement any hashing or salting code here; instead, we get to leverage the work we did earlier. Selecting SSHA gives us a salted SHA1 password, as shown in Figure 6.9, on page 114.

Talking to LDAP from Rails

At this point, we have our ApacheDS instance up and running. It will require several steps on our end to integrate LunchedIn (as shown in Figure 6.10, on page 114):

- Add test and development configurations for ApacheDS.
- Add logic in our User model to connect and authenticate with LDAP.
- Add LDAP create logic to our User model.

We need to set up a way to store our LDAP connections in our application. Rails provides a good pattern for this in `database.yml`. Let's mimic this approach with our own `ldap.yml`:

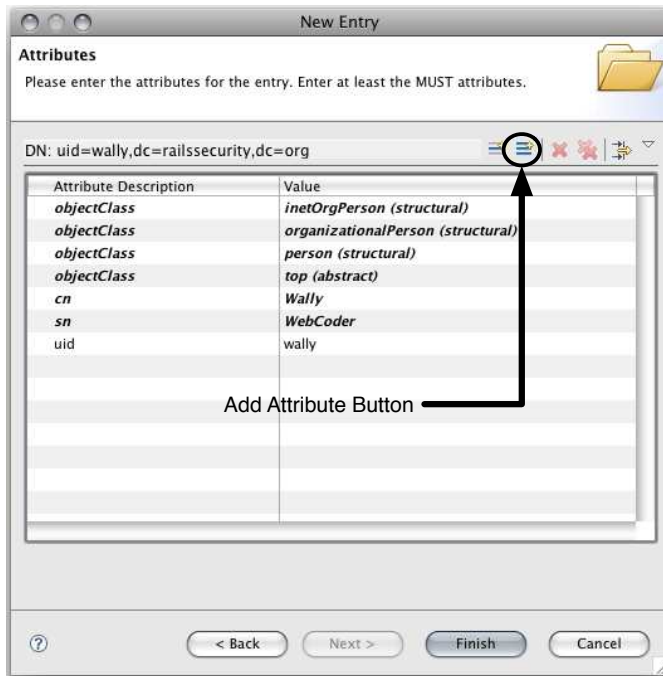


Figure 6.7: Apache Directory Studio: entering MUST attributes



Figure 6.8: Apache Directory Studio: selecting password as attribute type



Figure 6.9: Apache Directory Studio: Password Editor

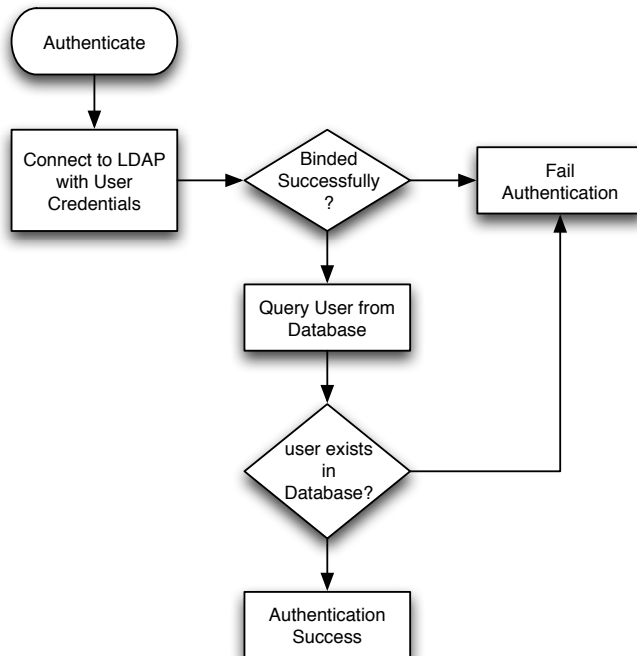


Figure 6.10: LDAP Authentication in LunchedIn

Download authentication/lunchedin_ldap/config/ldap.yml

LDAP Connectivity. Tailor this file to meet your own needs.

development:

```
port: 10636
auth:
  method: simple
  username: uid=admin,ou=system
  password: RailsRulez
host: localhost
encryption: simple_tls
basedn: dc=railssecurity,dc=org
```

test:

```
port: 10636
auth:
  method: simple
  username: uid=admin,ou=system
  password: RailsRulez
host: localhost
encryption: simple_tls
basedn: dc=railssecurity,dc=test
```

The test and development configurations look almost identical. However, if you look carefully, you will see that they use different `basedn:` values. This allows us to host the directories on the same server—much as we do with our databases in Rails.

To read `ldap.yml`, we need to add a line of code to our `environment.rb` that reads the settings into an `LDAP_CONF` constant. There is also an `AUTHENTICATION_MECHANISM` constant—ignore that one for now. Here is the code that loads `ldap.yml`:

Download authentication/lunchedin_ldap/config/environment.rb

```
RAILS_GEM_VERSION = '2.2.2' unless defined? RAILS_GEM_VERSION
```

```
require File.join(File.dirname(__FILE__), 'boot')
```

```
require 'yaml'
```

```
Rails::Initializer.run do |config|
  config.time_zone = 'UTC'
```

```
▶ LDAP_CONF = YAML.load_file("#{RAILS_ROOT}/config/ldap.yml")
```

```
  AUTHENTICATION_MECHANISM = "ldap"
```

```
end
```

This is pretty straightforward so far. Next, let's add a test:

```
Download authentication/lunchedin_ldap/test/unit/user_test.rb
```

```
def test_ldap_conf_yml
  assert_not_nil(LDAP_CONF)
  assert_equal(10636, LDAP_CONF["test"]["port"], LDAP_CONF.inspect)
end
```

At the time this book is being written, there are a few options for connecting to an LDAP directory. In these examples, we use the `ruby-net-ldap` gem. You will need to install the gem for the LDAP examples described in this book to work:

```
> gem install ruby-net-ldap
```

Once we have our Ruby LDAP support in place, we can change our `authenticate?` method in the `User` model so it performs an LDAP bind, rather than the database. Our previous database authentication code looked like this:

```
Download authentication/lunchedin/app/models/user.rb
```

```
def authenticate?(p)
  password_hash ==
    hashed_and_salted_password(p,salt)
end
```

Now our authentication code looks like this:

```
Download authentication/lunchedin_ldap/app/models/user.rb
```

```
def authenticate?(p)
  ldap = Net::LDAP::new({
    :host => "localhost",
    :port => 10636,
    :auth => {
      :method => :simple,
      :username =>
        "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}",
      :password => p
    },
    :encryption => :simple_tls
  })

  bind = ldap.bind
end
```

Next, let's run a unit test:

```
Download authentication/lunchedin_ldap/test/unit/user_test.rb
```

```
def test_authentic
  assert users(:wally).authenticate?("newpass1")
end
```

Finally, let's create an entry for our test fixture:

Download authentication/lunchedin_ldap/test/fixtures/users.yml

```

bobldap:
  id: 5
  first_name: Bob
  last_name: Ldap
  username: bldap
  email: bldap@railssecurity.org
  role_id: 1
  zip_code: 75024
  created_at: 2008-11-27 14:18:26
  updated_at: 2008-11-27 14:18:26

```

For `test_authentic()` to work correctly, you must have created `wally` in the `basedn` value specified in the `test:` section of `ldap.yml`.

The key thing to take away from this code is that the user now exists in both the database and the directory. The difference is that we are now validating the credentials against another datasource.

We want to connect to our LDAP instance in different methods, so let's refactor our LDAP connection code from `authentic?()` into another method:

Download authentication/lunchedin_ldap/app/models/user.rb

```

def self.ldap_server(conf = LDAP_CONF[RAILS_ENV], &block)
  ldap = Net::LDAP.new({:host => conf["host"],
    :port => conf["port"],
    :auth => {
      :method => conf["auth"]["method"].to_sym,
      :username => conf["auth"]["username"],
      :password => conf["auth"]["password"]
    },
    :encryption => conf["encryption"].to_sym})
  yield(ldap)
  ldap
end

```

We can then turn around and use this method to create users in our directory, as shown in this code:

Download authentication/lunchedin_ldap/app/models/user.rb

```

def create_in_ldap
  User.ldap_server do |ldap|
    lattrs = {
      :cn => "#{first_name} #{last_name}",
      :objectclass =>
        ["inetOrgPerson", "organizationalPerson",
          "person", "top"],
      :sn => "#{last_name}",
      :uid => "#{username}",
      :mail => "#{email}",

```

```

▶      :userPassword => User.ldap_ssha(@password)
    }

    dn = "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}"
    ldap.open {|l| l.add(:dn => dn, :attributes => lattrs)}
  end
end

```

We want to create the password properly during our insertion operation, so we create a properly formatted password string. We do it this way because we want to write raw string data to the directory. Here is the code to accomplish that task:

Download authentication/lunchedin_ldap/app/models/user.rb

```

def self.ldap_ssha(password)
  salt_byte_data = OpenSSL::Random.random_bytes(8)
  sha1_digest = Digest::SHA1.digest(password + salt_byte_data)
  pwd_sha1_bs_sa_bs = Base64::encode64(sha1_digest + salt_byte_data)
  "{SSHA}#{pwd_sha1_bs_sa_bs}"
end

```

This code starts by creating eight bytes of random data. Next, it creates a digest of the password with the salt appended. Then, it creates a Base64 representation of the digest, with the salt data appended. Isn't this cool? Now we have a nice little method to format a password for any LDAP inserts we might need in the future. Let's test this functionality:

Download authentication/lunchedin_ldap/test/unit/user_test.rb

```

def test_ldap_server_with_ssha_password
  username = 'bldap'
▶  plaintext = 'newpass1'
▶  password = User.ldap_ssha(plaintext)
  lattrs = {
    :cn => "Bob",
    :objectclass =>
      ["inetOrgPerson", "organizationalPerson",
       "person", "top"],
    :sn => "LDAPguy",
    :uid => username,
    :mail => "bob@railssecurity.org",
    :userPassword => password
  }
  dn = "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}"
  User.ldap_server do |ldap|
    ldap.open {|l| l.add(:dn => dn, :attributes => lattrs)}
▶  assert User.find_and_authenticate(username, plaintext)
    ldap.open {|l| l.delete(:dn => dn)}
  end
end

```

Note how we use password to store the digested and salted password, but we use plaintext to perform the actual authentication. This is, of course, the format we will receive from our users. But now let's consider another case, where we don't want to use salts for our passwords:

Download authentication/lunchedin_ldap/test/unit/user_test.rb

```
def test_ldap_server_with_plain_password
  username = 'bldap'
  ▶ plaintext = 'newpass1'
  lattrs = {
    :cn => "Bob",
    :objectclass =>
      ["inetOrgPerson", "organizationalPerson",
       "person", "top"],
    :sn => "LDAPguy",
    :uid => username,
    :mail => "bob@railssecurity.org",
    :userPassword => plaintext
  }

  dn = "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}"
  User.ldap_server do |ldap|
    ▶ ldap.open {|l| l.add(:dn => dn, :attributes => lattrs)}
    assert User.find_and_authenticate(username, plaintext)
    ldap.open {|l| l.delete(:dn => dn)}
  end
end
```

This code works because the LDAP server is performing the salt and digest based on what is stored in the password. The `ldap_ssha()` class method in our `User` model does some heavy lifting for us. Let's jump into a console session and have a look at some output:

```
>> User.ldap_ssha("password")
=> "{SSHA}7f4Bw11pdGQeG+7NfU/jtWkd24Sr0L0Psrsbrg==\n"
>>
```

The `{SSHA}` at the beginning of the string tells the LDAP server to apply a SHA1 digest and an eight-byte salt to this principal's password when a bind request is made. Can you guess what the first S stands for? Here's a hint—LDAP also supports SMD5, SHA, and MD5. S stands for salt. If we drop the `{SSHA}`, then the password is just plaintext.⁵

Talking to Microsoft Active Directory from Rails

Active Directory is probably the most popular LDAP implementation. Because it *is* compliant with X.509 and other standards, it seems

5. Well, not quite. Plaintext passwords are still Base64-encoded.

Configurable Authentication

We added the `AUTHENTICATION_MECHANISM` constant in `environment.rb` and ignored it. We didn't write LunchedIn to support LDAP authentication exclusively, so we need to provide a way to support both. Here's a *cheap* way we can add that needed flexibility:

Download authentication/lunchedin_ldap/app/models/user.rb

```

▶ if AUTHENTICATION_MECHANISM == "ldap"

  before_save :create_in_ldap

  def authentic?(p)
    ldap = Net::LDAP::new({
      :host => "localhost",
      :port => 10636,
      :auth => {
        :method => :simple,
        :username =>
          "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}",
        :password => p
      },
      :encryption => :simple_tls
    })

    bind = ldap.bind
  end
else
  def authentic?(p)
    password_hash ==
      hashed_and_salted_password(p, salt)
  end
end

```

Adding a single constant to our environment gives us a simple way of choosing authentication types. This approach might be a little ugly, but it works.

reasonable to hope that our code will work correctly. Let's write a unit test that connects to an Active Directory installation:

[Download](#) authentication/lunchedin_ldap/test/unit/user_test.rb

```
def test_active_directory_domain_name_success
  ldap = Net::LDAP::new({
    :host => "192.168.137.10",
    :port => 389,
    :auth => {
      :method => :simple,
      :username =>
        'RAILSSECURITY\draphael',
      :password => "RailsIs5uper"
    }
  })
  assert ldap.bind
end
```

Don't forget that draphael must exist on the RAILSSECURITY Windows domain for this code to work. Let's also add a failure test:

[Download](#) authentication/lunchedin_ldap/test/unit/user_test.rb

```
def test_active_directory_domain_name_failure
  ldap = Net::LDAP::new({
    :host => "192.168.137.10",
    :port => 389,
    :auth => {
      :method => :simple,
      :username =>
        'RAILSSECURITY\foobar',
      :password => "foobar"
    }
  })
  assert !ldap.bind
end
```

There are other ways we can build the username portion of our binding, as well. Here's a more traditional LDAP example:

[Download](#) authentication/lunchedin_ldap/test/unit/user_test.rb

```
def test_active_directory_ldap_name
  ldap = Net::LDAP::new({
    :host => "192.168.137.10",
    :port => 389,
    :auth => {
      :method => :simple,
      :username =>
        'cn=David Raphael,CN=Users,dc=railssecurity,dc=org',
      :password => "RailsIs5uper"
    }
  })
  assert ldap.bind
end
```

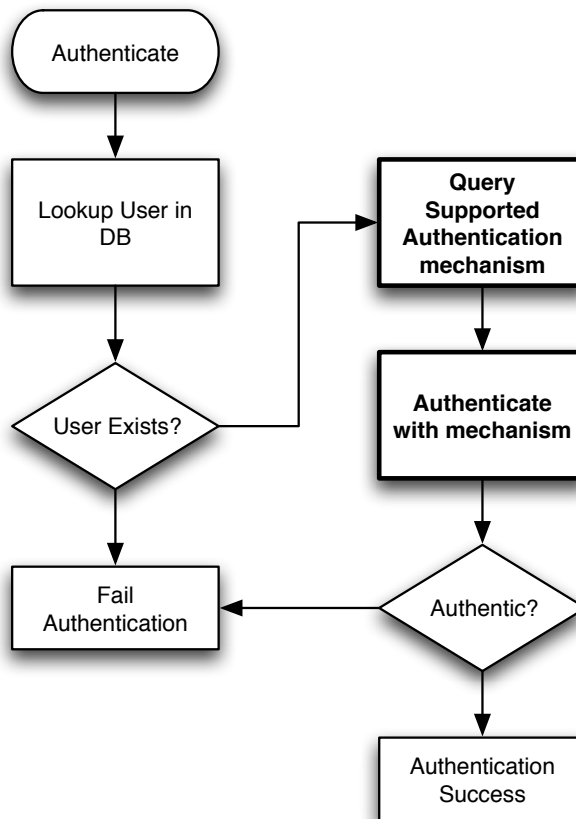


Figure 6.11: Choosing between LDAP or database at runtime

Figuring out the right combination of CNs and DCs is going to be dependent on a particular Active Directory installation. You will have to talk to the local Active Directory administrator for details on your particular setup.⁶

6.4 Flexible Authentication

We've added LDAP support to LunchedIn, but now we find ourselves stuck with either LDAP or database authentication at deployment—

6. Sometimes Active Directory administrators will set up their domains in a way that doesn't lend itself to LDAP authentication.

through our environment setting. This might be sufficient for some applications, but how can we make this more flexible? Specifically, how would we deal with this in an application like LunchedIn? Let's look at ways to modify LunchedIn so we can deal with this in a more dynamic manner.

A Smarter User Model

The basic flow that we want to achieve with our flexible authentication is shown in Figure 6.11, on the preceding page. We can solve this problem in quite a few ways, but let's start attacking this problem by leveraging one of my favorite ActiveRecord features: polymorphic models with single-table inheritance. Here is the migration we'll need to be able to subclass User:

[Download authentication/lunchedin_flex_auth/db/migrate/012_add_type_to_users.rb](#)

```
class AddTypeToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :type, :string, :default => "User"
    add_index :users, :type
  end

  def self.down
    remove_column :users, :type
  end
end
```

[Download at WoweBook.com](#)

Rethinking Our Users

You might recall that we hacked LDAP support into the code base in Section 6.3, *Talking to LDAP from Rails*, on page 112. And while that code passed all unit tests, we didn't really analyze the difference between our original User model and the LDAP-enabled version. For example, we added a `:before_save` callback that inserted the user into our directory. What about existing LDAP users? In fact, most organizations have facilities already in place for managing user accounts. We need to do a better job of integrating LunchedIn—as we would in the case of a real-world corporate integration.

Let's start by identifying what concerns users share, regardless of authentication mechanism. Here are the common concerns that we want to address for Users in LunchedIn:

- Users must have a defined role, such as admin or user.
- Users must be tied to comments through a join table.
- Users must have a first name, last name, email, and ZIP code.

- Users should be authenticated through a username and password, regardless of authentication mechanism. This means we can create a common contract for finding and authenticating any type of user.

Now let's list the common concerns we want to address specific to users authenticating against the database:

- Database-authenticated users must store their passwords in the database.
- Database-authenticated users must provide validation support for password creation and manipulation.
- Database-authenticated users must provide the necessary code to salt and hash the passwords.
- Database-authenticated users must provide the necessary code to authenticate based on a username and password.

Finally, we need to consider the concerns that are specific to users authenticating against LDAP:

- LDAP-authenticated users must exist in the directory. We will need to add this to the user validation.
- LDAP-authenticated users must hydrate an address, first name, last name, and ZIP code from the directory.
- LDAP-authenticated users must provide the necessary code to authenticate based on a username and password.

Polymorphic user models are described in a simple class diagram in [Figure 6.12](#), on the following page. We can use this diagram for guidance while we code.

User, DbUser, and LDAPUser are now separate models, but all of them share the same database table. This means we should be able to add authentication types without affecting any of our relationships with other models, such as Event or Comment.

User, DbUser, and LDAPUser

Let's begin by creating our DbUser. This should be an easy change because we don't need to add or subtract any functionality.

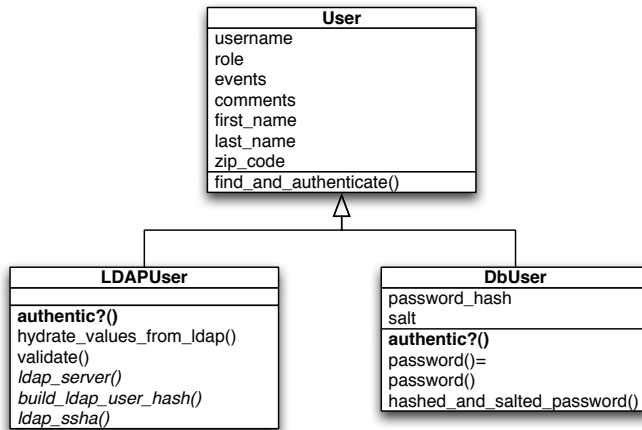


Figure 6.12: Polymorphic user models

The first step is to amputate portions of our original User until it looks like this:

[Download](#) authentication/lunchedin_flex_auth/app/models/user.rb

```

class User < ActiveRecord::Base
  attr_protected :role_id
  validates_presence_of :username
  validates_uniqueness_of :username
  has_many :events
  has_many :comments, :as => :commentable
  belongs_to :role

  def self.find_and_authenticate(username, password)
    user = find_by_username(username)
    return false if user.nil?
    if user.authentic?(password)
      return user
    end
    false
  end
end

```

Nothing too exciting happens here.

Next, we need to take the code that we removed from `User` and add it to our newly created `DbUser`:

[Download](#) authentication/lunchedin_flex_auth/app/models/db_user.rb

```
class DbUser < User
```

```

▶ # DbUser specific
▶ validates_presence_of :password_hash
▶ validates_length_of :zip_code, :is => 5
▶ validates_numericality_of :zip_code
▶ validates_presence_of :email, :first_name, :last_name
▶ validates_length_of :password, :in => 7..32
▶ validates_confirmation_of :password

  def password=(password)
    @password = password
    write_attribute(:salt,
      OpenSSL::Digest::SHA1.new(
        OpenSSL::Random.random_bytes(256)).hexdigest)
    write_attribute(:password_hash,
      hashed_and_salted_password(password, salt))
  end

  def password
    @password
  end

▶ def authentic?(p)
▶   password_hash ==
▶   hashed_and_salted_password(p, salt)
▶ end

  def hashed_and_salted_password(p, s)
    OpenSSL::Digest::SHA1.new([s,p].to_s).hexdigest
  end

end

```

The main thing worth noting here is that we've added the validations and `authentic?()` method. We also need to move the relevant test code from the old `UserTest` to `DbUserTest`:

[Download](#) authentication/lunchedin_flex_auth/test/unit/db_user_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'
```

```
class DbUserTest < ActiveSupport::TestCase
```

```

  def test_find_and_authenticate_success
    assert User.find_and_authenticate('wally', "newpass1")
  end

```

```

def test_authentic_failed
  assert !User.find_and_authenticate('wally', 'foo')
end

def test_authentic
  assert users(:wally).authentic?("newpass1")
end

```

end

For our unit test to work correctly, we also need to update all of our fixtures to have the correct type, as in this example:

[Download](#) authentication/lunchedin_flex_auth/test/fixtures/users.yml

```

bob:
  id: 1
  type: DbUser
  username: bob
  password_hash: <%= OpenSSL::Digest::SHA1.new("hello").hexdigest %>
  first_name: Bob
  last_name: Smith
  email: bob@gmail.com
  role_id: 1
  zip_code: 75024
  created_at: 2007-11-27 14:18:26
  updated_at: 2007-11-27 14:18:26

```

Here's a nice detail: we can add a type: of DbUser to all of our test data. Let's add one for LDAP:

[Download](#) authentication/lunchedin_flex_auth/test/fixtures/users.yml

```

wally:
  id: 3
  type: LDAPUser
  first_name: Wally
  last_name: Webcoder
  username: wally
  password_hash: <%= OpenSSL::Digest::SHA1.new("newpass1").hexdigest %>
  email: wally@yahoo.com
  role_id: 2
  zip_code: 75024
  created_at: 2007-11-27 14:18:26
  updated_at: 2007-11-27 14:18:26

```

At this point, we can create our LDAPUser model. We can start by stealing the authentic?() method code from the LDAP example in Section 6.3, *Talking to LDAP from Rails*, on page 112.

Download authentication/lunchedin_flex_auth/app/models/ldap_user.rb

```
class LDAPUser < User
  attr_accessor :password, :password_confirmation

  def authentic?(p)
    ldap = Net::LDAP::new({
      :host => "localhost",
      :port => 10636,
      :auth => {
        :method => :simple,
        :username =>
          "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}",
        :password => p
      },
      :encryption => :simple_tls
    })
    bind = ldap.bind
  end
end
```

We can test this code using the user we created in Section 6.3, *Create a New LDAP User*, on page 110:

Download authentication/lunchedin_flex_auth/test/unit/ldap_user_test.rb

```
def test_find_and_authenticate_success
  assert User.find_and_authenticate('wally', "newpass1")
end
```

Again, nothing too exciting occurs here. Just keep in mind that it would be ideal for the test LDAP data to be refactored out of the unit tests.⁷ Next, let's add the code that will pull the appropriate data from the directory and populate our database:

Download authentication/lunchedin_flex_auth/app/models/ldap_user.rb

```
before_save :hydrate_values_from_ldap

def hydrate_values_from_ldap
  LDAPUser.ldap_server do |ldap|
    results = ldap.search({
      :base => LDAP_CONF[RAILS_ENV]["basedn"],
      :filter => Net::LDAP::Filter.eq("uid", username)})
    raise "Ambiguous LDAP search. " if results.size > 1
    write_attribute(:email, results[0].mail[0])
    write_attribute(:zip_code, results[0].postalCode[0])
    write_attribute(:first_name, results[0].givenName[0])
    write_attribute(:last_name, results[0].sn[0])
  end
end
```

7. We didn't want to spend too much time focusing on non-authentication-related code here. It would be relatively trivial to leverage this code to move LDAP test data out of the unit tests and into fixtures.

Notice that we've attached the `hydrate_values_from_ldap()` as an ActiveRecord callback—before saving.

We require that an LDAPUser exist in LDAP before we create the account in LunchedIn, so we need to authenticate the user in LDAP:

Download authentication/lunchedin_flex_auth/app/models/ldap_user.rb

```
def validate
  unless authentic?(password)
    errors.add_to_base("LDAP Authentication failed")
  end
end
```

This code reuses the `authentic?()` method. Kinda cool, eh? The password is simply a value that we get from controller when it's posting.

We are never done until we've tested. This test code is a little bit bulky, but that's mainly because our fixtures are inline:

Download authentication/lunchedin_flex_auth/test/unit/ldap_user_test.rb

```
def test_create_success
  LDAPUser.ldap_server do |ldap|
    username = "test_create_username"
    lattr = LDAPUser.build_ldap_user_hash("test", "user",
      "test@railssecurity.org", username, "password", "75093")
    # create our test user in LDAP
    dn = "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}"
    ldap.open {|l| l.add(:dn => dn, :attributes => lattr)}
    user = LDAPUser.new do |u|
      u.username = username
      u.password = "password"
    end

    assert(user.save, user.errors.entries.join("\n"))
    user.reload
    # Make sure that the attributes are getting populated
    assert_equal("test@railssecurity.org", user.email)
    assert_equal("test", user.first_name)
    assert_not_nil("user", user.last_name)
    assert_equal("75093", user.zip_code)
    # This shouldn't be getting populated
    assert_nil(user.password_hash)
    assert(User.find_and_authenticate(username, "password"))
    # make sure we delete our test user afterwards
    ldap.open {|l| l.delete(:dn => dn)}
  end
end
```

Our next snippet uses the `build_ldap_user_hash()` class method to create a hash for LDAP inserts (again, this is a by-product of not having LDAP test fixtures).

Download authentication/lunchedin_flex_auth/app/models/ldap_user.rb

```
def self.build_ldap_user_hash(first_name, last_name,
  email, username, password, zip_code)
  lattrs = {
    :cn => "#{first_name} #{last_name}",
    :givenName => first_name,
    :objectclass =>
      ["inetOrgPerson", "organizationalPerson",
        "person", "top"],
    :sn => last_name,
    :uid => username,
    :mail => email,
    :postalCode => zip_code,
    :userPassword => ldap_ssha(password)
  }
end
```

The `build_ldap_user_hash()` uses the `ldap_ssha()` class method that we built previously. Here's that snippet of code again:

Download authentication/lunchedin_flex_auth/app/models/ldap_user.rb

```
def self.ldap_ssha(password)
  salt_byte_data = OpenSSL::Random.random_bytes(8)
  sha1_digest = Digest::SHA1.digest(password + salt_byte_data)
  pwd_sha1_bs_sa_bs = Base64::encode64(sha1_digest + salt_byte_data)
  "{SSHA}#{pwd_sha1_bs_sa_bs}"
end
```

With this code in place, we can proceed to the controller and views.

Updating Registration Form

We've extended our `User` model into `DbUser` and `LDAPUser`, which means need to make a few tweaks to `UserController` to accommodate the new classes.

Let's begin by updating the `new()` method:

Download authentication/lunchedin_flex_auth/app/controllers/users_controller.rb

```
def new
  user_type = params[:user].nil? ? '' : params[:user][:type]
  case user_type
  when 'LDAPUser'
    @user = LDAPUser.new(params[:user])
  else
    @user = DbUser.new(params[:user])
  end
end
```

It's worth noting that this code defaults to `DbUser` if the request doesn't specify a type.

We also need to add code that is similar to the `create()` method in the controller:

Download authentication/lunchedin_flex_auth/app/controllers/users_controller.rb

```
def create
  # Little glue to create correct account type
  case params[:user][:type]
  when 'LDAPUser'
    @user = LDAPUser.new(params[:user])
  else
    @user = DbUser.new(params[:user])
  end

  respond_to do |format|
    if @user.save
      flash[:notice] = 'your account was successfully created.'
      session[:user_id] = @user.id
      format.html { redirect_to(@user) }
    else
      format.html { render :action => 'new' }
    end
  end
end
```

We need to add two lines to `routes.rb` for `redirect_to()` to work correctly:

Download authentication/lunchedin_flex_auth/config/routes.rb

```
map.resources :users
# These are pointing to the User controller
map.resources :db_users, :controller => "users"
map.resources :ldap_users, :controller => "users"
```

This tells Rails to use the `UserController` as the controller for `DbUser` and `LDAPUser`.

Next, we need to update our functional test. First, we need to add a type parameter to our create requests—because the controller now expects a type for creation explicitly. This requires a handful of changes in our `test_create_user_success()` method:

Download authentication/lunchedin_flex_auth/test/functional/users_controller_test.rb

```
def test_create_user_success
  post :create, :user => {
    :type => "DbUser",
    :first_name => 'Juan',
    :last_name => 'Valdez',
    :email => 'juan@gmail.com',
```

```

      :username => 'juaninamillion',
      :zip_code => '75024',
      :password => 'hello34',
      :password_confirmation => 'hello34'
    }
    user = User.find_by_username('juaninamillion')
    assert_not_nil user, assigns(:user).errors.inspect
    assert_response :redirect
    assert_redirected_to user_path(assigns(:user))
  end
end

```

So far we've updated the existing test; next, we need to add a test case for our LDAP user creation:

Download authentication/lunchedin_flex_auth/test/functional/users_controller_test.rb

```

def test_create_user_success_in_ldap
  LDAPUser.ldap_server do |ldap|
    username = "juaninamillion"
    password = "hello34"
    ▶ lattrs = LDAPUser.build_ldap_user_hash("test", "user",
    ▶ "test@railssecurity.org", username, password, "75093")
    ▶ dn = "uid=#{username},#{LDAP_CONF[RAILS_ENV]["basedn"]}"
    ▶ ldap.open {|l| l.add(:dn => dn, :attributes => lattrs)}
    ▶ post :create, :user => {
    ▶   :type => "LDAPUser",
    ▶   :username => username,
    ▶   :password => password
    ▶ }
    user = User.find_by_username('juaninamillion')
    assert_not_nil user, assigns(:user).errors.inspect
    assert_response :redirect
    assert_redirected_to user_path(assigns(:user))
    ldap.open {|l| l.delete(:dn => dn)}
  end
end
end

```

This code has the funky LDAP-user creation embedded, just as in our unit tests. Once we add this, we can proceed to fixing our view code.

We're dealing with two different models in this case, which means we need to be careful not to split our views into separate partials. However, the LDAPUser requires only a subset of DbUser, so we can guard against this by adding a little glue to the existing form:

Download authentication/lunchedin_flex_auth/app/views/users/new.html.erb

```

<h1>Have lunch with us</h1>

<h2>Create A Profile</h2>

<%= error_messages_for 'user' %>

```

```

<% form_tag :action => 'create' do %>

<div class="highlighted-form">
  <table class="form">
    <tr>
      <td class="label">Account Type:</td>
      <td>
        <%= radio_button('user', 'type', 'LDAPUser') %> LDAP
        <%= radio_button('user', 'type', 'DbUser') %> Database
      </td>
    </tr>
    <tr>
      <td class="label">User Name:</td>
      <td><%= text_field('user', 'username', :size => 14) %></td>
    </tr>
    <tr>
      <td class="label">Password:</td>
      <td><%= password_field('user', 'password', :size => 14) %></td>
    </tr>
    <% if @user.is_a? DbUser %>
    <tr>
      <td class="label">Confirm Password:</td>
      <td><%= password_field('user', 'password_confirmation', :size => 14) %></td>
    </tr>
    <tr>
      <td class="label">First Name:</td>
      <td><%= text_field('user', 'first_name', :size => 16) %></td>
    </tr>
    <tr>
      <td class="label">Last Name:</td>
      <td><%= text_field('user', 'last_name', :size => 16) %></td>
    </tr>
    <tr>
      <td class="label">Email:</td>
      <td><%= text_field('user', 'email') %></td>
    </tr>
    <tr>
      <td class="label">Zip Code:</td>
      <td><%= text_field('user', 'zip_code', :size => 6) %></td>
    </tr>
    <% end %>
  </table>
</div>
<div style="padding: 10px;">
  <%= submit_tag 'Create' %>
</div>
<% end %>

```

These changes should produce the result we see in Figure 6.13, on the next page. If we select a database user, we should see a result similar to that displayed in Figure 6.14, on the following page.

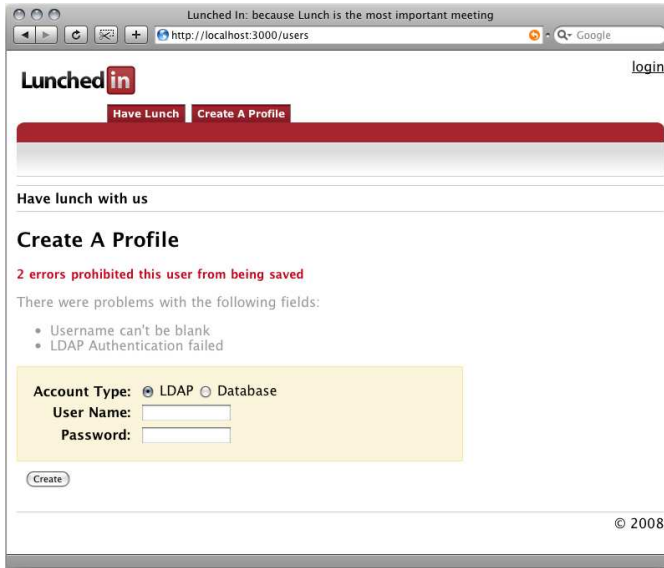


Figure 6.13: Creating an LDAP-authenticated user

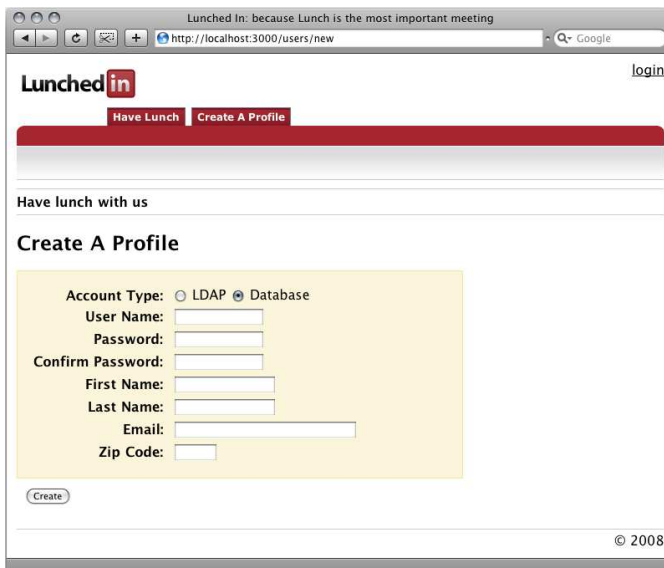


Figure 6.14: Creating a database-authenticated user

Authorization

7.1 What Is Authorization?

We defined authorization as the process of verifying privilege for a given principal in Chapter 6, *Authentication: Decentralized Authentication*, on page 95. To put it another way, authorization is the process of deciding which actors have access to what actions and what data.

Authorization fits into several models:

- Mandatory access control (MAC) grants access based on the sensitivity (denoted by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of a given sensitivity. Government information classification, such as Secret or Top Secret, is a good example of MAC-style authorization.
- Discretionary access control (DAC) grants access to objects based on the identity of subjects and/or groups to which they belong. Windows and Unix file permissions are good examples of DAC-style authorization.
- Role-based access control (RBAC) lets you associate access control with actions in a system that typically exist above the object level. A given action of an application can interact with a number of objects, but access to actions is controlled through permission based on role assignments, not at the level of individual data objects.

Several community development projects provide generic authorization support in Rails:

ActiveRbac <http://active-rbac.rubyforge.org/>
The ActiveRbac plug-in helps you implement role-based access control in your application.

ModelSecurity <http://perens.com/FreeSoftware/ModelSecurity/>
The ModelSecurity plug-in leverages the data model to implement access control.

ActiveRbac appears to remain in active development. Its primary goal is to provide programmatic support for attaching permissions to roles and attaching roles to users.

ModelSecurity appears to be stagnant at this point. However, for someone who needs tight control over ActiveRecord models, ModelSecurity takes an approach similar to the ACEGI project from the Java community. Essentially, it provides a wormhole from the controller layer to the model layer, enabling the model to be aware of the current user. This enables an additional layer of defense that you don't get in more controller-centric approaches.

7.2 Mandatory Access Control

Authorization systems that implement MAC generally do so on a system-wide basis. You need to use a MAC model when regulations or laws make specific classifications or mandates regarding data. For example, certain U.S. laws prevent the export of cryptographic libraries to certain countries labeled as restricted. Therefore, any site that provides software downloads *must* control access to cryptographic software downloads based on restriction labels. MAC is another method for enforcing the principle of least privilege, as defined in Chapter 1, *Security in Ruby on Rails*, on page 10; it accomplishes this by reducing privilege system-wide.

Let's take a look at a less political example, where we need to implement a corporate policy on access control for payroll data:

- The HR Coordinator has access to employee information.
- The Payroll Manager has access to employee information and payroll information.

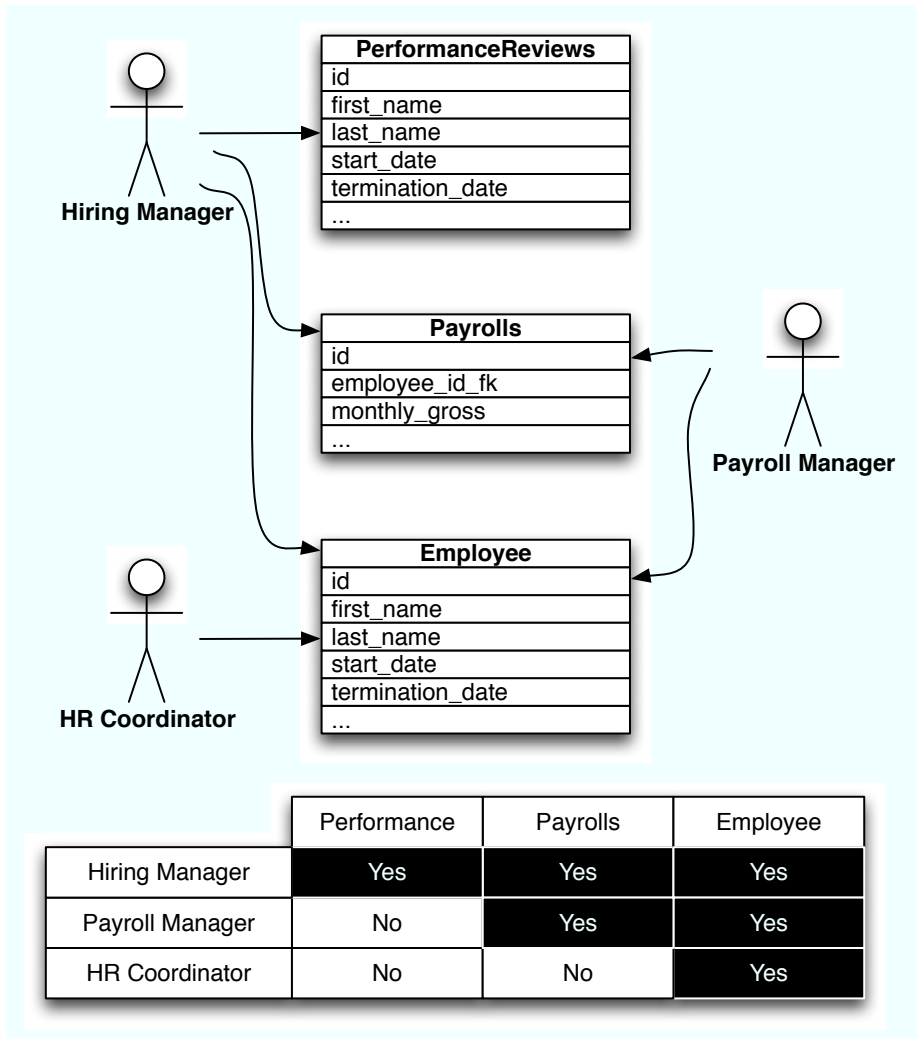


Figure 7.1: Mandatory access control

	Create	Read	Update	Delete	Target
Jim	n	y	n	n	PersonnelRecord
Jim	y	y	y	n	PayrollRecord
Wally	y	y	y	y	WallysPersonalData

Figure 7.2: Discretionary access control

- The Hiring Manager has access to employee information, employee performance reviews, and payroll information.

Let's take a look at Figure 7.1, on the previous page. Here we see three tables that contain data about employees. It is our lowest level of security control because the basic employee information, such as name and employment data, is a piece of data that anyone in an HR department can access (access to payroll data is essential for people who administer payroll or perform salary reviews). Finally, this example restricts performance reviews to the hiring managers.

7.3 Discretionary Access Control

We see DAC implemented most widely in operating systems, relational databases, or any kind of system-level software. We need DAC whenever we have generic (not activity-centric) access control concerns.

We can see an example that relies on DAC in Figure 7.2. Here we have a small list of principals, actions, and targets. Together, the items in this list comprise the generic building blocks of a DAC system. We derive actions from a well-defined interface. For files, Unix and Windows provide a contract for dealing with files that include a permission. In many languages, such as Java or C#, the virtual machine provides a security manager that understands how to deal with objects that implement a predetermined security interface.

Although we can create a collection of permissions, add them to a group, and call this a role, we haven't created a "role" from a security purist's standpoint. Our example separates the action (read, update, delete, and so on) from the target (some data). In RBAC, the action is the target.

	Create	Read	Update	Delete	Owner	Target
Everyone	n	y	n	n	Joe	Joe's Page
Owner	y	y	y	y		
Joe's Friends	n	y	n	n		
Bob	n	y	n	n		

Figure 7.3: Initial access list: Joe's page

	Create	Read	Update	Delete	Owner	Target
Everyone	n	n	n	n	Joe	Joe's Page
Owner	y	y	y	y		
Joe's Friends	n	n	n	n		
Bob	n	n	n	n		

Figure 7.4: Access list: Joe's page after he sets the page to private

If we want to implement a user-centric content-management system (CMS), such as a community wiki, DAC allows our users to set specific access based on the site's content.

Let's imagine a fictional wiki that we've built for users to exchange techniques for writing better web applications. Let's also assume that Joe and Bob participate on this wiki. In this scenario, the default behavior dictates that any user can read any page that any other user on the site creates. However, the wiki also supports discretionary control on its pages. Let's look at a more detailed example:

- Joe creates a new wiki page.
- The wiki application creates a new access list (see Figure 7.3).
- Joe sets access on the page to private.
- The wiki application modifies the page's access list so only the owner can read it (see Figure 7.4).
- Bob browses to the page that Joe created.
- The wiki application checks the page's access list. Bob doesn't meet the criteria to access the page, so the site denies access to Bob (see Figure 7.5, on the next page).

	Create	Read	Update	Delete	Owner	Target
Everyone	n	n	n	n	Joe	Joe's Page
Owner	y	y	y	y		
Joe's Friends	n	y	n	n		
Bob	n	y	n	n		

Figure 7.5: Access list: Joe's page after he grants additional rights

But now let's assume that Joe wants to grant *certain* people access to his page, so he creates a *group* called Joe's Friends.

We need to remember several important details when dealing with DAC:

- Groups are principals.
- Users are principals.
- Principals are the actors in a DAC-based system.
- Targets are assigned ownership, usually by the target's creator.

7.4 Role-Based Access Control

Often, individual roles or job functions drive access requirements within an organization. These roles can be associated with activities in a system. For example, a doctor can create a prescription that might move throughout the system. The doctor might have *created* the prescription, but she doesn't own the prescription object. The prescription object is a transient object—one that represents the authorization itself! Several entities can read the prescription, including the patient, the pharmacist, and the insurance company.

Assume we have a short list of roles associated with their respective controllers and actions (see Figure 7.6, on the following page). In this example, we create two roles: admin and user. This maps well to the following user stories:

- An admin can add or delete users.
- An admin can update or delete profiles.
- A user can register as a user.
- A user can update her profile.

Using a story-centric approach enables us to include our security requirements during our planning sessions. In other words, autho-

Roles	Controllers	Actions
:admin	UsersController	#add_user
		#delete_user
	ProfilesController	#update_profile
		#delete_profile
:user	UsersController	#register
	ProfilesController	#update_my_profile

Figure 7.6: Role-based access control

rization is not an afterthought. Let's simulate a dialogue between our user(s) and developer(s) about a new social-networking application for a site dedicated to discussing and sharing music lyrics.

Joe User: *We need the ability to flag potentially vulgar lyrics.*

Dave Developer: *Who is "we"?*

Joe User: *The Site Moderators.*

Dave Developer: *OK, so Site Moderators must have an interface to flag lyrics submitted by a Community User as Vulgar.*

Joe User: *Yes, and Community Users should also have the ability to mark lyrics as Vulgar, but a Site Moderator should review those particular Vulgar Flags.*

Dave Developer: *Got it.*

This dialogue reveals several things:

- We will need a `flag_vulgar()` action that we will assign this action to the Site Moderator role.
- We will need a `submit_as_vulgar()` action that we will assign to the Community User role.
- We will need an `approve_as_vulgar()` action that we will assign to the Site Moderator role.

Now, all we need is a way to implement these roles in our application.

7.5 Implementing RBAC

Before we jump into implementing RBAC in our application, let's take a look at where our authorization will reside in the request life cycle (see Figure 7.7, on the next page).

We will implement our authorization control inside a filter that executes only if our authentication filter returns `true`.

However, we need to provide a list of controller actions paired with roles for our filter to work its magic.

Role-Based Controller Authorization

We have several choices for enumerating access across our controllers:

- We can create a static list of controller actions and their roles in a separate file. This approach is simple, but it decouples our implementation from the role assertions. Decoupling the two creates unnecessary maintenance complexity.
- We can create a database-driven list of controller actions and their roles. This strategy can work if we want to configure our application roles at runtime. However, it eliminates the benefit of capturing our RBAC requirements during our planning sessions with the users. This approach also decouples the role assertions from the code.
- We can leverage the brilliantly dynamic nature of Ruby and create our own macro-like constructs to support role assertions on controller methods. This allows us to assign roles to the software during our code implementation. Additionally, we can expand our functional tests to assert authorization checks for each action we test.

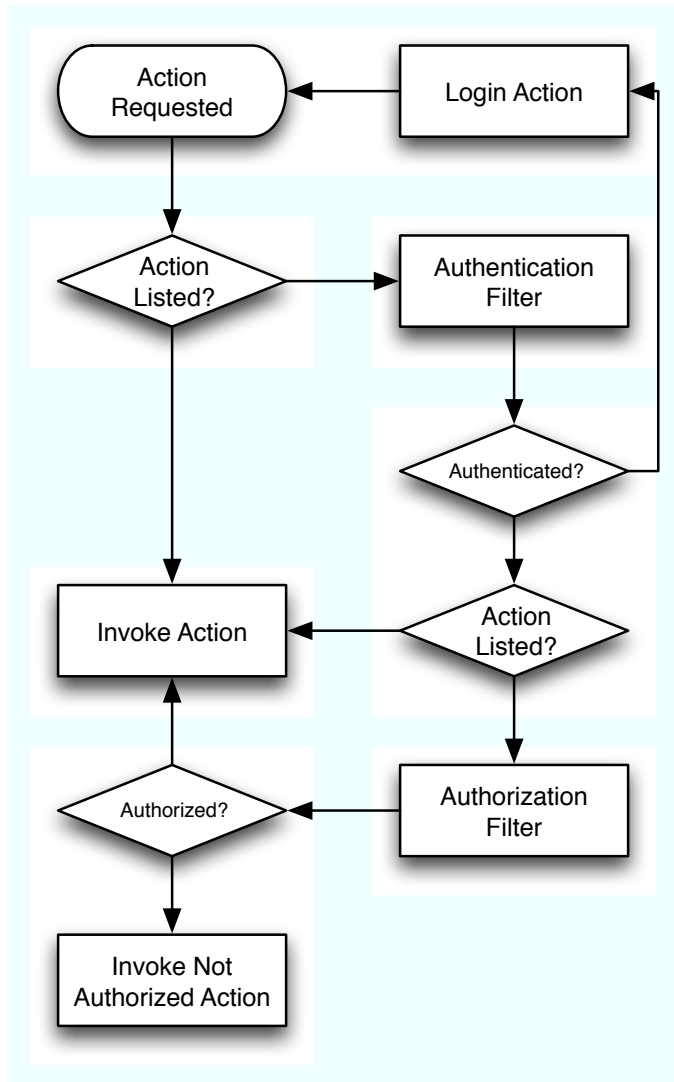


Figure 7.7: Application filter flow

A Note About Filters

It is a good practice to return true or false in filter methods explicitly. This is easy to forget because Ruby does not require an explicit return in methods. Rails depends on true|false to determine whether it should execute the next filter in the chain.

Let's try to implement the third option because it just feels right! This code illustrates what we want our syntax for a role(s) declaration to look like:

```
...
roles :admin
def delete_user

end
...
roles :admin, :user
def add_comment

end
...
```

To do this, we need to create another module and call it RoleBasedControllerAuthorization. This module implements several features:

- It provides class methods to mix in with our controllers.
- It provides a class instance variable read accessor that renders a method's role(s).
- It provides a class instance variable or accessor that renders the roles for a state variable as they are called by the macro.
- It implements a self.method_added for the controller. This method captures the methods as we add them. It then correlates the roles stored by the state variable into the correct hash value in the access list.
- It provides the authorization filter method for our controllers.

Let's get started. First, we define a `self.included()` method that also mixes the class methods in. This saves us from having to use a separate `extend()` in our controllers:

Download `authorization/lunchedin_old/app/security/role_based_controller_authorization.rb`

```
module RoleBasedControllerAuthorization

  def self.included(base)
    base.extend(AuthorizationClassMethods)
  end
end
```

Second, we need to create our class methods:

Download `authorization/lunchedin_old/app/security/role_based_controller_authorization.rb`

```
Line 1 module AuthorizationClassMethods
-   def self.extended(base)
-     class <<base
-       @access_list = {}
5       attr_reader :access_list
-     end
-   end
-
-   def roles(*roles)
10    @roles = roles
-   end
-
-   def method_added(method)
-     logger.debug "#{caller[0].inspect}"
15    logger.debug "#{method.inspect}"
-     @access_list[method] = @roles
-   end
- end
```

This code defines an `access_list()` reader on line 5. This method gives us read-only access to our role-based access list. Next, we define a `roles()` method that accepts an array of roles on line 10. Finally, we add the `@roles` state variable to the `access_list` when a method is defined in a controller on line 16.

Additionally, we need to include our newly crafted module into our `ApplicationController`:

Download `authorization/lunchedin_old/app/controllers/application.rb`

```
class ApplicationController < ActionController::Base
▶   include RoleBasedControllerAuthorization
end
```

Let's unit-test our work. We begin by creating a controller with role declarations that we can use for testing.



Joe Asks...

Wait! Aren't We Changing the Way Ruby/Rails Works?

One of Ruby's greatest benefits is that it lets you leverage its metaprogramming facilities. For example, our ability to participate in the class evaluation life cycle of our controller classes tremendously simplifies the effort required to implement the role-based authorization described in this chapter. It also greatly reduces the amount of code required to perform an authorization check for each action.

```
class RoleBasedController < ApplicationController
  roles :admin
  def admin_only

  end

  roles :admin, :user
  def user_and_admin

  end
end
```

Now let's write our test:

```
class RoleBasedControllerAuthorizationTest < Test::Unit::TestCase
  def test_admin_only
    assert(RoleBasedController.access_list[:admin_only] == [:admin])
    assert(RoleBasedController.access_list[:user_and_admin] == [:admin, :user])
  end
end
```

Our unit test should pass, assuming we've done everything correctly so far. Now that we've established a way to express roles on a per-action basis in our controllers, we need to perform authorization checks using this data. Let's add the filter code to our `RoleBasedControllerAuthorization` module:

[Download](#) `authorization/lunchedin_old/app/security/role_based_controller_authorization.rb`

```
def authorization_filter
  ▶ user = User.find(:first,
  ▶   :conditions => ["id = ?", session[:user_id]])

  ▶ action_name = request.parameters[:action].to_sym
  ▶ action_roles = self.class.access_list[action_name]
```

```

▶ if action_roles.nil?
  logger.error "You must provide a roles declaration\
    or add skip_before_filter :authorization_filter to\
    the beginning of #{self}."
  # We deliberately did not return any information to the user here.
  # This is considered a programmer error. We could also raise an
  # Error or Exception here. If we wanted to switch to a blacklist
  # version of the role based access control, this would allow the
  # Action. However, this violates two of our principles; Least
  # Privilege, Whitelist
  redirect_to :controller => 'root', :action => 'index'
  return false
elsif action_roles.include? user.role.name.to_sym
▶   return true
else
  logger.info "#{user.user_name} (role: #{user.role.name}) attempted to access\
    #{self.class}##{action_name} without the proper permissions."
  flash[:notice] = "Not authorized!"
  redirect_to :controller => 'root', :action => 'index'
▶   return false
end
end
end

```

The `authorization_filter()` executes several actions:

- It looks up the user from the session. We should always include this step if we chain an authentication filter before the authorization filter. Depending on our application's requirements, we might want to handle the case where a user is nil.
- It retrieves the action from the request.
- It retrieves the roles associated with the action.
- If the action's roles are nil, then it logs an error and redirects the request.
- If the user's role exists in the method access list, it returns true and ends the authorization-filter process.
- Finally, it fails securely. The default behavior of the filter is to redirect to the root controller.

There are many variations that we can implement in this filter. Depending on what our application's requirements dictate, we can alter the behavior of the authorization failures. However, we should always preserve a fail-close model in our authorization logic.

Previously, we performed an authorization check in our CommentsController:

[Download](#) authentication/lunchedin_old/app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController

  before_filter :check_authorization

  def check_authorization
    unless @user and @user.role.name == 'admin'
      flash[:notice] = "Not authorized!"
      redirect_to :controller => 'root', :action => 'index'
    end
  end

  def remove
    Comment.delete(params[:id])
    render :update do |page|
      page.visual_effect :fade, "comment_#{params[:id]}"
    end
  end
end
```

Let's revise this code with our new role-based authorization module:

[Download](#) authorization/lunchedin_old/app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController

  ▶ roles :admin
  def remove
    Comment.delete(params[:id])
    render :update do |page|
      page.visual_effect :fade, "comment_#{params[:id]}"
    end
  end
end
```

This change reduces the complexity of the code dramatically, while also increasing its readability and maintainability.

Let's add an additional test to the unit test of our comments controller:

[Download](#) authorization/lunchedin_old/test/functional/comments_controller_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'
require 'comments_controller'

# Re-raise errors caught by the controller.
class CommentsController; def rescue_action(e) raise e end; end
```

```

class CommentsControllerTest < Test::Unit::TestCase

  fixtures :users

  def setup
    @controller = CommentsController.new
    @request     = ActionController::TestRequest.new
    @request.session[:user_id] = users(:dave).id
    @response    = ActionController::TestResponse.new
  end

  # Replace this with your real tests.
  def test_truth
    assert true
  end

  def test_roles
    assert(CommentsController.access_list[:remove] == [:admin])
  end

end

```

Next, let's add a roles "viewer" for our application. This viewer will give us a nice view of actions and roles across all of our controllers. Let's begin by adding a controller to host this functionality:

```
script/generate controller Authorization
```

Now we add our logic to present the controllers, actions, and roles to the view:

[Download](#) authorization/lunchedin_old/app/controllers/authorization_controller.rb

```

class AuthorizationController < ApplicationController
  roles :admin

  def index
    @controllers =
      ActionController::Routing.possible_controllers.delete_if do |controller|
        is_it_a? controller
      end
    @controllers.collect! {|c| controller_class c }
  end

  private
  def controller_class(controller)
    "#{controller.camelize}Controller".constantize
  end

  def is_it_a?(controller)
    the_class = controller_class(controller)
    !the_class.is_a?(Class) ||

```

```

▶ !the_class.ancestors.include?(ApplicationController) ||
▶ !the_class.included_modules.include?(RoleBasedControllerAuthorization)
end
end

```

The preceding code sample has several goals:

- To declare our roles to be :admin only
- To gather all controllers through the ActionController::Routing class
- To delete all references to noncontroller classes and controllers that do not include our authorization module

Now we can create a view:

[Download](#) authorization/lunchedin_old/app/views/authorization/index.html

```

<!-- This is obviously designed for functionality - not beauty. -->
<% for controller in @controllers %>
<table style="width: 100%; font-size: 10px">
  <th style="background-color: #FFCF5C"><%= controller %></th>
  <% for method in controller.access_list.keys %>
  <tr style="background-color: #FFFB93">
    <td>#<%= method %></td>
  </tr>
    <% if !controller.access_list[method.to_sym].nil? %>
      <% for role in controller.access_list[method.to_sym] %>
        <tr><td style="padding-left: 15px;">:<%= role %> </td></tr>
      <% end %>
    <% end %>
  <% end %>
</table>
<% end %>

```

At this point, we should be able to view all of our controllers, actions, and roles (see Figure 7.8, on the following page).

We also need to make role declarations in the rest of our controllers. Most of them will be :user, :admin. It is a good exercise to debug the rest of the controllers, write unit tests to assert permissions, and validate that our controllers with roles behave as intended.

Beyond Controller-Based RBAC

A controller-based RBAC covers 90 percent of most application's authorization requirements, but we must also consider the user interface—we need to make sure we don't render links and other user interface controls that result in an authorization denial for the end user.

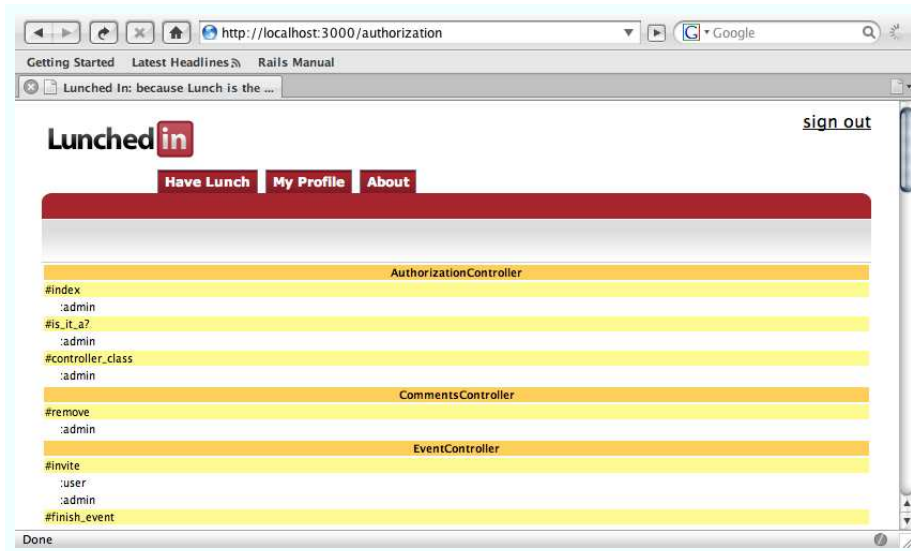


Figure 7.8: Role declarations

We can do this by providing something functionality along these lines in a view helper:

```
def if_authorized :controller, :action => :foo
  ...
end
```

We will leave this part as an exercise to the reader. The principle is very similar to our authorization filter. Based on the parameters, a lookup of the current user's role, combined with the controller and action, provides enough data to the view helper to determine whether it should render the contents.

7.6 A Simple Approach to Model Security

There are many cases when RBAC doesn't cover a specific requirement. If the application's requirements mandate granular runtime control over all data, then we have two basic options:

- We can implement a data-access class that is driven by an access-control list approach. The challenge with this approach is the fact that our `has_many` macros will bypass the data-access class. But

this might still be an acceptable option if its children inherit permission implicitly.

- We can write a library to intercept all ActiveRecord calls. This is the approach that Bruce Perens took with ModelSecurity. The challenge with this approach is that Rails changes over time, and it could be a maintenance nightmare. Also, ActiveRecord's aggressive approach in creating complex relationships with all the macros it provides could make creating a secure solution very challenging.

But the truth is, most applications don't require such granular control at the model level. In fact, the biggest gap in the RBAC model is generally between users of the same role. For example, if a user creates a blog entry in a multiuser blog system, that user owns the data. That user should be able to delete her blog entry, update it, and so on.

Fortunately, we can build this kind of authorization into our application quite easily. Let's look at how to add an `edit_my_profile()` action to our ProfileController. All users are members of the `:user` role, so we must prevent users from editing other users' profiles.

Let's begin by adding a couple methods to our ProfileController:

[Download](#) authorization/lunchedin_i2_old/app/controllers/profile_controller.rb

```
def edit_my_profile
  render :action => 'edit'
end

def save_my_profile
  if @user.update_attributes(params[:user])
    flash[:notice] = 'account successfully updated'
    redirect_to :action => 'my_profile', :id => @user
  else
    render :action => 'edit'
  end
end
```

So where is the extra authorization code? The answer: we don't need any. Let's look another look at the application controller we created in Chapter 6, *Authentication: Decentralized Authentication*, on page 95:

[Download](#) authorization/lunchedin_i2_old/app/controllers/application.rb

```
def check_authentication
  logger.debug "FILTER> #{self}#check_authentication called"
  if session[:user_id]
    begin
      @user = User.find(session[:user_id]) and return
```




```

    rescue
      reset_session
    end
  end
end
# You must return false for this condition.
# We do not want our authorization filter to be called
redirect_to :controller => "login", :action => "login" and return false
end

```

We query the `@user` object automatically for each authenticated request. This code uses `session(:user_id)` to fetch the `@user`, so it does not require a parameter provided from the user request. We can take advantage of this fact in a few ways:

- We can eliminate repetitive code in each of our methods that deal with a `@user`.
- We can scope all `:user` role methods to that particular user. This allows us to access data *through* the user, as opposed to using some request parameter—such as `user_id`.

Now let's add an edit link to the profile view:

Download authorization/lunchedin_i2_old/app/views/profile/show.rhtml

```

<tr>
  <td class="label">Zip Code:</td>
  <td><%= @user.zip_code %></td>
</tr>
<tr>
  <td><%= link_to "Edit", { :action => :edit_my_profile } %></td>
</tr>

```

Next, we can add a view:

Download authorization/lunchedin_i2_old/app/views/profile/edit.rhtml

```

<h1>Have lunch with us</h1>

<h2>Edit Profile</h2>

<%= error_messages_for 'user' %>

<% form_tag :action => 'save_my_profile' do %>
<%= render :partial => 'form' %>

<div style="padding: 10px;">
  <%= submit_tag 'Save' %>
</div>
<% end %>

```

Finally, we can add the partial:

[Download](#) authorization/lunchedin_i2_old/app/views/profile/_form.html

```
<div class="highlighted-form">
  <table class="form">
    <tr>
      <td class="label">First Name:</td>
      <td><%= text_field('user', 'first_name', :size => 16) %></td>
    </tr>
    <tr>
      <td class="label">Last Name:</td>
      <td><%= text_field('user', 'last_name', :size => 16) %></td>
    </tr>
    <tr>
      <td class="label">Email:</td>
      <td><%= text_field('user', 'email') %></td>
    </tr>
    <tr>
      <td class="label">User Name:</td>
      <td><%= text_field('user', 'user_name', :size => 14) %></td>
    </tr>
    <tr>
      <td class="label">Password:</td>
      <td><%= password_field('user', 'password', :size => 14) %></td>
    </tr>
    <tr>
      <td class="label">Confirm Password:</td>
      <td><%= password_field('user', 'password_confirmation', :size => 14) %></td>
    </tr>
    <tr>
      <td class="label">Zip Code:</td>
      <td><%= text_field('user', 'zip_code', :size => 6) %></td>
    </tr>
  </table>
</div>
```

There is a tendency when building applications to *add* authorization and authentication after the application is well under development. Because we made a good design decision in our authentication filter, we're in a strong position to reap the benefits of scoping the authority of :user role actions.

Cryptography is about communication in the presence of adversaries.

► Ron Rivest

Chapter 8

Data Protection Using Cryptography

Data protection is about keeping information safe. For example, you want to prevent the rest of the world from learning your passwords, credit card numbers, and Social Security numbers—any of these could cause you serious distress if they were to fall into the wrong hands. An additional complication: sometimes we need to communicate information across public channels, such as the Internet.

Security researchers seem fond of pointing out that cryptography does not ensure a secure system. This is true, but *not using* cryptography won't deliver a secure system, either—and it could be a recipe for disaster! Encryption is an important tool that we can use to hide information from potential eavesdroppers.

In this chapter, we'll focus on techniques we can use to prevent others from spying on our information. When protecting data, we must consider, not just whether an attacker can *read* protected information, but whether that information has been *altered*. In this chapter, we will cover the following:

- How to verify information using message digests
- How to use the Ruby OpenSSL bindings to encrypt and decrypt information
- Guidelines for selecting cryptography types and algorithms
- How to leverage ActiveRecord in order to encrypt and decrypt data transparently

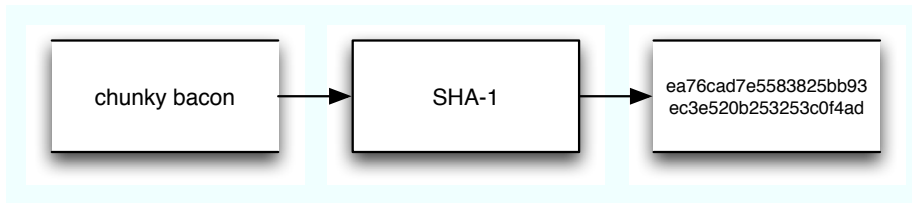


Figure 8.1: A one-way hash function

8.1 Message Digests

We touched on protecting passwords earlier in Chapter 6, *Authentication: Decentralized Authentication*, on page 95. As you might recall, message digests are unique one-way hashes of a variable-length input message. These hashes act like fingerprints for a message; no two *should* be alike. We'll talk more about this later in the chapter.

For example, assume you ran the input, chunky bacon, through the SHA1 digest function (see Figure 8.1). Let's re-create this digest using the OpenSSL module in the Ruby standard library:

```

irb(main):001:0> require 'openssl'
=> true
irb(main):002:0> OpenSSL::Digest::SHA1.new('chunky bacon')
=> ea76cad7e5583825bb93ec3e520b253253c0f4ad
  
```

In this example, we instantiate a SHA1 digest class, passing the message we want returned. SHA1 is not the only algorithm we can use. For example, the OpenSSL Ruby library supports additional functions in the same form:

[Download](#) data_protection/digests.rb

```

require 'openssl'

%w(DSS1 MD2 MD4 MD5 MDC2 RIPEMD160 SHA SHA1).each do |digest|
  h = eval("OpenSSL::Digest::#{digest}.new('chunky bacon')").hexdigest
  puts "#{digest}: #{h}"
end

data_protection> ruby digests.rb
DSS1: ea76cad7e5583825bb93ec3e520b253253c0f4ad
MD2: 414b03d14ed5a47ae0aea8cdac470578
MD4: 275043efc7c862d50b156a4a3447ea02
MD5: 792edef71cf5dbb64592311985d3daee
MDC2: f9f8e4bd7cf17fa4190422a22bd2d35e
  
```

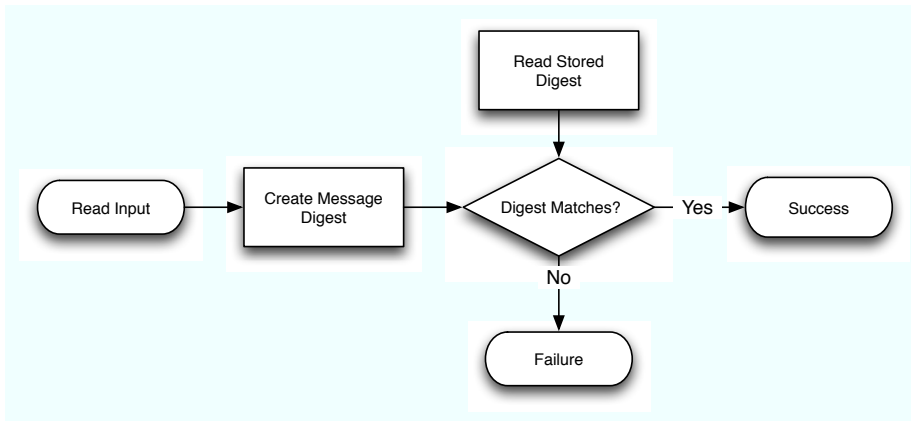


Figure 8.2: Message digest verification

```

RIPMD160: b43bacf76eacaab9bb87f151e4cc738aa4895086
SHA: 395cb5b149cf7fc7720297570e03360c0bd84dff
SHA1: ea76cad7e5583825bb93ec3e520b253253c0f4ad
  
```

The standard library includes many standard Digest classes. In addition to using subclasses such as SHA1, we can also use Digest directly:

```

irb(main):001:0> require 'openssl'
=> true
irb(main):002:0> OpenSSL::Digest::Digest.new('SHA1', 'chunky bacon')
=> ea76cad7e5583825bb93ec3e520b253253c0f4ad
  
```

We don't need to use any other cryptography library in Ruby—just about everything you might need is available in the OpenSSL classes in the Ruby standard library. Using hand-rolled cryptography algorithms is never a good idea!

Hashes and Passwords

In an ideal world, a password database will never be visible to the public. However, in the case of a system or application vulnerability, this data may become visible to an attacker. Message digests help us address this problem by providing a *fingerprint* we can verify against.

The steps used to verify a password are shown in Figure 8.2. What makes this a powerful concept is that we can determine whether a password matches the fingerprint without having to know the actual value!

We can also employ a *salt*, which is a small amount of random data added to the original message before the digest function processes it. The salt is not a secret, and you need it to verify that a message matches the digest. We cover salts in depth in Section 6.2, *Adding Salts to the Passwords*, on page 102.

When Hashes Attack

Earlier we said the hash values produced for different inputs should themselves be different. This determines the quality of a hash function. When two inputs are found that return the same message digest, a *collision* is said to have occurred. We know collisions exist because of how hash functions are calculated; in fact, all hash functions have an infinite number of collisions! How can this be? Hash functions yield a fixed-length output with a variable-length input, so an infinitely large input divided by a finite number is infinity. Don't be alarmed just yet. The goal is to create a hash function where collisions can't occur. Collision attacks do exist for many hash functions in use today, but the context of their use plays a larger role in selection.

Choosing a Hash Function

We have only a handful of options when it comes to choosing a cryptographic hash function. If our application must interoperate with an existing application, the choice is typically made for us. For new development, the SHA family of hash digests is preferred. While recent developments signal a possible collision attack against the SHA1 algorithm, it remains the most widely accepted algorithm. It is also the current U.S. standard. That could (and probably will) change in the future, but the likely replacements are SHA-2 variants (SHA-224, SHA-256, SHA-384, or SHA-512).

We, the authors, believe you should avoid the MD5 algorithm. Many well-known collision attacks exist for MD5, and the cost of using the SHA family is minimal.

8.2 Encrypting Data

It's not uncommon to hear someone say, "We must encrypt this data." In most cases, the person who says it expects the person handling the encryption to use *block cipher* encryption. Block ciphers are black boxes that house the algorithms. We use block ciphers to encrypt information

called the *plaintext*. The plaintext is whatever we want to protect. Block ciphers operate on a fixed length of data, called a *block*.

Many variable length block ciphers exist, but they are not the norm.

Types of Cryptography

For the purposes of the book, we can divide cryptography into two main groups: symmetric and asymmetric. Each group has its own strengths and weaknesses, and neither group is always better than the other.

Symmetric cryptography

Symmetric cryptography gets its name from the way the same key is used to encrypt and decrypt the information protected. We refer to encrypted information as the *ciphertext*.

The following example illustrates basic usage of the OpenSSL module in the Ruby standard library:

[Download](#) data_protection/aes_test.rb

```
Line 1 #!/bin/env ruby
- require 'openssl'
-
- plain_text = "To the limit!"
5 algorithm = "AES-128-CBC"
- aes = OpenSSL::Cipher::Cipher.new(algorithm)
- key = aes.random_key
-
- puts %(clear plain_text:   "#{plain_text}")
10 puts %(symmetric key:    "#{key.inspect}")
- puts %(cipher alg:      "#{algorithm}")
-
- aes.encrypt(key)
- cipher_text = aes.update(plain_text)
15 cipher_text << aes.final
- puts %(encrypted plain_text: #{cipher_text.inspect})
-
- # clear the aes cipher
- aes.reset
20
- # decrypt and display output
- aes.decrypt(key)
- out = aes.update(cipher_text)
- out << aes.final
25 puts %(decrypted plaintext: "#{out}")

data_protection> ruby aes_test.rb
clear plain_text:   "To the limit!"
symmetric key:    "\"\204\016<#p\216\310;\256\227\366\2064\017#\355\""
cipher alg:      "AES-128-CBC"
encrypted plain_text:  "\376KS\001I\020\317\307i\202)\271+\0225K"
decrypted plaintext: "To the limit!"
```

Joe Asks...

Why Should I Use an OpenSSL random_key?

The question that really needs to be answered is this: why should we use the `OpenSSL::Cipher.random_key()` class instead of the `Kernel::rand()` class? The OpenSSL library uses a pseudo-random number algorithm designed specifically for cryptography. Producing a cryptographically random number generator is a nontrivial, computationally intensive task. Ruby's `Kernel::rand()` class is better than many others due to its underlying Mersenne Twister algorithm, but our goal was never to replace a true cryptographically random number generator. In short, we should use OpenSSL because it provides an easy way to defend against attacks that rely on sequence predictability.

This code begins by instantiating the `OpenSSL::Cipher::Cipher` class, passing the algorithm, key length, and block mode to the `initialization()` method. In our example, we use AES, 128-bit key length, and block mode *cipher block chaining*.

We can generate a random key from an initialized Cipher object by invoking the `random_key()` method. We can adjust the length of the key based upon the key length value passed to the `initialize()` method. Encryption and decryption occur by triggering the cipher mode with the `encrypt(key)` and `decrypt(key)` methods. Once we put the cipher into a mode, we call the `update()` method, passing it the plaintext or ciphertext.

The next example illustrates how to use an *initialization vector*, or IV. An initialization vector is an array of bits used to introduce additional randomness to the ciphertext stream:

[Download](#) `data_protection/aes_iv_test.rb`

```
Line 1 #!/bin/env ruby
- require 'openssl'
-
- plain_text = "To the limit!"
5 algorithm = "AES-128-CBC"
- aes = OpenSSL::Cipher::Cipher.new(algorithm)
- key = aes.random_key
- iv = aes.random_iv
-
```



```

10 puts %(clear plain_text:   "#{plain_text}")
-   puts %(symmetric key:   #{key.inspect})
-   puts %(initialization vector: "#{iv.inspect}")
-   puts %(cipher alg:     "#{algorithm}")
-
15 aes.encrypt(key, iv)
-   cipher_text = aes.update(plain_text)
-   cipher_text << aes.final
-   puts %(encrypted plain_text: #{cipher_text.inspect})
-
20 # clear the aes cipher
-   aes.reset
-
-   # decrypt and display output
-   aes.decrypt(key, iv)
25 out = aes.update(cipher_text)
-   out << aes.final
-   puts %(decrypted plaintext: "#{out}")

data_protection> ruby aes_iv_test.rb
clear plain_text:   "To the limit!"
symmetric key: "]"364\323T\317\377\216\264\221\216b\314;\237\021\267"
initialization vector: ""\004\332\235\320\315\250K\244\227}\232`\326_\237H""
cipher alg:        "AES-128-CBC"
encrypted plain_text: "\3162jqS\224\305B\347:\267\350><N\203"
decrypted plaintext: "To the limit!"

```

You can see the only significant change between these two examples on line 15. To use an initialization vector, we must pass it as the second parameter to the `encrypt()` and `decrypt()` methods of the `Cipher` class.

When using block ciphers that support initialization vectors, we must take care to use those vectors! Initialization vectors add additional strength to the ciphertext, and they can help reduce the amount of information that can be gleaned from the encrypted information. For example, if we did not use an initialization vector and two plaintexts were the same, the ciphertext of each message would be the same. This might not pose an immediate problem, but it could become a serious issue if someone learned one of the plaintext values.

Keep in mind the initialization vector is not a secret. It's an array of bytes required to decrypt the message. If we were to lose this information, we would lose any means of decrypting the message. We should store our initialization vector with the ciphertext. Some developers choose to prepend the IV to the ciphertext, while others prefer to create a separate field to hold the value. Each approach works, but prepending the value to the ciphertext confers a slight advantage because an attacker would need to determine its location.

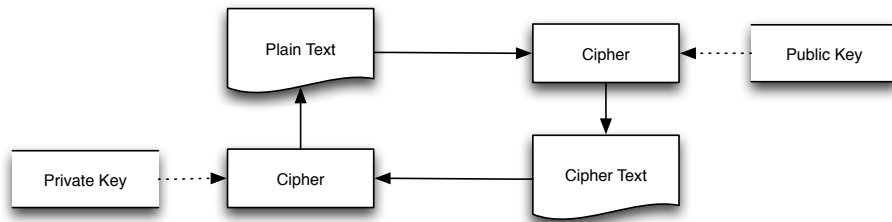


Figure 8.3: Asymmetric encryption

We should use symmetric encryption when we face each of the following situations:

- We need to encrypt and decrypt large amounts of data in a timely manner. Symmetric cryptography is an order of magnitude faster than asymmetric cryptography.
- We do not need key distribution because the same application encrypts and decrypts the information.
- Key exchange can occur out of band, in a secure manner.

Asymmetric cryptography

Asymmetric cryptography uses two keys. Each key can be used to decrypt ciphertexts, as long as the other key was used to encrypt it (see Figure 8.3). This approach enables bidirectional message communication without divulging the private key. This is a remarkable concept. Key distribution is a serious pain, and asymmetric cryptography can help us alleviate it.

Let's look at an example that uses the RSA algorithm to encrypt and decrypt a message:

[Download](#) data_protection/rsa_test.rb

```

Line 1  #!/bin/env ruby
-      require 'openssl'
-      include OpenSSL
-
5      rsa = PKey::RSA.generate(512)
-      txt = "Chunky Bacon"
-      puts "Plain Text: #{txt}"
-

```

```

- cipher_text_with_priv = rsa.private_encrypt(txt)
10 cipher_text_with_pub = rsa.public_encrypt(txt)
-
- puts "== Plain Text Decrypted With Public Key =="
- puts pt_with_pub = rsa.public_decrypt(cipher_text_with_priv)
-
15 puts "== Plain Text Decrypted With Private Key =="
- puts pt_with_priv = rsa.private_decrypt(cipher_text_with_pub)

data_protection> ruby rsa_test.rb
Plain Text: Chunky Bacon
== Plain Text Decrypted With Public Key ==
Chunky Bacon
== Plain Text Decrypted With Private Key ==
Chunky Bacon

```

On line 5, we instantiate an RSA public key. Next, on lines 9 and 10, we generate two ciphertexts using the private and public keys. Finally, on lines 13 and 16, we decrypt each ciphertext using the opposite key.

Choosing the right cryptography

So far we've learned about symmetric block ciphers like AES and asymmetric algorithms like RSA. Next, we need to determine when to use a specific algorithm and the consequences of choosing one over the other.

We can choose from several well-respected algorithms for symmetric block ciphers. The current number-one choice is the *advanced encryption standard*, or *AES*. AES was established using a contest where various groups submitted proposals, and the top submission was chosen. This approach contrasts distinctly with a design-by-committee approach, where any specific vision is normalized to the lowest common denominator. An algorithm known as *Rijndael* was selected as the winner, and it now bears the name AES. AES is the current U.S. government standard for symmetric encryption, and it is ideal for cases where third parties do not need to send or receive encrypted data.

If choosing the same algorithm the U.S. government uses makes you uncomfortable, don't fret; you have many other options. *Serpent* was another AES finalist, as was the *Twofish* algorithm. These algorithms currently represent a sound approach, but new attacks are being developed constantly, so what's sound for now might not hold true for the future. The *DES* algorithm is a good example of this; attacks targeting the DES algorithm have rendered it undesirable, except in the case of backwards compatibility.



Joe Asks...

Why Is ECB Undesirable?

From the previous examples, you might have noticed some extra information passed into the `initialize()` method of the Cipher classes. These values instruct the cipher how to handle messages that grow beyond a single block. Block sizes are typically in the eight-byte range, so this situation occurs frequently:

```
$ ruby ecb_versus_cbc.rb
```

```
aes with mode cbc
```

```
"\267X\254I\216M\021\255\300`\251\220\272\243"  
"\036\237\234\224p\324D{i\036\266\342MTiV"  
"\203r\246\232\220\301=\205\267cN\314\376n?. "  
"\331\v\225\024j\301\310\037I C\222>\246\274\230T"  
"s\213n\212\270\232\022\230v\327\035\215\237\ez\005"
```

```
aes with mode ecb
```

```
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"  
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"  
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"  
"\0356\215\017\332\246\034\000$\357\252D\f\016I\374"  
"d\026\325\252d\305D\304\031H C\310\310\215{"
```

In our example we can see that the plaintext is identical, but the corresponding ciphertext is not. The output of each line corresponds to one AES block. The first group of lines displays a pseudo-random distribution of block contents; it used *cipher block chaining* to achieve that effect. The second example shows four equal consequence blocks within the cipher text; it appears this way because the cipher wasn't instructed to extend its algorithm across blocks. This is bad because it allows for known plaintext attacks or attacks where the part of the message is known. An example of this would be attacking protocols that contain standard headers included in the plaintext. By limiting the protection to a single block and knowing the order of certain parts of the plaintext, we can narrow drastically how much of the plaintext must be deciphered.

To sum up, *don't use ECB!*

If an application requires that you exchange messages with encrypted information, then managing symmetric keys will become a nightmare once any party involved is no longer trusted. This is where asymmetric, or public-key cryptography, shines. Unfortunately, using public-key cryptography is more difficult because it has significantly more going on within it. Given this fact, we must concede proper encryption is beyond the scope of this book. That said, both RSA and DSA serve as excellent starting points for encryption.

8.3 Using Cryptography with ActiveRecord

Now that we're acquainted with the basics of using cryptography with Ruby, it's time to improve our data-protection implementation. One way to improve it: we can use extension techniques common to the construction of Rails.

ActiveRecord Encryptor

The book *Agile Web Development with Rails* [RTH08], includes an example of leveraging ActiveRecord hooks to provide transparent encryption and decryption of a model's properties. That example focuses on teaching developers how to use ActiveRecord callbacks, but it leaves out the finer details of using cryptography. Our next task is to provide an implementation for that book's example.

Add a security question

When authenticating users, it is important to provide additional factors. One example of this involves incorporating a security question to further challenge the user. Let's add a security question to LunchedIn that helps us authenticate a user who has lost her password.

We can run the following command to create a Migration class that builds the additional fields required for our security question:

```
$ ruby script/generate migration add_secret_user_question
```

Next add the following code to 011_add_secret_user_question.rb:

Download `data_protection/lunchedin/db/migrate/011_add_secret_user_question.rb`

```
class AddSecretUserQuestion < ActiveRecord::Migration
  def self.up
    add_column "users", "secret_question", :string
    add_column "users", "secret_answer", :string
  end
end
```

```

def self.down
  remove_column "users", "secret_question"
  remove_column "users", "secret_answer"
end
end

```

This migration adds the `secret_question` and `secret_answer` columns to the `users` table. We can use these columns to store the plaintext question and the ciphertext answer.

Creating the encryptor

We will use the AES symmetric block cipher to encrypt the answer to the associated secret question. Create the `encryptor` directory under the `app/security` folder. If there is no `security` folder under the `app` folder, we need to go ahead and create this now. Next, let's create a file named `routines.rb` and add the following code to it:

Download `data_protection/lunchedin/app/security/encryptor/routines.rb`

```

module Encryptor
  module Routines
    def encrypt(plain_text, key, opts)
      cipher = OpenSSL::Cipher::Cipher.new(algorithm(opts))
      iv = cipher.random_iv
      cipher.encrypt(key, iv)
      cipher_text = cipher.update(plain_text)
      cipher_text << cipher.final
      cipher_text.insert(0, iv)
      Base64.encode64(cipher_text)
    end

    def decrypt(cipher_text, key, opts)
      cipher = OpenSSL::Cipher::Cipher.new(algorithm(opts))
      decoded_cipher_text = Base64.decode64(cipher_text)
      cipher.decrypt(key, decoded_cipher_text.slice!(0..15))
      out = cipher.update(decoded_cipher_text)
      out << cipher.final
    end

    def algorithm(opts)
      "#{opts[:cipher]}-#{opts[:keylength]}-#{opts[:block_mode]}"
    end
  end
end

```

In the `Encryptor::Routines` module, we define the methods to use for encrypting and decrypting the plaintext and ciphertext. These methods require no internal state, so we can mix them safely into many types of classes.

Next, we create a file called `active_record.rb` under the `app/security/encryptor` directory and add the following code:

[Download](#) `data_protection/lunchedin/app/security/encryptor/active_record.rb`

```
module ActiveRecord
  class Base
    def self.encrypted_key=(key)
      write_inheritable_attribute :encrypted_key, key
    end

    def self.encrypted_key
      read_inheritable_attribute :encrypted_key
    end
  end
end
```

The preceding code adds the `encrypted_key()` and `encrypted_key=(key)` class methods to `ActiveRecord::Base`. These class methods serve as the accessors for key storage.

In the body of each method, we call `read_inheritable_attribute(attribute)` and `write_inheritable_attribute(attribute, value)`. These two ActiveSupport methods facilitate retrieval and storage of *class instance variables*. We can use these to define a base value, but we need to keep in mind that subclasses can redefine this base value without changing its parent's value.

Next, we add the following class definition to the same `active_record.rb` file:

[Download](#) `data_protection/lunchedin/app/security/encryptor/active_record.rb`

```
module Encryptor
  module ActiveRecord
    class Encryptor
      include ::Encryptor::Routines

      @@default_options = {
        :cipher => 'AES', :block_mode => 'CBC', :keylength => 128
      }

      def self.default_options
        @@default_options
      end

      def initialize(field, opts = {})
        @field = field
        @options = @@default_options.merge(opts)
      end
    end
  end
end
```

```

def before_save(model)
  unless model[@field].blank?
    key = model.class.encrypted_key
    model[@field] = encrypt(model[@field], key, @options)
  end
end

def after_save(model)
  unless model[@field].blank?
    key = model.class.encrypted_key
    model[@field] = decrypt(model[@field], key, @options)
  end
end

alias_method :after_find, :after_save
end
end
end

```

We use the `Encryptor::ActiveRecord::Encryptor` class as a delegate of ActiveRecord callback hooks.¹ We can use these callbacks to intercept the lifecycle events associated with the retrieval and persistence of ActiveRecord objects. After the object is retrieved from the database, we decrypt the ciphertext and replace the attribute's value with the plaintext. A similar operation occurs right before the object is saved, except that here we can encrypt the plaintext and replace the attribute with the ciphertext.

Now we need to add a class method to `ActiveRecord::Base` that will let us declare that a field can be persisted only in an encrypted state. Next, let's create the `class_methods.rb` file under the `app/security/encryptor` directory and add the following code:

Download `data_protection/lunchedin/app/security/encryptor/class_methods.rb`

```

module Encryptor
  module ActiveRecord
    def self.included(base)
      base.extend(ClassMethods)
    end

    module ClassMethods
      def encrypt(attribute, options = {})
        encryptor = Encryptor.new(attribute, options)
        before_save(encryptor)
        after_save(encryptor)
        after_find(encryptor)
      end
    end
  end
end

```

1. ActiveRecord callbacks are defined in <http://api.rubyonrails.org/classes/ActiveRecord/Callbacks.html>.


```

        define_method(:after_find) { }
      end
    end
  end
end

```

In this code, we declare the `Encryptor::ActiveRecord` module that contains the `included(base)` class method and the `ClassMethods` module. The `included(base)` method is a callback that is invoked *whenever the receiver is included in another module or class*.² We define this method so we can call `extend(module)` on the receiving module or class. This effectively allows us to define class methods in a module and mix them into any other class.

In our example, we mix the `Encryptor::ActiveRecord::ClassMethods` module into `ActiveRecord::Base`. This enables us to invoke our `encrypt(attribute, options)` method from any model object.

This method creates an `Encryptor` instance for a specific attribute and registers itself with the `before_save()`, `after_save()`, and `after_find()` callbacks.

Now we need to set up a file to load and mix in the dependent Ruby files for our encryptor implementation. Create the `encryptor.rb` file in the `app/security` folder, and then add the following code:

Download `data_protection/lunchedin/app/security/encryptor.rb`

```

require "encryptor/routines"
require "encryptor/active_record"
require "encryptor/class_methods"

```

```
ActiveRecord::Base.send(:include, Encryptor::ActiveRecord)
```

In this file, we add the `require` statements for the dependent files and invoke the `include(klass)` method, passing it the `Encryptor::ActiveRecord` module.

Integrating ActiveRecord encryptor

Now that we have constructed the encryptor, we can integrate it into our `LunchedIn` application. If we didn't do so in a previous chapter, we need to enable the `security` folder under `app` folder.

2. See the module documentation at <http://www.ruby-doc.org/core/classes/Module.html>.

We can accomplish this by editing the `environment.rb` file so it looks like this:

Download `data_protection/lunchedin/config/environment.rb`

```
# Add additional load paths for your own custom dirs
config.load_paths += %W( #{RAILS_ROOT}/app/security )
```

Now let's create the initializer for our encryptor. Let's begin by creating a file named `security.rb` under `config/initializers` and adding the following code:

Download `data_protection/lunchedin/config/initializers/security.rb`

```
require 'encryptor'

if RAILS_ENV['production']
  ActiveRecord::Base.encrypted_key = File::read('/etc/lunchedin.key')
else
  # test and development configuration
  ActiveRecord::Base.encrypted_key = '1234567890abcdef'
end
```

This sets a default key for the development and test configurations. When LunchedIn is run in production, the key is read from the filesystem.

Next, let's declare that the `secret_answer` field must be encrypted when persisted. We can do this by adding the following class method call to the `user.rb` file:

Download `data_protection/lunchedin/app/models/user.rb`

```
class User < ActiveRecord::Base
  encrypt :secret_answer
```

Now let's add a test case to `user_test.rb` under `test/unit`:

Download `data_protection/lunchedin/test/unit/user_test.rb`

```
def test_encryptor
  @user = users(:bob)
  @user.secret_answer = 'Porsche Carrera GT'
  assert @user.save
  @answer = User.connection.select_all(
    "select secret_answer from users where id = 1"
  )
  cipher_text = @answer.first['secret_answer']
  opts = Encryptor::ActiveRecord::Encryptor.default_options
  @plain_text = decrypt(cipher_text, User.encrypted_key, opts)
  assert_equal 'Porsche Carrera GT', @plain_text
end
```

At this point, we can run through our unit tests:

```
data_protection> rake test
```

Keep in mind that we used this example to illustrate the mechanism of symmetric cryptography with ActiveRecord. A small number of simple, secret questions isn't a very strong test of authenticity—just ask Paris Hilton!³

Ruby's power enables us to provide transparent confidentiality of ActiveRecord properties with only a single class method invocation of the model object. To accomplish this, we used ActiveRecord callbacks to intercept persistence life-cycle events.

Protecting data in the context of a typical Rails application is well within the power of the OpenSSL bindings in the Ruby standard library. We, the authors, recommend that you avoid third-party libraries that duplicate the functionality within OpenSSL. Our reasoning is this: OpenSSL is a very stable project that receives significant scrutiny from the open source community. This doesn't assure that nothing will go wrong, but it definitely inspires greater confidence.

3. <http://www.macdevcenter.com/pub/a/mac/2005/01/01/paris.html>

Digital Signatures and Email

We have touched on data-protection topics; now it's time to build on these ideas and strengthen our protection techniques using *digital signatures*.

Specifically, we're going to cover X.509 digital certificates, the same certificates used to secure HTTP traffic with Transport Layer Security (TLS).

Transport Layer Security is often referred to by its predecessor protocol, Secure Sockets Layer (SSL). TLS was designed to prevent eavesdropping by establishing an authenticated and secure channel between parties. In this chapter, we will cover how to use techniques similar to what TLS uses to protect messages between two parties. For example, we will cover how to do the following:

- Create and verify digital signatures in the Ruby language
- Use digital signatures in the context of email messages
- Add signed email messages to LunchedIn
- Authenticate received mail by verifying digital signatures

9.1 Digital Signatures

As a resident of the World Wide Web, you've probably heard about digital signatures. Digital signatures are crypto-systems built using message digests and asymmetric cryptography. *Crypto-system* is just a fancy way of describing a collection of cryptography techniques used as a system. The reasoning behind this is simple: coordinated properly, the elements of a cryptography system will form a stronger defense than

individual techniques, enabling you to mitigate the various downsides of individual techniques that you combine into a system. For example, TLS is a cryptographic system that combines the use of asymmetric ciphers, certificates, and symmetric ciphers to achieve several goals, including authentication, confidentiality, and performance.

In this chapter, we will focus on digital signatures as implemented by the ITU-T X.509 standard. While there are many different mechanisms for creating a *digital substitute for a signature*, the X.509 standard is the most widely adopted.

[Download at WoweBook.com](#)

X.509

We create digital signatures using a message called a *certificate*. The standard for this is X.509. A certificate is an organization's or individual's digital representation. These representations are issued by a third party, called a *certificate authority*. Certificate authorities are responsible for verifying identities, tracking compromised certificates, and issuing certificates. At least, this is how it works in theory.

In reality, public key infrastructure (PKI) is more about paying a fee to certificate authorities so that our applications won't display warning messages when browsing to sites using certificates they created themselves. How does this work?

Any application using X.509 certificates maintains a database that contains the root certificates of major certificate authorities. Certificate authorities pay for the privilege of being included in major applications such as web browsers, mail clients, and mobile phones.

The reason only root certificates are included in these browsers et al. is because all certificates are issued under the authority of another when they are created. This means that we have a chain of trust. An application—your web browser, for example—trusts the root certificate holders, and those root certificate companies can issue other certificates based on that trust. Effectively, a browser trusts a certificate because it's signed by some third party that the browser trusts.

The process a client uses to verify a certificate is straightforward (see Figure 9.1, on the next page). The process is initiated when the client receives a certificate and then verifies the signature against its list of known certificate authorities. If the certificate checks out, the client resumes normal communication with the entity that presented the certificate.

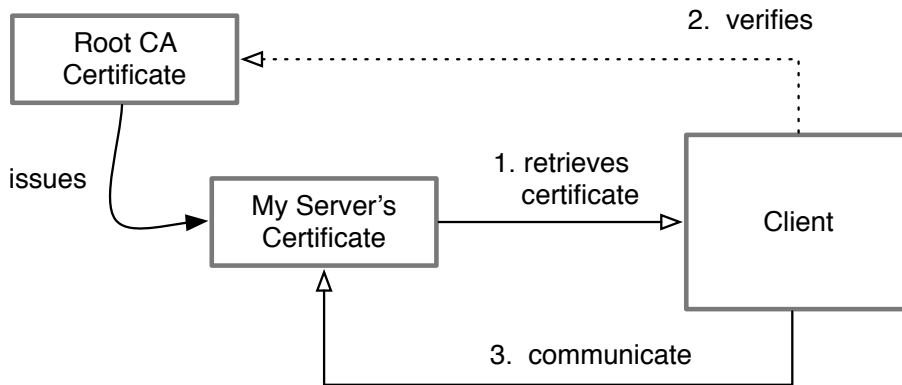


Figure 9.1: Digital certificate verification

Security for all

For public-facing web applications that require the use of TLS, a certificate issued by a major CA is a necessity. The main difference between each certificate authority has nothing to do with the cryptographic strength of a given CA's secure channel, but with the amount of background verification and paperwork performed before a given CA issues a certificate. In our experience, the so-called *credibility* conferred on a site by purchasing a \$400 VeriSign certificate versus a \$100 Thawte certificate is minimal. We, the authors, recommend that you save your money and go with the cheapest certificate signed by a CA present in your targeted browsers' trusted CA lists.

If the clients connecting to your server are not end users, but instead machine-to-machine interfaces controlled entirely by you, then creating your own certificate authority might be an option for you.

Creating Self-Signed Certificates

Regardless of how your production environment is configured, a *self-signed certificate*—one that is not issued from a certificate authority—is useful for testing and development. There is no certificate chain in self-signed certificates—the presence of the certificate cannot be attested to by a higher authority.

To create a self-signed certificate, we begin by creating a private key that we will derive the public key from. This code enables us to create an RSA private key for use in digital signature:

```
digital_signatures> openssl genrsa 1024 > host.key
```

It's possible to create the same key using the Ruby OpenSSL bindings. To do this, we create a file called `create_host_key.rb` and add this code:

[Download](#) `digital_signatures/create_host_key.rb`

```
#!/bin/env ruby
require 'openssl'

File.open("host.key", "w") do |f|
  f.write(OpenSSL::PKey::RSA.new(1024))
end
```

After we create the script, we can generate an RSA private key by executing the following code:

```
digital_signatures> ruby create_host_key.rb
```

So far we've created a private key; now it's time to generate an X.509 certificate. This command creates a certificate using the OpenSSL command-line toolkit:

```
digital_signatures> openssl req -new -x509 -nodes -sha1 -days 365 \
    -key host.key > host.cert
```

As an alternative to using the command-line toolkit, we can generate a certificate using the Ruby bindings. We do this by creating a file called `create_host_cert.rb` and adding the following code:

[Download](#) `digital_signatures/create_host_cert.rb`

```
#!/bin/env ruby
require 'openssl'

File.open("host.cert1", "w") do |f|
  private_key = OpenSSL::PKey::RSA.new(File::read('host.key'))
  cert = OpenSSL::X509::Certificate.new
  cert.subject = OpenSSL::X509::Name.new('/CN=')
  cert.not_before = Time.now
  cert.not_after = Time.now + 365 * 24 * 60 * 60
  cert.public_key = private_key.public_key
  cert.serial = 3
  cert.version = 2
  cert.sign(private_key, OpenSSL::Digest::SHA1.new)
  f.write(cert)
end
```

After we create the script, we can generate the X.509 certificate from the private key we generated previously:

```
digital_signatures> ruby create_host_cert.rb
```

The Ruby bindings provide a nice object-oriented abstraction to the OpenSSL library. Let's inspect the certificate we just created using the OpenSSL module. Create a file called `inspect_certificate.rb` and add this code:

[Download](#) digital_signatures/inspect_certificate.rb

```
#!/bin/env ruby
require 'openssl'
include OpenSSL::X509

# inspect the certificate
cert = Certificate.new(File.read("host.cert"))
puts "Subject:      #{cert.subject}"
puts "Algorithm:    #{cert.signature_algorithm}"
puts "Serial:       #{cert.serial}"
puts "Not Before:    #{cert.not_before}"
puts "Not After:     #{cert.not_after}"
puts "#{cert.public_key.to_pem}"
```

Next, we invoke this command to execute the script:

```
digital_signatures> ruby inspect_certificate.rb

Subject:      /C=US/ST=TX/L=Plano/O=Internet Widgits Pty Ltd
Algorithm:    sha1WithRSAEncryption
Serial:       14414595003654168849
Not Before:   Sat Nov 10 20:04:25 UTC 2007
Not After:    Sun Nov 09 20:04:25 UTC 2008
-----BEGIN RSA PUBLIC KEY-----
MIGJAoGBAPCKM2zkrTuSuCsORktAjNq4w9bBUCYbj9PS2tzLghCumgi2+C/tUY79
YWUfnWeGdjBgCr1SgBV0PznwIZDrIvtmHpjNMS0FbFhefnUi2DK+U8izVveqFCvq
SrHQMcvk+1MimYR3c0rppmvTKt7ktDvQAsTCBoyKHrxHelpmhs/jAgMBAAE=
-----END RSA PUBLIC KEY-----
```

In this example, we begin by instantiating the `OpenSSL::X509::Certificate` class and passing it the contents of the certificate in the `host.cert` file. PEM encoding is a standard way to store a single certificate. The `Certificate` class aliases the `to_s()` method to the `to_pem()` method.

Now that we have had a little experience using X.509 certificates in Ruby, let's create a signature and verify the contents.

To DER-encode a certificate, execute the `to_der()` method.

Converting Certificates

Sometimes certificates don't arrive in the required format. Fortunately, the OpenSSL toolkit has us covered: we can execute the following command to convert a PEM-encoded certificate to a DER-encoded certificate:

```
digital_signatures> openssl x509 -in file.der -inform der
                        -out file.pem
```

This works great if all we ever get are DER-encoded certificates. This code enables us to go in the opposite direction:

```
digital_signatures> openssl x509 -in file.pem -outform pem
                        -out file.der
```

The following code shows how to sign and verify data using the Certificate class:

[Download](#) digital_signatures/digital_signature_test.rb

```
Line 1 require 'openssl'
2 include OpenSSL
3
4 cert = X509::Certificate.new(File.read('host.cert'))
5 private_key = PKey::RSA.new(File.read("host.key"))
6 input = "Loafing is not permitted"
7
8 signature = private_key.sign(Digest::SHA1.new, input)
9 is_verified = cert.public_key.verify(Digest::SHA1.new, signature, input)
```

In this example, we start by instantiating a Certificate object. We do this by passing the PEM-encoded certificate located in the host.cert file. Next we load the private key from the host.key file.

At this point, we have successfully loaded all of the keys and certificates needed to generate a digital signature. This means we can digitally sign any stream of data by invoking the sign(cipher, data) method and passing it the data we want signed, as well as the desired hash cipher. On line 9, we verify that the signature is correct by invoking the verify(cipher, signature, data) method, passing it the same hash cipher used to create the signature and the plaintext received

The return value of the verify(cipher, signature, data) method gives us confidence that the data hasn't changed and that the message was sent by a party with access to the private key associated with the certificate.

9.2 Storing Certificates

When storing or transporting certificates and key pairs, it is advisable to keep them in a protected form. We can achieve this using the PKCS12 format #12. This format is a file-based key store that utilizes password-based symmetric encryption.

The code shows how to create a PKCS12 key store:

[Download](#) digital_signatures/pkcs12_test.rb

```
require 'openssl'

pkey = OpenSSL::PKey::RSA.new(512)
cert = OpenSSL::X509::Certificate.new
cert.version = 1
cert.subject = cert.issuer = OpenSSL::X509::Name.parse("/C=F00")
cert.public_key = pkey.public_key
cert.not_before = Time.now
cert.not_after = Time.now + (365*24*60*60)
cert.sign(pkey, OpenSSL::Digest::SHA1.new)
p12 = OpenSSL::PKCS12.create("supers3cr3t", "P12 Certificate", pkey, cert)
File.open("test.p12", "w") { |s| s.write(p12.to_der) }
```

We can use the `OpenSSL::PKCS12.create(password, description, pkey, cert)` method to create the PKCS12 store. To persist this store on the file system, we combine the standard Ruby file operations with the `to_der()` instance method of the PKCS12 object.

The following shows how we can read and extract the private key and the certificate from a PKCS12 store:

[Download](#) digital_signatures/pkcs12_test.rb

```
require 'openssl'

p12 = OpenSSL::PKCS12::PKCS12.new(File::read('test.p12'), "supers3cr3t")
puts p12.certificate
puts p12.key
```

We can read a PKCS12 store by invoking the `new(contents, password)` method of the `OpenSSL::PKCS12::PKCS12` class.

9.3 Secure Email Exchange

Securing email might sound like an impossible task, given the proliferation of pharmaceutical and anatomy-enhancement advertisements using the same medium, but there is a way! Using asymmetric cryptography and PKI, we can digitally sign and encrypt messages for

Ext	MIME Type	Used For...
.p7m	application/pkcs7-mime	SignedData, EnvelopedData
.p7c	application/pkcs7-mime	Degenerate SignedData certificate management message
.p7z	application/pkcs7-mime	CompressedData
.p7s	application/pkcs7-signature	SignedData

Figure 9.2: S/MIME file attachments

transit. This is an especially important development because of the asynchronous nature of how mail is transferred through untrusted SMTP infrastructure.

The most widely accepted technique for accomplishing this is to use a set of standard extensions built on the Multipurpose Internet Mail Extensions—the *MIME* standard called S/MIME.

The S/MIME Message Format

The MIME standard allows for attachments of varying type to be included within the body of an email message. The S/MIME standard builds upon this idea by creating special encoding types for signatures and ciphertexts. We can see the list of file extensions associated with each MIME type in Figure 9.2.

Creating a Root Certificate

Let's begin our S/MIME adventure by creating a root certificate for LunchedIn; we'll use this certificate to issue all client certificates. We can execute the following commands to create our root certificate:

```
digital_signatures> openssl genrsa 1024 > email_ca.key
digital_signatures> openssl req -new -x509 -nodes -sha1 -days 3650 \
    -key email_ca.key > email_ca.cer
```

Next, we create a certificate for the email address associated with our application, as outlined in Figure 9.3, on the next page. When dealing with certificate authorities, we must create several intermediate files (you can find a description of those files in Figure 9.4, on the following page):

```
digital_signatures> openssl genrsa 1024 > lunchedin.gmail.email.key
digital_signatures> openssl req -new -key lunchedin.gmail.email.key \
    -out lunchedin.gmail.email.csr
```

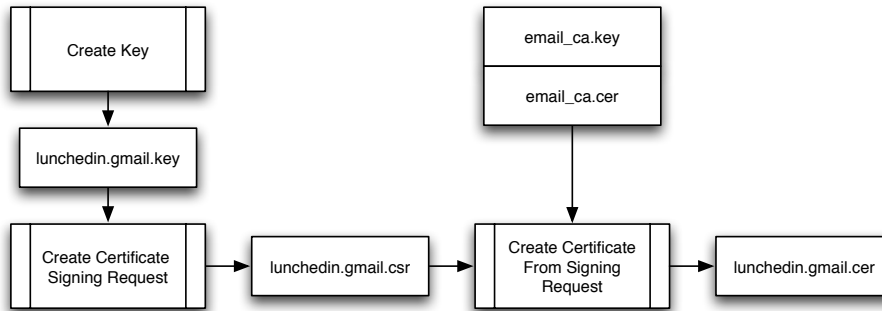


Figure 9.3: Creating the LunchedIn email certificate

File	Description
email_ca.key	Private key of certificate authority
email_ca.cer	The X.509 certificate and public key of the certificate authority, created using <i>email_ca.key</i>
lunchedin.gmail.key	Private key associated with the email account for the LunchedIn application.
lunchedin.gmail.csr	The <i>certificate signing request</i> created from the private key <i>lunchedin.gmail.key</i> and the public key of the certificate authority <i>email_ca.cer</i> .
lunchedin.gmail.cer	The X.509 certificate created from the certificate signing request <i>lunchedin.gmail.csr</i> and the private key of the certificate authority <i>email_ca.key</i> .

Figure 9.4: Certificate file descriptions

In this example, we create an RSA key pair and save the output in the `lunchedin.gmail.email.key` file. Let's start by creating a special file called a *certificate signing request*, or CSR. This file contains all the information required to construct the certificate, which we then sign using our private key. The purpose of a CSR is to package our *soon to be certificate* securely, so that the CA can generate a valid X.509 certificate using its private key.

This command generates a valid certificate using the CSR we created in the previous command; it also generates the private key and a certificate for our chosen CA:

```
digital_signatures> openssl x509 -req -days 3650 -in lunchedin.gmail.email.csr \
    -CA email_ca.cer -CAkey email_ca.key -set_serial 01 \
    -out lunchedin.gmail.email.cer
```

Now that we have the necessary certificates, let's see how we can create S/MIME signatures that we might embed within email messages.

Creating S/MIME Signatures and Ciphertexts

The OpenSSL extension in the Ruby standard library includes classes to create and read S/MIME data under the PKCS7 module. This example shows how to sign a message using the PKCS7 format:

[Download](#) `digital_signatures/smime_sign_test.rb`

```
Line 1 require 'openssl'
      2 include OpenSSL
      3
      4 msg = "Hello\r\n"
      5 cert = X509::Certificate.new(File::read('lunchedin.gmail.email.cer'))
      6 key = PKey::RSA.new(File::read('lunchedin.gmail.email.key'))
      7 p7sig = PKCS7::sign(cert, key, msg, [], OpenSSL::PKCS7::DETACHED)
      8 smime = PKCS7::write_smime(p7sig)
```

On line 7, we create the PKCS7 signature by invoking the `sign(cert, key, data, certs, flags)` method. On line 8, we create the S/MIME output. The S/MIME message contains the signature, the plaintext message, and a wrapper.

Once we create the S/MIME message, we can verify it by reversing the steps we used to create it. This code shows how to verify an S/MIME message using the email certificate authority created earlier:

[Download](#) `digital_signatures/smime_sign_test.rb`

```
Line 1 to_verify = PKCS7::read_smime(smime)
      2 store = X509::Store.new
      3 store.add_cert(X509::Certificate.new(File::read('email_ca.cer')))
      4 is_verified = to_verify.verify([], store)
```

On line 1, we read the S/MIME message using the `read_smime(message)` method from the `PKCS7` module. Then we instantiate a certificate store the object and add the contents of the `email_ca.cer` certificate. Next, we verify the message against the certificate store.

Wait, there's more! Not only can we sign messages, but we can also encrypt them using the same techniques. The only requirement is that we have access to the certificate of the person at the destination email address. This code illustrate how to encrypt messages:

[Download](#) `digital_signatures/smime_protect_test.rb`

```
require 'openssl'
include OpenSSL

sender_cert = X509::Certificate.new(File::read('host.cert'))
sender_key = PKey::RSA.new(File::read('host.key'))

data = "This is a clear-signed message.\r\n"
sender_p7sig = PKCS7::sign(sender_cert, sender_key, data, [], PKCS7::TEXT)
sender_smime = PKCS7::write_smime(sender_p7sig)

lunchedin_cert = X509::Certificate.new(File::read('lunchedin.gmail.email.cert'))
sender_p7enc = PKCS7::encrypt([lunchedin_cert], sender_smime)
msg = PKCS7::write_smime(sender_p7enc)

# decrypt
lunchedin_key = PKey::RSA.new(File::read('lunchedin.gmail.email.key'))
lunchedin_p7enc = PKCS7::read_smime(msg)
lunchedin_data = lunchedin_p7enc.decrypt(lunchedin_key, lunchedin_cert)
puts lunchedin_data.inspect
```

Creating the email message is only part of the task at hand as we create a secure email application. Next we need to learn how to package and send our secure messages within Rails.

9.4 Sending Signed Messages

LunchedIn, which has become the influential restaurant-rating site we always hoped it would be, is now experiencing issues with competing sites sending spoofed customer emails! As part of a comprehensive approach, we will now implement digital signatures on sensitive emails that originate from our site. This enables the recipient to verify our emails.

To send signed email in LunchedIn, we can use an ActionMailer class to prepare and send the messages. We create the ActionMailer stub with this command:

```
digital_signatures> ruby script/generate_mailer MailPerson
```

The ActionMailer generator creates the `mail_person.rb` file under the `app/models` directory, along with a unit-test stub and directory under fixtures.

Creating Digital Signatures with ActionMailer

Now that we have the basic ActionMailer functionality in place, let's add the welcome message. Add this code to the `app/models/mail_person.rb` file:

[Download](#) `digital_signatures/lunchedin/app/models/mail_person.rb`

```
def welcome_message(user)
  from 'lunchedin@gmail.com'
  recipients user.email
  subject "Thank you for registering with our website"
  body :first_name => user.first_name, :last_name => user.last_name
end
```

We use this method to set up the recipients, subject, and accessible variables for the ActionMailer view.

Create the `welcome_message.html.erb` view under `app/views/mail_person`, and add this code:

[Download](#) `digital_signatures/lunchedin/app/views/mail_person/welcome_message.html.erb`

```
Dear <%= @first_name %> <%= @last_name %>,
```

```
Thanks for signing up at LunchedIn!
```

```
Your account will allow you to search, add and view restaurants in
your local area. Remember...with great power, comes great responsibility.
```

```
Thanks.
```

```
-Management
```

At this point, we have the basic pieces necessary to send email. Now let's look at how we can add S/MIME functionality. We can use the following test case to prototype this functionality in LunchedIn:

[Download](#) `digital_signatures/lunchedin/test/unit/mail_person_test.rb`

```
Line 1 def test_create_signed_welcome_message
-     message = MailPerson.create_welcome_message(users(:wally))
-
-     # instantiate openssl objects
5     key = OpenSSL::PKey::RSA.new(
-       File::read("#{RAILS_ROOT}/config/lunchedin.gmail.email.key"))
-     cert = OpenSSL::X509::Certificate.new(
-       File::read("#{RAILS_ROOT}/config/lunchedin.gmail.email.cer"))
-
```

```

10  # sign encoded message
-   p7sig = OpenSSL::PKCS7::sign(cert, key, message.encoded, [],
-     OpenSSL::PKCS7::DETACHED)
-   smime = OpenSSL::PKCS7::write_smime(p7sig, message.encoded)
-
15  new_message = TMail::Mail.parse(smime)
-   # assert_equal(new_message.encoded, smime) doesn't work!
-
-
-   def smime.encoded; self end
20  def smime.from; TMail::Mail.parse(self).parts.first.from end
-   def smime.destinations; TMail::Mail.parse(self).parts.first.destinations end
-
-   MailPerson.deliver(smime)
-   assert_equal ['lunchedin@gmail.com'], smime.from
25  assert_equal ['wally@yahoo.com'], smime.destinations
- end

```

In this example, we start by invoking `create_welcome_message()` on line 2. This is actually not a method at all, but an interception on behalf of our good friend, `method_missing(method_symbol, *parameters)`.¹

Once we create the `TMail::Mail` object and assign it to `message`, we can instantiate the necessary `OpenSSL` objects. On line 12, we use our key and certificate to sign the message. The `PKCS7::sign()` method expects a `String`, so we must invoke the `encoded()` method on `message`. This is significant because, from this point forward, *the signature will be invalid if the message is changed*. But we face trouble ahead: `ActionMailer` still needs access to some of the `TMail::Mail` methods to deliver the message properly. No sweat; here comes metaprogramming to the rescue!

In just about any language, we wouldn't be able to accomplish our next task without inducing a major headache. But here the solution is simple: we add the methods we need directly to the string instance, referenced by the `smime` variable. Now we have a `String` object that *acts like* a `TMail::Mail` object, so the `deliver(message)` method accepts it with no problems.

We've figured out how to send a signed email with `ActionMailer`; next, let's add a convenience method to `MailPerson` that enables us to keep our model code tidy.

1. This is true as of `ActionMailer` 1.99.0. See [digital_signatures/action_mailer/base.rb](#).

Add the following method to `digital_signatures/lunchedin/app/models/mail_person.rb`:

[Download](#) `digital_signatures/lunchedin/app/models/mail_person.rb`

```
def self.deliver_signed(method, *vargs)
  message = send("create_#{method}", vargs.first)
  key = OpenSSL::PKey::RSA.new(
    File::read("#{RAILS_ROOT}/config/lunchedin.gmail.email.key"))
  cert = OpenSSL::X509::Certificate.new(
    File::read("#{RAILS_ROOT}/config/lunchedin.gmail.email.cer"))
  p7sig = OpenSSL::PKCS7::sign(cert, key, message.encoded, [],
    OpenSSL::PKCS7::DETACHED)
  smime = OpenSSL::PKCS7::write_smime(p7sig)

  def smime.encoded; self end
  def smime.from; TMail::Mail.parse(self).parts.first.from end
  def smime.destinations
    TMail::Mail.parse(self).parts.first.destinations
  end
  def smime.method_missing(*vargs); end
  self.deliver(smime)
end
```

To verify the functionality, let's add a test case for good measure. Add this code to `digital_signatures/lunchedin/test/unit/mail_person_test.rb`:

[Download](#) `digital_signatures/lunchedin/test/unit/mail_person_test.rb`

```
def test_deliver_signed
  MailPerson.deliver_signed(:welcome_message, users(:wally))
  smime = OpenSSL::PKCS7::read_smime(ActionMailer::Base.deliveries.first)
  ca = File::read("#{RAILS_ROOT}/config/email_ca.cer")
  cacert = OpenSSL::X509::Certificate.new(ca)
  store = OpenSSL::X509::Store.new
  store.add_cert(cacert)
  assert smime.verify([], store)
end
```

We should now be able to run the unit tests and verify that all is well. We have created the necessary changes for ActionMailer, so now we can add an email event that fires when a new user is created in LunchedIn. We can execute this command to create a SignupObserver class:

```
digital_signatures> ruby script/generate observer signup
```

Next, update `app/models/signup_observer.rb` so it looks like the following:

`Download` digital_signatures/lunchedin/app/models/signup_observer.rb

```
class SignupObserver < ActiveRecord::Observer
  observe :user

  def after_create(user)
    MailPerson.deliver_signed(:welcome_message, user)
  end
end
```

Some email servers require TLS-enabled connections, but ActionMailer doesn't support this functionality out of the box. Fortunately, Kyle Maxwell wrote a Rails plug-in to solve this problem. If you're using a TLS-enabled SMTP gateway, such as Gmail, install the `action_mailer_optional_tls` plug-in by executing the following command:

```
digital_signatures> ruby script/plugin install \
  http://svn.kylemaxwell.com/rails_plugins/action_mailer_optional_tls/trunk
```

Now let's configure the `development.rb` file to use Gmail's TLS-enabled SMTP gateway. Edit the configuration file with your Gmail username and password or another SMTP gateway:

`Download` digital_signatures/lunchedin/config/environments/development.rb

```
ActionMailer::Base.smtp_settings = {
  :address => "smtp.gmail.com",
  :port => 25,
  :user_name => "lunchedin",
  :password => "lunchedin1",
  :authentication => :plain,
  :tls => true
}
```

We need to implement the preceding step to activate the observer, which we accomplish by uncommenting and editing the `config.active_record.observers` setting in the `config/environment.rb` file. Update this line in your `environment.rb` file to enable the `SignupObserver` observer for the `User` class:

`Download` digital_signatures/lunchedin/config/environment.rb

```
config.active_record.observers = [:signup_observer]
```

When a user is created, the `after_save()` callback triggers the delivery of the signed welcome email.

For Extra Credit

It's always possible to make something good even better. Delivering signed emails using ActionMailer is no exception. For starters, we could convert the preceding code into a Rails plug-in. Here are a handful of additional ways we might improve on how we integrate S/MIME signatures into ActionMailer:

- We could use the `alias_method_chain()` method to copy the existing `method_missing()` method and redefine the existing one with a check for `deliver_signed_#{method}()`. If the check fails, we could call the previous version's missing method.
- We could add class method calls to define keys and certificates for the various ActionMailer methods.
- We could package the ActionMailer modifications into a Rails plug-in for easy distribution and reuse.

9.5 Verifying Received Messages

Sending secure mail is only one part of the equation when it comes to creating a comprehensive, email-enabled application. Next, we can use similar S/MIME verifications techniques to authenticate incoming messages. For example, ActionMailer supports receiving mail with the `receive(message)` method. Before we begin stuffing our `MailPerson` class with encoded messages, let's create the email that the user will reply to.

Sending Out the Reply Email

Let's add an observer to send out an email the recipient can respond to. We can use the an issued digital certificate as the authentication mechanism. Start by executing the following command, which creates our event observer:

```
digital_signatures> ruby script/generate observer event
```

Now add this code to the generated EventObserver class located in `app/models/event_observer.rb`:

[Download](#) digital_signatures/lunchedin/app/models/event_observer.rb

```
class EventObserver < ActiveRecord::Observer
  observe :event

  def after_save(event)
    if event.status == 'organized'
      MailPerson.deliver_venue_comment(event)
    end
  end
end
```

This observer hooks into the life cycle of our Event class. When we update the model with the organized status, an email is sent out requesting feedback. Add the following method to the MailPerson class:

[Download](#) digital_signatures/lunchedin/app/models/mail_person.rb

```
def venue_comment(event)
  from 'lunchedin@gmail.com'
  recipients event.hosted_by.email
  subject "Tell us what you think!"
  body :event => event
end
```

Next let's add the view for the associated ActionMailer method. Create the `venue_comment.html.erb` file under `app/views/mail_person`, and add this code:

[Download](#) digital_signatures/lunchedin/app/views/mail_person/venue_comment.html.erb

```
Dear <%= @event.hosted_by.first_name %> <%= @event.hosted_by.last_name %>,
```

```
Thanks for using LunchedIn to organize your event!
```

```
Please respond to this email and tell us about <%= @event.venue.name %>.
```

```
This message is about:
```

```
[<%= @event.venue.name %>=<%= @event.venue.id %>]
```

As before, we need to tell Rails to include our Observer in the life cycle of our models. Add the following line to the `environment.rb` file:

[Download](#) digital_signatures/lunchedin/config/environment.rb

```
config.active_record.observers += [:event_observer]
```

Improving Our Application

We can take several steps to improve the email feature in LunchedIn. For starters, the application could benefit from a proper object model that represents the issued certificate. This feature would allow us to create *certificate revocation lists* (curls) that we could use to *revoke* the ability to automate create comments in our application.

Another improvement we could make is to add the option to send fully encrypted emails using the certificates we issued. We could achieve this by storing a copy of the issued certificate and relating it to the user.

Verifying Email Messages

Now let's add S/MIME *authentication* for incoming messages.

Add the following code to `mail_person.rb` to handle the processing of received messages (we will add the code to receive messages from our mail server later):

Download `digital_signatures/lunchedin/app/models/mail_person.rb`

```
def receive(email)
  begin
    pkcs7 = OpenSSL::PKCS7::read_smime(email.port.to_s)
    store = OpenSSL::X509::Store.new
    ca = File::read("#{RAILS_ROOT}/config/email_ca.cer")
    store.add_cert(OpenSSL::X509::Certificate.new(ca))

    if pkcs7.verify([], store)
      id = email.parts.first.body.match(/\[(\d)\]/).captures.first
      elements = pkcs7.certificates.first.subject.to_a
      cert_email = elements.detect { |e| e.first == 'emailAddress' }
      parts = email.parts.first.body.split('--- LunchedIn@gmail.com wrote:')
      body_text = parts.first
      venue = Venue.find(id)
      comment = Comment.new
      comment.user = User.find_by_email(cert_email[1])
      comment.subject = 'Well I thought...'
      comment.body = body_text
      venue.comments << comment
    end
  rescue OpenSSL::PKCS7::PKCS7Error
    logger.debug 'PKCS7 error'
  end
end
```

The `receive(message)` method contains the logic to verify the incoming message using the supplied signature. If the messages check out, we create a `Comment` and attach it to the `Venue` the message was sent for. A key point: we use the `Certificate` to determine the email address of the associated `User`.

Retrieving Email Using IMAP

Fortunately, the Ruby standard library supports both IMAP and POP out of the box. This enables us to retrieve email messages from a remote server and then process them using the `receive(message)` method of `MailPerson`. For this example, we use the IMAP functionality provided by a well-known search company. Add the following code to `mail_person.rb`:

Download `digital_signatures/lunchedin/app/models/mail_person.rb`

```
require 'net/imap'
require 'openssl'

class MailPerson < ActionMailer::Base

  def self.check_mail
    imap = Net::IMAP.new('imap.gmail.com', 993, true)
    imap.login('lunchedin', 'lunchedin1')
    imap.select('INBOX')
    imap.search(['ALL']).each do |m|
      msg = imap.fetch(m, 'RFC822')[0].attr['RFC822']
      receive(msg)
      imap.store(m, "+FLAGS", [:Deleted])
    end
    imap.expunge()
  end
end
```

The `check_mail()` class method creates an IMAP connection to the mail server and issues the login command. Once authenticated, the `ActionMailer` class selects the “INBOX” mailbox and proceeds to invoke `receive(message)` on each message. After the `receive(message)` is invoked, we set the `Deleted` flag and then invoke the `expunge()` to clean up everything. Now we should have a fully functional email `Comment` feature in `LunchedIn`.

In this chapter, we looked at the basics of creating X.509 certificates and keys that use the `OpenSSL` bindings for Ruby. Yes, there are many parts of an application that can use digital signatures; however, the good news is that they are relatively easy to integrate and provide a lot of value.

SSO: Centralized Authentication

You might recall that we categorized authentication as either *centralized* or *decentralized* back in Chapter 6, *Authentication: Decentralized Authentication*, on page 95.

Another term for centralized authentication is *single sign-on*, or SSO. There are many different flavors of SSO. Some (like Kerberos) were designed and intended for use on intranets, and they place the most emphasis on user control and management. Others, such as OpenID, were designed for public access; for example, you might use OpenID to provide credentials when you log onto someone's blog. OpenID places its emphasis on user convenience.

In a nutshell, SSO provides the infrastructure necessary to authenticate users across multiple applications without requiring that the user log in every time he crosses an application boundary.

SSO implementations can be quite complex. In this chapter, we will enable LunchedIn to authenticate against the following systems:

- OpenID
- CAS
- Kerberos, GSS-API, and SPNEGO

10.1 Public Versus Private

The critical question is this: how do we choose the right SSO solution for our applications?



Joe Asks...

Isn't OpenID Decentralized?

According to OpenID's official website, "OpenID is an open, decentralized, free framework for user-centric digital identity."* You probably recall, however, that we listed OpenID as a centralized-authentication system. We believe that centralized is a loaded term. In defense of those who created OpenID, the architecture of the OpenID system *is* decentralized because there are many providers. In defense of our definition, we refer to any authentication mechanism that resides outside an application as centralized—in other words, if we were to write three separate applications that use the same authentication system, we would consider that to be centralized authentication for those applications.

*. <http://openid.net/>

This short list of guidelines can help us decide:

- Use OpenID if your users are public, as is the case with our standard LunchedIn users. You can read more about this option in Section 10.2, *OpenID*, on the following page.
- If your users are private, as is the case with our corporate LunchedIn users, you have a couple options:
 - Use Kerberos, GSSAPI, and SPNEGO if your user environments are tightly controlled by the organization with Active Directory or some other centralized user management. You can learn more about this option in Section 10.4, *Kerberos, GSSAPI, and SPNEGO*, on page 219.
 - Use CAS if your user environments are not tightly controlled by the organization. This is common for companies that use contractors and there is a large mix of configurations, such as Windows, Macs, Linux, and on. You can learn more about option this in Section 10.5, *CAS*, on page 251.

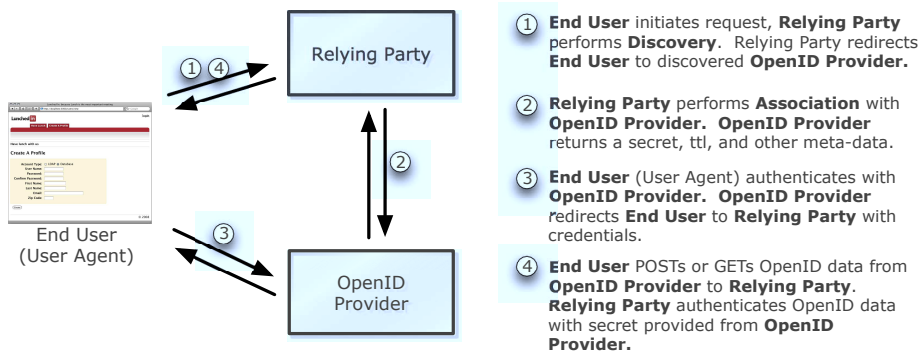


Figure 10.1: OpenID authentication flow

10.2 OpenID

OpenID is a series of specifications that the OpenID Foundation¹ supports and promotes. At the time we write this, Google, Microsoft, IBM, VeriSign, and Yahoo! hold board positions with the OpenID foundation. It appears that industry support is only increasing and that we can look forward to seeing more OpenID-compatible sites in the future.

OpenID combines the usability of not having to remember multiple usernames and passwords with the improved security of delegating authentication. You might wonder how delegating authentication improves security. Let's answer this question by delving a bit deeper into the inner workings of OpenID.

Any OpenID authentication scenario involves the participation of three actors:

- The End User is the user being authenticated. We will use the terms End User and User Agent interchangeably.
- The OpenID Provider is the End User authentication server. OpenID Providers, often abbreviated as OP, store sensitive information such as passwords and personal data.
- The Relying Party is the site requesting authentication of a user. This is LunchedIn for our examples.

1. <http://openid.net/foundation>

A good starting point is to take a careful look at Figure 10.1, on the previous page. In this example, the End User never has to give his password to the Relying Party. We know from our experience in previous chapters how difficult it is to implement robust password-based authentication. With OpenID, the OpenID Provider decides the factor(s) used to authenticate. This allows our End Users to be selective about whom they trust. End Users can select their provider based on reputation and features. Some providers might provide more robust options for authentication, such as biometrics or anti-phishing measures.

Implementing an OpenID Relying Party

Before we get started, make sure you've installed the OpenID gem:

```
> sudo gem install ruby-openid
```

Next, let's consider how our LunchedIn sample application fits into Step 1 in Figure 10.1, on the preceding page.

Performing OpenID discovery

We will give our End Users the option to use OpenID instead of our normal login mechanism. Let's add the appropriate fields to our login form.

```
Download sso/lunchedin_openid/app/views/session/login.html.erb
```

```
<!-- Standard Login -->
<div style="margin-top:14px">
<% form_tag :controller => 'session', :action => "check_login" do %>
  <label for="user_user_name">Username</label>
    <%= text_field 'user', 'username' %>
  <br/>
  <label for="user_password">Password</label>
    <%= password_field 'user', 'password' %>
  <br/>
    <%= submit_tag 'Login' %>
<% end %>
</div>

<!-- Added for OpenID -->
<div>
  Or you can login with your OpenID
</div>

<div style="margin-top:14px">
<% form_tag :controller => 'session', :action => "begin_openid" do %>
  <label for="user_user_name">OpenID </label>
  <!-- We'll use the standard 'openid_identifier' name so that User
    agents that are OpenID aware can deal with it accordingly -->
```

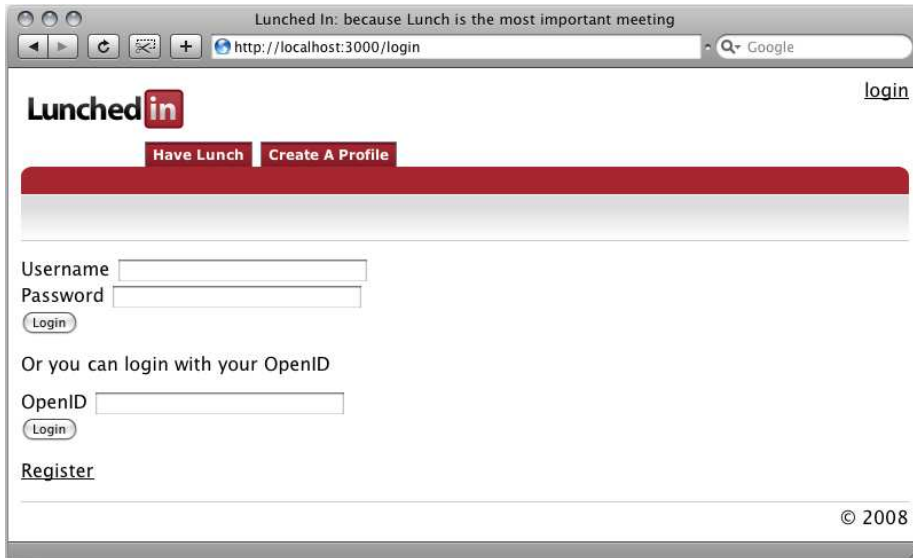


Figure 10.2: OpenID login added

```

    <%= text_field_tag 'openid_identifier', "", :size => 30 %>
  <br/>
  <%= submit_tag 'Login' %>
<% end %>
</div>

```

We can see the changes to our login form in Figure 10.2.

Now, our users can initiate a request with their OpenID identifier. We need to perform *discovery* on this identifier. Fortunately, the OpenID gem provides an easy way to use its API for this. First, you need to instantiate a `OpenID::Consumer` object, along with an `OpenID::Store` implementation. In this example, we add the following code to our `SessionController` class, which we can reuse throughout our example:

```

def consumer
  @store = OpenID::Store::Filesystem.new("#{RAILS_ROOT}/tmp/openid")
  @consumer ||= consumer = OpenID::Consumer.new(session, @store)
end

```

Now let's add the code to perform discovery to our controller:

```

def begin_openid
  identifier = params[:openid_identifier]
  openid_request = consumer.begin(identifier)

```

Before we continue, let's take a look at what's happening in the initial lines of our `begin_openid()` method inside an irb session:

```
>> identifier = "http://www.davesblog.com"
=> "http://www.davesblog.com"
>> consumer = OpenID::Consumer.new(session={}, store)
=> #<OpenID::Consumer:0x24e6ae0 @store=nil, @session={}, \
@session_key_prefix="OpenID::Consumer::">
>> openid_request = consumer.begin(identifier)
=> #<OpenID::Consumer::CheckIDRequest:0x2491504 ...>
>> openid_request.endpoint
=> #<OpenID::OpenIDServiceEndpoint:0x2492814 \
@local_id="https://me.yahoo.com/a/hqUGQw967o90LCQE0UJ7Rm70tNMyZmc-", \
@display_identifier=nil, \
@type_uris=["http://specs.openid.net/auth/2.0/signon"], \
@used_yadis=false,
@server_url="https://open.login.yahooapis.com/openid/op/auth", \
@canonical_id=nil, @claimed_id="http://www.davesblog.com/">
```

At this point, we've passed only a single piece of information to our API—<http://www.davesblog.com>. This is known as the *claimed identifier*. From the claimed identifier, the API is able to discover a server URL² and a local identifier.³ So how did the API derive these values? We can gain insight into this by looking at the source of <http://www.davesblog.com>. For the sake of brevity, this snippet includes only the lines of code related to OpenID:

```
...
<link rel="openid2.provider"
      href="https://open.login.yahooapis.com/openid/op/auth" />
<link rel="openid2.local_id"
      href="https://me.yahoo.com/a/hqUGQw967o90LCQE0UJ7Rm70tNMyZmc-" />
...
```

Now, we see quite plainly that these are special links included in the blog at <http://www.davesblog.com>. Alternatively, it would have been possible to use the local identifier that Yahoo! provided directly.

So what happens if no one pays the blog's domain registration bill and some other Dave gains control over this domain?

From the standpoint of a Relying Party, we cannot rely on the claimed identifier as a nonrepudiable form of identification. Therefore, we might want to rely on the combination of claimed identifier and local identity if our application uses any personal data from the user.

2. <https://open.login.yahooapis.com/openid/op/auth>

3. <https://me.yahoo.com/a/hqUGQw967o90LCQE0UJ7Rm70tNMyZmc->

Performing OpenID association

The OpenID association occurs inside the `OpenID::Consumer::begin()` method in step 2 of Figure 10.1, on page 193. So we don't need to do much in our code for this to work. Let's take a look at this method's source:

Download `sso/ruby-openid-2.1.2/lib/openid/consumer.rb`

```
def begin(openid_identifier, anonymous=false)
  manager = discovery_manager(openid_identifier)
  service = manager.get_next_service(&method(:discover))

  if service.nil?
    raise DiscoveryFailure.new("No usable OpenID services were found "\
                              "for #{openid_identifier.inspect}", nil)
  else
    begin_without_discovery(service, anonymous)
  end
end
```

The service variable is nil in our first call to `OpenID::Consumer::begin()`—which means we need to establish an association:

Download `sso/ruby-openid-2.1.2/lib/openid/consumer.rb`

```
def begin_without_discovery(service, anonymous)
  assoc = association_manager(service).get_association
  checkid_request = CheckIDRequest.new(assoc, service)
  checkid_request.anonymous = anonymous

  if service.compatibility_mode
    rt_args = checkid_request.return_to_args
    rt_args[Consumer.openid1_return_to_nonce_name] = Nonce.mk_nonce
    rt_args[Consumer.openid1_return_to_claimed_id_name] =
      service.claimed_id
  end

  self.last_requested_endpoint = service
  return checkid_request
end
```

This association makes a call to `association_manager()`:

Download `sso/ruby-openid-2.1.2/lib/openid/consumer.rb`

```
def association_manager(service)
  AssociationManager.new(@store, service.server_url,
                        service.compatibility_mode, negotiator)
end
```

Now `association_manager()` makes a call to `OpenID::Consumer::AssociationManager.new()` with the server URL that was discovered. Depending on

which `OpenID::Store` we use, the data is stored for future sessions. We can look at this association data on the filesystem if we use `OpenID::Store::Filesystem`. Here is a sample of that data:

[Download](#) `sso/lunchedin_openid/doc/association_data.txt`

```
version:2
handle:KBAXHgob1NPMY66pJ9RjJrA0xqpLyJDLtAqmWXUBzntNGOGIiRqb.a1T5ESUxz1Veno2a\
s9hxAqQJ5khVAXmjMNHlwOP.iBkDRn10AirkYFjY5br
W92D3wvdJWICLt0
secret:I00U9Ujbdy6c9A9Lr4eKM/rQemA=
issued:1232851295
lifetime:14400
assoc_type:HMAC-SHA1
```

This is the data that OpenID uses to verify data from our End Users. There is an association for each OpenID Provider our application has authenticated users against.

So far we've completed steps 1 and 2 from Figure 10.1, on page 193. It's time to move to step 3, End User authentication with the OpenID Provider.

Redirecting the End User to the OpenID Provider

We use the information we learned in step 1 to tell our End User to authenticate against his OpenID Provider. We do that with a redirect or Ajax call. We begin by constructing the URL for the redirect:

```
def begin_openid
  identifier = params[:openid_identifier]
  openid_request = consumer.begin(identifier)

  return_to = url_for(:action => :complete_openid, :only_path => false)
  realm = url_for(:action => :index, :only_path => false)
```

Next, we redirect the user:

```
redirect_to openid_request.redirect_url(realm, return_to, params[:immediate])
```

Note that we pass the `realm` and the `return_to` variables to the `OpenID::Consumer::CheckIDRequest::redirect_url()` method. The OpenID Provider uses the `realm` variable to inform the user which website has requested authentication. When the authentication process completes, the End User is redirected to the value in the `return_to` variable.

In some cases, we might want to support an OpenID Provider that has an Ajax interface. We can add support for this using the `:immediate` parameter and an alternate set of markup to support the inlined form:

```

if openid_request.send_redirect?(realm, return_to, params[:immediate])
  redirect_to openid_request.redirect_url(realm, return_to, params[:immediate])
else
  render :text => openid_request.html_markup(realm,
    return_to, params[:immediate], {'id' => 'openid_form'})
end

```

We also need to add code to handle our registration process:

```

sregreq = OpenID::SReg::Request.new
# required registration fields
sregreq.request_fields(['email', 'fullname', 'postcode'], true)
openid_request.add_extension(sregreq)
# We are hard coding this to keep the implementation simple. This
# could be provided as an option in the form.
openid_request.return_to_args['did_sreg'] = 'y'

```

Just note for now that this code requests the End User's email, fullname, and postcode. We will use this to populate our user data.

Here is the complete `begin_openid()` method:

```

# We want to handle the client openid request
def begin_openid
  identifier = params[:openid_identifier]
  openid_request = consumer.begin(identifier)

  return_to = url_for(:action => :complete_openid, :only_path => false)
  realm = url_for(:action => :index, :only_path => false)
  # Registry data extension

  sregreq = OpenID::SReg::Request.new
  # required registration fields
  sregreq.request_fields(['email', 'fullname', 'postcode'], true)
  openid_request.add_extension(sregreq)
  # We are hard coding this to keep the implementation simple. This
  # could be provided as an option in the form.
  openid_request.return_to_args['did_sreg'] = 'y'
  # This if/else provides support for AJAX flow. This is not a complete
  # implementation.
  if openid_request.send_redirect?(realm, return_to, params[:immediate])
    redirect_to openid_request.redirect_url(realm, return_to, params[:immediate])
  else
    render :text => openid_request.html_markup(realm,
      return_to, params[:immediate], {'id' => 'openid_form'})
  end
end

```

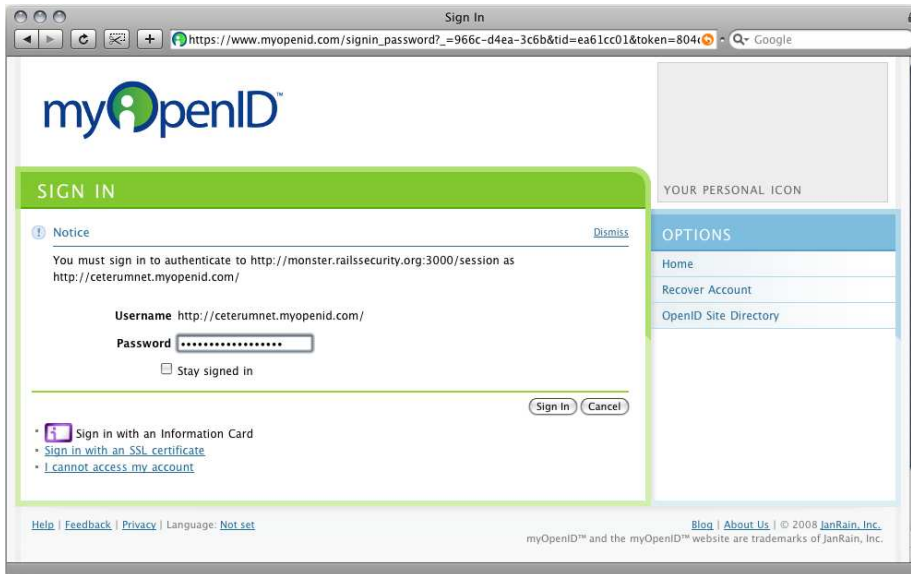


Figure 10.3: myOpenID OpenID Provider authentication dialog presented to End User

Let's try out what we have so far. For our test to work, we need an OpenID—I've created one at Yahoo!⁴ and one at myOpenID.⁵ For this example, we recommend using myOpenID because this group supports more extensions for the standard this book's example illustrates. Regardless, we will write this code to the OpenID specifications, so it should work with any OpenID provider.

When we enter our OpenID claimed identifier in the login form and submit it, the site should redirect us to the appropriate OpenID Provider. For example, our login form redirects us to the myOpenID authentication screen if that's our OpenID provider (see Figure 10.3). Similarly, our login form redirects us to the Yahoo! authentication screen if that's our OpenID Provider (see Figure 10.4, on the next page).

Once the End User authenticates the user, the OpenID Provider redirects the End User back to the `return_to` URL.⁶

4. <http://openid.yahoo.com/>

5. <https://www.myopenid.com/>

6. http://monster.railssecurity.org/session/complete_openid

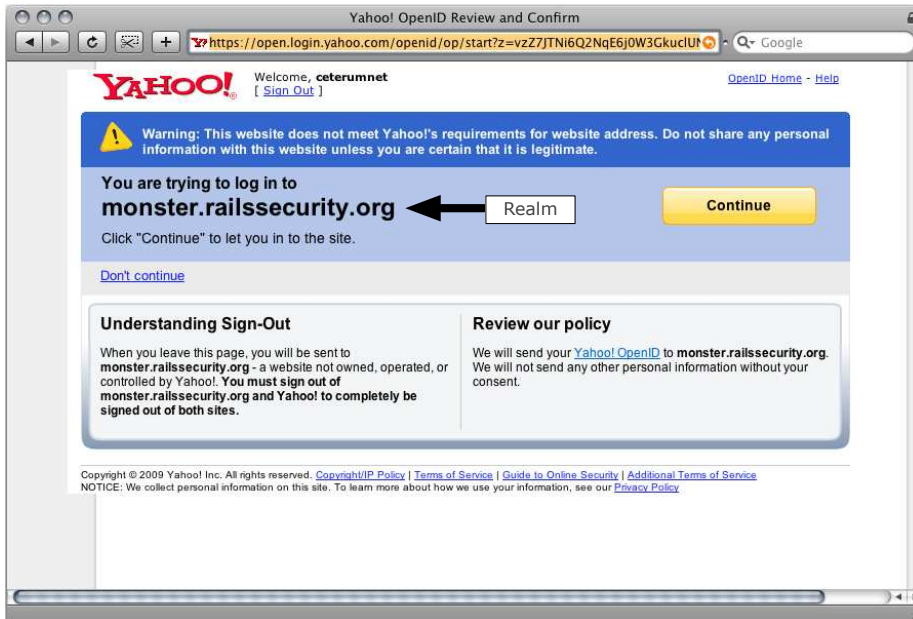


Figure 10.4: Yahoo! OpenID OpenID Provider authentication dialog presented to End User

The End User will now have a collection of parameters that we must handle (see Figure 10.5, on the following page).

Authenticating the End User

Let's look at the first part of our `complete_openid()` controller method:

```
# This implementation was strongly derived from the
# OpenID example included with the OpenID gem.
def complete_openid
  current_url = url_for(:action => 'complete_openid', :only_path => false)
  parameters = params.reject{|k,v|request.path_parameters[k]}
  oidresp = consumer.complete(parameters, current_url)
```

Here, we construct a `OpenID::Consumer::Response` object from our request parameters, filtering out any Rails parameters.

Standard OpenID 2.0 Response Parameters

openid.assoc_handle	{HMAC-SHA1}{49a1a6ba}{diWL0A==}
openid.claimed_id	http://ceterumnet.myopenid.com/
openid.identity	http://ceterumnet.myopenid.com/
openid.mode	id_res
openid.ns	http://specs.openid.net/auth/2.0
openid.op_endpoint	http://www.myopenid.com/server
openid.response_nonce	2009-02-22T20:14:18Zv4Mp5Y
openid.return_to	http://monster.railssecurity.org:3000/session/complete_openid?did_sreg=y
openid.sig	VfIRVx3GS9vWN8rIrlnK2e90Qyo=
openid.signed	assoc_handle,claimed_id,identity,mode,ns,ns.sreg,op_endpoint,response_nonce,return_to,signed,sreg.email,sreg.fullname,sreg.postcode

Sreg Extension Response Parameters

openid.ns.sreg	http://openid.net/extensions/sreg/1.1
openid.sreg.email	draphael@gmail.com
openid.sreg.fullname	David Raphael
openid.sreg.postcode	75093

Figure 10.5: OpenID parameters

The next chunk of code checks whether our End User was authenticated successfully:

```
if oidresp.status == OpenID::Consumer::SUCCESS
  identifier = oidresp.identity_url
  if @authenticated_user = User.find_by_username(identifier)
    session[:user_id] = @authenticated_user.id
```

Note that we retrieve a User object from the database. We need a way to populate the user in the database so we can store our application's specific data. We populate the user by leveraging the OpenID Simple Registration Extension.⁷

7. http://openid.net/specs/openid-simple-registration-extension-1_0.html

```

else
  if params[:did_sreg]
    sreg_resp = OpenID::SReg::Response.from_success_response(oidresp)
    if sreg_resp.empty?
      # This is where we would want to direct the user to a form where they
      # could fill in their profile data.
      logger.debug "NO SREG DATA RETURNED"
      flash[:notice] = 'We are unable to setup
                        an account with the provided OpenID.'
    else
      @authenticated_user = User.create_from_openid_data(identifier, sreg_resp)
      session[:user_id] = @authenticated_user.id
    end
  end
end
end
redirect_to user_path(@authenticated_user)

```

We need to create a User from the parameters returned. The parameters we need are listed in Figure 10.5, on the previous page. Refer to the OpenID specification for a list of all possible values that we can request from an OpenID Provider. In this case, we include email, fullname, and postcode, the same parameters we specified in our initial begin_openid() method. We can create the user by adding this code to our User class:

[Download](#) sso/lunchedin_openid/app/models/user.rb

```

def self.create_from_openid_data(identifier, openid_data)
  user = User.new do |u|
    u.first_name = openid_data['fullname'].split(' ').first
    u.last_name  = openid_data['fullname'].split(' ').last
    u.email      = openid_data['email']
    # Just create a random password
    rand_data = OpenSSL::Random.random_bytes(255)
    u.password_hash = OpenSSL::Digest::SHA1.new(rand_data).hexdigest
    u.username     = identifier
    u.zip_code     = openid_data['postcode']
  end
  # We are going to skip validation for this kind of creation.
  user.save(false)
  user
end

```

The finished code for the complete_openid() method looks like this:

```

# This implementation was strongly derived from the
# OpenID example included with the OpenID gem.
def complete_openid
  current_url = url_for(:action => 'complete_openid', :only_path => false)
  parameters = params.reject{|k,v|request.path_parameters[k]}
  oidresp = consumer.complete(parameters, current_url)

```

```

if oidresp.status == OpenID::Consumer::SUCCESS
  identifier = oidresp.identity_url
  if @authenticated_user = User.find_by_username(identifier)
    session[:user_id] = @authenticated_user.id
  else
    if params[:did_sreg]
      sreg_resp = OpenID::SReg::Response.from_success_response(oidresp)
      if sreg_resp.empty?
        # This is where we would want to direct the user to a form where they
        # could fill in their profile data.
        logger.debug "NO SREG DATA RETURNED"
        flash[:notice] = 'We are unable to setup
                        an account with the provided OpenID.'
      else
        @authenticated_user = User.create_from_openid_data(identifier, sreg_resp)
        session[:user_id] = @authenticated_user.id
      end
    end
  end
  redirect_to user_path(@authenticated_user)
else
  redirect_to :action => 'index'
end
end
end

```

Assuming we've implemented everything correctly, we see a welcome screen for the just authenticated user, along with his registration data (see Figure 10.6, on the following page).

So far we've built an OpenID support as a Relying Party into LunchedIn. Let's take the next step and implement an OpenID provider.

10.3 Implementing an OpenID Provider

It takes a bit more work to implement an OpenID Provider, but the OpenID gem still makes it relatively straightforward to do so.

We can see the three request handlers we will implement in Figure 10.7, on the next page. We can add these handlers in an OpenID Controller in the LunchedIn application.

OpenID Provider: Controller Implementation

Let's begin by generating a controller that acts as the primary interface to our provider implementation:

```
> script/generate controller OpenId
```

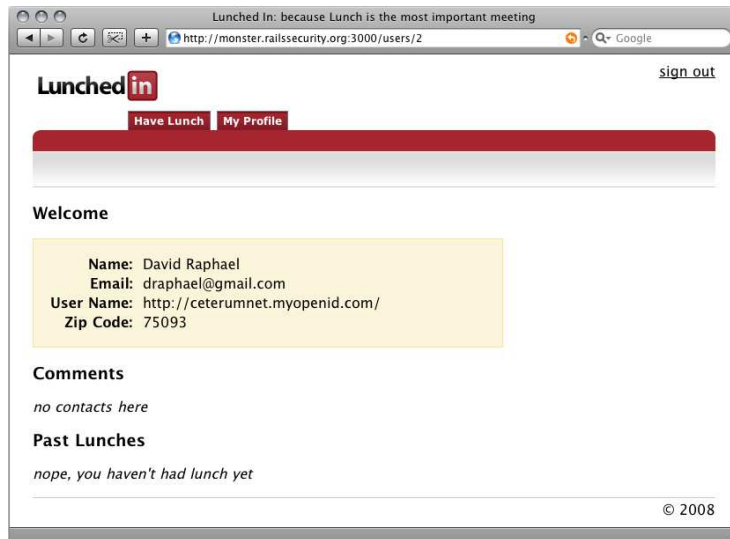


Figure 10.6: A successful authentication using OpenID, along with registration data

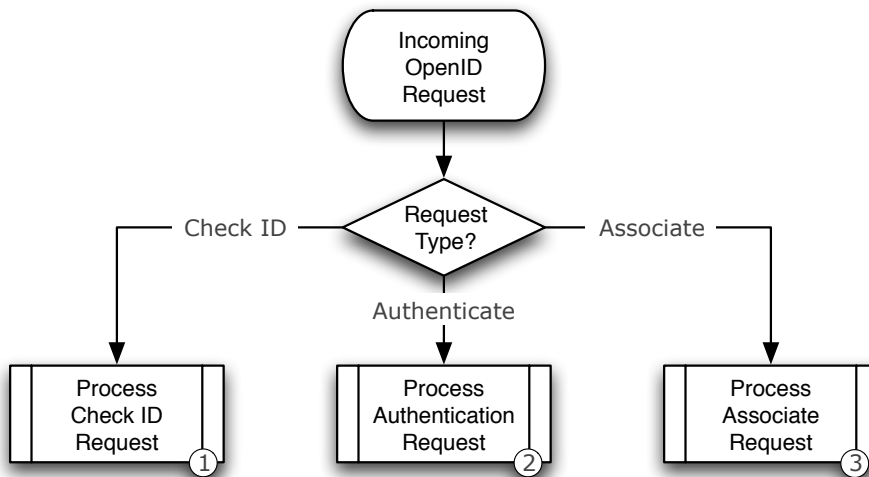


Figure 10.7: OpenID Provider request flow

Next, we can implement an `index()` method that acts as our OpenID request router:

```
def index
  begin
    openid_request = session[:last_oidreq]
    openid_request ||= server.decode_request(params)
    session[:last_oidreq] = nil

    case openid_request
    when nil
      render :text => "This is an OpenID server.", :status => 500
    when CheckIDRequest
      ❶ handle_check_id_request(openid_request)
    when CheckAuthRequest
      ❷ handle_check_auth_request(openid_request)
    when AssociateRequest
      ❸ handle_associate_request(openid_request)
    else
      render :text => "Some general problem", :status => 500
    end
  rescue ProtocolError => e
    render :text => "Problem processing OpenID Request", :status => 500
    logger.debug { e.to_s }
  end
end
```

❶ If the `index()` method receives a `CheckIDRequest` request, it routes that to `handle_check_id_request()`.

❷ If the `index()` method receives a `CheckAuthRequest` request, it routes the request to `handle_check_auth_request()`.

❸ If the `index()` method receives an `AssociateRequest` request, it routes the request to `handle_associate_request()`.

Next, we need to implement these request handlers. We'll start with the two easiest handlers first.

Implementing `handle_check_auth_request()` and `handle_associate_request()`

Let's look at `handle_check_auth_request()` first.

```
# Responding to Authentication Requests
# http://openid.net/specs/\
# openid-authentication-2_0.html#responding_to_authentication
def handle_check_auth_request(openid_request)
  render_response(server.handle_request(openid_request))
end
```

OpenID check_authentication Request Parameters

openid.assoc_handle	{HMAC-SHA1}{49a1a6ba}{diWL0A==}
openid.claimed_id	http://ceterumnet.myopenid.com/
openid.identity	http://ceterumnet.myopenid.com/
openid.mode	check_authentication
openid.ns	http://specs.openid.net/auth/2.0
openid.op_endpoint	http://www.myopenid.com/server
openid.response_nonce	2005-05-15T17:11:51ZUNIQUE
openid.return_to	http://monster.railssecurity.org:3000/session/complete_openid?did_sreg=y
openid.sig	VfIRVx3GS9vWN8rIrnK2e90Qyo=
openid.signed	op_endpoint,return_to,response_nonce,assoc_handle,claimed_id,identity
openid.sig	5en6G6MezRroT3XKqkdPOMY/BfQ=\n

Figure 10.8: OpenID check_authentication params

The `handle_check_auth_request()` processes a request that looks something like the data in Figure 10.8. The OpenID library verifies the data by ensuring that the OpenID provider signed the data; it also makes sure that the data has not been used by verifying the nonce. This work happens inside the OpenID library.

We also have to implement the `render_response()` and `server()` methods for this approach to work:

```
def render_response(openid_response)
  if openid_response.needs_signing
    signed_response = server.signatory.sign(openid_response)
  end
  web_response = server.encode_response(openid_response)
  case web_response.code
  when HTTP_OK
    render :text => web_response.body, :status => 200
  when HTTP_REDIRECT
    redirect_to web_response.headers['location']
  else
    render :text => web_response.body, :status => 400
  end
end
```

OpenID associate Request Parameters

openid.ns	http://specs.openid.net/auth/2.0
openid.mode	check_authentication
openid.assoc_type	HMAC-SHA1
openid.session_type	no-encryption

Figure 10.9: OpenID associate params

```
def server
  if @server.nil?
    server_url = url_for(controller => :open_id,
                        :action => 'index', :only_path => false)
    dir = Pathname.new(RAILS_ROOT).join('db').join('openid-store')
    store = OpenID::Store::Filesystem.new(dir)
    @server = Server.new(store, server_url)
  end
  @server
end
```

The `server()` method returns the `@server` instance variable, which is an instance of the `OpenID::Server` class that has been instantiated with a file-based store.

The `render_response()` method is a simple wrapper that adapts the OpenID responses so they work with our Rails controller.

Now, let's do the same thing for `handle_associate_request()`:

```
# Establishing Associations
# http://openid.net/specs/openid-authentication-2_0.html#associations
# This is when the openid.mode == "associate"
def handle_associate_request(openid_request)
  render_response(server.handle_request(openid_request))
end
```

The `handle_associate_request()` processes a request that looks something like the data in Figure 10.9. The OpenID server takes the data and creates an association in the `OpenID::Store`. Once the association is created, the server responds to the Relying Party with a shared secret that they use in future information exchanges.

OpenID Storage Concerns

For larger systems that span multiple physical devices, you need to use a database to store the OpenID data. The OpenID gem includes examples of ActiveRecord-based stores.

OpenID check_setup Request Parameters

openid.assoc_handle	{HMAC-SHA1}{49a1a6ba}{diWL0A==}
openid.claimed_id	http://monster.railssecurity.org:3001/open_id/user_page/open_id_demo
openid.identity	http://monster.railssecurity.org:3001/open_id/user_page/open_id_demo
openid.mode	checkid_setup
openid.ns	http://specs.openid.net/auth/2.0
openid.return_to	http://monster.railssecurity.org:3000/session/complete_openid?did_sreg=y
openid.realm	http://localhost:3000/session
openid.sreg.required	email,fullname,postcode
openid.ns.sreg	http://openid.net/extensions/sreg/1.1

Figure 10.10: OpenID checkid_setup params

That was easy. Because the gem includes most of the logic we needed for those requests, we were able to add some basic code to wrap the OpenID library's functionality, enabling it to work nicely with Rails.⁸

It's also worth noting that both of these methods handle server-to-server communication requests. There is no view layer code to implement. The next handler, `handle_check_id_request()`, will be much more complex, include view code, and interact with the user.

⁸ Some of this code was derived from the examples used in the OpenID gem examples directory.

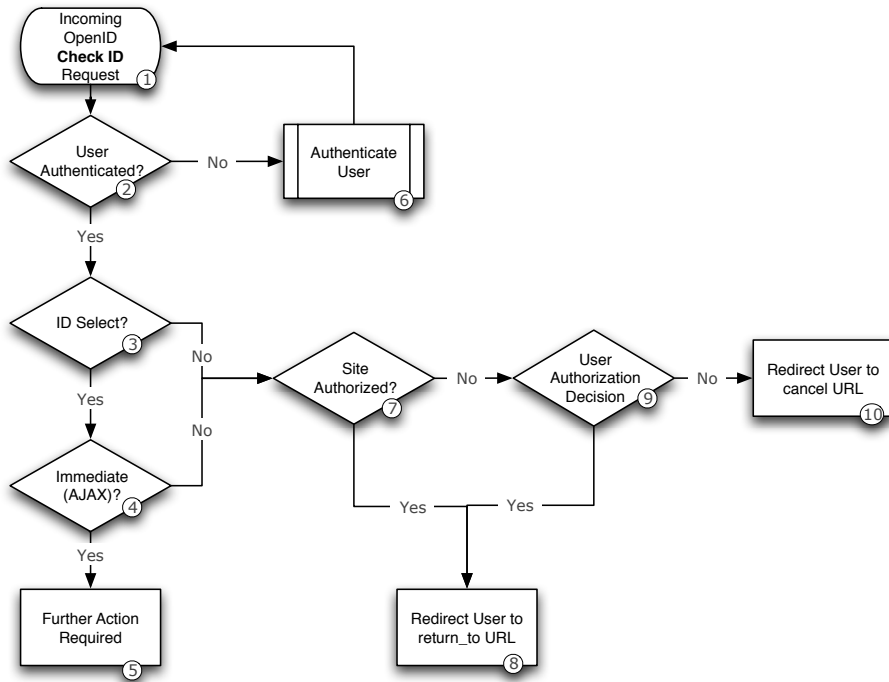


Figure 10.11: OpenID Check ID request flow

Implementing `handle_check_id_request()`

The `handle_check_id_request()` method processes the parameters shown in Figure 10.10, on the preceding page. It also processes a similar set of parameters, but with the mode set to `checkid_immediate`.

Our first task when handling a `checkid_setup` or `checkid_immediate` OpenID request is to authenticate the user, according to decision 2 in Figure 10.11. Let's add that to our controller:

```

def handle_check_id_request(openid_request)
  unless session[:user_id]
    session[:last_oidreq] = openid_request
    redirect_to :controller => :session
  else

```

Here we face a situation where we need to duplicate some of the code we wrote earlier in this chapter, repeating the logic from our application authentication. We'd prefer to avoid duplicating code wherever possible,

but doing so here allows us to stash the OpenID request in a special session key, `:last_oidreq`.

You might wonder we stash the OpenID request in this session. Let's look at our change to `SessionController` that uses this request:

```
def check_login
  username, password = params[:user][:username], params[:user][:password]
  if @authenticated_user = User.find_and_authenticate(username, password)
    session[:user_id] = @authenticated_user.id
    ▶ if session[:last_oidreq]
    ▶   redirect_to :controller => :open_id
    else
      redirect_to user_path(@authenticated_user)
    end
  else
    flash[:notice] = "the user #{params[:user][:username]}, "
    flash[:notice] << "was not found with the specified password"
    redirect_to :controller => 'session', :action => 'login'
  end
end
```

This code routes us back to our `OpenIdController` after authenticating our user, as shown in box 6 of Figure 10.11, on the preceding page.

After we authenticate the End User, we need to determine whether we have the `claimed_id` or an `id_select`. We also need to determine whether the request is immediate. We can accomplish both tasks in this bit of code:

```
openid_response = nil
identity = openid_request.identity
# We need to choose an identifier if the RP has send us
# "http://specs.openid.net/auth/2.0/identifier_select"
if openid_request.id_select
  if openid_request.immediate
    # If it's 'immediate' and the identifier_select
    # has been specified we must ask that the User
    # take further action.
    openid_response = openid_request.answer(false)
    render_response(openid_response)
  else
    # Since the user is logged in, we can provide the
    # identity.
    identity = url_for_user
  end
end
```

Notice that we invoke `openid_request.answer(false)` if the request is immediate and we have an `id_select`. This particular combination means that the user is attempting to use Ajax, and she is not passing a `claimed_id`.

This user will need to log in before she can leverage Ajax without passing a valid identity.

Once we have a valid identity, we need to check whether the user authorizes the site that has requested authentication:

```
if is_authorized(identity, openid_request.trust_root)
  # Since the site is authorized, we will render
  # and authorize response
  openid_response = openid_request.answer(true, nil, identity)
  render_response(openid_response)
```

This code creates the `is_authorized()` method:

```
def is_authorized(identity_url, trust_root)
  session[:user_id] and (identity_url == url_for_user) and approved(trust_root)
end
```

We also need to create an `approved()` method and a `url_for_user()` method:

```
def approved(trust_root)
  if session[:approvals].nil?
    false
  else
    session[:approvals].member?(trust_root)
  end
end

def url_for_user
  server_url = url_for(:action => 'index', :only_path => false)
  user = User.find(session[:user_id])
  "#{server_url}/user_page/#{user.username}"
end
```

If the End User authorizes the site, we redirect to the site or render the appropriate Ajax response to the user.

To support discovery properly, we need to render something for the `user_page()` method that we include as part of the `url_for_user`. Let's look at that method:

```
def render_user_xrds
  @types = [ OpenID::OPENID_2_0_TYPE,
            OpenID::OPENID_1_0_TYPE,
            OpenID::SREG_URI ]
  render :action => :xrds, :format => :xrds
end
```

This method also includes a corresponding view for HTML and XRDS:

[Download](#) sso/lunchedin_openid_provider/app/views/open_id/user_page.html.erb

```
<html>
  <head>
    <meta http-equiv="X-XRDS-Location" content="<%= @xrds_url %>" />
    <link rel="openid.server" href="<%= url_for :action => 'index' %>" />
  </head>
  <body>
    <p>OpenID identity page for <%= params[:username] %></p>
  </body>
</html>
```

[Download](#) sso/lunchedin_openid_provider/app/views/open_id/xrds.xrds.erb

```
<?xml version="1.0" encoding="UTF-8"?>
<xrds:XRDS
  xmlns:xrds="xri://$xrds"
  xmlns="xri://$xrd*(($v*2.0))">
  <XRD>
    <Service priority="0">
      <% @types.each do |uri| %>
      <Type><%= uri %></Type>
      <% end %>
      <URI><%= url_for(:controller => 'open_id', :only_path => false) %></URI>
    </Service>
  </XRD>
</xrds:XRDS>
```

Next, we need to deal with sites that the End User has not yet placed in her approved list:

```
elsif openid_request.immediate
  # We have to answer negatively since 'immediate'
  # has been requested and the user has not authorized.
  server_url = url_for(:action => 'index')
  openid_response = openid_request.answer(false, server_url)
  render_response(openid_response)
else
  # We will give the user the opportunity to authorize
  # this particular request.
  session[:last_oidreq] = openid_request
  @openid_request = openid_request
  @url_for_user = url_for_user
  render :action => :decide
end
end
end
```

The first `elsif` in the code handles the Ajax use case. We simply respond that the request cannot be authorized without further action.

In the else part of the code, we stash the `openid_request` in the session (as we did in the previous example) and then render the `:decision` action.

This code creates the `:decision` view:

Download `sso/lunchedin_openid_provider/app/views/open_id/decide.erb`

```
<% form_for :decision, :url =>
  { :controller => :open_id, :action => :decision },
  :method => :post do |f|%>
  <table>
    <!-- <tr><td>The following site:</td></tr> -->
    <tr><td>Do you want to allow the following site</td></tr>
    <tr><td><%= @openid_request.trust_root %></td></tr>

    <tr><td>to use the identity?</td></tr>
    <tr><td><%= h(@openid_request.identity) %></td></tr>
    <tr><td><%= h(@url_for_user) %></td></tr>

  </table>
  <%= f.submit "Yes", :name => "yes", :value => "yes" %>
  <%= f.submit "No", :name => "no", :value => "no" %>
<% end %>
```

This view posts to the `decision()` action:

```
def decision
  openid_request = session[:last_oidreq]
  # Not sure we should nil this out here.
  session[:last_oidreq] = nil
  if params[:yes].nil?
    redirect_to openid_request.cancel_url
  else
    identity = openid_request.identity
    session[:approvals] ||= []
    session[:approvals] << openid_request.trust_root
    openid_response = openid_request.answer(true, nil, identity)
    add_sreg(openid_request, openid_response)
    render_response(openid_response)
  end
end
```

The `decision()` action accumulates the approved site into the session if the user chooses to allow this. OpenID doesn't specify the precise behavior for providers. In this example, we throw away the approval data with the session, but depending on your needs in other circumstances, you might want to persist this data across sessions in some user data that you store in the database.

The preceding code also adds the Sreg data for user registration data.

We also need to add the following code for this to work:

```
# This could be refactored into the User Model
def add_sreg(openid_request, openid_response)
  # check for Simple Registration arguments and respond
  sregreq = OpenID::SReg::Request.from_openid_request(openid_request)
  if sregreq
    # In a real application the user should be asked for
    # permission to release registration data like this.
    user = User.find(session[:user_id])
    sreg_data = {
      'nickname' => user.username,
      'fullname' => "#{user.first_name} #{user.last_name}",
      'email' => user.email,
      'postcode' => user.zip_code
    }
    sregresp = OpenID::SReg::Response.extract_response(sregreq, sreg_data)
    openid_response.add_extension(sregresp)
  end
end
```

We saw this functionality in action with our Relying Party implementation above in Section 10.2, *Implementing an OpenID Relying Party*, on page 194. Now let's see if we can get our Relying Party implementation and our provider implementation to engage in a successful OpenID communication.

Logging into Our LunchedIn OpenID Provider

Let's begin by starting up an instance of our provider on a different port (assuming we are running on the same system):

```
> script/server -p 3001
```

In this example, we test the provider running on port 3001.

We can start the Relying Party on port 3000:

```
> script/server -p 3000
```

Once we get both servers running, we need to create a user to test the communication process. Let's create a user like the one shown in Figure 10.12, on the following page.

After we create the user successfully, let's check out the username in Figure 10.13, on page 217. This is the local username that the provider uses.

OpenID doesn't specify how we must provide the user identity. In our implementation, we followed the format demonstrated in Figure 10.14, on page 217.

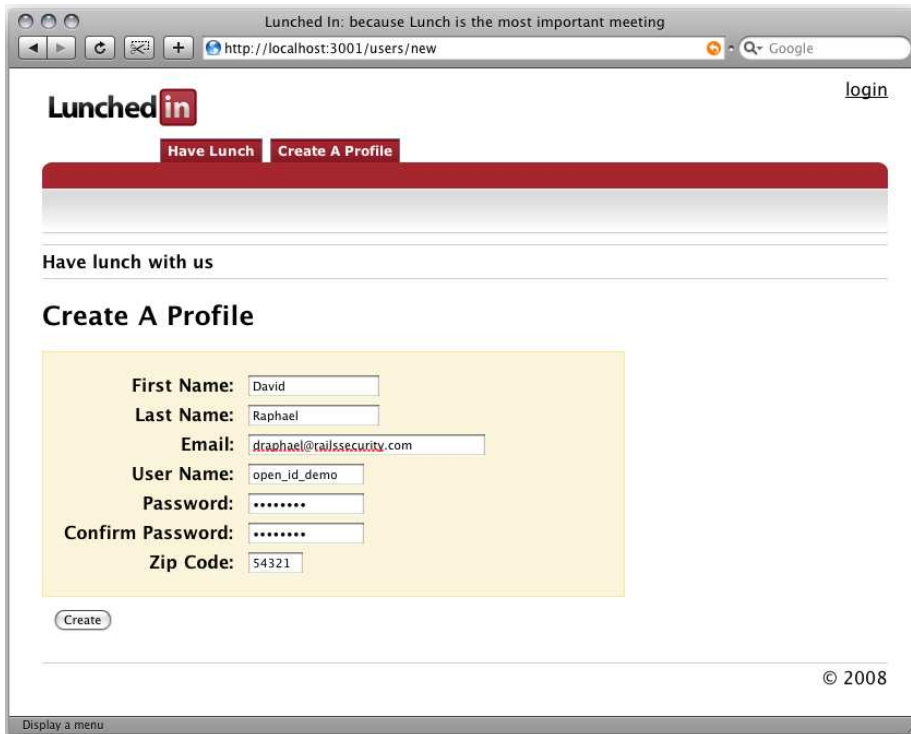


Figure 10.12: Registering demo user

Now let's launch a session with our Relying Party running on port 3000. Our OpenID login should be `http://localhost:3001/open_id/user_page/open_id_demo`, as shown in Figure 10.15, on page 218.

Clicking the login button sets several actions in motion:

- The Relying Party performs discovery.
- The Relying Party redirects the user to the OpenID Provider.
- The OpenID Provider presents the user with the decide page, as shown in Figure 10.16, on page 218.

Once we authorize the site, the provider adds the site to the approved list and redirects the user to the `return_to` URL provided in the original request. In this case, it's our user page (see Figure 10.17, on page 219). Note that the User Name is the claimed ID for our user.

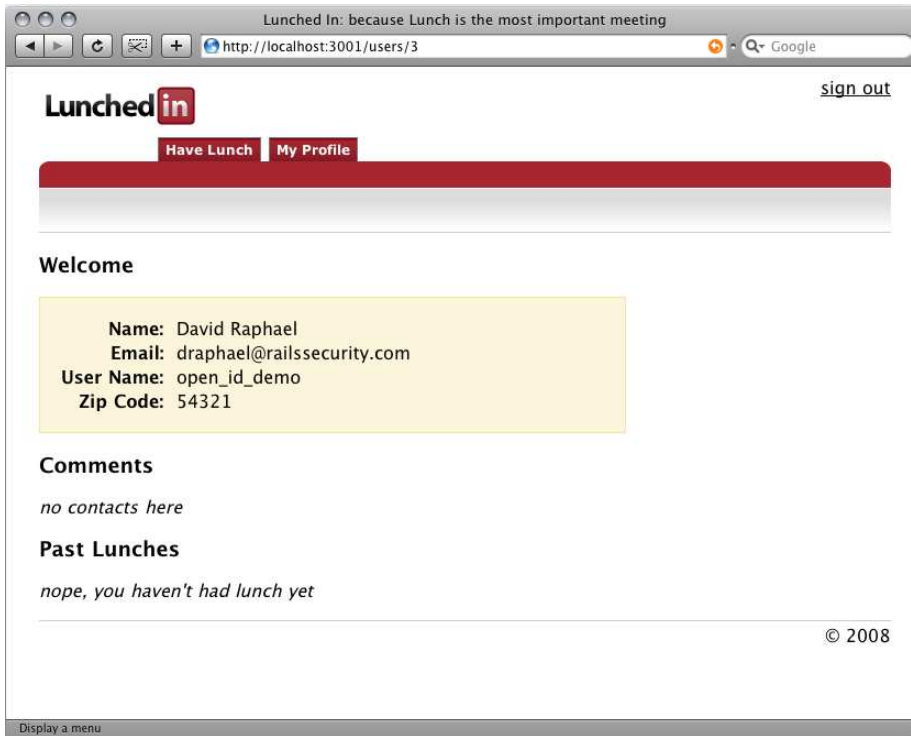


Figure 10.13: Demo user registration data on provider



Figure 10.14: A user page on provider

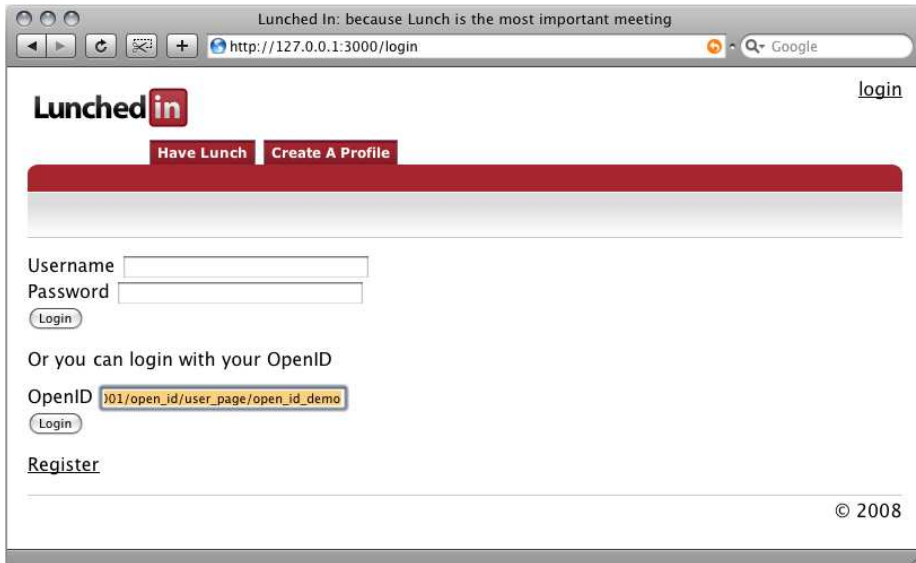


Figure 10.15: The login page on Relying Party

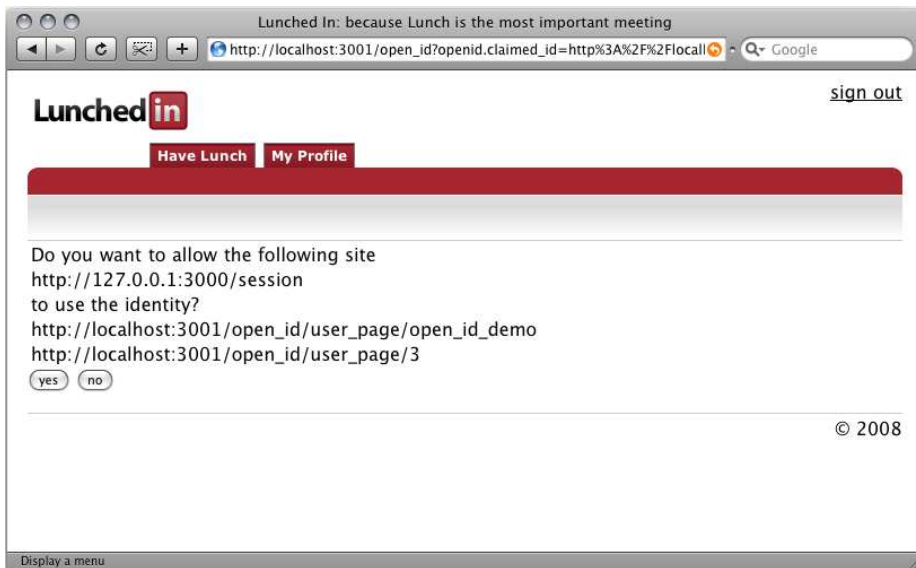


Figure 10.16: The decide page on the provider

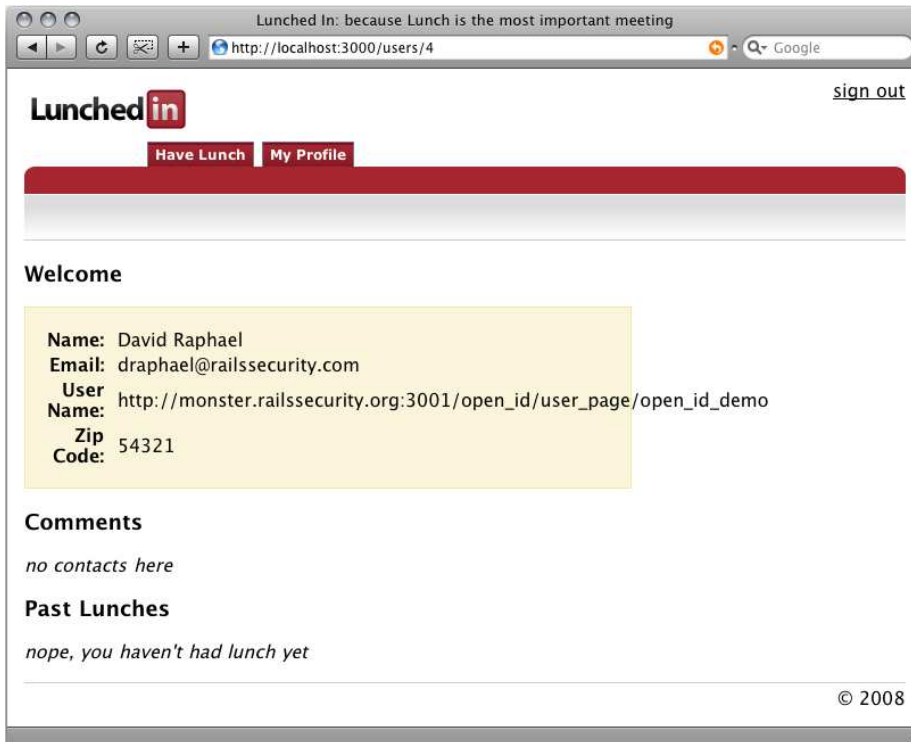


Figure 10.17: The user page for the OpenID user on Relying Party

10.4 Kerberos, GSSAPI, and SPNEGO

Kerberos is a network-authentication protocol that provides strong authentication for client-server applications by employing secret-key cryptography.⁹

The first published version, Kerberos 4, emerged sometime in the late 1980s from Project Athena—a joint MIT, DEC, and IBM project. Its primary intended use was to provide bidirectional authentication for client-server architectures.

All modern operating systems use the current version of this protocol, Kerberos 5.

9. http://web.mit.edu/Kerberos/#what_is

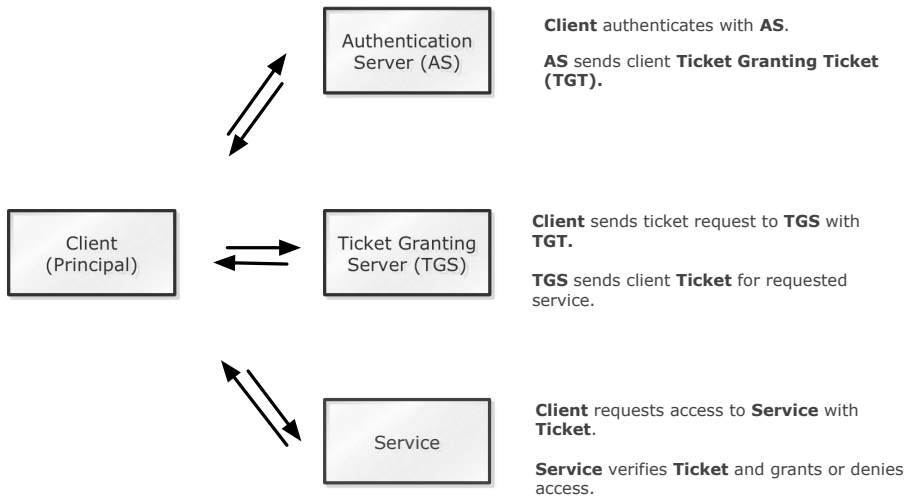


Figure 10.18: A simplified Kerberos/GSSAPI flow chart

[Download at WoweBook.com](http://WoweBook.com)

Our goal here is to demonstrate how to use Kerberos from a Ruby or Ruby on Rails environment. As such, we will avoid any complex theory, but you can find a plethora of information available on Kerberos for free on the Web. In any case, a relatively simple representation of ticket-based authentication should be enough to see you through this chapter's basic Kerberos examples (see Figure 10.18).

Building Ruby Support for GSSAPI

At the time of writing this book, there are not any ready-to-use APIs or libraries for Kerberos available for Ruby.

We, the authors, have written some basic SWIG wrappers around Kerberos, GSSAPI, and SPNEGO for these examples, but won't cover the implementation of these wrappers exhaustively. Instead, we will cover the important interactions with the APIs. You can find the complete listings for these wrappers in Chapter 12, *Authentication Appendix*, on page 263. You will need to compile these libraries before you can run the next set of examples. Do this by executing the following code in the top level of `gssapi_ruby` and `spnego_ruby`:

```
> ruby extconf.rb
> make
> sudo make install
```

Setting Up the Kerberos Server

Before we get started, let's make sure we have a good development environment for learning GSSAPI/Kerberos.

These examples use Microsoft's Active Directory server. Active Directory is probably the most common Kerberos system that developers need to integrate with. However, most LDAP servers support Kerberos, and Linux and OS X both ship with Kerberos support out of the box, as do most Unix systems.

If you run Windows 2003—simply install the OS with Active Directory. For Windows 2000 Server, you need to install the Kerberos tools manually to get complete support.

Once we set up a basic Active Directory installation, it's time to create a couple of users and service principal names (SPNs).

Let's begin by creating the test user that we will use in our first example, `rubything` (feel free to substitute any name you like).

Next, we follow these steps to create the user with the normal Active Directory dialogs:

1. Open Active Directory Users and Computers.
2. Right-click Users > New > User.
3. Fill out the data appropriately, and click Next.

Now we need to create the keytab and the associated SPN:

```
C:\Documents and Settings\Administrator\Desktop\krb>ktpass -princ\
rubything/monster.railssecurity.org@RAILSSECURITY.ORG -mapuser\
RAILSSECURITY\rubything +rndPass -ptype KRB5_NT_PRINCIPAL -crypto\
RC4-HMAC-NT -out ruby4.keytab
```

It's important that we follow the command-line arguments precisely, paying particular attention to the `-princ` and the `-crypto` parameters.

If we right-click the user in the administrative console, we should see something very close to Figure 10.19, on the following page.

Let's repeat these steps and create another account with the username, `monster`, and the principal, `HTTP/monster.railssecurity.org`. If we right-click the user in the administrative console, we should see something like Figure 10.20, on page 223.

We have a couple of test users, so let's tackle some Kerberos/GSSAPI code.

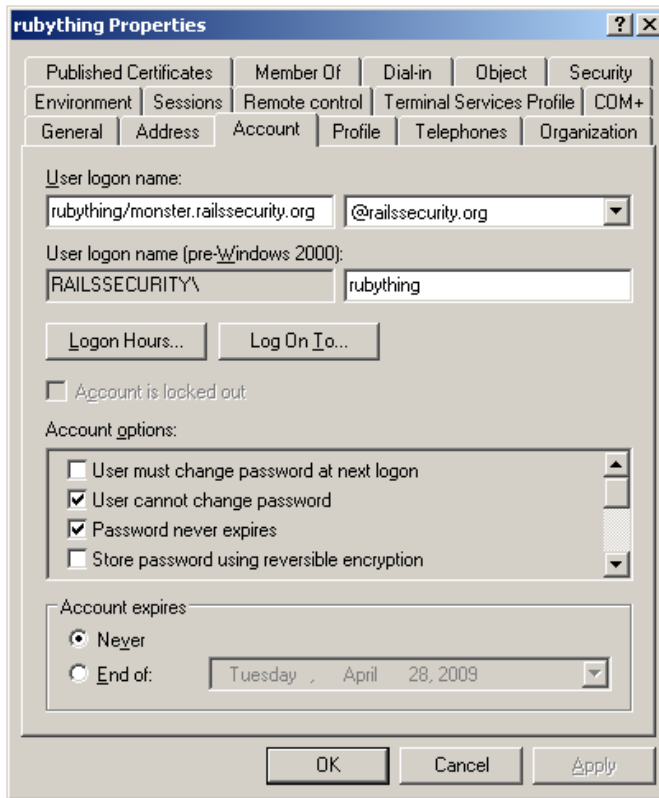


Figure 10.19: rubything properties

Kerberos/GSSAPI

The Generic Security Services API (GSSAPI) was developed as an abstraction around ticket-granting network-authentication protocols. The main purpose of GSSAPI is to abstract all the variations across Kerberos implementations.¹⁰

Microsoft calls its proprietary version of the GSSAPI the Security Support Provider Interface (SSPI). It differs slightly from the RFCs for GSSAPI, but it does support protocol interoperability for Kerberos 5.

Let's write a basic client-server application to explore the inner workings of Kerberos/GSSAPI.

10. GSSAPI is defined in RFCs 1964, 2743, 2744, and 4121.

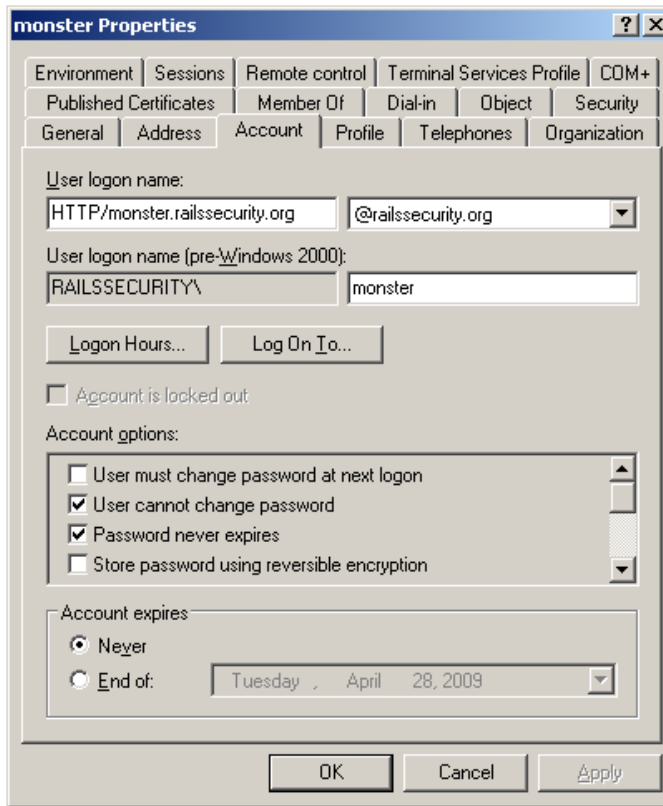


Figure 10.20: Monster properties

GSSServer and GSSClient

GSSAPI and Kerberos are written in C, so they are very procedural in their structure. We should also keep in mind that our Ruby bindings don't make an effort to reconcile the interface into an object-oriented approach.

Before we implement any server or client code, let's create a simpler API that we can reuse for our example. This will give us an opportunity to learn GSSAPI/Kerberos along the way.

The GSSAPI interface defines credential management calls, context-level calls, per-message calls, and support calls.

Let's begin with some basic credential management. Specifically, we want to gather credentials for a user. Thus, we need to define `draphael@RAILSSECURITY.ORG` in Active Directory.

We need to create a `GssBufferDescStruct` class. This is a Ruby class that wraps the GSSAPI-defined C struct, `gss_buffer_desc`. This struct is a buffer that holds data for various GSSAPI calls. Let's create an empty Ruby file and add this code:

[Download](#) `sso/gssapi_ruby/client_credentials.rb`

```
require 'gssapi_api'
include GSSAPI

client_name = "draphael@RAILSSECURITY.ORG"
buffer = GssBufferDescStruct.new
buffer.set_value_from_string(client_name)
```

So far we've created a buffer with our username; now let's import it into a format that GSSAPI can use:

```
results = gss_import_name(buffer, NT_USER_NAME)
major_status, minor_status, client_opaque_name = results
print_results(results, "gss_import_name")
results = gss_release_buffer buffer
print_results(results, "gss_release_buffer")
```

Here we've obtained an opaque name that we can pass to the next call. We also release the buffer—GSSAPI provides memory management as part of the standard. `NT_USER_NAME` indicates to GSSAPI that we are using a named user, which means we can now retrieve the credentials:

```
results = gss_acquire_cred(client_opaque_name, 0, NULL_OID_SET, GSS_C_INITIATE)
print_results(results, "gss_acquire_cred")
```

At this point, we can run the example. Depending on the OS and where it runs, whether in a graphical environment or from the command line, what we see will look something like Figure 10.21, on page 226.

If we've authenticated successfully, the acquired credentials should be in the credentials cache. We can check this using the `klist` command included with Kerberos:

```
Kerberos 5 ticket cache: 'API:Initial default ccache'
Default principal: draphael@RAILSSECURITY.ORG

Valid Starting      Expires            Service Principal
04/11/09 12:06:55  04/11/09 22:06:54  krbtgt/RAILSSECURITY.ORG@RAILSSECURITY.ORG
renew until 04/18/09 12:06:55
```


Some of the Calls Here Are Not GSSAPI

For example, the `GssBufferDescStruct` class enhances the SWIG interface-generation process. In this case, we added the `set_value_from_string()` method; GSSAPI doesn't ship with this functionality by default:

[Download](#) `sso/gssapi_ruby/gssapi.i`

```
%extend gss_buffer_desc_struct {
  void setValueFromString(char * new_value) {
    if(new_value == NULL)
      return;
    int len = strlen(new_value) + 1;
    char *str;
    str = (char*)malloc(sizeof(char) * len);
    strncpy(str, new_value, len);
    $self->value = str;
    $self->length = strlen(str) + 1;
  }

  void setValueFromArray(char * new_value, int length) {
    $self->value = new_value;
    $self->length = length;
  }
  char *getValueAsString() {
    return (char*) $self->value;
  }
}

%typemap(in,numinputs=0) char *OUT_ARRAY($1_ltype temp) {
  temp = ($1_ltype) malloc(sizeof($1_ltype));
  $1 = temp;
}

%typemap(in,numinputs=0) int *OUT_LEN($1_ltype temp) {
  temp = ($1_ltype) malloc(sizeof($1_ltype));
  $1 = temp;
}

%typemap(argout,fragment="output_helper") char *OUT_ARRAY {
  VALUE o = SWIG_FromCharPtrAndSize(arg1->value, arg1->length);
  $result = output_helper($result, o);
}

void getValueAsByteArray(char *OUT_ARRAY, int *OUT_LEN) {
  OUT_ARRAY = (char *)$self->value;
  OUT_LEN    = (int *)&$self->length;
}

};
```

SWIG generates Ruby classes from C structs. Sometimes, as in this case, those interfaces don't reconcile very well. The default method for handling strings from Ruby breaks in SWIG if we don't implement these specialized setters.

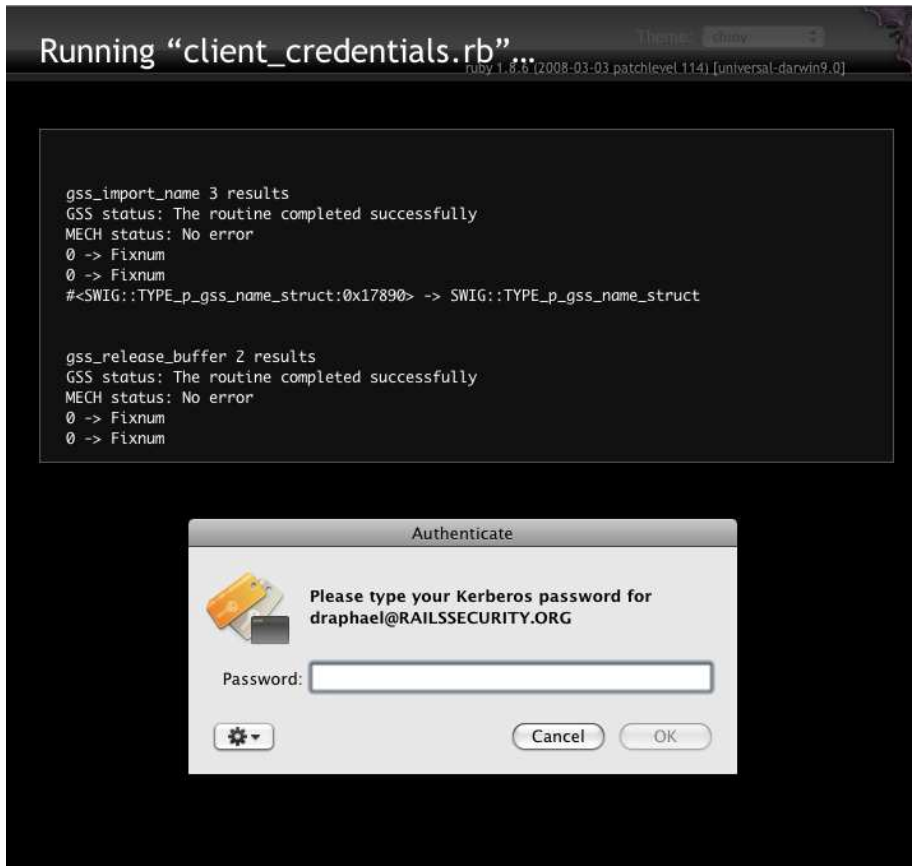


Figure 10.21: Acquiring credentials

If you're running OS X, you can also use the Kerberos extras GUI located in `/System/Library/CoreServices/Kerberos`. This tool shows a graphical interface to many of the options for Kerberos (see Figure 10.22, on the next page).

You have probably noticed that the GSSAPI methods return an Array object. Every array includes two results: Major Status and Minor Status. The values of these results translate into implementation-specific issues.

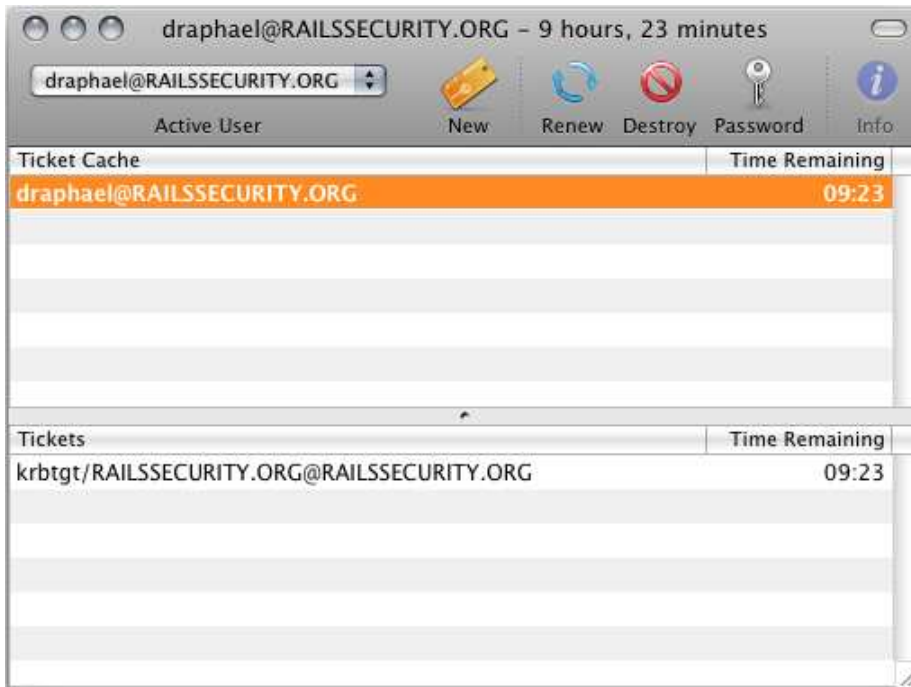


Figure 10.22: Kerberos GUI

We can make sense of these return values by adding a helper method:

```
def print_results(results, _name="UNDEFINED")
  puts "\n\n"
  puts "#{_name} #{results.length} results"
  print_errors(results[0], results[1])
  results.each do |r|
    puts "#{r} -> #{r.class}"
  end
end
```

Here is the supporting C code:

[Download](#) sso/gssapi_ruby/gssapi_wrap.c

```
void printErrors(OM_uint32 iMajorStatus, OM_uint32 iMinorStatus) {
  OM_uint32 message_context = 0;
  gss_buffer_desc status_string;

  OM_uint32 tempStatus = 0;
  gss_display_status( &tempStatus, iMajorStatus,
                    GSS_C_GSS_CODE, GSS_C_NULL_OID,
                    &message_context, &status_string );
}
```

<i>KNOCK_KNOCK</i>	Initial message to start communication
<i>SRV_REQ_CREDS</i>	Credentials Request
<i>INVALID_TOKEN</i>	Invalid Token
<i>CLIENT_ACK</i>	Client Acknowledgement
<i>ACCESS_GRANTED</i>	Access Granted
<i>ACCESS_DENIED</i>	Access Denied
<i>SECRET_REQUEST</i>	Client requests secret message

Figure 10.23: GssServer tokens

```

printf("GSS status: %s\n", status_string.value);

gss_display_status( &tempStatus, iMinorStatus,
                  GSS_C_MECH_CODE, GSS_C_NULL_OID,
                  &message_context, &status_string );

printf("MECH status: %s\n", status_string.value);

return;
}

```

This approach gives us have readable error messages—a nontrivial part of getting GSSAPI code up and running.

As we progress through our implementation, we will continue to add features to our API that ease the mismatch of GSSAPI and Ruby.

We have a basic understanding of how we intend to interact with GSS-API; next, we want to implement our client and server.

GssServer

Let's begin by defining a communication protocol for our GssServer and GssClient classes. You can find the list of tokens we'll exchange between our client and server in Figure 10.23.

Now let's define our token-based server-control flow (see Figure 10.24, on the next page).

So far we've specified our protocol; next, we need to create a server.rb file and add the tokens as constants.

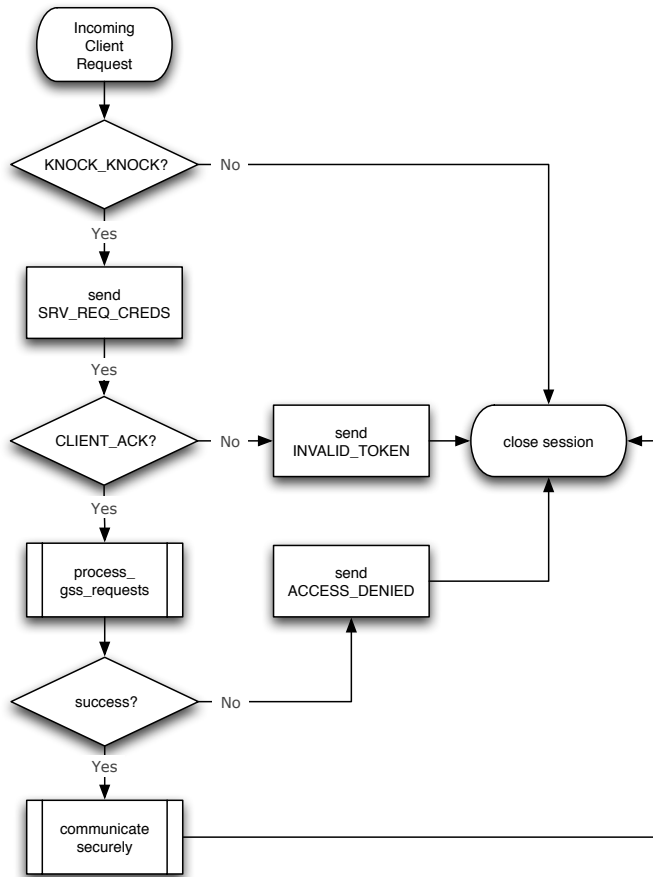


Figure 10.24: GssServer main flow

module Tokens

```

KNOCK_KNOCK    = "knock knock\n"
SRV_REQ_CREDS  = "credentials\n"
INVALID_TOKEN   = "I don't understand your request\n"
CLIENT_ACK     = "OK\n"
ACCESS_GRANTED  = "Access Granted\n"
ACCESS_DENIED   = "Access Denied\n"
SECRET_REQUEST  = "Secret Request\n"

```

end

We're now ready to add some network code for basic transmission. We do this by adding a method called `send_message()`:

```
def send_message(msg, socket)
  len = [msg.size].pack('N')
  socket.write(len)
  socket.write(msg)
end
```

This method packs the size of the data in a 4-byte integer and then transmits the size of the message, followed by the message data.

We also need to add a `read_message()` method:

```
def read_message(socket)
  if len = socket.read(4)
    msg = socket.read(len.unpack('N').to_s.to_i)
    msg
  end
end
```

This method reads the first 4 bytes of the message to determine its size. It then reads the amount of data specified by the size. Obviously, this code is not the most reliable network communication code, but we've kept it simple to demonstrate how Kerberos/GSSAPI can be implemented with any protocol.

Next, we need some higher-level network code to move our GSSAPI packets. Let's create another method called `send_msg()`:

```
def send_msg(gss_token, socket)
  srv_str = gss_token.get_value_as_byte_array
  token_string = Base64.encode64(srv_str)
  send_message(token_string, socket)
  gss_release_buffer gss_token
end
```

This method accepts a `gss_token` variable that we need to send over the network. This method uses the `send_message()` method that we just created. It also Base64-encodes the message.

Let's create the complementary method `recv_msg()` that Base64-decodes the message and constructs a new `GssBufferDescStruct` from the message:

```
def recv_msg(socket)
  if recv_string_b64 = read_message(socket)
    recv_string = Base64.decode64(recv_string_b64)
    recv_token = GssBufferDescStruct.new
    recv_token.set_value_from_byte_array(recv_string, recv_string.size)
    recv_token
  end
end
```

Notice that we have to pass the length of the string—we do this because the underlying wrappers assume that the string is null terminated. We must treat these opaque data types as raw data.

Now that we've created some basic network code, let's dive into our server logic. Let's begin by creating a constructor:

```
def initialize
  puts "[#{self.class}] Initializing"
  @secret_info = "Winning lotto: 1 2 4 11 21 9 8"
  @ctx = nil
  keytab_filename = "ruby4.keytab"
  service_name = "rubything/monster.railssecurity.org@RAILSSECURITY.ORG"
  @server_creds = acquire_service_cred(service_name, keytab_filename)
```

This first part of the initialize() method acquires credentials from the keytab file that we created in Section 10.4, *Setting Up the Kerberos Server*, on page 221. We also add a @secret_info variable; we need to send this variable to our client once it provides the correct credentials.

The code to acquire credentials depends on the acquire_server_cred() method:

```
def acquire_service_cred(service_name, keytab_filename)
  krb_5_gss_register_acceptor_identity(keytab_filename)
  buffer = GssBufferDescStruct.new
  buffer.set_value_from_string(service_name)
  results = gss_import_name(buffer, NT_USER_NAME)
  major_status, minor_status, opaque_name = results
  gss_release_buffer buffer
  results = gss_acquire_cred(opaque_name, 0, NULL_OID_SET, GSS_C_ACCEPT)
  major, minor, creds, oid_set, usage = results
  creds
end
```

We will continue creating our initialize() method by adding the main server routine:

```
server = TCPServer.new("127.0.0.1", 5000)
while(session = server.accept)
  hello = read_message(session)
  if(hello == KNOCK_KNOCK)
    send_message(SRV_REQ_CREDS, session)
    msg = read_message(session)
    if CLIENT_ACK == msg
      send_message(service_name,session)
      if process_gss_requests(session)
        req = read_message(session)
        if req == SECRET_REQUEST
          # Without crypto -> send_message(@secret_info,session)
          tell_secret(@secret_info, session)
        else
```

```

        send_message(INVALID_TOKEN,session)
    end
    else
        send_message(Access_Denied,session)
    end
end
else
    send_message(INVALID_TOKEN,session)
end
session.close
end
end

```

In the while loop, we handle a new request from the client, as shown in Figure 10.24, on page 229.

We also incorporate a `tell_secret()` method:

```

def tell_secret(msg, socket)
    buffer_to_encrypt = GssBufferDescStruct.new
    buffer_to_encrypt.set_value_from_byte_array(msg.clone, msg.size)
    session = g_wrap_message(@ctx, buffer_to_encrypt)
    encrypted_buffer = session.token
    send_msg(encrypted_buffer, socket)
end

```

The preceding code leverages GSSAPI message-level encryption. This aspect of GSSAPI leverages the encryption keys that have already been exchanged to protect data. Let's add the `g_wrap_message()` method to our API.

```

def g_wrap_message(ctx, buffer)
    results = gss_wrap(ctx, 1, 0, buffer)
    session = GSSSession.new
    session.results = results
    session.major = results[0]
    session.minor = results[1]
    session.token = results[3]
    session
end

```

We also need a `read_secret()` for the client we will write:

```

def read_secret(socket)
    recv_token = recv_msg(socket)
    decrypted_buffer = g_unwrap_message(@ctx, recv_token).token
    secret = decrypted_buffer.get_value_as_byte_array.clone
    gss_release_buffer decrypted_buffer
    secret
end

```


The `read_secret()` method decrypts the message using the `g_unwrap_message()` method:

```
def g_unwrap_message(ctx, buffer)
  results      = gss_unwrap(ctx, buffer)
  session      = GSSSession.new
  session.results = results
  session.major = results[0]
  session.minor = results[1]
  session.token = results[2]
  session
end
```

The `process_gss_requests()` method performs the authentication check (see Figure 10.24, on page 229). In this method, we handle the tokens that GSSAPI sends back and forth:

```
def process_gss_requests(session)
  results = []
  authenticated = false
  major_status = -5
  client_princ = nil
  begin
    recv_token = recv_msg(session)

    ▶ # Pass the token received from the GssClient
    ▶ gss_session =
    ▶   accept_sec_context(:ctx => nil,
    ▶                     :creds => @server_creds,
    ▶                     :token => recv_token)
    ▶ # We reset our context to whatever is passed
    ▶ # back from GSSAPI.
    @ctx = gss_session.ctx
    msg = gss_session.token.get_value_as_byte_array

    ▶ d_results = gss_display_name(gss_session.principal)
    ▶ client_princ = d_results[2].get_value_as_string
    ▶ puts "[#{self.class}] User requesting
    ▶       authentication: #{client_princ}"

    if(msg)
      send_msg(gss_session.token, session)
    end
    major_status = gss_session.major
    ▶ end while major_status == GSSAPI::GSS_S_CONTINUE_NEEDED
    ▶ if major_status == GSSAPI::GSS_S_COMPLETE
      authenticated = true
      puts "Client Authenticated!" if DEBUG
    else
      puts "[#{self.class}] Failure: #{major_status}"
      print_results(gss_session.results, self.class.to_s) if DEBUG
    end
    authenticated
  end
end
```

This method calls the `accept_sec_context()` method, which is the main GSSAPI method we use for server-side authentication.

In the `begin/end while` of `process_gss_requests()`, we loop and pass the current tokens to the `accept_sec_context()` method. Remember that GSSAPI abstracts the implementation details—some Kerberos implementations require that the client perform multiple steps. These steps are sent back to the client:

- `GSSAPI::GSS_S_CONTINUE_NEEDED` is returned until no more information is needed from the client.
- `GSSAPI::GSS_S_COMPLETE` is returned if the server has authenticated the client's credentials.
- All other return values indicate that the client was not authenticated.

This code implements the `accept_sec_context()` method for our developing API:

```
def accept_sec_context(args={})
  args.merge!(args)
  results = gss_accept_sec_context(
    args[:ctx],
    args[:creds],
    args[:token],
    args[:bindings]
  )

  session = GSSSession.new
  session.major = results[0]
  session.minor = results[1]
  session.ctx = results[2]
  session.principal = results[3]
  session.token = results[5]
  session.creds = results[8]
  session.results = results
  session
end
```

We also need a basic container to transport our C-like data; let's implement a `GSSSession` class:

```
class GSSSession
  attr_accessor :ctx
  attr_accessor :token
  attr_accessor :major
  attr_accessor :minor
  attr_accessor :creds
```

```

attr_accessor :results
attr_accessor :principal
attr_accessor :name
attr_accessor :oid
attr_accessor :oid_set
end

```

This class is nothing more than a data container that we can use to wrap our unfriendly GSSAPI results into a more readable format.

We can create a small, simple command-line executable file to run the server:

[Download](#) sso/gssapi_ruby/gss_server

```

#!/usr/bin/ruby
require 'server'
GssServer.run

```

Let's review the entire GssServer code and the API we've developed before we implement GssClient:

[Download](#) sso/gssapi_ruby/server.rb

```

require 'gssapi_api'
require 'socket'
require 'digest'
require 'base64'

class GssServer
  DEBUG = true

  attr_accessor :ctx
  module Tokens
    KNOCK_KNOCK      = "knock knock\n"
    SRV_REQ_CREDS    = "credentials\n"
    INVALID_TOKEN    = "I don't understand your request\n"
    CLIENT_ACK       = "OK\n"
    ACCESS_GRANTED   = "Access Granted\n"
    ACCESS_DENIED    = "Access Denied\n"
    SECRET_REQUEST   = "Secret Request\n"
  end

  module TokenMechanisms
    include GSSAPI
    include Tokens
    def send_msg(gss_token, socket)
      srv_str = gss_token.get_value_as_byte_array
      token_string = Base64.encode64(srv_str)
      send_message(token_string, socket)
      gss_release_buffer gss_token
    end
    def recv_msg(socket)

```

```

    if recv_string_b64 = read_message(socket)
      recv_string = Base64.decode64(recv_string_b64)
      recv_token = GssBufferDescStruct.new
      recv_token.set_value_from_byte_array(recv_string, recv_string.size)
      recv_token
    end
  end
  def send_message(msg, socket)
    len = [msg.size].pack('N')
    socket.write(len)
    socket.write(msg)
  end
  def read_message(socket)
    if len = socket.read(4)
      msg = socket.read(len.unpack('N').to_s.to_i)
      msg
    end
  end
  def tell_secret(msg, socket)
    buffer_to_encrypt = GssBufferDescStruct.new
    buffer_to_encrypt.set_value_from_byte_array(msg.clone, msg.size)
    session = g_wrap_message(@ctx, buffer_to_encrypt)
    encrypted_buffer = session.token
    send_msg(encrypted_buffer, socket)
  end
  def read_secret(socket)
    recv_token = recv_msg(socket)
    decrypted_buffer = g_unwrap_message(@ctx, recv_token).token
    secret = decrypted_buffer.get_value_as_byte_array.clone
    gss_release_buffer decrypted_buffer
    secret
  end
end

include Socket::Constants
include Tokens
include TokenMechanisms
include GSSAPI

def initialize
  puts "[#{self.class}] Initializing"
  @secret_info = "Winning lotto: 1 2 4 11 21 9 8"
  @ctx = nil
  keytab_filename = "ruby4.keytab"
  service_name = "rubything/monster.railssecurity.org@RAILSSECURITY.ORG"
  @server_creds = acquire_service_cred(service_name, keytab_filename)
  server = TCPServer.new("127.0.0.1", 5000)
  while(session = server.accept)
    hello = read_message(session)
    if(hello == KNOCK_KNOCK)
      send_message(SRV_REQ_CREDS, session)
    end
  end
end

```

```

msg = read_message(session)
if CLIENT_ACK == msg
  send_message(service_name,session)
  if process_gss_requests(session)
    req = read_message(session)
    if req == SECRET_REQUEST
      # Without crypto -> send_message(@secret_info,session)
      tell_secret(@secret_info, session)
    else
      send_message(INVALID_TOKEN,session)
    end
  else
    send_message(ACCESS_DENIED,session)
  end
end
end
else
  send_message(INVALID_TOKEN,session)
end
session.close
end
end

```

```

def process_gss_requests(session)
  results = []
  authenticated = false
  major_status = -5
  client_princ = nil
  begin
    recv_token = recv_msg(session)

    # Pass the token received from the GssClient
    gss_session =
      accept_sec_context(:ctx => nil,
                        :creds => @server_creds,
                        :token => recv_token)

    # We reset our context to whatever is passed
    # back from GSSAPI.
    @ctx = gss_session.ctx
    msg = gss_session.token.get_value_as_byte_array

    d_results = gss_display_name(gss_session.principal)
    client_princ = d_results[2].get_value_as_string
    puts "[#{self.class}] User requesting
          authentication: #{client_princ}"

    if(msg)
      send_msg(gss_session.token, session)
    end
    major_status = gss_session.major
  end while major_status == GSSAPI::GSS_S_CONTINUE_NEEDED
  if major_status == GSSAPI::GSS_S_COMPLETE

```

```

    authenticated = true
    puts "Client Authenticated!" if DEBUG
  else
    puts "[#{self.class}] Failure: #{major_status}"
    print_results(gss_session.results, self.class.to_s) if DEBUG
  end
  authenticated
end

def self.run
  GssServer.new
end
end

```

[Download](#) sso/gssapi_ruby/gssapi_api.rb

```
require File.expand_path(File.dirname(__FILE__) + "/GSSAPI")
```

```
module GSSAPI
```

```

  NT_USER_NAME = nt_user_name
  NULL_OID_SET = null_oid_set

```

```

class GSSSession
  attr_accessor :ctx
  attr_accessor :token
  attr_accessor :major
  attr_accessor :minor
  attr_accessor :creds
  attr_accessor :results
  attr_accessor :principal
  attr_accessor :name
  attr_accessor :oid
  attr_accessor :oid_set
end

```

```

def init_sec_context(args={})
  args.merge!(args)
  results = gss_init_sec_context(
    args[:creds],
    args[:ctx],
    args[:opaque_name],
    args[:mech],
    args[:flags] ||=
      GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG | GSS_C_DELEG_FLAG,
    args[:time_req] ||= 0,
    args[:bindings],
    args[:token_from_server]
  )
  session = GSSSession.new
  session.major = results[0]
  session.minor = results[1]
end

```

```

    session.ctx = results[2]
    session.token = results[4]
    session.results = results
  session
end

def accept_sec_context(args={})
  args.merge!(args)
  results = gss_accept_sec_context(
    args[:ctx],
    args[:creds],
    args[:token],
    args[:bindings]
  )

  session = GSSSession.new
  session.major = results[0]
  session.minor = results[1]
  session.ctx = results[2]
  session.principal = results[3]
  session.token = results[5]
  session.creds = results[8]
  session.results = results
  session
end

def display_name(opaque_name)
  results = gss_display_name(opaque_name)
  session = GSSSession.new
  session.major = results[0]
  session.minor = results[1]
  session.token = results[2]
  session
end

def print_results(results, _name="UNDEFINED")
  puts "\n\n"
  puts "#{_name} #{results.length} results"
  print_errors(results[0], results[1])
  results.each do |r|
    puts "#{r} -> #{r.class}"
  end
end

def import_name(buffer, name_type)
  results = gss_import_name(buffer, name_type)
  major_status, minor_status, opaque_name = results
  major, minor, creds, oid_set, usage = results
  session = GSSSession.new
  session.results = results
  session.major = results[0]

```

```

    session.minor = results[1]
    session.name  = results[2]
    session
end

def g_wrap_message(ctx, buffer)
  results      = gss_wrap(ctx, 1, 0, buffer)
  session      = GSSSession.new
  session.results = results
  session.major = results[0]
  session.minor = results[1]
  session.token = results[3]
  session
end

def g_unwrap_message(ctx, buffer)
  results      = gss_unwrap(ctx, buffer)
  session      = GSSSession.new
  session.results = results
  session.major = results[0]
  session.minor = results[1]
  session.token = results[2]
  session
end

def acquire_service_cred(service_name, keytab_filename)
  krb_5_gss_register_acceptor_identity(keytab_filename)
  buffer = GssBufferDescStruct.new
  buffer.set_value_from_string(service_name)
  results = gss_import_name(buffer, NT_USER_NAME)
  major_status, minor_status, opaque_name = results
  gss_release_buffer buffer
  results = gss_acquire_cred(opaque_name, 0, NULL_OID_SET, GSS_C_ACCEPT)
  major, minor, creds, oid_set, usage = results
  creds
end
end
end

```

GssClient

So far we have a server implementation; we still need to write a client that communicates with it. We can see our token-based client-control flow in [Figure 10.25](#), on page [242](#).

Before we start any network communication with the server, we need to have credentials available for authentication. Let's implement this logic in a `preauthenticate()` method:

```

def preauthenticate
  # Attempt to use logged in user account
  results = gss_acquire_cred(nil, 0, NULL_OID_SET, GSS_C_INITIATE)

```



```

unless(results[0] == GSSAPI::GSS_S_COMPLETE &&
        results[1] == GSSAPI::GSS_S_COMPLETE)
  puts "Please enter username:"
  client_name = gets.chomp
  buffer = GssBufferDescStruct.new
  buffer.set_value_from_string(client_name)
  results = gss_import_name(buffer, NT_USER_NAME)
  major_status, minor_status, client_opaque_name = results
  gss_release_buffer buffer
  results = gss_acquire_cred(client_opaque_name, 0,
                             NULL_OID_SET, GSS_C_INITIATE)
end

if(results[0] == GSSAPI::GSS_S_COMPLETE &&
   results[1] == GSSAPI::GSS_S_COMPLETE)
  true
else
  false
end
end
end

```

This method begins by calling the `gss_acquire_cred()` method with no principal specified; `nil` tells GSSAPI that we want to use the default credentials cache.

If we don't receive `GSS_S_COMPLETE` as the result, we need to gather credentials. The following steps enable us to request TGT from Authentication Server:

1. Request a principal name from our user.
2. Wrap the principal name in a `GssBufferDescStruct`.
3. Invoke `gss_acquire_cred()` with the freshly created buffer.

Once the call to `gss_acquire_cred()` is complete, we need to return `true` if the call completed successfully, but `false` if the call not complete successfully.

Let's put our `preauthenticate()` method to use in our `initialize()` method:

```

def initialize
  puts "[#{self.class}] Initializing"
  if preauthenticate
    socket = TCPSocket.new("127.0.0.1", 5000)
    send_message(KNOCK_KNOCK, socket)
    recv = read_message(socket)
  end
end

```

Once we preauthenticated the client successfully, we connect to the server, send `KNOCK_KNOCK`, and receive the response.

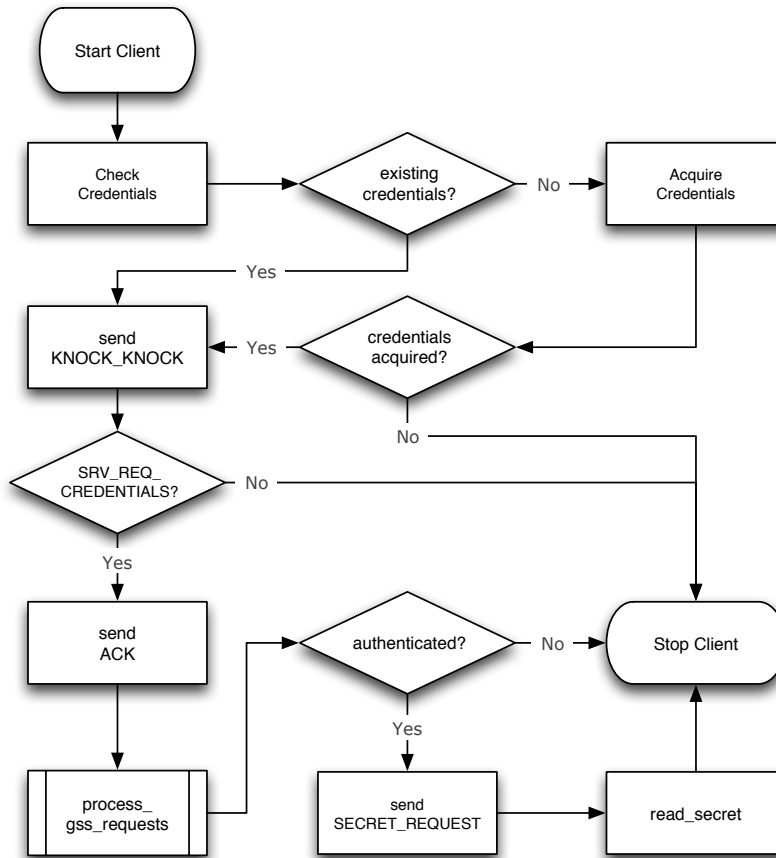


Figure 10.25: GssClient main flow

Next, we check the response:

```

authenticated = false
if(recv == SRV_REQ_CREDS)
    send_message(CLIENT_ACK, socket)
    authenticated = process_gss_requests(socket)
end

```

If the server response is `SRV_REQ_CREDS`, then we invoke the `process_gss_requests()` method.

If `process_gss_requests()` is successful—which means we’ve successfully authenticated against the server—we send the `SECRET_REQUEST` token:

```

if authenticated
  puts "[#{self.class}] Getting secret" if DEBUG
  send_message(SECRET_REQUEST, socket)
  secret = read_secret(socket)
  puts "[#{self.class}] Secret: #{secret}"
else
  msg = read_message(socket)
  puts "[#{self.class}] #{msg}"
end
socket.close
else
  puts "Unable to collect Valid Credentials."
end
end

```

Here, we decrypt the secret message using `read_secret()`, which we defined in Section 10.4, `GssServer`, on page 228. Also, we can close the socket. As we did in our `GssServer` class, we must implement the `process_gss_requests()` method in our client class. Let’s take a look at this method’s code:

```

def process_gss_requests(socket)
  service_name = read_message(socket)
  puts "[#{self.class}] Service name: #{service_name.inspect}"

```

We gather the name of the service and print it to the screen. This call simply reads the `service_name` variable that the server sends after the client sends `CLIENT_ACK`.

Next, let’s process the service name into a GSSAPI-compatible format:

```

buffer = GssBufferDescStruct.new
buffer.set_value_from_string(service_name.chomp)
results = gss_import_name(buffer, NT_USER_NAME)
major_status, minor_status, opaque_name = results
gss_release_buffer buffer

```

The `opaque_name` plays a key role in our next chunk of code:

```

recv_token = nil
gss_session = nil
authenticated = false
begin
  puts "[#{self.class}] init_sec_context =>" if DEBUG
  gss_session = init_sec_context({
    :opaque_name => opaque_name,
    :ctx => self.ctx,
    :token_from_server => recv_token })
  major_status = gss_session.major

```

```

# We have to use 'self' here to ensure that the variable is
# scoped to the object instance level and not the block level.
self.ctx      = gss_session.ctx
srv_str       = gss_session.token.get_value_as_byte_array
if srv_str && major_status == GSSAPI::GSS_S_CONTINUE_NEEDED
  puts "[#{self.class}] #{self.class} Sending token" if DEBUG
  send_msg(gss_session.token, socket)
  puts "[#{self.class}] receiving token" if DEBUG
  rcv_token = rcv_msg(socket)
else
  puts "[#{self.class}] no more tokens to process" if DEBUG
end

end while(major_status == GSSAPI::GSS_S_CONTINUE_NEEDED)

```

As we did in our server code, we loop until the GSSAPI calls complete. Next, we verify that we are authenticated:

```

if(major_status == GSSAPI::GSS_S_COMPLETE)
  puts "[#{self.class}] Client successfully authenticated!" if DEBUG
  puts "[#{self.class}] Major Status: #{major_status}" if DEBUG
  authenticated = true
else
  puts "[#{self.class}] Client failed"
  print_results(gss_session.results, self.class.to_s) if DEBUG
  send_message("Failed\n.", socket)
end
authenticated
end

```

Let's perform a final review of this code before we run the example:

Download `ss0/gssapi_ruby/client.rb`

```

require 'gssapi_api'
require 'socket'
require 'server'
require 'digest'
require 'base64'
require 'pp'

class GssClient
  include GssServer::Tokens
  include GssServer::TokenMechanisms
  include GSSAPI

  attr_accessor :ctx
  DEBUG = true

  def initialize
    puts "[#{self.class}] Initializing"
    if preauthenticate
      socket = TCPSocket.new("127.0.0.1", 5000)

```

```

send_message(KNOCK_KNOCK, socket)
recv = read_message(socket)
puts "[#{self.class}] Received: #{recv}" if DEBUG
authenticated = false
if(recv == SRV_REQ_CREDS)
  send_message(CLIENT_ACK, socket)
  authenticated = process_gss_requests(socket)
end
if authenticated
  puts "[#{self.class}] Getting secret" if DEBUG
  send_message(SECRET_REQUEST, socket)
  secret = read_secret(socket)
  puts "[#{self.class}] Secret: #{secret}"
else
  msg = read_message(socket)
  puts "[#{self.class}] #{msg}"
end
socket.close
else
  puts "Unable to collect Valid Credentials."
end
end

def preauthenticate
  # Attempt to use logged in user account
  results = gss_acquire_cred(nil, 0, NULL_OID_SET, GSS_C_INITIATE)

  unless(results[0] == GSSAPI::GSS_S_COMPLETE &&
          results[1] == GSSAPI::GSS_S_COMPLETE)
    puts "Please enter username:"
    client_name = gets.chomp
    buffer = GssBufferDescStruct.new
    buffer.set_value_from_string(client_name)
    results = gss_import_name(buffer, NT_USER_NAME)
    major_status, minor_status, client_opaque_name = results
    gss_release_buffer buffer
    results = gss_acquire_cred(client_opaque_name, 0,
                              NULL_OID_SET, GSS_C_INITIATE)
  end

  if(results[0] == GSSAPI::GSS_S_COMPLETE &&
     results[1] == GSSAPI::GSS_S_COMPLETE)
    true
  else
    false
  end
end

def process_gss_requests(socket)
  service_name = read_message(socket)
  puts "[#{self.class}] Service name: #{service_name.inspect}"

```

```

buffer = GssBufferDescStruct.new
buffer.set_value_from_string(service_name.chomp)
results = gss_import_name(buffer, NT_USER_NAME)
major_status, minor_status, opaque_name = results
gss_release_buffer buffer
recv_token = nil
gss_session = nil
authenticated = false
begin
  puts "[#{self.class}] init_sec_context =>" if DEBUG
  gss_session = init_sec_context({
    :opaque_name => opaque_name,
    :ctx => self.ctx,
    :token_from_server => recv_token })
  major_status = gss_session.major
  # We have to use 'self' here to ensure that the variable is
  # scoped to the object instance level and not the block level.
  self.ctx = gss_session.ctx
  srv_str = gss_session.token.get_value_as_byte_array
  if srv_str && major_status == GSSAPI::GSS_S_CONTINUE_NEEDED
    puts "[#{self.class}] #{self.class} Sending token" if DEBUG
    send_msg(gss_session.token, socket)
    puts "[#{self.class}] receiving token" if DEBUG
    recv_token = recv_msg(socket)
  else
    puts "[#{self.class}] no more tokens to process" if DEBUG
  end
end while(major_status == GSSAPI::GSS_S_CONTINUE_NEEDED)
if(major_status == GSSAPI::GSS_S_COMPLETE)
  puts "[#{self.class}] Client successfully authenticated!" if DEBUG
  puts "[#{self.class}] Major Status: #{major_status}" if DEBUG
  authenticated = true
else
  puts "[#{self.class}] Client failed"
  print_results(gss_session.results, self.class.to_s) if DEBUG
  send_message("Failed\n.", socket)
end
authenticated
end
end
GssClient.new

```

Running GssClient and GssServer

We've written the server and client code; now let's try to run them. Let's start by running the server in a terminal window:

```

[monster - draphael][gssapi_ruby]> ./gss_server
[GssServer] Initializing

```

Before we can run our client, let's make sure we don't have any tickets in the Kerberos credentials cache by deleting any Kerberos credentials from that cache:

```
[monster - draphael][gssapi_ruby]> kdestroy
```

We can verify we don't have any Kerberos tickets in the cache by running `klist`:

```
[monster - draphael][gssapi_ruby]> klist
klist: No Kerberos 5 tickets in credentials cache
```

We can start the client after we verify that the credentials cache is empty:

```
⇐ [monster - draphael][gssapi_ruby]> ruby client.rb
⇒ [GssClient] Initializing
Please enter username:
⇐ draphael@RAILSSECURITY.ORG
⇒ Please enter the password for draphael@RAILSSECURITY.ORG:
⇐ *****
⇒ [GssClient] Received: credentials
[GssClient] Service name: "rubything/monster.railssecurity.org@RAILSSECURITY.ORG"
[GssClient] init_sec_context =>
[GssClient] GssClient Sending token
[GssClient] receiving token
[GssClient] init_sec_context =>
[GssClient] no more tokens to process
[GssClient] Client successfully authenticated!
[GssClient] Major Status: 0
[GssClient] Getting secret
[GssClient] Secret: Winning lotto: 1 2 4 11 21 9 8
```

Let's check `klist` again:

```
[monster - draphael][gssapi_ruby]> klist
Kerberos 5 ticket cache: 'API:Initial default ccache'
Default principal: draphael@RAILSSECURITY.ORG
```

```
Valid Starting      Expires              Service Principal
04/18/09 15:32:07   04/19/09 01:32:06   krbtgt/RAILSSECURITY.ORG@RAILSSECURITY.ORG
renew until 04/25/09 15:32:07
04/18/09 15:32:06   04/19/09 01:32:06   rubything/monster.railssecurity.org@RAI...
renew until 04/25/09 15:32:06
```

You can see here that we have credentials that were previously not in our cache. So what happens if we run our client again?

```
⇐ [monster - draphael][gssapi_ruby]> ruby client.rb
⇒ [GssClient] Initializing
[GssClient] Received: credentials
[GssClient] Service name: "rubything/monster.railssecurity.org@RAILSSECURITY.ORG"
[GssClient] init_sec_context =>
```

```
[GssClient] GssClient Sending token
[GssClient] receiving token
[GssClient] init_sec_context =>
[GssClient] no more tokens to process
[GssClient] Client successfully authenticated!
[GssClient] Major Status: 0
[GssClient] Getting secret
[GssClient] Secret: Winning lotto: 1 2 4 11 21 9 8
```

Notice that we didn't have to enter our credentials this time. You might be wondering how this is useful for Rails applications; let's look at how we integrate GSSAPI into our Rails applications via SPNEGO.

SPNEGO

The Simple and Protected GSSAPI Negotiation Mechanism—commonly referred to as SPNEGO—was designed as a standard for negotiating authentication between two peers. It fills a gap in GSSAPI—GSSAPI provides standards for peers that have already acquired the same type of credentials, but it doesn't deal with peers that support multiple authentication mechanisms over the same network protocol.

We will focus on HTTP as the network protocol for SPNEGO in this section, but be aware that SPNEGO can operate over many different of network protocols.

Microsoft started integrating Kerberos and NTLM authentication into IIS and Internet Explorer several years ago. In conjunction with this work, Microsoft published an extension to SPNEGO that defines NTLM and Kerberos authentication over HTTP.¹¹

Until recently, Internet Explorer and IIS were the only systems that supported SPNEGO; however, Firefox, Safari, and several others have now joined the list of browsers that support SPNEGO.

So how do we support SPNEGO in Rails?

Rails and SPNEGO

It turns out that we don't have to do much to support SPNEGO in Rails—that is, we don't have to do much more than we've already done to support GSSAPI in Section 10.4, *Kerberos/GSSAPI*, on page 222.

11. <http://tools.ietf.org/html/rfc4559>

Let's open up our SessionController and add some initialization code:

```
include GSSAPI
def check_login
  @ctx = nil
  keytab_filename = "#{RAILS_ROOT}/config/monster.keytab"
  @server_creds = acquire_service_cred(nil, keytab_filename)
```

This code is very similar to what we included in GssServer. The only difference here is that we use a different keytab file. We created this keytab with the service name, HTTP/monster.railssecurity.org. Note the use of HTTP in the beginning of the name; including that is a critical aspect of making our code work correctly.

Next, we need to strip the GSSAPI Token from the HTTP header:

```
if auth_header = request.headers['HTTP_AUTHORIZATION']
  logger.debug { "first 10: [#{auth_header[0..9]}]" }
  token = auth_header[10..-1]
```

This code strips off the first ten characters of the header value; the rest of the header value is a Base64-encoded token.

Now, let's pass this token along to GSSAPI:

```
recv_string = Base64.decode64(token)
recv_token = GssBufferDescStruct.new
recv_token.set_value_from_byte_array(recv_string,recv_string.size)
gss_session =
  accept_sec_context(:creds => @server_creds,
                    :token => recv_token)
```

As this code illustrates, we need to decode the token first, but the rest of the code operates like our previous GSSAPI example.

Finally, let's process the result:

```
case gss_session.major
when GSSAPI::GSS_S_CONTINUE_NEEDED
  render :text => "not quite there.", :status => 401
when GSSAPI::GSS_S_COMPLETE
  d_results = gss_display_name(gss_session.principal)
  username = d_results[2].get_value_as_string
  # render(:text => "Authenticated. User: #{username}",
  #       :status => 200)
  @authenticated_user = User.create_or_find_from_kerb Princ(username)
  session[:user_id] = @authenticated_user.id
  redirect_to user_path(@authenticated_user)
else
  render(:text => "Access Denied: #{gss_session.major}",
        :status => 401)
end
```

```

else
  render(:text => "\nHeaders:\n#{request.headers.to_a.join("<br/>")}",
        :status => 401)

  logger.debug { "Negotiate initiated" }
  response.headers['WWW-Authenticate'] = 'Negotiate'
end
end

def logout
  reset_session
end
end

```

This code is pretty rough, but it demonstrates the basic idea: the browser does the heavy lifting for us. We also need to tweak our login screen (see Figure 10.26, on the following page):

Download [sso/lunchedin_spnego/app/views/session/login.html.erb](#)

```

<% if flash[:notice] %>
<div class="notice">
  <%= h(flash[:notice]) %>
</div>
<% end %>

<div style="margin-top:14px">
<% form_tag :controller => 'session', :action => "check_login" do %>
  <br/>
  <%= submit_tag 'Login with Integrated Kerberos Login' %>
<% end %>
</div>

```

Finally, we need to add a method for user creation in our User model:

Download [sso/lunchedin_spnego/app/models/user.rb](#)

```

# This is not the ideal implementation
# You will have to adapt this to your needs
def self.create_or_find_from_kerb Princ(kerb Princ)
  user = find_by_username(kerb Princ)
  unless user
    user = User.new do |u|
      u.username = kerb Princ
      u.email = "(email)"
      u.first_name = "(first_name)"
      u.last_name = "(last_name)"
      u.zip_code = "75093"
    end
    user.save(false)
  end
  user
end

```

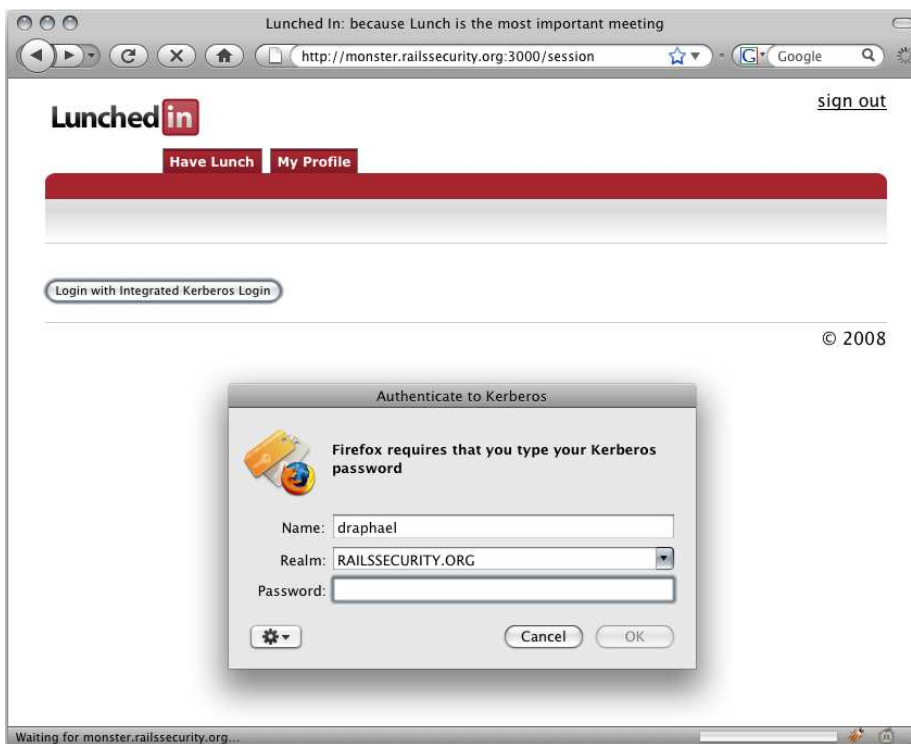


Figure 10.26: SPNEGO login dialog

Running this example should enable us to log in and see a screen like Figure 10.27, on the next page.

10.5 CAS

Central Authentication Service, or CAS, was developed at Yale University as a way to decouple authentication from each individual web application on its networks, as well as to provide users with the ability to authenticate against many systems without providing their passwords to each individual system.

CAS also introduces the notion of a ticket-granting cookie (TGC), which acts very much like a TGT from Kerberos. The TGC provides the user with an SSO experience. We can see a simplified overview of CAS's authentication flow in Figure 10.28, on the following page.

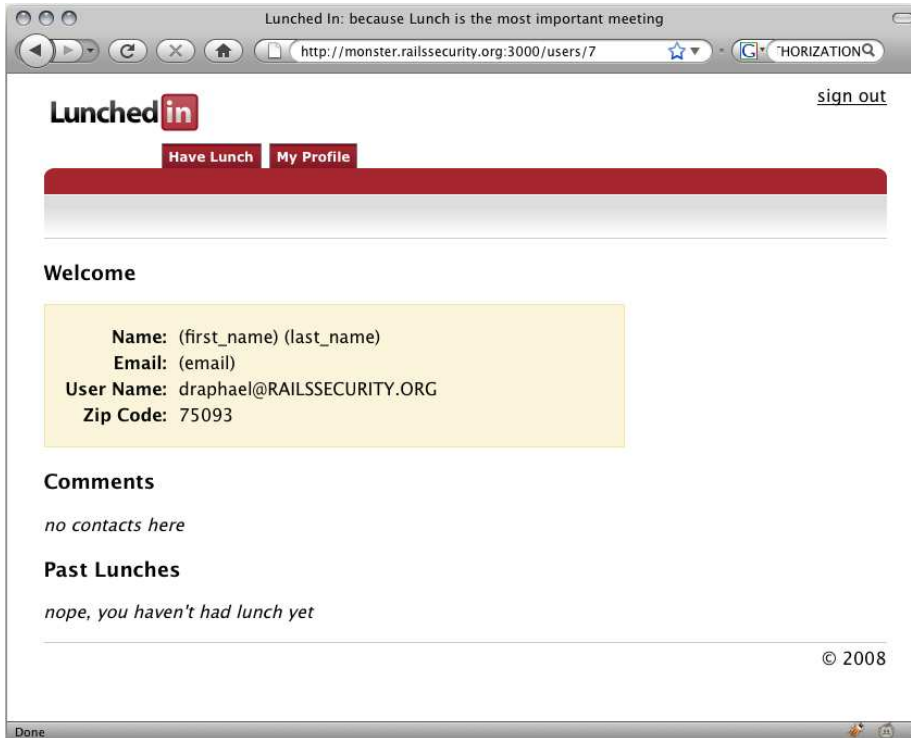


Figure 10.27: SPNEGO login success

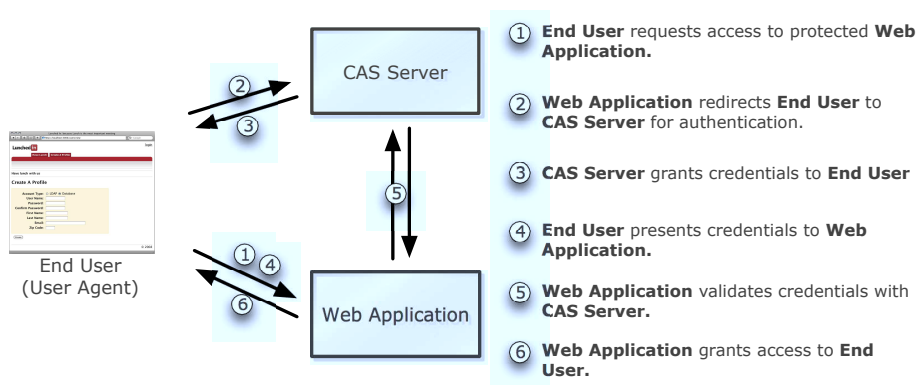


Figure 10.28: CAS authentication flow

Yale University created its implementation of CAS in Java. However, you can find a pair of CAS implementations written in Ruby: Castronaut and RubyCAS.

Our example will use the Castronaut implementation, which was implemented by Relevance LLC and stands out as the best Ruby-based implementation of CAS in the opinion of this book's authors.

Castronaut

The folks at Relevance implemented Castronaut as an open source project, and you can find a link to install a gem for it at Relevance's website.¹²

To get started, let's install the gem:

```
> sudo gem install relevance-castronaut --source http://gems.github.com
```

We also need the client plug-in:

```
> ./script/plugin install\
http://rubycas-client.googlecode.com/svn/trunk/rubycas-client
```

Next, we need to create a config file for Castronaut:

[Download](#) sso/lunchedin_castronaut/config/castronaut.yml

```
organization_name: Rails Security
```

```
environment: development
```

```
server_port: 4567
```

```
log_directory: log
```

```
log_level: Logger::DEBUG
```

```
ssl_enabled: false
```

```
cas_database:
```

```
  adapter: sqlite3
```

```
  database: db/cas.db
```

```
  timeout: 5000
```

```
cas_adapter:
```

```
  adapter: ldap
```

```
  host: localhost
```

```
  port: 10389
```

```
  prefix: uid=
```

```
  base: dc=railssecurity,dc=org
```

12. <http://thinkrelevance.com/open-source>

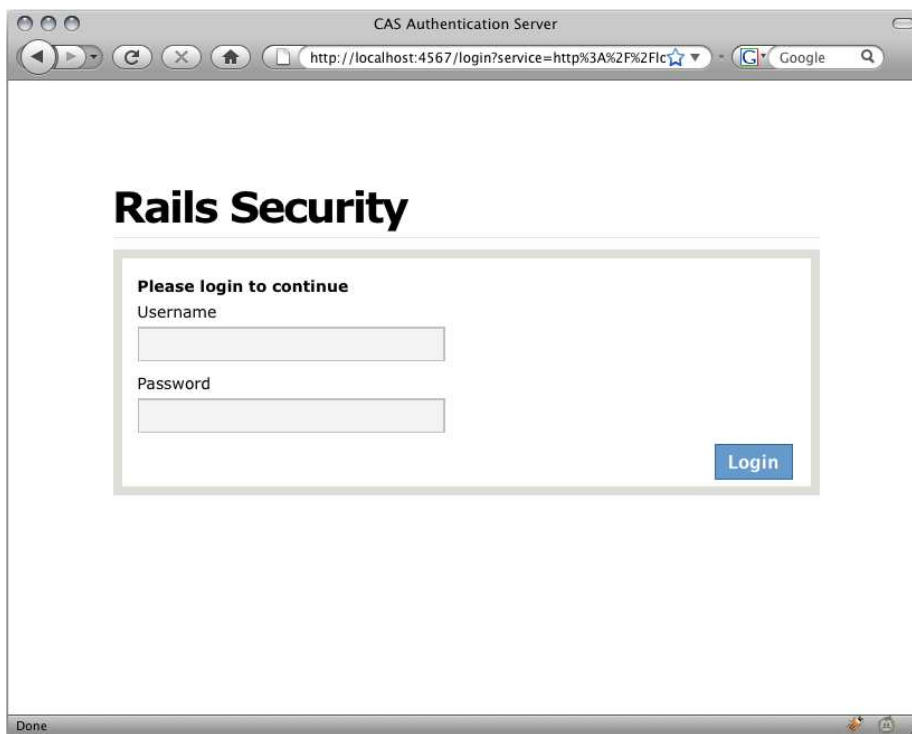


Figure 10.29: Castronaut login screen

Finally, let's fire up our Castronaut server and make sure it's working:

```
[monster - draphae][lunchedin_castronaut]> castronaut
No configuration file specified. Defaulting to ./config/castronaut.yml
== Sinatra/0.9.0.2 has taken the stage on 4567 for development with backup from Thin
>> Thin web server (v1.0.0 codename That's What She Said)
>> Maximum connections set to 1024
>> Listening on 0.0.0.0:4567, CTRL+C to stop
```

Navigating to <http://localhost:4567> should bring up something like Figure 10.29.

Adding CAS Authentication to LunchedIn

Now that we have a running Castronaut server, let's create CAS support for LunchedIn.

First, add the necessary support code to our Rails environment:

[Download](#) sso/lunchedin_castronaut/config/environment.rb

```
RAILS_GEM_VERSION = '2.2.2' unless defined? RAILS_GEM_VERSION

require File.join(File.dirname(__FILE__), 'boot')

Rails::Initializer.run do |config|
  config.action_controller.session = {
    :session_key => '_lunchedin_session',
    :secret      => '2b6d57c18f8884b65d248d72bf97e0f62e3eaa5ad3ba9eb6fef402109'+
      '386ad0468758d2a3f19bb8f707e6a387531dfe344c9e99f699a8735f661a536f054d0de'
  }
end

# BEGIN_HIGHLIGHT
CASClient::Frameworks::Rails::Filter.configure(
  :cas_base_url => "http://localhost:4567/",
  :login_url   => "http://localhost:4567/login",
  :logout_url  => "http://localhost:4567/logout"
)
```

We do need to make some changes to the ApplicationController class:

[Download](#) sso/lunchedin_castronaut/app/controllers/application.rb

```
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time

  protect_from_forgery # :secret => 'cf9d327f2d16793b606c6689f76e90c3'

  before_filter CASClient::Frameworks::Rails::Filter, :except => [ :logout ]
  before_filter :check_authentication, :except => [ :logout ]

  def check_authentication
    if session[:cas_user]
      @authenticated_user = User.find_by_username(session[:cas_user])
      if @authenticated_user
        session[:user_id] = @authenticated_user.id
        return true
      else
        redirect_to :controller => "users", :action => :new and return
      end
    end
  end
end
```

[Download at WoweBook.com](#)

The first change you will probably notice is the addition of a new filter provided by the CAS plug-in.

In our previous examples, we checked the value of `session(:user_id)`. Here we need to check the value of `session(:cas_user)`, which is injected by

CASClient, a class included in the CAS client plug-in. If we can't find the user, we redirect him to the new action of UsersController.

Next, let's take a look at our UsersController class:

[Download](#) sso/lunchedin_castronaut/app/controllers/users_controller.rb

```

class UsersController < ApplicationController
  skip_before_filter :check_authentication, :only => [:new, :create]
  def new
    @user = User.new
  end

  def show
    @user = User.find(params[:id])
    respond_to do |format|
      format.html
    end
  end

  def create
    @user = User.new(params[:user])
    @user.username = session[:cas_user]
    respond_to do |format|
      if @user.save
        flash[:notice] = 'your account was successfully created.'
        session[:user_id] = @user.id
        format.html { redirect_to(@user) }
      else
        format.html { render :action => 'new' }
      end
    end
  end
end
end

```

In this file, all we modify only the `skip_before_filter` variable and then tweak the `create()` method to add the username to the `@user` prior to saving. We also need to change our User model:

[Download](#) sso/lunchedin_castronaut/app/models/user.rb

```

class User < ActiveRecord::Base
  attr_protected :role_id
  validates_presence_of :email, :first_name, :last_name
  validates_presence_of :username
  validates_uniqueness_of :username
  validates_length_of :zip_code, :is => 5
  validates_numericality_of :zip_code
  has_many :events
  has_many :comments, :as => :commentable
  belongs_to :role
end

```


We don't need password validation because we rely on CAS to provide authentication. Indeed, we could remove those columns from the database altogether. Additionally, this code removes the password and authentication methods that existed previously in our User model.

Let's have a look at the corresponding user view:

Download `sso/lunchedin_castronaut/app/views/users/new.html.erb`

```
<h1>Have lunch with us</h1>
<h2>Create A Profile</h2>

<%= error_messages_for 'user' %>
<% form_tag :action => 'create' do %>

<div class="highlighted-form">
  <table class="form">
    <tr>
      <td class="label">CAS Username:</td>
      <td><%= session[:cas_user] %></td>
    </tr>
    <tr>
      <td class="label">First Name:</td>
      <td><%= text_field('user', 'first_name', :size => 16) %></td>
    </tr>
    <tr>
      <td class="label">Last Name:</td>
      <td><%= text_field('user', 'last_name', :size => 16) %></td>
    </tr>
    <tr>
      <td class="label">Email:</td>
      <td><%= text_field('user', 'email') %></td>
    </tr>
    <tr>
      <td class="label">Zip Code:</td>
      <td><%= text_field('user', 'zip_code', :size => 6) %></td>
    </tr>
  </table>
</div>
<div style="padding: 10px;">
  <%= submit_tag 'Create' %>
</div>
<% end %>
```

Here, we remove the password fields and hard-code the username with the `session(:cas_user)` value.

Assuming we input all the changes correctly, an attempt to log in to LunchedIn should result in a page like the one in Figure 10.30, on the following page.

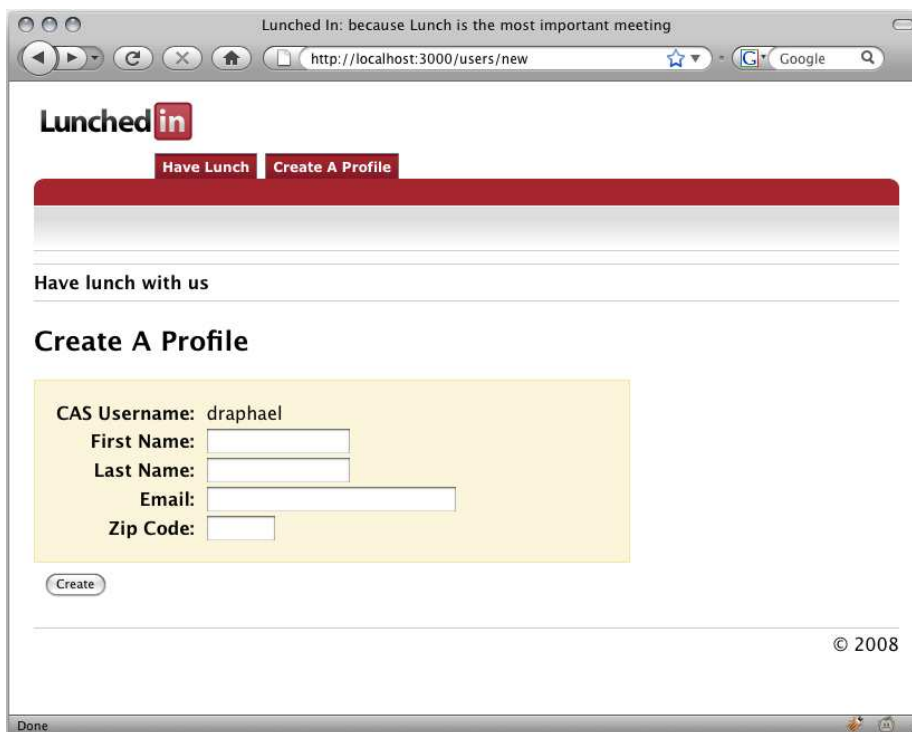


Figure 10.30: LunchedIn registration screen with Castronaut

And if we create the user, we should be redirected to a screen like the one in Figure 10.31, on the next page.

The last thing we need to modify is the `SessionController` class:

Download `sso/lunchedin_castronaut/app/controllers/session_controller.rb`

```
class SessionController < ApplicationController
  skip_before_filter :check_authentication

  def index
    render :action => 'login'
  end

  def logout
    reset_session
    CASClient::Frameworks::Rails::Filter.logout(self)
  end
end
```

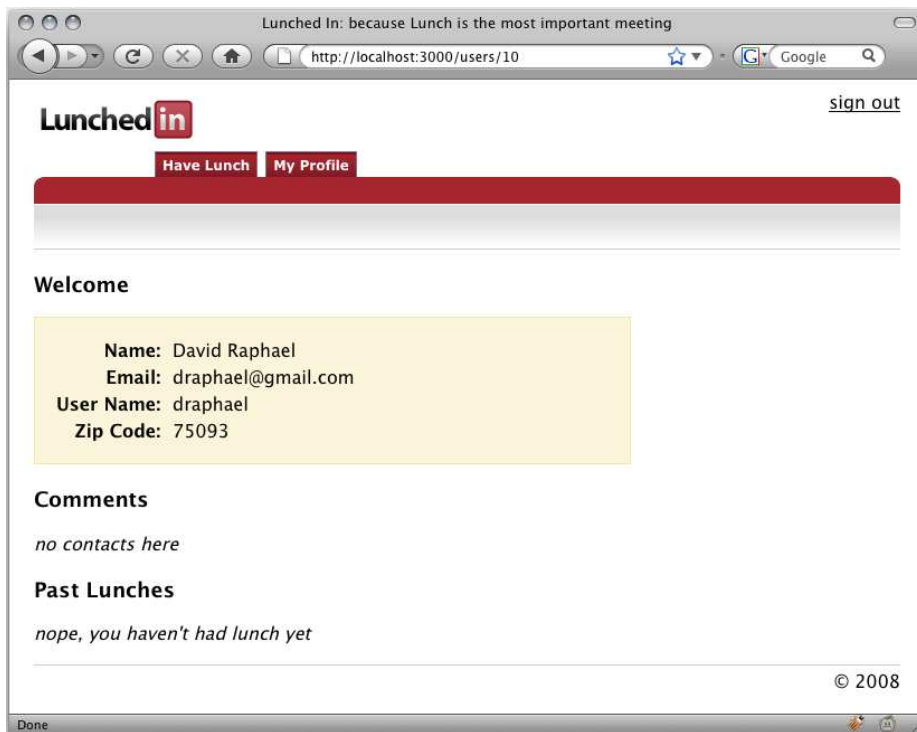


Figure 10.31: LunchedIn registration success with Castronaut

We don't change much in this code, apart from the call to `CASClient::Frameworks::Rails::Filter.logout()`, which redirects a user to the CAS logout screen.

Part III

Reference

Web Application Proxies

11.1 Web Application Proxies

MouseHole isn't the only way to manipulate the requests and responses of an application. There is a list of additional proxy applications that serve a similar purpose:

SPIKE Proxy <http://www.immunitysec.com>
SPIKE Proxy is a professional-grade tool for looking for application-level vulnerabilities in web applications. SPIKE Proxy covers the basics, such as SQL Injection and cross-site scripting, but it also features a completely open Python infrastructure that allows advanced users to customize it for web applications that other tools fall apart on. SPIKE Proxy is available for Linux and Windows.

Paros <http://www.parosproxy.org>
Paros is a proxy for those who need to evaluate the security of their web applications. It is free of charge and written completely in Java. Paros's proxy nature lets you intercept and modify all HTTP and HTTPS data between server and client, including cookies and form fields.

Burp Proxy <http://www.portswigger.net>
Burp Proxy is an interactive HTTP/S proxy server for attacking and debugging web-enabled applications. It operates as a man-in-the-middle between the end browser and the target web server, and it allows the user to intercept, inspect, and modify the raw traffic passing in both directions.

Achilles Proxy <http://www.mavensecurity.com/achilles>
Achilles acts as an HTTP/HTTPS proxy that allows a user to intercept, log, and modify web traffic on the fly.

Odysseus Proxy <http://www.wastelands.gen.nz/odysseus/>
Odysseus is a proxy server, which acts as a man-in-the-middle during an HTTP session. A typical HTTP proxy will relay packets to and from a client browser

and a web server. Odysseus will intercept an HTTP session's data in either direction and give the user the ability to alter the data before transmission.

Webstretch Proxy <http://sourceforge.net/projects/webstretch>

Webstretch enables a user to view and alter all aspects of communications with a website via a proxy. Developers use this tool primarily for security-based penetration testing of websites, but it can also prove useful for debugging during development.

Firefox Developer Tools <http://www.mozdev.org>

The Mozilla developer site contains a host of plug-ins that might be useful for security testing.

Fiddler <http://www.fiddler2.com/fiddler2/>

Fiddler is a web application proxy written in .NET that provides deep integration with Internet Explorer and decryption functionality of HTTPS (with appropriate certificates and keys).

Authentication Appendix

12.1 OpenLDAP Setup

On Linux or OS X systems, OpenLDAP typically comes preinstalled on Linux and OS X systems. If you already have an LDAP server running, then you can use that configuration to run this example. Otherwise, you will need to obtain OpenLDAP. You must include the following schema files in your configuration file (their specific location is platform-dependent):

[Download](#) authentication/lunchedin_i3_old/doc/slapd.conf

```
include      /etc/openldap/schema/core.schema
include      /etc/openldap/schema/cosine.schema
include      /etc/openldap/schema/inetorgperson.schema
```

Note that these schemas aren't included by default, but they are important because they define the constraints of our directory's data. Additionally, we need to define a rootdn:

[Download](#) authentication/lunchedin_i3_old/doc/slapd.conf

```
database      bdb
suffix        "dc=railssecurity,dc=com"
rootdn        "cn=Manager,dc=railssecurity,dc=com"
# Cleartext passwords, especially for the rootdn, should
# be avoid. See slapasswd(8) and slapd.conf(5) for details.
# Use of strong authentication encouraged.
rootpw        secret
```

In a production environment, we don't want to store a plaintext password in our configuration. Indeed, we should follow the same practices in our directory server that we would follow for any database server. Once we configure and start our server, we can initialize it with some data.

Here is a sample:

[Download](#) authentication/lunchedin_i3_old/doc/railssecurity.ldif

```
dn: dc=railssecurity, dc=com
dc: railssecurity
objectClass: organization
objectClass: dcObject
o: railssecurity.com
```

```
dn: ou=people, dc=railssecurity, dc=com
ou: people
objectClass: organizationalUnit
```

```
dn: cn=wally,ou=people, dc=railssecurity, dc=com
mail: wally@yahoo.com
userPassword:: cGFzc3dvcmQ=
uid: wally
ou: people
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: World
cn: Wally World
cn: wally
```

```
dn: cn=bob,ou=people, dc=railssecurity, dc=com
mail: bob@bobsworld.com
userPassword:: aGVsbG8=
uid: bob
ou: people
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: World
cn: Bob World
cn: bob
```

```
dn: cn=dave,ou=people, dc=railssecurity, dc=com
mail: dave@davesworld.com
userPassword:: aGVsbG8=
uid: dave
ou: people
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: World
cn: Dave World
cn: Dave
```


We load with the preceding sample with this command:

```
> ldapadd -x -w 'secret' -D\
'cn=Manager,dc=railssecurity,dc=com' -f\
railssecurity.ldif \

adding new entry "dc=railssecurity, dc=com"
adding new entry "ou=people, dc=railssecurity, dc=com"
adding new entry "cn=wally,ou=people, dc=railssecurity, dc=com"
```

Commercial LDAP implementations typically come with nice GUI-based management tools. However, this is a GUI tool that comes in handy when working with OpenLDAP.

University of Chicago LDAP Browser

A Java Swing application that allows you to browse and manipulate LDAPv2- and LDAPv3-compliant servers.

If we've set up our OpenLDAP server correctly, we should be able to execute the following command:

```
> ldapsearch -x -b 'dc=railssecurity,dc=com'
```

Running this code produces a result that looks something like this:

```
# extended LDIF
#
# LDAPv3
# base <dc=railssecurity,dc=com> with scope sub
# filter: (objectclass=*)
# requesting: ALL
#
# railssecurity.com
dn: dc=railssecurity,dc=com
dc: railssecurity
objectClass: organization
objectClass: dcObject
o: railssecurity.com

# people, railssecurity.com
dn: ou=people,dc=railssecurity,dc=com
ou: people
objectClass: organizationalUnit

# wally, people, railssecurity.com
dn: cn=wally,ou=people,dc=railssecurity,dc=com
mail: wally@wallysworld.com
userPassword:: cGFzc3dvcmQ=
uid: wally
ou: people
objectClass: top
```

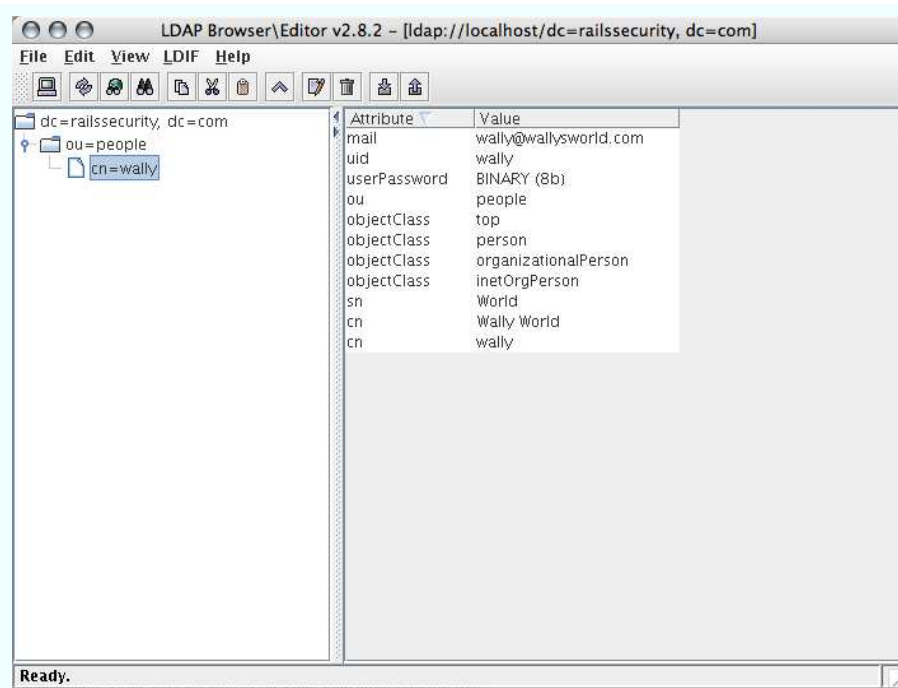


Figure 12.1: Browsing your OpenLDAP directory

```
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
sn: World
cn: Wally World
cn: wally
```

```
# search result
search: 2
result: 0 Success
```

```
# numResponses: 4
# numEntries: 3
```

And if you've installed the LDAP Browser, you should see something like Figure 12.1.

12.2 GSSAPI

GSSAPI SWIG Bindings

[Download](#) sso/gssapi_ruby/gssapi.i

```
%feature("docstring") {
  /*
   Document-module: GSSAPI
   GSSAPI Module
  */
}

%module GSSAPI

#include "typemaps.i"
#include "gssapi_wrap.h"

%{

#include <stdio.h>
#include <string.h>
#include <gssapi.h>
#include <gssapi/gssapi.h>
#include <gssapi/gssapi_generic.h>
#include <gssapi/gssapi_krb5.h>
#include "gssapi_wrap.h"

%}

typedef unsigned int    uint32_t;
typedef int             int32_t;
typedef uint32_t        gss_uint32;
typedef int32_t         gss_int32;
typedef gss_uint32      OM_uint32;
typedef gss_int32       OM_int32;
typedef int             gss_cred_usage_t;
typedef int             gss_qop_t;

%typedef unsigned int  uint32_t;
%typedef int           int32_t;
%typedef uint32_t      gss_uint32;
%typedef int32_t       gss_int32;
%typedef gss_uint32    OM_uint32;
%typedef gss_int32     OM_int32;
%typedef int           gss_cred_usage_t;
%typedef int           gss_qop_t;

#ifdef KRB5_CALLCONV
#define KRB5_CALLCONV
#define KRB5_CALLCONV_C
#endif
```

```

struct gss_name_struct;
typedef struct gss_name_struct * gss_name_t;
%typedef struct gss_name_struct * gss_name_t;

struct gss_cred_id_struct;
typedef struct gss_cred_id_struct * gss_cred_id_t;
%typedef struct gss_cred_id_struct * gss_cred_id_t;

struct gss_ctx_id_struct;
typedef struct gss_ctx_id_struct * gss_ctx_id_t;
%typedef struct gss_ctx_id_struct * gss_ctx_id_t;

typedef struct gss_OID_desc_struct {
    OM_uint32 length;
    void *elements;
} gss_OID_desc, *gss_OID;

typedef struct gss_OID_set_desc_struct {
    size_t count;
    gss_OID elements;
} gss_OID_set_desc, *gss_OID_set;

typedef struct gss_buffer_desc_struct {
    size_t length;
    void *value;
} gss_buffer_desc, *gss_buffer_t;

typedef struct gss_channel_bindings_struct {
    OM_uint32 initiator_addrtype;
    gss_buffer_desc initiator_address;
    OM_uint32 acceptor_addrtype;
    gss_buffer_desc acceptor_address;
    gss_buffer_desc application_data;
} *gss_channel_bindings_t;

%extend gss_buffer_desc_struct {
    void setValueFromString(char * new_value) {
        if(new_value == NULL)
            return;
        int len = strlen(new_value) + 1;
        char *str;
        str = (char*)malloc(sizeof(char) * len);
        strncpy(str, new_value, len);
        $self->value = str;
        $self->length = strlen(str) + 1;
    }

    void setValueFromByteArray(char * new_value, int length) {
        $self->value = new_value;
        $self->length = length;
    }
}

```

```

char *getValueAsString() {
    return (char*) $self->value;
}

%typemap(in,numinputs=0) char *OUT_ARRAY($1_ltype temp) {
    temp = ($1_ltype) malloc(sizeof($1_ltype));
    $1 = temp;
}

%typemap(in,numinputs=0) int *OUT_LEN($1_ltype temp) {
    temp = ($1_ltype) malloc(sizeof($1_ltype));
    $1 = temp;
}

%typemap(argout,fragment="output_helper") char *OUT_ARRAY {
    VALUE o = SWIG_FromCharPtrAndSize(arg1->value, arg1->length);
    $result = output_helper($result, o);
}

void getValueAsByteArray(char *OUT_ARRAY, int *OUT_LEN) {
    OUT_ARRAY = (char *)$self->value;
    OUT_LEN    = (int *)&$self->length;
}

};

/*
 * Flag bits for context-level services.
 */

#define GSS_C_DELEG_FLAG 1
#define GSS_C_MUTUAL_FLAG 2
#define GSS_C_REPLAY_FLAG 4
#define GSS_C_SEQUENCE_FLAG 8
#define GSS_C_CONF_FLAG 16
#define GSS_C_INTEG_FLAG 32
#define GSS_C_ANON_FLAG 64
#define GSS_C_PROT_READY_FLAG 128
#define GSS_C_TRANS_FLAG 256

/*
 * Credential usage options
 */
#define GSS_C_BOTH 0
#define GSS_C_INITIATE 1
#define GSS_C_ACCEPT 2

/*
 * Status code types for gss_display_status
 */
#define GSS_C_GSS_CODE 1

```

```

#define GSS_C_MECH_CODE 2

/*
 * The constant definitions for channel-bindings address families
 */
#define GSS_C_AF_UNSPEC      0
#define GSS_C_AF_LOCAL      1
#define GSS_C_AF_INET       2
#define GSS_C_AF_IMPLINK    3
#define GSS_C_AF_PUP        4
#define GSS_C_AF_CHAOS      5
#define GSS_C_AF_NS         6
#define GSS_C_AF_NBS        7
#define GSS_C_AF_ECMA      8
#define GSS_C_AF_DATAKIT    9
#define GSS_C_AF_CCITT     10
#define GSS_C_AF_SNA        11
#define GSS_C_AF_DECnet     12
#define GSS_C_AF_DLI        13
#define GSS_C_AF_LAT        14
#define GSS_C_AF_HYLINK     15
#define GSS_C_AF_APPLETALK  16
#define GSS_C_AF_BSC        17
#define GSS_C_AF_DSS        18
#define GSS_C_AF_OSI        19
#define GSS_C_AF_X25        21

#define GSS_C_AF_NULLADDR   255

/*
 * Various Null values.
 */
#define GSS_C_NO_NAME        ((gss_name_t) 0)
#define GSS_C_NO_BUFFER     ((gss_buffer_t) 0)
#define GSS_C_NO_OID        ((gss_OID) 0)
#define GSS_C_NO_OID_SET    ((gss_OID_set) 0)
#define GSS_C_NO_CONTEXT    ((gss_ctx_id_t) 0)
#define GSS_C_NO_CREDENTIAL ((gss_cred_id_t) 0)
#define GSS_C_NO_CHANNEL_BINDINGS ((gss_channel_bindings_t) 0)
#define GSS_C_EMPTY_BUFFER  {0, NULL}

#define GSS_KRB5_INTEG_C_QOP_DES_MD5 0x0002
#define GSS_KRB5_INTEG_C_QOP_DES_MAC 0x0003

/*
 * Some alternate names for a couple of the above values.  These are defined
 * for V1 compatibility.
 */
#define GSS_C_NULL_OID      GSS_C_NO_OID
#define GSS_C_NULL_OID_SET GSS_C_NO_OID_SET

```

```

#define GSS_C_QOP_DEFAULT 0

/*
 * Expiration time of 2^32-1 seconds means infinite lifetime for a
 * credential or security context
 */
#define GSS_C_INDEFINITE ((OM_uint32) 0xffffffff)

/* Major status codes */

#define GSS_S_COMPLETE 0

#define GSS_C_CALLING_ERROR_OFFSET 24
#define GSS_C_ROUTINE_ERROR_OFFSET 16
#define GSS_C_SUPPLEMENTARY_OFFSET 0
#define GSS_C_CALLING_ERROR_MASK ((OM_uint32) 0377u1)
#define GSS_C_ROUTINE_ERROR_MASK ((OM_uint32) 0377u1)
#define GSS_C_SUPPLEMENTARY_MASK ((OM_uint32) 0177777u1)
/*
 * Supplementary info bits:
 */

#define GSS_S_CONTINUE_NEEDED (1 << (GSS_C_SUPPLEMENTARY_OFFSET + 0))
#define GSS_S_DUPLICATE_TOKEN (1 << (GSS_C_SUPPLEMENTARY_OFFSET + 1))
#define GSS_S_OLD_TOKEN (1 << (GSS_C_SUPPLEMENTARY_OFFSET + 2))
#define GSS_S_UNSEQ_TOKEN (1 << (GSS_C_SUPPLEMENTARY_OFFSET + 3))
#define GSS_S_GAP_TOKEN (1 << (GSS_C_SUPPLEMENTARY_OFFSET + 4))

#if defined (_WIN32) && defined (_MSC_VER)
# ifdef GSS_DLL_FILE
#  define GSS_DLLIMP __declspec(dllexport)
# else
#  define GSS_DLLIMP __declspec(dllimport)
# endif
#else
# define GSS_DLLIMP
#endif

const gss_OID NT_USER_NAME();
const gss_OID NT_MACHINE_UID_NAME();
const gss_OID NT_STRING_UID_NAME();
const gss_OID NT_HOSTBASED_SERVICE();
const gss_OID NT_ANONYMOUS();
const gss_OID NT_EXPORT_NAME();
const gss_OID_set NULL_OID_SET();

/*
 * Typemaps: These give us the ability to reconcile the API to Ruby nicely.
 */

```

```

#define NEW_FROM_SWIG_PTR(type,swigtype,_ptr)
    VALUE o = SWIG_NewPointerObj(SWIG_as_voidptr(_ptr),
        swigtype,
        SWIG_POINTER_OWN );
#endif

#define IN_OUT_TYEMAPPER(_type, _swigtype, _ptr)
%typemap(in,numinputs=0) _type OUTPUT($1_ltype temp) {
    temp = ($1_ltype) malloc(sizeof($1_ltype));
    $1 = temp;
}

%typemap(argout,fragment="output_helper") _type OUTPUT {
    NEW_FROM_SWIG_PTR(type, _swigtype, _ptr);
    $result = output_helper($result, o);
}
#endif

IN_OUT_TYEMAPPER(gss_name_t *, SWIGTYPE_p_gss_name_struct, *$1);
IN_OUT_TYEMAPPER(gss_OID *, SWIGTYPE_p_gss_OID_desc_struct, $1);
IN_OUT_TYEMAPPER(gss_OID_set *, SWIGTYPE_p_gss_OID_set_desc_struct, $1);
IN_OUT_TYEMAPPER(gss_buffer_t, SWIGTYPE_p_gss_buffer_desc_struct, $1);
IN_OUT_TYEMAPPER(gss_cred_id_t *, SWIGTYPE_p_gss_cred_id_struct, *$1);

%typemap(in,numinputs=0) gss_buffer_t OUTPUT2(gss_buffer_desc temp) {
    $1 = GSS_C_NO_BUFFER;
}

%typemap(in,numinputs=0) gss_buffer_t OUTPUT3($1_ltype temp) {
    $1 = malloc(sizeof(gss_buffer_t));
}

%typemap(argout,fragment="output_helper") gss_buffer_t OUTPUT3 {
    NEW_FROM_SWIG_PTR((gss_buffer_t), SWIGTYPE_p_gss_buffer_desc_struct, $1);
    $result = output_helper($result, o);
}

%typemap(in) gss_name_t * IN($*1_ltype temp) {
    SWIG_ConvertPtr($input, (void **) &temp, $1_descriptor, SWIG_POINTER_OWN);
    $1 = &temp;
}

%typemap(in,numinputs=1) gss_ctx_id_t * OUTPUT($*1_ltype tmp_ctx) {
    if($input == Qnil) {
        tmp_ctx = GSS_C_NO_CONTEXT;
        $1 = &tmp_ctx;
    } else {
        void *vPtr = 0;
        // Check if vPtr is just a NULL -> (void *) NULL
        SWIG_ConvertPtr($input, &vPtr, $1_descriptor, SWIG_POINTER_OWN);
    }
}

```



```

    $1 = vPtr;
}
}

%typemap(argout,fragment="output_helper") gss_ctx_id_t *OUTPUT {
    NEW_FROM_SWIG_PTR((gss_ctx_id_t *), SWIGTYPE_p_p_gss_ctx_id_struct, *$1);
    $result = output_helper($result, o);
}

%typemap(in) gss_ctx_id_t IN($1_ltype temp) {
    SWIG_ConvertPtr($input, (void **) &temp, $&1_descriptor, SWIG_POINTER_OWN);
    $1 = temp;
}

%apply gss_name_t          *OUTPUT { gss_name_t *OUT };
%apply gss_name_t          *IN { gss_name_t *IN };
%apply int                 *OUTPUT { OM_uint32 *OUT };
%apply gss_buffer_t        OUTPUT { gss_buffer_t OUT };
%apply gss_buffer_t        OUTPUT2 { gss_buffer_t OUT2 };
%apply gss_cred_id_t       *OUTPUT { gss_cred_id_t *OUT };
%apply gss_ctx_id_t        *OUTPUT { gss_ctx_id_t *INOUT };
%apply gss_OID_set         *OUTPUT { gss_OID_set *OUT };
%apply gss_OID             *OUTPUT { gss_OID *OUT };
%apply int                 *OUTPUT { gss_cred_usage_t *OUT };
%apply int                 *OUTPUT { int *OUT };

void printErrors(OM_uint32 iMajorStatus, OM_uint32 iMinorStatus);

%feature("docstring") {
/*
    Document-method: gss_import_name

    call-seq:
        gss_import_name(GssBufferDesc name_buffer, opaque_name_type) ->
        [MajorStatus, MinorStatus, gss_name_t]

    A module function.

*/
}

OM_uint32 KRB5_CALLCONV gss_import_name
(
    OM_uint32 *OUT, /* minor_status */
    gss_buffer_t, /* input_name_buffer */
    gss_OID, /* input_name_type(used to be const) */
    gss_name_t *OUT /* output_name */
);

```

```

// Returns:
// [MajorStatus, MinorStatus, MessageContext, StatusString]
// %feature("autodoc", "0");
OM_uint32 KRB5_CALLCONV gss_display_status
(
    OM_uint32 *OUT,      /* minor_status */
    OM_uint32,          /* status_value => GSS_C_GSS_CODE | GSS_C_MECH_CODE */
    int,                /* status_type */
    gss_OID,            /* mech_type (used to be const) => GSS_C_NULL_OID */
    OM_uint32 *OUT,     /* message_context */
    gss_buffer_t OUT    /* status_string */
);

// gss_acquire_cred(...) -> [MajorStatus, MinorStatus,
//                             gss_cred_id_t, gss_OID_set, TimeRec]
OM_uint32 KRB5_CALLCONV gss_acquire_cred
(
    OM_uint32 *OUT,      /* minor_status */
    gss_name_t IN,      /* desired_name -> NULL means default */
    OM_uint32 IN,       /* time_req */
    gss_OID_set IN,     /* desired_mechs -> NULL means default */
    gss_cred_usage_t IN, /*
        cred_usage
        -- 0=INITIATE-AND-ACCEPT,
        -- 1=INITIATE-ONLY,
        -- 2=ACCEPT-ONLY
    */
    gss_cred_id_t *OUT, /* output_cred_handle */
    // if returned non-NULL,
    // -- caller must release with GSS_Release_cred()
    gss_OID_set *OUT,   /* actual_mechs */
    // if returned non-NULL,
    // -- caller must release with GSS_Release_oid_set()
    OM_uint32 *OUT     /* time_rec */
);

OM_uint32 KRB5_CALLCONV gss_display_name
(OM_uint32 *OUT, /* minor_status */
  gss_name_t IN, /* input_name */
  gss_buffer_t OUT, /* output_name_buffer */
  gss_OID *OUT /* output_name_type */
);

OM_uint32 KRB5_CALLCONV gss_inquire_cred
(
    OM_uint32 *OUT,      /* minor_status */
    gss_cred_id_t,      /* cred_handle */
    gss_name_t *OUT,    /* name */
    OM_uint32 *OUT,     /* lifetime */
    gss_cred_usage_t *OUT, /* cred_usage */
    gss_OID_set *OUT    /* mechanisms */
);

```

Download at WoweBook.com

```

// gss_release_cred(gss_cred_id_t *) -> [MajorStatus, MinorStatus]
OM_uint32 KRB5_CALLCONV gss_release_cred
(
    OM_uint32      *OUT,      /* minor_status */
    gss_cred_id_t  *IN       /* cred_handle */
);

// -> [MajorStatus, MinorStatus, ReturnedContext, actual_mech_type,
//      output_token, ret_flags, time_rec]
OM_uint32 KRB5_CALLCONV gss_init_sec_context
(
    OM_uint32      *OUT,      /* minor_status */
    gss_cred_id_t  IN,       /* claimant_cred_handle */
    gss_ctx_id_t   *INOUT,   /* context_handle */
    gss_name_t     IN,       /* target_name */
    gss_OID        IN,       /* mech_type (used to be const)
                             should use -> GSS_C_NO_OID for KRB5*/
    OM_uint32     IN,       /* req_flags */
    OM_uint32     IN,       /* time_req */
    gss_channel_bindings_t IN, /* input_chan_bindings */
    gss_buffer_t   IN,       /* input_token */
    gss_OID *     OUT,       /* actual_mech_type */
    gss_buffer_t   OUT,       /* output_token */
    OM_uint32 *   OUT,       /* ret_flags */
    OM_uint32 *   OUT,       /* time_rec */
);

OM_uint32 KRB5_CALLCONV gss_accept_sec_context
(
    OM_uint32      *OUT,      /* minor_status */
    gss_ctx_id_t   *INOUT,   /* context_handle */
    gss_cred_id_t  IN,       /* acceptor_cred_handle */
    gss_buffer_t   IN,       /* input_token_buffer */
    gss_channel_bindings_t IN, /* input_chan_bindings */
    gss_name_t     *OUT,     /* src_name */
    gss_OID        *OUT,     /* mech_type */
    gss_buffer_t   OUT,       /* output_token */
    OM_uint32 *   OUT,       /* ret_flags */
    OM_uint32 *   OUT,       /* time_rec */
    gss_cred_id_t *   OUT,   /* delegated_cred_handle */
);

OM_uint32 KRB5_CALLCONV gss_process_context_token
(
    OM_uint32 *   OUT,       /* minor_status */
    gss_ctx_id_t  IN,       /* context_handle */
    gss_buffer_t  IN,       /* token_buffer */
);

```

```

OM_uint32 KRB5_CALLCONV gss_delete_sec_context
(OM_uint32 *
  gss_ctx_id_t *
  gss_buffer_t
  OUT, /* minor_status */
  IN, /* context_handle */
  OUT2 /* output_token */
);

OM_uint32 KRB5_CALLCONV gss_release_name
(OM_uint32 *OUT, /* minor_status */
  gss_name_t *IN /* input_name */
);

OM_uint32 KRB5_CALLCONV gss_release_buffer
(OM_uint32 *OUT, /* minor_status */
  gss_buffer_t IN /* buffer */
);

OM_uint32 KRB5_CALLCONV gss_export_name
(OM_uint32 *OUT, /* minor_status */
  const gss_name_t, /* input_name */
  gss_buffer_t OUT /* exported_name */
);

OM_uint32 KRB5_CALLCONV gss_wrap
(
  OM_uint32 *OUT, /* minor_status */
  gss_ctx_id_t IN, /* context_handle */
  int IN, /* conf_req_flag */
  gss_qop_t IN, /* qop_req */
  gss_buffer_t IN, /* input_message_buffer */
  int *OUT, /* conf_state */
  gss_buffer_t OUTPUT3 /* output_message_buffer */
);

OM_uint32 KRB5_CALLCONV gss_unwrap
(
  OM_uint32 *OUT, /* minor_status */
  gss_ctx_id_t IN, /* context_handle */
  gss_buffer_t IN, /* input_message_buffer */
  gss_buffer_t OUT, /* output_message_buffer */
  int *OUT, /* conf_state */
  gss_qop_t *OUT /* qop_state */
);

/*
  KERBEROS / GSSAPI Stuff
*/

GSS_DLLIMP extern const gss_OID_desc * const GSS_KRB5_NT_PRINCIPAL_NAME;

OM_uint32 KRB5_CALLCONV krb5_gss_register_acceptor_identity(const char *);

```

```
OM_uint32 KRB5_CALLCONV gss_krb5_get_tkt_flags
(OM_uint32 *OUT,
 gss_ctx_id_t context_handle,
 krb5_flags *ticket_flags);
```

```
OM_uint32 KRB5_CALLCONV gss_krb5_copy_ccache
(OM_uint32 *OUT,
 gss_cred_id_t cred_handle,
 krb5_ccache out_ccache);
```

```
OM_uint32 KRB5_CALLCONV gss_krb5_ccache_name
(OM_uint32 *OUT, const char *name,
 const char **out_name);
```

GSSAPI SWIG C Helpers

[Download](#) sso/gssapi_ruby/gssapi_wrap.h

```
#include <gssapi.h>
#include <stdio.h>
#include <gssapi/gssapi_generic.h>
#include <string.h>

typedef struct gssapi_data {
    gss_name_t      gss_name; //opaque type
    gss_cred_id_t   credentials; //opaque type
    OM_uint32      major_status; //basic unsigned int
    OM_uint32      minor_status; //basic unsigned int
    gss_ctx_id_t    ctx; //opaque type
    gss_buffer_desc recvToken; //value / length buffer
    gss_buffer_desc sendToken; //value / length buffer
} GSSAPI_RESULT;

const char * getMechErrorMessage(OM_uint32 iMinorStatus);
const char * getGssErrorMessage(OM_uint32 iMajorStatus);
int GssImportNameHelper(const char * serviceName);
```

[Download](#) sso/gssapi_ruby/gssapi_wrap.c

```
#include "gssapi_wrap.h"

const gss_OID NT_USER_NAME() {
    return GSS_C_NT_USER_NAME;
};

const gss_OID NT_MACHINE_UID_NAME() {
    return GSS_C_NT_MACHINE_UID_NAME;
};

const gss_OID NT_STRING_UID_NAME() {
    return GSS_C_NT_STRING_UID_NAME;
};
```

```

const gss_OID NT_HOSTBASED_SERVICE() {
    return GSS_C_NT_HOSTBASED_SERVICE;
};

const gss_OID NT_ANONYMOUS() {
    return GSS_C_NT_ANONYMOUS;
};

const gss_OID NT_EXPORT_NAME() {
    return GSS_C_NT_EXPORT_NAME;
};

const gss_OID_set NULL_OID_SET() {
    return GSS_C_NULL_OID_SET;
}

const char * getGssErrorMessage(OM_uint32 iMajorStatus) {
    OM_uint32 message_context = 0;
    gss_buffer_desc status_string;
    OM_uint32 tempStatus = 0;
    printf("\nAttempting to display error message::\n");
    gss_display_status( &tempStatus, iMajorStatus,
                      GSS_C_GSS_CODE, GSS_C_NULL_OID,
                      &message_context, &status_string );

    printf("\nprinting GSS status:\n %s\n", status_string.value);
    return status_string.value;
}

const char * getMechErrorMessage(OM_uint32 iMinorStatus) {
    OM_uint32 message_context = 0;
    gss_buffer_desc status_string;
    OM_uint32 tempStatus = 0;
    printf("\nAttempting to display error message::\n");
    gss_display_status( &tempStatus, iMinorStatus,
                      GSS_C_MECH_CODE, GSS_C_NULL_OID,
                      &message_context, &status_string );

    printf("\nprinting MECH status:\n %s\n", status_string.value);
    return "";
}

void printErrors(OM_uint32 iMajorStatus, OM_uint32 iMinorStatus) {
    OM_uint32 message_context = 0;
    gss_buffer_desc status_string;

    OM_uint32 tempStatus = 0;
    gss_display_status( &tempStatus, iMajorStatus,
                      GSS_C_GSS_CODE, GSS_C_NULL_OID,
                      &message_context, &status_string );
    printf("GSS status: %s\n", status_string.value);
}

```

```

gss_display_status( &tempStatus, iMinorStatus,
                   GSS_C_MECH_CODE, GSS_C_NULL_OID,
                   &message_context, &status_string );

printf("MECH status: %s\n", status_string.value);

return;
}

int GssImportNameHelper(const char * serviceName) {
    OM_uint32 iMajorStatus;
    OM_uint32 iMinorStatus;
    gss_buffer_desc stServerNameBuffer;
    gss_name_t stServerName;

    int len = strlen(serviceName) + 1;
    stServerNameBuffer.value = (char *)serviceName;
    stServerNameBuffer.length = len;

    iMajorStatus = gss_import_name(
        &iMinorStatus,
        &stServerNameBuffer,
        GSS_C_NT_USER_NAME,
        &stServerName );

    if( iMajorStatus != GSS_S_COMPLETE ) {
        printf("Error importing.");
        OM_uint32 message_context = 0;
        gss_buffer_desc status_string;

        OM_uint32 tempStatus = 0;

        gss_display_status( &tempStatus, iMajorStatus,
                           GSS_C_GSS_CODE, GSS_C_NULL_OID,
                           &message_context, &status_string );
        printf("\nGSS status: %s", status_string.value);

        gss_display_status( &tempStatus, iMinorStatus,
                           GSS_C_MECH_CODE, GSS_C_NULL_OID,
                           &message_context, &status_string );

        printf("MECH status: %s\n", status_string.value);

        return -1;
    }

    return 0;
}

```

extconf

[Download](#) sso/gssapi_ruby/extconf.rb

```
require 'mkmf'

system('swig -v -autorename -ruby -I/usr/include -o gssapi.c gssapi.i')
have_library "gssapi_krb5"
create_makefile "gssapi"
```

12.3 SPNEGO

SPNEGO SWIG Bindings

[Download](#) sso/lunchedin_spnego/lib/spnego_ruby/spnego.i

```
%module R_spnego

%apply char *INPUT { unsigned char *pbMechToken };

#include ".././././spnegotokenhandler/src/spnego.h"

%{

#include ".././././spnegotokenhandler/src/spnego.h"
#include ".././././spnegotokenhandler/src/spnegodata.h"
#include ".././././spnegotokenhandler/src/derparse.h"
#include ".././././spnegotokenhandler/src/spnegoparse.h"

%}
```

extconf

[Download](#) sso/lunchedin_spnego/lib/spnego_ruby/extconf.rb

```
require 'mkmf'

system('swig -v -autorename -ruby -I/usr/include -o r_spnego.c spnego.i')
have_library "spnego"
create_makefile "r_spnego"
```


Chapter 13

Bibliography

- [HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.
- [RTH08] Sam Ruby, David Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

Index

A

- `access_list()`, 145
- Achilles Proxy, 261
- ActionMailer
 - digital signatures, 183–187
- Active Directory, 119–122
- ActiveRbac plug-in, 136
- ActiveRecord, 92–94
 - `attr_protected()`, 52
 - callbacks, 168
 - calls, intercepting, 152
 - cryptography and, 165–171
 - encryptor integration, 169
 - `h()`, 62
 - HTTP parameter binding, 52
 - placeholders, 93
 - sanitization and, 60
 - validation and, 62
- ActiveRecord::ConnectionAdapters::Quoting module, 92
- Administrator privileges, 33f
- Administrator role, enabling, 28, 32f
- Advanced encryption standard (AES), 163, 166
- Agile Web Development with Rails* (Ruby et al), 165
- Apache Directory Studio, 109f
 - distinguished name, 112f
 - installation, 108
 - LDAP entry, 110f
 - MUST attributes, 113f
 - object class, 111f
 - password editor, 114f
 - password, attribute type, 113f
- Application filter flow, 143f
- Application proxies, 25f, 24–25
- `approved()`, 212
- Asterisks, 90
- Asymmetric cryptography, 162f, 162–163
- Attacks. *see* Countermeasures; Web application hacking techniques
- `attr_protected()`, 52
- Authentication, 95–133
 - vs. authorization, 95
 - centralized, 95
 - concepts in, 95–98
 - decentralized, 95
 - defined, 70, 95
 - flexible, 122f, 122–133
 - LDAP/Directory-based
 - authentication, 109f, 110f, 111f, 112f, 113f, 114f, 107–122
 - OpenLDAP setup, 263
 - passwords, 165
 - sessions and contexts, 98
 - testing, 69, 71–73
 - user registration, 99–107
 - users against database, 124
 - see also* Single sign-on (SSO)
- Authentication process, 39
- Authorization, 52–57, 135–154
 - vs. authentication, 95
 - defined, 96, 135
 - discretionary access control, 138f, 139f, 140f, 138–140
 - implementing role-based access control, 143f, 151f, 142–151
 - mandatory access control, 137f, 136–138
 - planning for, 140
 - Rails support for, 136
 - role-based access control, 141f, 140–142
 - security approaches, 151–154
- Authorization checks, 35, 38f, 54
- `authorization_filter()`, 147

Authorization model, [28](#), [53](#)
 Authorization vulnerabilities, [31](#)
 Authorization, testing, [69](#), [73–74](#)

B

`begin_openid()`, [199](#)
 Behavior-driven development (BDD), [83](#)
 Bind variables, [59](#), [92](#), [93](#)
 Biometric factors, [97](#)
 Blacklist technique, [90](#)
 Blacklisting input patterns, [61](#)
 Block cipher encryption, [158](#)
 Broken authorization, [35f](#), [36f](#), [31–36](#),
 [37](#), [38f](#)
 BURP Proxy, [261](#)

C

CAPTCHA, [65](#), [99](#), [100](#), [102f](#)
 Castronaut, [253–259](#)
 authentication, [259f](#), [254–259](#)
 gem for installation, [253](#)
 registration, [254f](#), [258](#), [259f](#)
 Central Authentication Service (CAS),
 [252f](#), [254f](#), [258f](#), [259f](#), [251–259](#)
 authentication flow, [252f](#)
 Castronaut gem, [253](#)
 Centralized authentication, [95](#), [192](#)
 see also Single-sign-on (SSO)
 Certificate authority, [173](#)
 Certificate file descriptions, [180f](#)
 Certificate-signing request, [181](#)
 Certificates
 converting from PEM to DER, [177](#)
 root, [179](#)
 Character escaping, [92](#)
 Characters
 SQL construction, [89–92](#)
 Unicode, [89](#)
`check_login()`, [74](#)
 Cipher block chaining, [160](#), [164](#)
 Ciphers, [75](#), [76](#), [158](#)
 Ciphertext, [159](#), [181](#)
 Claimed identifier, [196](#)
 Collisions, [158](#)
 Command injection, [85f](#), [85–86](#), [87f](#)
 Community code, security and, [11](#)
 Complexity, [19–20](#)
 Configurable authentication, [120](#)
 Contexts, [98](#)
 Controller actions, [142](#)
 Convention, [33](#), [50](#)

Cookies, theft of, [43](#), [44](#)
 Countermeasures
 authorization, [52–57](#)
 CSRF prevention, [64–66](#)
 form fields and, [27](#)
 parameter manipulation protection,
 [49–52](#)
 SQL injection, [57–60](#)
 XSS prevention, [61–64](#)
 Credentials, [97](#)
 Cross-site request forgery (CSRF), [46f](#),
 [45–48](#)
 protecting against, [64–66](#)
 vs. XSS, [47](#)
 Cross-site scripting (XSS), [41f](#), [42f](#), [43f](#),
 [44f](#), [40–45](#)
 cookie theft, [43](#)
 vs. cross-site request forgery, [47](#)
 HTML injection, [40](#)
 overview of, [45](#)
 protecting against, [61–64](#)
 reflected XSS, [41](#)
 testing against known hacks, [78](#)
 Cryptographic software downloads, [136](#)
 Cryptography, [155–171](#)
 ActiveRecord and, [165–171](#)
 encryption and, [158–165](#)
 message digests and, [156f](#), [157f](#),
 [156–158](#)
 Ruby libraries for, [157](#)
 CSRF, *see* Cross-site request forgery

D

DAC, *see* Discretionary access control
 Data protection, [57](#), [69](#), [155–171](#)
 ActiveRecord and, [165–171](#)
 encryption, [158–165](#)
 message digests, [156f](#), [157f](#),
 [156–158](#)
 tests for, [75–78](#)
 Database-authenticated users, [124](#),
 [134f](#)
 Decentralized authentication, [95](#)
`decision()`, [214](#)
`decrypt()`, [160](#)
 Defense in Depth concept, [86](#)
 complexity and, [19–20](#)
 DRY principle, [19](#)
 fail close, [15–16](#)
 least privilege, [17–18](#)
 overview, [13–14](#)

weakest link, 14f, 14
whitelisting, 16–17

Defenses, *see* Countermeasures;

Security measures

DES algorithm, 163

Digital signatures, 172–190

certificate verification, 174f

described, 172–177

email security, 179f, 180f, 179–182

incoming email messages, 187–189

overview of, 172

signed messages, 182–187

storing certificates, 178

Discretionary access control (DAC),

135, 138f, 139f, 140f, 138–140

DRY principal, 19

E

ECB, 164

Email, digital signatures, 172–190

incoming messages, 187–189

overview of, 174f, 172–177

securing exchanges, 179f, 180f,

179–182

signed messages, 182–187

storing certificates, 178

encrypt(), 160

Encryption, 158–165, 182

encryption_key(), 167

escape(), 42

escapeHTML(), 64

extend(), 145

F

Factors

biometric, 97

defined, 97

secret, 97

Fail close, 15–16

Fail safe, 54

Fiddler, 262

Filter logic, 58

Filter methods, 60

filter_parameter_logging(*params), 75

Filters, 143f, 144

Firefox Developer Tools, 262

Fixes, *see* Countermeasures; Security measures

Fixtures, 127

flash object, 63

Flexible authentication, 122f, 122–133

Fuzz testing, 82

G

Gmail's TL enabled SMTP gateway, 186

GSSAPI, 220, 222, 267

GSSClient, 242f, 223–248

GSSServer, 228f, 229f, 223–248

H

h(), 62, 63

Hacking

convention and, 33

known XSS hacks, 78

see also Web application hacking techniques

handle_associate_request(), 209f

handle_associate_request(), 206, 208

handle_check_auth_request(), 206

handle_check_id_request(), 209

handle_check_auth_request(), 207

handle_check_id_request(), 210

Hash functions

collisions and, 158

message digests, 156

passwords and, 157

selecting, 158

HEX encoding, 43f

Hilton, Paris, 171

HTML injection, 40, 61

HTTP GET requests, 46

HTTP protocol, 65, 66

Hunt, Andrew, 19

I

ID access

protecting, 55–57

unprotected, 35, 38f

IMAP, 190

, 65

inetOrgPerson, 110

Initialization vectors (IVs), 160, 161

Injection, *see* Command injection; SQL injection

Input

rules about, 84

safe vs. unsafe, 61

string handling, improper, 88

testing, 70, 78–81

validation of, 61, 84–94

command injection, 85f, 85–86,

87f

- SQL injection, [88–92](#)
- SQL injection, Rails module for, [92–94](#)

K

- Kerberos, [220f](#), [219–220](#)
 - code with GSSAPI, [222](#)
 - credentials, [226f](#)
 - GSSAPI, [220](#)
 - GSSServer and GSSClient, [228f](#), [229f](#), [242f](#), [223–248](#)
 - GUI, [227f](#)
 - server setup, [221](#)
 - SPNEGO and, [251f](#), [248–251](#), [252f](#)
- Key distribution, [162](#)
- Keys
 - asymmetric cryptography, [162](#)
 - PKCS12 store, [178](#)
 - private, [175](#)
 - storage of, ActiveRecord, [167](#)

L

- LDAP binding, [108](#)
- LDAP/Directory-based authentication, [109f](#), [110f](#), [111f](#), [112f](#), [113f](#), [114f](#), [107–122](#)
 - Active Directory, [119–122](#)
 - flexible, [122f](#), [122–133](#)
- Least privilege, [17–18](#)
- Lightweight Directory Access Protocol, [see](#) LDAP/Directory-based authentication
- `<link>`, [65](#)
- Login
 - attacks on, [57](#)
 - testing, [71](#)
- Login attacks, [38](#)
- Login process, testing
 - see also* Authentication entries
- Logout, testing, [72](#)
- LunchedIn
 - ActiveRecord encryptor, [169](#)
 - authorization model, [28](#)
 - broken authorization, [35f](#), [36f](#), [31–36](#), [37](#), [38f](#)
 - CAPTCHA for, [100](#), [102f](#)
 - CAS authentication, [258f](#), [259f](#), [254–259](#)
 - CAS authentication flow, [252f](#)
 - Castronaut, [254f](#)
 - code for, [24](#)

- CSRF, [46f](#), [45–48](#)
- digital email signatures, [182–187](#)
- email certificate, [180f](#)
- EventController weaknesses, [55](#)
- LDAP authentication, [114f](#)
- LDAP vs. database authentication, [123](#)
 - overview of, [23–24](#)
 - parameter manipulation, [26–31](#)
 - salts, adding, [104–107](#)
 - security questions, [165](#)
 - SQL injection, [39f](#), [37–40](#)
 - voting more than once, [34](#)
- XSS, [41f](#), [42f](#), [43f](#), [44f](#), [40–45](#)

M

- Mandatory access control (MAC), [137f](#), [135–138](#)
- Message digests, [156f](#), [157f](#), [156–158](#)
- Metaprogramming, [184](#)
- Metaprogramming, Ruby, [146](#)
- MIME, *see* S/MIME
- ModelSecurity, [152](#)
- ModelSecurity plug-in, [136](#)

N

- Named bind variables, [59](#), [92](#), [93](#)

O

- Odysseus Proxy, [262](#)
- OpenID, [192](#)
 - association, [197](#)
 - authentication flow, [193f](#)
 - described, [193](#)
 - discovery, [194](#), [195f](#)
 - gem for, [194](#)
 - overview of, [193–204](#)
 - parameters, [202f](#)
 - provider implementation, [205f](#), [207f](#), [208f](#), [209f](#), [210f](#), [216f](#), [204–216](#), [217–219f](#)
 - redirect process, [200](#), [201f](#)
 - relaying party authentication, [205f](#)
 - storage, [209](#)
- OpenLDAP
 - directory, browsing, [266](#)
 - setup, [263](#)
- OpenSSL, [159](#)
- OpenSSL::Cipher.random_key(), [160](#)
- Output, sanitization of, [62](#)

P

Parameter manipulation, 28f, 29f, 30f, 31f, 26–31, 32, 33f
 protecting against, 49–52
 testing, 75–78

Parameters, binding of, 50

Paros, 261

Passwords

- as factors, 97
- hash verification, 157
- hashes and, 157
- parameters and, 75
- plaintext, 119
- salts, 103–104
- salts and, 119
- security questions for, 165
- strength, 73
- web application security, 103
see also LDAP

Patches, *see* Countermeasures;
 Security measures

Payroll data example, 136

Pen-testing, 27

Perens, Bruce, 152

Persistent XSS, cookie theft, 43, 44f

PKCS12 key store, 178

Placeholders, 93

Plaintext, 119, 159

Polymorphic user models, 125f

*The Pragmatic Programmer: From
 Journeyman to Master* (Hunt &
 Thomas), 19

Prevention, *see* Security measures

Principals, 53, 96, 97
see also Authorization entries

Private keys, 175

ProfileController, 152

protect_from_forgery(), 65

Proxy applications, 261–262

Public vs. private users, 192

Q

Query construction, 58

Quotations, 90

quote_string(s), 92

R

Rails testing stack, 71f

rails_rcov plug-in, 81

RatingsController, 53

Rcov, 82, 83f

Reflected cross-site scripting, 41

Registration, user, 99–107

Remote access, GET requests for, 46

Reply email, 187–188

Rijndael, 163

RMagick, 99

Role declarations, 151f

Role-based access control (RBAC), 53,
 135, 141f, 140–142
 approaches to, 142
 implementing, 143f, 151f, 142–151

RoleBasedControllerAuthorization, 144

Roles, 53, 138, 140, 144

Root certificates, 179

RSA private keys, 175

RSpec, 83

ruby-net-ldap gem, 116

S

S/MIME authentication, 189

S/MIME message attachments, 179f,
 179

S/MIME signatures, 181

Salts, 103–104, 158
 adding to LunchedIn, 104–107
 defined, 103
 LDAP and, 119

Sanitization, 60, 62, 70
 SQL injection and, 90
 testing, 78–81

Sanitization logic, 58

save(), 61

save_with_captcha(), 101

Secret factors, 97

Secure Sockets Layer (SSL), 98, 172

Security

- addressing vulnerabilities, 49
- community code and, 11
- complexity and, 19–20
- concerns with, 10
- Defense in Depth concept, 13–14
- fail close and, 15–16
- least privilege, 17–18
- mind-set of, 12
- weakest link and, 14f, 14
- whitelisting, 16–17
see also Security measures; Testing

Security boundaries, 18

Security measures

- authorization, 52–57

CSRF prevention, 64–66
 parameter manipulation, 49–52
 SQL injection prevention, 57–60
 XSS prevention, 61–64
see also Data protection; Testing
 Security questions, 165
 Security through obscurity, 40
 Selenium, 78–81
 self.included(), 145
 Self-signed certificates, 174–177
 Serpent algorithm, 163
 Service accounts, 96
 Services, 96
 Session harvesting, 44
 Session timeouts, 65
 SessionController, 57, 71, 76
 Sessions, 98
 show(), 55
 simple_captcha plug-in, 99
 Single quotation marks, 37
 Single sign-on (SSO), 191–259
 CAS, 252f, 254f, 258f, 259f, 251–259
 Kerberos, 220f, 219–220
 OpenID, 195f, 202f, 193–204, 205f
 association, 197
 authentication flow, 193f
 redirect process, 200, 201f
 OpenID provider, implementing,
 205f, 207f, 208f, 209f, 210f, 216f,
 204–216, 217–219f
 public vs. private, 192
 SPIKE Proxy, 261
 SPNEGO, 251f, 248–251, 252f, 280
 SQL injection, 39f, 37–40
 characters, dangerous, 89–92
 consequences of, 40
 defined, 88
 protecting against, 57–60
 Rails module for, 92–94
 validation, 88–92
 SQL, true statements, 39
 ssl_required plug-in, 80
 ssl_requirement plug-in, 74, 76
 SSO, *see* Single sign-on (SSO)
 Symmetric cryptography, 159–162
 Symmetric encryption, 163

T

Tamper Data, 25f
 broken authorization, 34, 36f
 form elements, adding, 31f

requests, outgoing, 37f
 requests, replaying, 36f
 starting, 29f
 users, creating, 30f
 Tarantula, 83
 Testing, 68–83
 authentication, 71–73
 authorization, 73–74
 frameworks for, 82
 input validation, 78–81
 layers, 70, 71f
 LDAP data, 128
 LDAP user authentication, 129
 measuring coverage of, 81–82, 83f
 parameters, 75–78
 role declarations, 145
 self-signed certificates, 174
 what to test for, 68–70
 see also Countermeasures; Security
 measures
 Thomas, Dave, 19
 Timeouts, 65
 TLS-enabled SMTP gateway, 186
 to_i(), 91
 Transport Layer Security (TLS), 172
 Twofish algorithm, 163

U

Unicode characters, 89
 unless keyword, 54
 update(), 56
 url_for_user(), 212
 URL-encoded values, 43f
 User interface, 150
 User registration, 99–107, 130, 134f
 User stories, authorization, 140
 Users, 96, *see* Authorization entries
 concerns, in LunchedIn, 123
 database authentication, 124
 demo, 215
 LDAP and, 110
 LDAP authentication, 124
 as nil, 147
 polymorphic model, 125f
 public vs. private, 192

V

validates_format_of(), 78
 Validation, 84–94
 command injection, 85f, 85–86, 87f
 data rules for, 84

- defined, [84](#)
- of input, [61](#)
- SQL injection, [88–92](#)
- SQL injection, Rails module for, [92–94](#)
- Variable interpolation, [58](#)
- Verification
 - digital certificates, [174f](#)
 - of email messages, [189](#)
- verify(), [66](#)
- Voting links
 - fixing, [53](#)
 - hacking, [34](#)
- Vulgar lyrics example, [141](#)
- Vulnerabilities, *see* Web application hacking techniques

W

- “Wally Webcoder”, [36](#), [110](#)
- Weakest link, securing, [14f](#), [14](#)

- Web application hacking techniques, [23–48](#)
 - application proxies, [25f](#), [24–25](#)
 - broken authorization, [35f](#), [36f](#), [31–36](#), [37](#), [38f](#)
 - cross-site request forgery, [46f](#), [45–48](#)
 - LunchedIn, overview of, [23–24](#)
 - parameter manipulation, [28f](#), [29f](#), [30f](#), [31f](#), [26–31](#), [32](#), [33f](#)
 - SQL injection, [39f](#), [37–40](#)
 - XSS, [41f](#), [42f](#), [43f](#), [44f](#), [40–45](#)
 - see also* Countermeasures
- Webstretch Proxy, [262](#)
- Whitelisting, [16–17](#)
- Whitelists, [90](#)
- win32-open3 gem, [79](#)

X

- X.509 digital certificates, [174f](#), [173–174](#)
- XSS, *see* Cross-site scripting

The Pragmatic Bookshelf

Download at WoweBook.com

Available in paperback and DRM-free PDF, our titles are here to help you stay on top of your game. The following are in print as of December 2009; be sure to check our website at pragprog.com for newer titles.

Title	Year	ISBN	Pages
<i>Advanced Rails Recipes: 84 New Ways to Build Stunning Rails Apps</i>	2008	9780978739225	464
<i>Agile Coaching</i>	2009	9781934356432	250
<i>Agile Retrospectives: Making Good Teams Great</i>	2006	9780977616640	200
<i>Agile Web Development with Rails, Third Edition</i>	2009	9781934356166	784
<i>Augmented Reality: A Practical Guide</i>	2008	9781934356036	328
<i>Behind Closed Doors: Secrets of Great Management</i>	2005	9780976694021	192
<i>Best of Ruby Quiz</i>	2006	9780976694076	304
<i>Core Animation for Mac OS X and the iPhone: Creating Compelling Dynamic User Interfaces</i>	2008	9781934356104	200
<i>Core Data: Apple's API for Persisting Data on Mac OS X</i>	2009	9781934356326	256
<i>Data Crunching: Solve Everyday Problems using Java, Python, and More</i>	2005	9780974514079	208
<i>Debug It! Find, Repair, and Prevent Bugs in Your Code</i>	2009	9781934356289	232
<i>Deploying Rails Applications: A Step-by-Step Guide</i>	2008	9780978739201	280
<i>Design Accessible Web Sites: 36 Keys to Creating Content for All Audiences and Platforms</i>	2007	9781934356029	336
<i>Desktop GIS: Mapping the Planet with Open Source Tools</i>	2008	9781934356067	368
<i>Developing Facebook Platform Applications with Rails</i>	2008	9781934356128	200
<i>Enterprise Integration with Ruby</i>	2006	9780976694069	360
<i>Enterprise Recipes with Ruby and Rails</i>	2008	9781934356234	416
<i>Everyday Scripting with Ruby: for Teams, Testers, and You</i>	2007	9780977616619	320
<i>FXRuby: Create Lean and Mean GUIs with Ruby</i>	2008	9781934356074	240
<i>From Java To Ruby: Things Every Manager Should Know</i>	2006	9780976694090	160
<i>GIS for Web Developers: Adding Where to Your Web Applications</i>	2007	9780974514093	275
<i>Google Maps API, V2: Adding Where to Your Applications</i>	2006	PDF-Only	83
<i>Grails: A Quick-Start Guide</i>	2009	9781934356463	200

Continued on next page

Title	Year	ISBN	Pages
<i>Groovy Recipes: Greasing the Wheels of Java</i>	2008	9780978739294	264
<i>Hello, Android: Introducing Google's Mobile Development Platform</i>	2009	9781934356494	272
<i>Interface Oriented Design</i>	2006	9780976694052	240
<i>Land the Tech Job You Love</i>	2009	9781934356265	280
<i>Learn to Program, 2nd Edition</i>	2009	9781934356364	230
<i>Manage It! Your Guide to Modern Pragmatic Project Management</i>	2007	9780978739249	360
<i>Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects</i>	2009	9781934356296	200
<i>Mastering Dojo: JavaScript and Ajax Tools for Great Web Experiences</i>	2008	9781934356111	568
<i>Modular Java: Creating Flexible Applications with OSGi and Spring</i>	2009	9781934356401	260
<i>No Fluff Just Stuff 2006 Anthology</i>	2006	9780977616664	240
<i>No Fluff Just Stuff 2007 Anthology</i>	2007	9780978739287	320
<i>Pomodoro Technique Illustrated: The Easy Way to Do More in Less Time</i>	2009	9781934356500	144
<i>Practical Programming: An Introduction to Computer Science Using Python</i>	2009	9781934356272	350
<i>Practices of an Agile Developer</i>	2006	9780974514086	208
<i>Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications</i>	2004	9780974514031	176
<i>Pragmatic Thinking and Learning: Refactor Your Wetware</i>	2008	9781934356050	288
<i>Pragmatic Unit Testing in C# with NUnit</i>	2007	9780977616671	176
<i>Pragmatic Unit Testing in Java with JUnit</i>	2003	9780974514017	160
<i>Pragmatic Version Control Using Git</i>	2008	9781934356159	200
<i>Pragmatic Version Control using CVS</i>	2003	9780974514000	176
<i>Pragmatic Version Control using Subversion</i>	2006	9780977616657	248
<i>Programming Clojure</i>	2009	9781934356333	304
<i>Programming Cocoa with Ruby: Create Compelling Mac Apps Using RubyCocoa</i>	2009	9781934356197	300
<i>Programming Erlang: Software for a Concurrent World</i>	2007	9781934356005	536
<i>Programming Groovy: Dynamic Productivity for the Java Developer</i>	2008	9781934356098	320
<i>Programming Ruby: The Pragmatic Programmers' Guide, Second Edition</i>	2004	9780974514055	864
<i>Programming Ruby 1.9: The Pragmatic Programmers' Guide</i>	2009	9781934356081	960
<i>Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine</i>	2009	9781934356319	250

Continued on next page

Title	Year	ISBN	Pages
<i>Prototype and script.aculo.us: You Never Knew JavaScript Could Do This!</i>	2007	9781934356012	448
<i>Rails Recipes</i>	2006	9780977616602	350
<i>Rails for .NET Developers</i>	2008	9781934356203	300
<i>Rails for Java Developers</i>	2007	9780977616695	336
<i>Rails for PHP Developers</i>	2008	9781934356043	432
<i>Rapid GUI Development with QtRuby</i>	2005	PDF-Only	83
<i>Release It! Design and Deploy Production-Ready Software</i>	2007	9780978739218	368
<i>Scripted GUI Testing with Ruby</i>	2008	9781934356180	192
<i>Ship it! A Practical Guide to Successful Software Projects</i>	2005	9780974514048	224
<i>Stripes ...and Java Web Development Is Fun Again</i>	2008	9781934356210	375
<i>TextMate: Power Editing for the Mac</i>	2007	9780978739232	208
<i>The Definitive ANTLR Reference: Building Domain-Specific Languages</i>	2007	9780978739256	384
<i>The Passionate Programmer: Creating a Remarkable Career in Software Development</i>	2009	9781934356340	200
<i>ThoughtWorks Anthology</i>	2008	9781934356142	240
<i>Ubuntu Kung Fu: Tips, Tricks, Hints, and Hacks</i>	2008	9781934356227	400
<i>iPhone SDK Development</i>	2009	9781934356258	576

More Ruby in Depth...

Metaprogramming Ruby

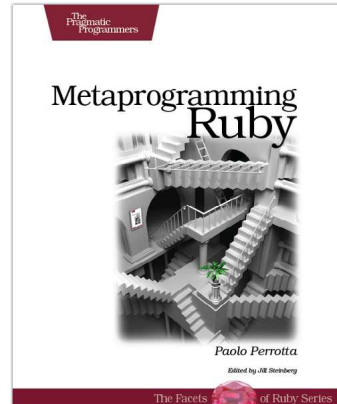
This book describes metaprogramming as an essential component of Ruby. Once you understand the principles of Ruby, including the object model, scopes, and eigenclasses, you're on your way to applying metaprogramming both in your daily work and in your fun, after-hours projects. Whether you're a Ruby apprentice on the path to mastering the language or a Ruby wiz in search of new tips, this book is for you.

Metaprogramming Ruby

Paolo Perrotta

(240 pages) ISBN: 9781934356470. \$32.95

<http://pragprog.com/titles/ppmetr>



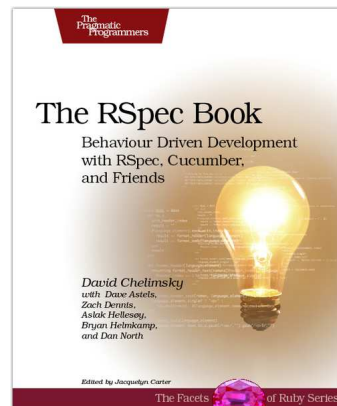
The RSpec Book

RSpec, Ruby's leading Behaviour Driven Development tool, helps you do TDD right by embracing the design and documentation aspects of TDD. It encourages readable, maintainable suites of code examples that not only test your code, they document it as well. *The RSpec Book* will teach you how to use RSpec, Cucumber, and other Ruby tools to develop truly agile software that gets you to market quickly and maintains its value as evolving market trends drive new requirements.

The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends

David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellestøy, Bryan Helmkamp, Dan North
(350 pages) ISBN: 978-1-9343563-7-1. \$42.95

<http://pragprog.com/titles/achbd>



The Home of Ruby and Rails

Agile Web Development with Rails

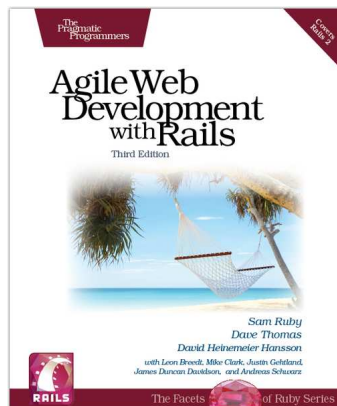
Rails is a full-stack, open-source web framework, with integrated support for unit, functional, and integration testing. It enforces good design principles, consistency of code across your team (and across your organization), and proper release management. This is the newly updated Third Edition, which goes beyond the award winning previous editions with new material covering the latest advances in Rails 2.0.

Agile Web Development with Rails: Third Edition

Sam Ruby, Dave Thomas, and David Heinemeier Hansson, et al.

(784 pages) ISBN: 978-1-9343561-6-6. \$43.95

<http://pragprog.com/titles/rails3>



Programming Ruby 1.9 (The Pickaxe for 1.9)

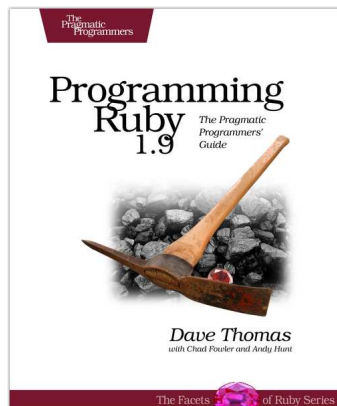
The Pickaxe book, named for the tool on the cover, is the definitive reference to this highly-regarded language.

- Up-to-date and expanded for Ruby version 1.9
- Complete documentation of all the built-in classes, modules, and methods
- Complete descriptions of all standard libraries
- Learn more about Ruby's web tools, unit testing, and programming philosophy

Programming Ruby 1.9: The Pragmatic Programmers' Guide

Dave Thomas with Chad Fowler and Andy Hunt
(992 pages) ISBN: 978-1-9343560-8-1. \$49.95

<http://pragprog.com/titles/ruby3>



More for Your Career...

The Passionate Programmer

This book is about creating a remarkable career in software development. Remarkable careers don't come by chance. They require thought, intention, action, and a willingness to change course when you've made mistakes. Most of us have been stumbling around letting our careers take us where they may. It's time to take control.

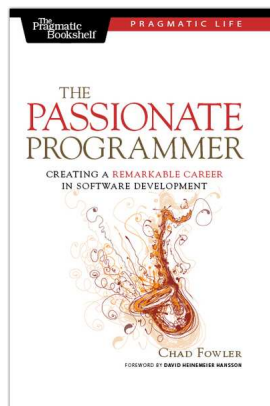
This revised and updated second edition lays out a strategy for planning and creating a radically successful life in software development (*the first edition was released as My Job Went to India: 52 Ways To Save Your Job*).

The Passionate Programmer: Creating a Remarkable Career in Software Development

Chad Fowler

(200 pages) ISBN: 978-1934356-34-0. \$23.95

<http://pragprog.com/titles/cfcar2>



Land the Tech Job You Love

You've got the technical chops—the skills to get a great job doing what you love. Now it's time to get down to the business of planning your job search, focusing your time and attention on the job leads that matter, and interviewing to wow your boss-to-be.

You'll learn how to find the job you want that fits you and your employer. You'll uncover the hidden jobs that never make it into the classifieds or Monster. You'll start making and maintaining the connections that will drive your future career moves.

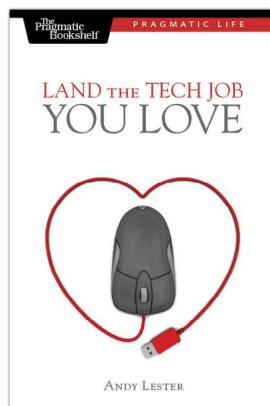
You'll land the tech job you love.

Land the Tech Job You Love

Andy Lester

(225 pages) ISBN: 978-1934356-26-5. \$23.95

<http://pragprog.com/titles/algh>



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Security on Rails' Home Page

http://pragprog.com/titles/fr_secure

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/fr_secure.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)