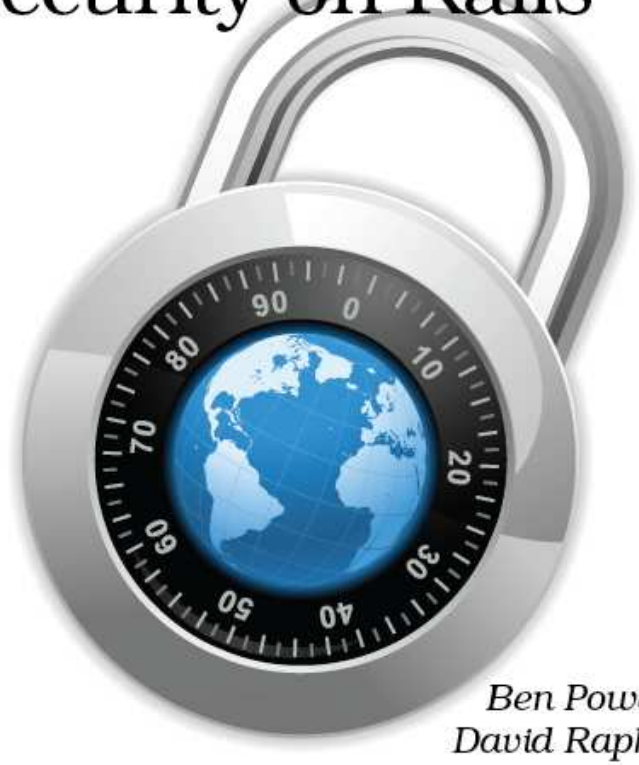
Extracted from:

Security on Rails

The Pragmatic Programmers

Security on Rails



Ben Poweski
David Raphael

Edited by Colleen Toporek

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

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: 2010-1-16

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.

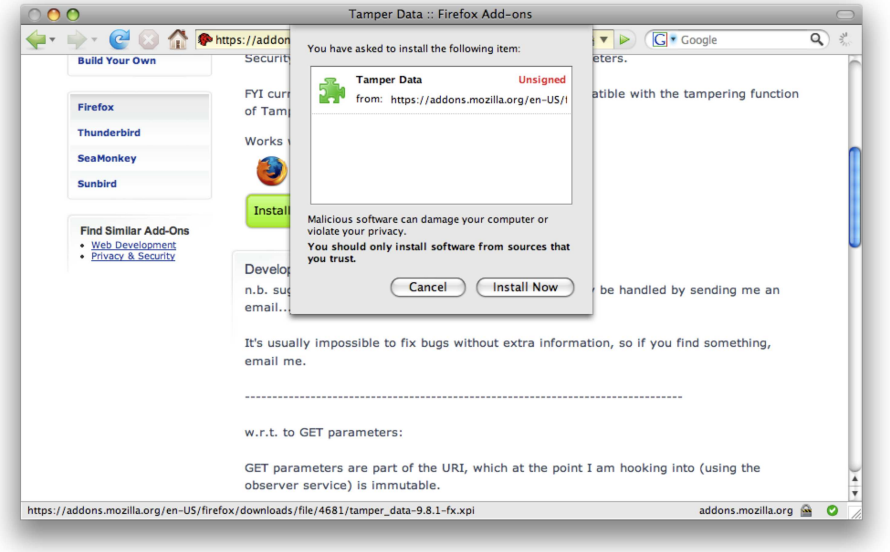


Figure 2.1: Installing Tamper Data

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 263. 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.

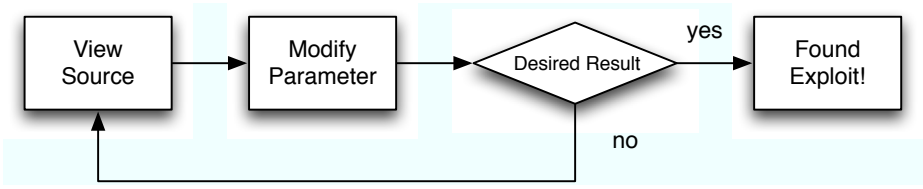


Figure 2.2: Testing process for parameter manipulation

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"/>
```

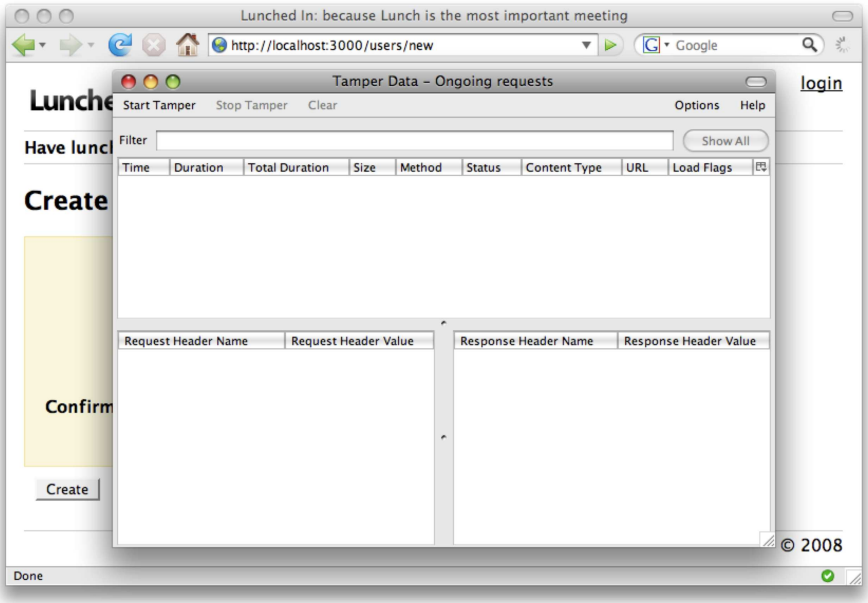


Figure 2.3: Starting Tamper Data

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 34.

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.

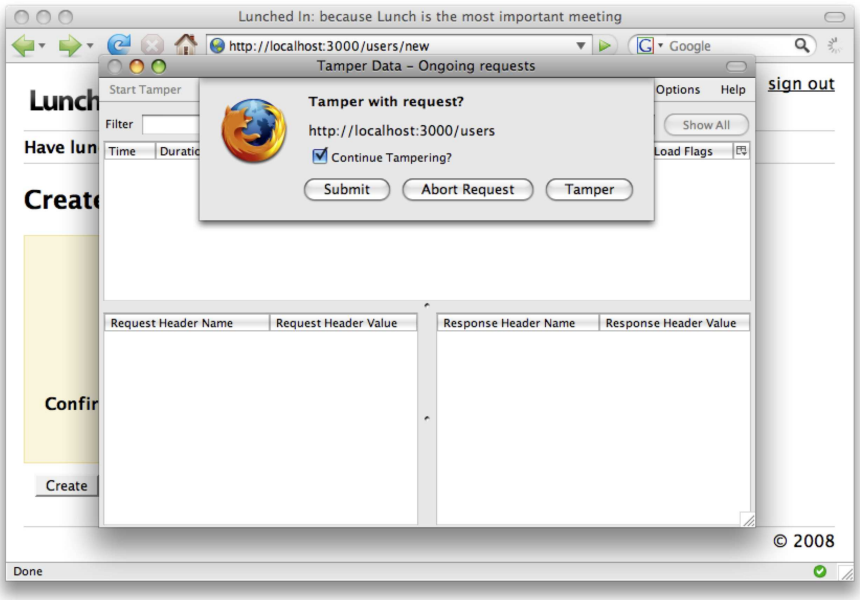


Figure 2.4: Creating a user

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 OK.

```
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 OK 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 35. We have changed the role of the user, so the

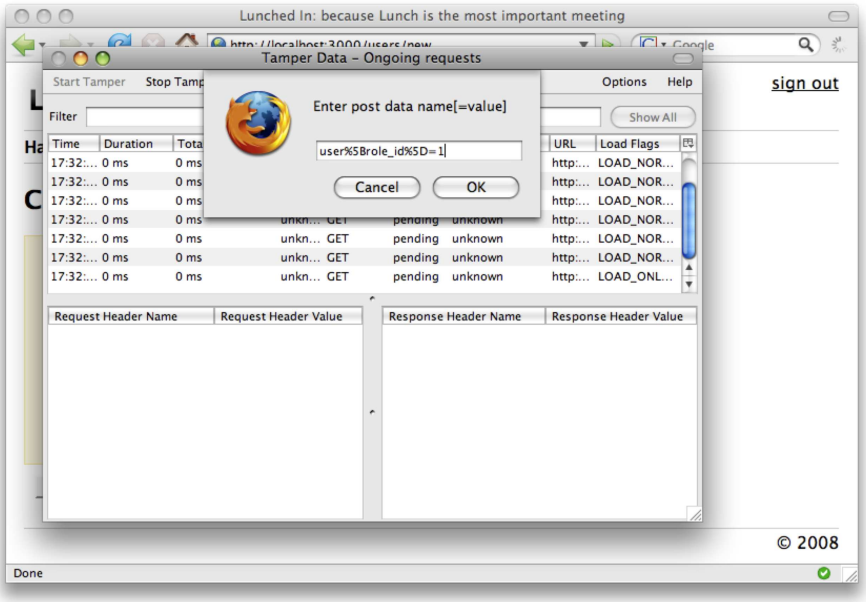


Figure 2.5: Adding a form element with Tamper Data

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.

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)